TALLER 1

“BIBLIOTECA-DIGITAL”

Presentado por:

Manuel Cabrera Silva

Luis Alejandro Fuentes Corzo

Miguel José Mendoza Samper

Romario Aldair Martínez Fuentes

UNIVERSIDAD POPULAR DEL CESAR

ESPECIALIZACIÓN DE INGENIERÍA DE SOFTAWARE

VALLEDUPAR/CESAR

2025

Tabla de contenido

Sistema de Biblioteca Digital

1. Introducción

Hoy en día desarrollar software no se trata solo de que funcione, sino de hacerlo bien, con una estructura que permita crecer, cambiar y mantenerse en el tiempo sin volverse un caos. Eso fue justo lo que nos propusimos con este proyecto. Teníamos un sistema de Biblioteca Digital hecho en Java con Spring Boot que cumplía con su función, pero que internamente tenía muchos problemas. Era difícil de mantener, complicado de entender y aún más difícil de escalar o modificar sin romper otras partes.

Lo que encontramos fue un código lleno de responsabilidades mezcladas. Controladores que hacían validaciones, entidades que enviaban correos o calculaban multas, servicios que manejaban de todo sin un orden claro. Todo eso iba en contra de las buenas prácticas y hacía que cualquier cambio fuera un dolor de cabeza. Por eso decidimos aplicar dos enfoques muy potentes: los principios SOLID y los patrones GRASP.

Más que un cambio técnico, esto fue un cambio de mentalidad. Empezamos a separar responsabilidades, a dividir el sistema por capas, a usar interfaces para depender menos de clases concretas y a darle a cada parte del código un propósito claro. Los controladores se enfocaron solo en recibir solicitudes, los servicios se encargaron de la lógica de negocio, las entidades dejaron de tener tareas que no les correspondían. También agregamos DTOS para controlar mejor los datos que se reciben y se envían.

Con todo este proceso, el sistema no solo se volvió más ordenado, sino más flexible, más fácil de entender y mucho más preparado para el futuro. Ya no da miedo tocar el código, porque sabemos que está hecho con una estructura pensada para durar. Este trabajo es prueba de que aplicar

buenas prácticas no es solo teoría, sino algo que mejora de verdad el día a día de un equipo de desarrollo.

2. Análisis de la Arquitectura Original

Principales Deficiencias Detectadas

Acoplamiento excesivo entre componentes: los controladores realizaban validaciones y conversiones de objetos, en lugar de delegar estas tareas a servicios o capas intermedias.

Entidades del dominio con lógica embebida, como métodos de cálculo de multas, restricciones de préstamo o envío de notificaciones.

Clases con múltiples responsabilidades como BibliotecaService, que combinaba operaciones CRUD, validaciones, envío de correos, y generación de reportes.

Repositorios con lógica de negocio, contradiciendo el principio de responsabilidad única (SRP) y dificultando su reutilización.

Ausencia de separación de preocupaciones entre entidades, controladores, y lógica de aplicación.

Ejemplos de Violaciones a SOLID

Principio	Violación Detectada
SRP	LibroController genera reportes y valida lógica de negocio
OCP	Libro.calcularCostoRetraso() con condicionales rígidas por tipo
OCP	Libro.calcularCostoRetraso() con condicionales rígidas por tipo
ISP	Libro.calcularCostoRetraso() con condicionales rígidas por tipo

DIP	Instanciación directa de NotificacionService en entidades y servicios

3. Principios Aplicados en el Rediseño

3.0. Aplicación de SOLID

SRP (Responsabilidad Única): Se separaron funcionalidades por capas: controladores manejan peticiones HTTP, servicios se encargan de la lógica de negocio, y DTOs gestionan el transporte de datos.

OCP (Abierto/Cerrado): Se rediseñaron comportamientos como el cálculo de costos por categoría mediante estrategias que faciliten la extensión sin modificar código existente.

LSP (Sustitución de Liskov): Se prepararon las entidades para una futura herencia válida, con métodos genéricos definidos mediante interfaces.

