

Análisis de Principios de Diseño de Software

Aplicados en el Proyecto MyGymJavaWeb

Sergio Leonardo Moreno Granado

Código: 20242020091

27 de octubre de 2025

Resumen

Este documento detalla la aplicación de principios fundamentales de diseño de software en el desarrollo del proyecto MyGymJavaWeb. El objetivo es evidenciar cómo estos principios han guiado la arquitectura del sistema para hacerlo más mantenible, escalable y robusto, utilizando un enfoque que combina patrones de diseño como el Template Method y Factory.

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

El ejemplo más claro de OCP es el sistema de generación de rutinas.

- **Cerrado a la modificación:** La clase `GeneradorRutinaBase` contiene la lógica principal en su método final `generar()`. Este método define el algoritmo (ajuste por edad, series/repeticiones, etc.) y no puede ser modificado por las subclases. El código cliente, como el `GenerarRutinaServlet`, tampoco necesita cambios.
- **Abierto a la extensión:** Para añadir una nueva rutina (ej. para 5 días), no se modifica el código existente. Simplemente se crea una nueva clase `GeneradorRutina5Dias` que herede de `GeneradorRutinaBase` y se actualiza la `GeneradorRutinaFactory` para que la reconozca.

```
1 public class GeneradorRutinaFactory {
2     public static GeneradorRutinaBase getGenerador(int diasDisponibles) {
3         switch (diasDisponibles) {
4             case 2:
5                 return new GeneradorRutina2Dias();
6             case 3:
7                 return new GeneradorRutina3Dias();
8             case 4:
9                 return new GeneradorRutina4Dias();
10            // Para extender, se añadiría aquí un "case 5:"
11            default:
12                throw new IllegalArgumentException("Número de días no
13                    ↳ soportado.");
14        }
15    }
16 }
```

15 }

Listing 1: La Factory es el único punto de extensión, manteniendo el resto del código cerrado

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 LSP se cumple perfectamente en el diseño del generador de rutinas. El `GenerarRutinaServlet` (cliente) opera con la abstracción `GeneradorRutinaBase`, sin conocer la implementación concreta. Puede recibir un objeto `GeneradorRutina2Dias`, `GeneradorRutina3Dias`, o cualquier otra subclase futura de la factory, y el programa funcionará correctamente porque todas las subclases respetan el contrato de la clase base.

```
1 // En GenerarRutinaServlet.java
2 // 1. El servlet no sabe qué generador concreto es.
3 GeneradorRutinaBase generador = GeneradorRutinaFactory.getGenerador(
4     ↪ usuario.getDiasDisponibles());
5
6 // 2. Llama al método de la clase base. LSP garantiza que funcionará
7 //    con cualquier subclase (GeneradorRutina2Dias, 3Dias, etc.).
8 Rutina rutina = generador.generar(usuario, todosLosEjercicios);
9
10 // 3. El resto del código funciona igual sin importar la implementación.
11 request.setAttribute("rutina", rutina);
12 request.getRequestDispatcher("rutina.jsp").forward(request, response);
```

Listing 2: El Servlet depende de la abstracción, permitiendo la sustitución

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

3.1. Definición

Este principio establece que una clase debe tener **una, y solo una, razón para cambiar**.

3.2. Aplicación en MyGymJavaWeb

La arquitectura de servlets sigue este principio. Cada servlet tiene una única responsabilidad bien definida:

- `LoginServlet`: Su única razón para cambiar es si la lógica de autenticación se modifica.
- `RegistroServlet`: Solo se encarga de crear nuevos usuarios.
- `GuardarRutinaServlet`: Su única responsabilidad es persistir la rutina generada.

De igual forma, clases como `JsonUtil` tienen la única responsabilidad de gestionar la persistencia de datos, y `GeneradorRutinaFactory` solo se encarga de crear objetos.

```
1 @WebServlet("/login")
2 public class LoginServlet extends HttpServlet {
3     @Override
4     protected void doPost(HttpServletRequest request, HttpServletResponse
5         response)
6         throws ServletException, IOException {
7         // Lógica exclusiva de autenticación...
8     }
9 }
```

Listing 3: `LoginServlet.java`, un ejemplo claro de SRP

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

4.1. Definición

El DIP establece que los módulos de alto nivel no deben depender de los módulos de bajo nivel; ambos deben depender de abstracciones.

