Análisis de Principios de Diseño de Software
Aplicados en el Proyecto MyGymJavaWeb

Sergio Leonardo Moreno Granado
Código: 20242020091

24 de octubre de 2025

Resumen

Este documento detalla la aplicación de principios fundamentales de diseño de software en el desarrollo del proyecto MyGymJavaWeb, una aplicación web de gestión de gimnasio. El objetivo es evidenciar cómo estos principios han guiado la arquitectura del sistema para hacerlo más mantenible, escalable y robusto, incluso utilizando un enfoque simple sin frameworks complejos.

1. Principio de Abierto/Cerrado (Open/Closed Principle - OCP)

1.1. Definición

El principio de Abierto/Cerrado establece que las entidades de software deben estar **abiertas para la extensión, pero cerradas para la modificación**.

1.2. Aplicación en MyGymJavaWeb

Este principio se observa en el `GenerarRutinaServlet`. El método `generarRutinaLogica` construye la rutina base. Esta parte está çerrada". Cuando añadimos funcionalidades como el ajuste por edad o la priorización muscular, lo hacemos extendiendo la lógica *alrededor* de este núcleo, no modificándolo.

```
private Rutina generarRutinaLogica(Usuario usuario, List<Ejercicio>
    ↪ todosLosEjercicios) {
    // ... (cálculo de edad y series base) ...

    // --- PARTE CERRADA: Creación de la rutina base ---
    switch (usuario.getDiasDisponibles()) {
        // ... Lógica para 2, 3, o 4 días ...
    }

    // --- PARTE ABIERTA: Extensión de la funcionalidad ---
    String prioridad = usuario.getPrioridadMuscular();
    if (prioridad != null && !prioridad.equals("sin_preferencia")) {
        agregarEjercicioPrioritario(rutina, todosLosEjercicios, usuario,
            ↪ ...);
    }
    return rutina;
}
```

Listing 1: Extensión de la lógica en `GenerarRutinaServlet.java`

2. Principio de Sustitución de Liskov (Liskov Substitution Principle - LSP)

2.1. Definición

Este principio afirma que los objetos de una superclase deben poder ser reemplazados por objetos de una subclase sin afectar la correctitud del programa.

2.2. Aplicación en MyGymJavaWeb

El diseño de MyGymJavaWeb no hace un uso extensivo de la herencia, por lo que no hay un ejemplo claro de LSP. Esta decisión arquitectónica en sí misma honra el principio al evitar los problemas comunes de jerarquías de herencia mal diseñadas. El sistema no viola el LSP porque no crea las condiciones para hacerlo.

3. Principio KISS (Keep It Simple, Stupid)

3.1. Definición

KISS aboga por la simplicidad y sostiene que la mayoría de los sistemas funcionan mejor si se mantienen simples en lugar de complicados.

3.2. Aplicación en MyGymJavaWeb

El principio KISS es la piedra angular del proyecto. Se manifiesta en:

- **Persistencia de Datos:** Se usaron archivos JSON en lugar de una base de datos SQL completa, una solución más simple para la escala del proyecto.
- **Lógica Directa:** Las reglas de negocio, como el ajuste por edad, se implementan de forma legible y sin complejidad innecesaria.

```
// Ajuste por edad: Simple, directo y efectivo.
if (edad > 50 && seriesYRepeticionesBase.startsWith("4")) {
    seriesYRepeticionesBase = "3" + seriesYRepeticionesBase.substring(1);
} else if (edad > 50 && seriesYRepeticionesBase.startsWith("5")) {
     seriesYRepeticionesBase = "4" + seriesYRepeticionesBase.substring(1);
}
```

Listing 2: Lógica KISS para el ajuste por edad en `GenerarRutinaServlet.java`

4. Principio DRY (Don’t Repeat Yourself)

4.1. Definición

El principio DRY establece que çada pieza de conocimiento debe tener una representación única”. En la práctica, significa evitar la duplicación de código.

4.2. Aplicación en MyGymJavaWeb

El ejemplo más claro es `JsonUtil`. En lugar de que cada servlet implemente su propia lógica para leer/escribir JSON, esta responsabilidad se abstrae en una única clase. Si necesitamos cambiar la persistencia, solo modificamos `JsonUtil`.

```java
public class JsonUtil {
    // ... rutas a los archivos JSON ...

    public static List<Usuario> leerUsuarios(String realPath) throws
        ↪ IOException { /* ... */ }
    public static void escribirUsuarios(List<Usuario> usuarios, String
        ↪ realPath) throws IOException { /* ... */ }
    // ... otros métodos para ejercicios y rutinas ...
}
```

Listing 3: Centralización de la lógica de persistencia en `JsonUtil.java`

5. Principio YAGNI (You Ain't Gonna Need It)

5.1. Definición

YAGNI establece que no se deben añadir funcionalidades hasta que no sean estrictamente necesarias, evitando la sobreingeniería.