ISP (Segregación de Interfaces): Los repositorios se segmentaron para no obligar a las clases a implementar métodos innecesarios.

DIP (Inversión de Dependencias): Se eliminó la creación directa de servicios dentro de entidades, reemplazándola por inyección de dependencias mediante interfaces.

3.1. Aplicación de GRASP

Controller: Coordinadores claros para cada tipo de entidad (LibroController, UsuarioController, etc.)

Creator: Los servicios instancian objetos relacionados cuando son responsables directos de su gestión.

Low Coupling: Desacoplamiento entre controladores y lógica de negocio.

High Cohesion: Cada clase tiene una única función bien definida.

Polymorphism y Pure Fabrication: Se crearon servicios auxiliares para reportes y notificaciones, evitando responsabilidades múltiples en clases del dominio.

4. Mejoras Técnicas Implementadas

Elemento	Antes	Después
Controlador	Realizaba validaciones y generaba reports	Solo gestiona solicitudes y respuestas HTTP
Entidad Libro	Calculaba costos, validaba restricciones, enviaba correos	Solo representa datos del libro

Servicio	Mezclaba lógica de libros, usuarios y notificaciones	Dividido en múltiples servicios según su propósito
Notificaciones	Enviadas desde cualquier parte del código	Centralizadas en NotificacionService inyectado
Repositorio	Incluía consultas complejas y cálculos	Limitado a funciones de acceso a datos

5. Posibles Riesgos y Recomendaciones Futuras

5.0. Riesgos Persistentes

- Algunas validaciones específicas aún se encuentran embebidas en servicios, lo cual podría migrarse a capas de validación.
- El sistema podría beneficiarse del uso de patrones adicionales como Factory, Strategy o Decorator para casos de extensión más complejos.
- Aún se utilizan métodos if/else para decisiones lógicas que podrían refactorizarse usando principios orientados a objetos más sólidos.

5.1. Recomendaciones

- Implementar pruebas unitarias por cada capa de responsabilidad.
- Usar herramientas de análisis estático para verificar cumplimiento de SOLID.
- Documentar cada módulo y su responsabilidad según GRASP.
- Aplicar principios de DDD (Domain-Driven Design) para enriquecer el modelo de dominio.

6. Conclusión

La refactorización del sistema de Biblioteca Digital representa un caso claro de evolución arquitectónica positiva mediante la aplicación disciplinada de principios de diseño. Al adoptar SOLID y GRASP, el sistema ha pasado de ser una estructura rígida y propensa a errores, a convertirse en una base flexible, clara y preparada para el cambio.

Este proceso no solo mejoró la calidad del código, sino que también sentó las bases para su escalabilidad, reutilización y sostenibilidad a largo plazo. Se recomienda continuar con esta línea de mejora continua, promoviendo buenas prácticas de diseño en cada nueva funcionalidad.

7.Análisis Crítico Consolidado – Proyecto Biblioteca Digital

7.0. Contextualización del Proyecto

El presente análisis crítico consolidado expone el proceso de evolución arquitectónica del sistema de Biblioteca Digital, desarrollado en Java utilizando el framework Spring Boot. El sistema fue concebido para la gestión integral de libros, préstamos y usuarios, permitiendo el control automatizado de operaciones habituales en bibliotecas institucionales.

En su concepción inicial, el sistema fue construido de forma funcional pero sin una orientación clara a principios de diseño. Esto conllevó a un crecimiento desordenado del código, alta dependencia entre módulos y escasa claridad en la asignación de responsabilidades. Como

consecuencia, el mantenimiento y la incorporación de nuevas funcionalidades se volvieron tareas costosas y propensas a errores.

Este proyecto se propuso aplicar los principios SOLID y GRASP como ejes estructurales del rediseño, orientando cada componente del software a cumplir un rol específico y cooperativo dentro de una arquitectura orientada a objetos moderna, coherente y escalable.

8. Diagnóstico de la Arquitectura Inicial

El análisis de la versión original permitió identificar una serie de deficiencias graves tanto a nivel de estructura como de mantenimiento. Uno de los principales problemas era el uso de clases con múltiples responsabilidades, lo que violaba directamente el Principio de Responsabilidad Única (SRP).