4.2. Aplicación en MyGymJavaWeb

El DIP se observa en dos puntos clave:

1. **Generación de Rutinas:** El `GenerarRutinaServlet` (módulo de alto nivel) no depende de `GeneradorRutina2Dias` (módulo de bajo nivel). Ambos dependen de la abstracción `GeneradorRutinaBase`.
2. **Persistencia:** Los servlets (alto nivel) no dependen de la implementación de lectura/escritura de archivos JSON. Dependen de la clase `JsonUtil`, que actúa como una abstracción sobre el mecanismo de persistencia. Si la persistencia cambiara a una base de datos, los servlets no necesitarían ser modificados.

```
1 // Dentro de un Servlet (módulo de alto nivel)
2 String realPath = getServletContext().getRealPath("/");
3
4 // No sabe que esto es un archivo JSON. Solo llama a la abstracción.
5 List<Usuario> usuarios = JsonUtil.leerUsuarios(realPath);
6
7 // De nuevo, no le importa el mecanismo de guardado.
8 JsonUtil.escribirUsuarios(usuarios, realPath);
```

Listing 4: Servlet dependiendo de la abstracción de persistencia `JsonUtil`

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

5.1. Definición

SoC aboga por dividir un programa en secciones distintas, donde cada sección aborda una preocupación separada. Es la base del patrón Modelo-Vista-Controlador (MVC).

5.2. Aplicación en MyGymJavaWeb

La aplicación sigue una arquitectura MVC, que es un ejemplo claro de SoC:

- **Modelo ('com.agym.modelo'):** Representa los datos (Usuario, Ejercicio). No sabe cómo se muestran ni manipulan.
- **Vista ('web/*.jsp'):** Se encarga de la presentación de los datos al usuario. No contiene lógica de negocio.
- **Controlador ('com.agym.controlador'):** Actúa como intermediario, recibe peticiones, interactúa con el Modelo y decide qué Vista mostrar.

```
1 // Al final del doPost() en GenerarRutinaServlet (Controlador)
2
3 // 1. Prepara los datos (Modelo) para la Vista
4 request.setAttribute("rutina", rutina);
5
6 // 2. Delega la responsabilidad de renderizar a la Vista (rutina.jsp)
7 request.getRequestDispatcher("rutina.jsp").forward(request, response);
```

Listing 5: El Controlador delega la presentación a la Vista

6. Principio KISS (Keep It Simple, Stupid)

6.1. Definición

KISS aboga por la simplicidad y sostiene que los sistemas funcionan mejor si se mantienen simples.

6.2. Aplicación en MyGymJavaWeb

El principio KISS es una guía en todo el proyecto:

- **Persistencia de Datos:** Se usaron archivos JSON en lugar de una base de datos, una solución simple y adecuada para la escala del proyecto.
- **Lógica Directa:** La lógica de negocio, como el ajuste de series por edad, es clara y fácil de entender.

```
1 // Lógica simple y directa dentro del Template Method
2 int edad = calcularEdad(usuario.getFechaNacimiento());
3 if (edad > 50) {
4     if (seriesYRepeticionesBase.startsWith("4")) {
5         seriesYRepeticionesBase = "3" + seriesYRepeticionesBase.substring
6             ↳ (1);
7     }
8 }
```

Listing 6: Lógica KISS para el ajuste por edad en GeneradorRutinaBase.java

7. Principio DRY (Don't Repeat Yourself)

7.1. Definición

DRY establece que cada pieza de conocimiento debe tener una representación única, evitando la duplicación de código.

7.2. Aplicación en MyGymJavaWeb

Se aplica en varios lugares:

- **JsonUtil:** Centraliza toda la lógica de lectura/escritura de JSON, evitando que cada servlet la implemente por su cuenta.
- **crearDiaRutina():** El método helper en `GeneradorRutinaBase` encapsula la lógica para crear un día de entrenamiento, evitando que `GeneradorRutina2Dias`, `3Dias`, etc., dupliquen este código.

```
1 // En GeneradorRutinaBase.java
2 protected Rutina.DiaRutina crearDiaRutina(String nombre, ..., String...
   ↳ grupos) {
3     // ... Lógica común para crear un día de rutina ...
4 }
5
6 // En GeneradorRutina2Dias.java (reutiliza el método)
7 @Override
8 protected void construirRutina(Rutina r, ...) {
9     r.agregarDia(crearDiaRutina("Día 1", ...));
10    r.agregarDia(crearDiaRutina("Día 2", ...));
11 }
```

Listing 7: Método helper para evitar duplicación en las subclases

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

8.1. Definición

YAGNI establece que no se deben añadir funcionalidades hasta que no sean estrictamente necesarias.

8.2. Aplicación en MyGymJavaWeb

Se aplicó al implementar el seguimiento de progreso. Se implementó la versión más simple (guardar y marcar como “Completada”) en lugar de un sistema complejo de registro de pesos o comentarios, que no era un requisito inicial.

```
1 public class RutinaGuardada {
2     private long id;
3     private int usuarioId;
4     private Date fechaGuardada;
5     private String estado; // Solo "Guardada" o "Completada"
6     private Rutina rutina;
7
8     // No se añadieron campos innecesarios como:
9     // private String comentariosUsuario;
10    // private Map<Integer, String> pesosLevantados;
11 }
```

Listing 8: Modelo de datos simple en `RutinaGuardada.java` siguiendo YAGNI