5.2. Aplicación en MyGymJavaWeb

Se aplicó al construir la app de forma incremental. El seguimiento de progreso es un ejemplo: se implementó la versión más simple (guardar y marcar como completada) en lugar de un sistema complejo de registro de pesos que no se había solicitado.

```java
public class RutinaGuardada {
    private long id;
    private int usuarioId;
    private Date fechaGuardada;
    private String estado; // Solo "Guardada" o "Completada"
    private Rutina rutina;

    // No se añadieron campos como:
    // private String comentariosUsuario;
    // private Map<Integer, String> pesosLevantados;
}
```

Listing 4: Modelo de datos simple en `RutinaGuardada.java` siguiendo YAGNI

6. Principio de Responsabilidad Única (Single Responsibility Principle - SRP)

6.1. Definición

Este principio establece que una clase debe tener **una, y solo una, razón para cambiar**. Esto significa que cada clase debe tener una única responsabilidad o propósito bien definido.

6.2. Aplicación en MyGymJavaWeb

Nuestra arquitectura de servlets sigue este principio de forma natural. Cada servlet tiene una única responsabilidad:

- `LoginServlet`: Su única razón para cambiar es si la lógica de autenticación se modifica. No se preocupa por el registro ni por la generación de rutinas.
- `RegistroServlet`: Solo se encarga de crear nuevos usuarios.
- `GenerarRutinaServlet`: Su única responsabilidad es la compleja lógica de negocio de crear un plan de entrenamiento.

Esta separación hace que el sistema sea mucho más fácil de entender y mantener. Si hay un error en el login, sabemos exactamente qué archivo revisar.

```java
@WebServlet("/login")
public class LoginServlet extends HttpServlet {

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

        String email = request.getParameter("email");
        String password = request.getParameter("password");

        // Lógica exclusiva de autenticación...
        // ... itera sobre los usuarios y verifica las credenciales ...

        // Si tiene éxito:
        // ... crea la sesión y redirige al dashboard ...

        // Si falla:
        // ... redirige de vuelta al login con un error ...
    }
}
```

Listing 5: `LoginServlet.java`, un ejemplo de SRP

7. Principio de Inversión de Dependencias (Dependency Inversion Principle - DIP)

7.1. Definición

El DIP establece que los módulos de alto nivel no deben depender de los módulos de bajo nivel; ambos deben depender de abstracciones. Además, las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones.

7.2. Aplicación en MyGymJavaWeb

Aunque no usamos interfaces explícitas, el espíritu del DIP está presente en la relación entre los Servlets (alto nivel) y `JsonUtil` (bajo nivel). Los servlets no saben *cómo* se guardan los datos. No contienen código que manipule archivos, escriba bytes o parse JSON. La clase `JsonUtil` actúa como una **abstracción** sobre el mecanismo de persistencia. Los servlets solo dependen de los métodos públicos de `JsonUtil`. Si mañana decidimos cambiar la persistencia de JSON a una base de datos SQL, solo necesitaríamos re-implementar los métodos dentro de `JsonUtil` (o crear una nueva clase que cumpla el mismo çontrato") sin tener que cambiar una sola línea de código en nuestros servlets.

```java
// Dentro de un Servlet (módulo de alto nivel)
String realPath = getServletContext().getRealPath("/");
```

```

// No sabe que esto es un archivo JSON. Solo llama a la abstracción.
List<Usuario> usuarios = JsonUtil.leerUsuarios(realPath);

// ... lógica de negocio ...

// De nuevo, no le importa el mecanismo de guardado.
JsonUtil.escribirUsuarios(usuarios, realPath);
```

Listing 6: Servlet dependiendo de la abstracción `JsonUtil`

8. Separación de Conceptos (Separation of Concerns - SoC)

8.1. Definición

Este es un principio de diseño más general que aboga por dividir un programa en secciones distintas, donde cada sección aborda una preocupación o responsabilidad separada. Es la base de muchos patrones de diseño, incluido el patrón Modelo-Vista-Controlador (MVC).

8.2. Aplicación en MyGymJavaWeb

Nuestra aplicación sigue una arquitectura MVC clásica, que es el ejemplo perfecto de SoC:

- **Modelo (‘com.agym.modelo‘):** Se preocupa únicamente de representar los datos de la aplicación (Usuario, Ejercicio, etc.). No sabe cómo se muestran ni cómo se manipulan.
- **Vista (‘webapp‘):** Se preocupa únicamente de la presentación de los datos al usuario (archivos JSP, HTML, CSS). No contiene lógica de negocio.
- **Controlador (‘com.agym.controlador‘):** Actúa como el intermediario. Se preocupa de recibir las peticiones del usuario, interactuar con el Modelo para procesar datos y decidir qué Vista mostrar.

Esta separación es fundamental para la organización del proyecto. Por ejemplo, el `GenerarRutinaServlet` (Controlador) prepara los datos y luego simplemente los ”pasa.ª `rutina.jsp` (Vista) para que esta se encargue de mostrarlos.

```
// Al final del doPost() en GenerarRutinaServlet (Controlador)

// 1. Prepara los datos (atributos) para la Vista
request.setAttribute("rutina", rutina);
request.setAttribute("objetivoUsuario", usuario.getObjetivo());

// 2. Delega la responsabilidad de renderizar a la Vista (rutina.jsp)
request.getRequestDispatcher("rutina.jsp").forward(request, response);
```

Listing 7: El Controlador delega la presentación a la Vista