Los controladores, por ejemplo, manejaban lógica de negocio, validaciones y respuestas HTTP, funciones que debieron estar separadas. Las entidades del dominio como `Libro` contenían métodos que iban más allá de la representación de datos, incluyendo lógica para notificaciones, restricciones de préstamo e incluso generación de reportes. Esto generaba una arquitectura rígida, difícil de escalar y con alta probabilidad de generar errores al introducir cambios.

El servicio principal (`BibliotecaService`) centralizaba operaciones de distintos módulos, mezclando lógica de usuarios, libros y préstamos en un solo componente. Este diseño iba en contra de los principios de cohesión y desacoplamiento, pilares de una arquitectura sostenible.

9. Evaluación de Violaciones a Principios de Diseño

Durante el diagnóstico se evidenciaron diversas violaciones a los principios SOLID:

SRP (Responsabilidad Única): Clases como `LibroController` gestionaban validaciones, reportes y persistencia.

OCP (Abierto/Cerrado): Métodos como `calcularCostoRetraso()` estaban diseñados con múltiples condicionales que limitaban su extensión.

LSP (Sustitución de Liskov): Se detectaron métodos en entidades que podrían no comportarse correctamente si se generalizaban a través de herencia.

ISP (Segregación de Interfaces): Repositorios con métodos que solo usaban ciertos clientes, forzando dependencias innecesarias.

DIP (Inversión de Dependencias): Uso directo de clases concretas como `NotificacionService` dentro del modelo, violando el principio de inversión.

Asimismo, en el marco de los patrones GRASP, se omitieron principios clave como:

Controller: el controlador no actuaba solo como coordinador, sino como ejecutor directo de lógica.

Creator: entidades no generaban sus propias relaciones de forma coherente.

Low Coupling / High Cohesion: se evidenció una fuerte dependencia entre módulos con baja cohesión interna.

Polymorphism y Pure Fabrication: ausentes en el diseño original, dificultando la extensibilidad.

10. Estrategia de Rediseño Arquitectónico

La estrategia implementada se fundamentó en una reestructuración modular por capas, separando las responsabilidades de presentación, lógica de negocio, persistencia y utilidades.

Cada entidad del dominio se enfocó únicamente en representar datos. Las responsabilidades de procesamiento y lógica fueron delegadas a servicios especializados. Los controladores quedaron reducidos a intermediarios entre las peticiones externas y los servicios internos, respetando así los principios de cohesión y bajo acoplamiento.

El rediseño también incluyó la definición de DTOs (Data Transfer Objects), con el objetivo de evitar la exposición directa del modelo de dominio en los controladores, brindando una capa de seguridad y flexibilidad en el intercambio de datos.

Adicionalmente, se introdujo la inversión de dependencias a través del uso de interfaces e inyección de dependencias, lo que permite desacoplar el uso de clases concretas y facilitar la implementación de pruebas unitarias.

11.Impacto del Rediseño

El impacto del rediseño ha sido evidente a múltiples niveles del sistema. Desde el punto de vista técnico, se logró:

- Mejorar significativamente la legibilidad del código.
- Aumentar la reutilización de componentes, gracias a la separación lógica por dominio.

- Facilitar la implementación de pruebas automáticas para cada componente individual.
- Reducir el tiempo necesario para implementar nuevas funcionalidades.

Desde la perspectiva de mantenimiento, el sistema ahora permite aislar errores con mayor facilidad y realizar mejoras sin temor a efectos colaterales no deseados. La nueva arquitectura, alineada a los principios SOLID y GRASP, constituye una base sólida para futuros desarrollos.

12.Reflexión Profesional

Este proyecto ha permitido evidenciar la importancia de la arquitectura en el ciclo de vida del software. Un sistema que no respeta los principios de diseño puede funcionar inicialmente, pero terminará colapsando a medida que se incorporan cambios, generando retrabajo, deuda técnica y pérdida de eficiencia.

Aplicar principios como SOLID y GRASP no es una práctica teórica, sino una necesidad en entornos profesionales. Estos principios no solo mejoran el código, sino que mejoran al equipo de desarrollo: obligan a pensar, a planificar, a diseñar antes de programar.

Este rediseño también muestra que nunca es tarde para mejorar. Aun cuando el código inicial presentaba errores estructurales serios, fue posible modularizarlo y transformarlo en una base sólida, clara y profesional.

13.Recomendaciones Futuras

Para fortalecer aún más la calidad del proyecto, se recomienda:

- Integrar herramientas como SonarQube para análisis de código estático.
- Adoptar TDD (Desarrollo guiado por pruebas) para nuevas funcionalidades.
- Ampliar la documentación técnica con diagramas UML.
- Introducir patrones como Strategy y Factory donde existan múltiples
- comportamientos posibles.
- Implementar CI/CD para automatizar pruebas y despliegues.
- Ampliar el sistema con autenticación, roles y permisos escalables.

El éxito del rediseño actual sienta las bases para convertir este proyecto en una plataforma robusta, adaptable y con proyección a largo plazo.

14. Conclusión

El rediseño del sistema de Biblioteca Digital representa una mejora sustancial tanto desde la óptica técnica como metodológica. Gracias a la aplicación disciplinada de los principios SOLID y los patrones GRASP, se logró consolidar un sistema moderno, eficiente y preparado para evolucionar.

Este proyecto constituye un claro ejemplo de cómo el diseño consciente puede marcar la diferencia entre un código funcional y uno verdaderamente profesional. El camino recorrido demuestra que las buenas prácticas no solo se aprenden, sino que se aplican y perfeccionan con cada línea de código.

15.Propuesta de Rediseño Detallado – Sistema Biblioteca Digital

16. Introducción

La presente propuesta de rediseño tiene como objetivo abordar las principales falencias encontradas en la arquitectura del sistema de Biblioteca Digital, desarrollado inicialmente en Java con Spring Boot. A lo largo de su uso, se identificaron deficiencias en la organización del código, responsabilidades mal asignadas y un bajo grado de cohesión, lo cual derivó en dificultades para la incorporación de nuevas funcionalidades, mantenimiento general y prueba de componentes aislados.

El rediseño se apoya en principios de ingeniería de software consolidados: los principios SOLID, que promueven una estructura modular y sostenible, y los patrones GRASP, que definen estrategias óptimas para la asignación de responsabilidades entre clases. Aplicar estos principios contribuirá no solo a mejorar la calidad técnica del código, sino también a facilitar el trabajo colaborativo en equipos de desarrollo, mejorar la comprensión del sistema y garantizar su adaptabilidad a largo plazo.

17. Objetivos del Rediseño

Los objetivos fundamentales que guían esta propuesta son:

- Eliminar el acoplamiento entre componentes para favorecer la evolución del sistema.
- Mejorar la separación de responsabilidades mediante la reorganización por capas.
- Facilitar la implementación de pruebas unitarias gracias a una estructura modular.
- Garantizar la comprensión y mantenibilidad del código para nuevos desarrolladores.

- Establecer una base robusta que permita incorporar características futuras de forma segura.

En consecuencia, esta propuesta no busca únicamente reparar errores, sino transformar radicalmente la arquitectura para adecuarla a las buenas prácticas de desarrollo de software moderno.

18. Diagnóstico del Sistema Actual

La versión actual del sistema presenta una serie de deficiencias estructurales que comprometen su calidad. En primer lugar, los controladores se encargan de realizar múltiples tareas, como validaciones de datos, instanciación de entidades, y en algunos casos, generación de reportes. Este comportamiento rompe el Principio de Responsabilidad Única (SRP).

Asimismo, las entidades del modelo de dominio, como `Libro`, contienen métodos que ejecutan lógica de negocio y hasta interactúan directamente con servicios como `NotificacionService`, violando la separación entre modelo y lógica de aplicación. Este tipo de diseño impide reutilizar las entidades y hace más difícil testearlas de forma aislada.

Por otro lado, la lógica del sistema está centralizada en un único servicio `BibliotecaService`, que gestiona operaciones relacionadas con libros, usuarios, préstamos y reportes, lo cual reduce la cohesión interna de cada módulo y crea una dependencia innecesaria entre dominios diferentes. El repositorio, lejos de limitarse a acceder a datos, también contiene consultas especializadas que deberían estar en la capa de servicio.

19. Arquitectura Propuesta

El nuevo diseño propuesto se fundamenta en una arquitectura multicapa, estructurada de la siguiente manera:

Capa de Presentación: compuesta por controladores REST encargados exclusivamente de recibir peticiones HTTP, extraer parámetros y delegar a los servicios.
Capa de Aplicación: constituida por servicios especializados que contienen la lógica de negocio agrupada por dominio (por ejemplo: `LibroService`, `PrestamoService`).
Capa de Persistencia: implementada mediante interfaces tipo `JpaRepository`, centrada exclusivamente en consultas y persistencia de datos.
Capa de DTOs: define estructuras de datos específicas para entrada y salida, promoviendo la seguridad y claridad del intercambio de datos.
Capa de Utilidades: incluye validadores, formateadores, generadores de reportes y servicios auxiliares que dan soporte a otras capas.
Interfaces y Contratos: utilizados en todos los puntos de comunicación entre componentes, facilitando el desacoplamiento y permitiendo cambios sin afectar al resto del sistema.

Este enfoque mejora la separación de preocupaciones, facilita el mantenimiento y soporta la escalabilidad del sistema.

20. Cambios Propuestos por Módulo

20.1 Controladores

Se eliminará toda lógica de negocio de los controladores. Su única función será manejar

solicitudes, extraer datos del cuerpo o los parámetros de la URL y enviarlos al servicio correspondiente. No realizarán validaciones ni conversiones.

20.2 Servicios

Los servicios estarán organizados por dominio funcional. Cada uno tendrá una responsabilidad clara y manejará una única entidad del modelo. Esto permitirá dividir los equipos de trabajo y aplicar pruebas unitarias más efectivas.

20.3 Entidades

Las clases del modelo dejarán de tener lógica. Serán simples POJOs con anotaciones JPA. Esto facilitará su reutilización en otros contextos, como exportación, migraciones o servicios externos.

20.4 DTOs

La creación de DTOs permitirá separar el modelo interno del sistema de los datos que se exponen a través de la API. También facilitará validaciones con anotaciones como `@NotNull`, `@Size`, etc., y garantizará un control detallado sobre la serialización.

20.5 Repositorios

Los repositorios se centrarán en el acceso a datos. Métodos como búsquedas personalizadas o estadísticas se implementarán usando `@Query` o `Specifications`, manteniendo la lógica en los servicios.

21. Aplicación de Principios SOLID

Se aplicarán los siguientes principios SOLID:

SRP: Cada clase se enfocará en una sola tarea específica.

OCP: Se podrán extender comportamientos como filtros o reglas de negocio sin tocar el código existente.

LSP: Todas las clases derivadas respetarán el contrato de sus superclases sin romper la lógica.

ISP: Se crearán interfaces pequeñas y enfocadas, de modo que ninguna clase dependa de métodos que no necesita.

DIP: Todas las dependencias se abstraerán mediante interfaces y se inyectarán desde el exterior.

22. Aplicación de Patrones GRASP

Para distribuir correctamente las responsabilidades, se aplicarán los siguientes patrones GRASP:

Creator: las clases que contienen datos importantes serán responsables de crear instancias relacionadas.

Controller: los controladores actuarán como coordinadores, no como ejecutores.

Low Coupling: se reducirá la dependencia entre componentes, facilitando los cambios y pruebas.

High Cohesion: cada clase cumplirá un rol específico y bien definido.

Polymorphism: se utilizarán interfaces y herencia para desacoplar y extender comportamientos

según sea necesario.

23. Beneficios Esperados

La implementación de esta propuesta traerá beneficios clave como:

- Mejora de la organización y claridad del código fuente.
- Mayor facilidad para realizar pruebas unitarias e integración.
- Reducción significativa del acoplamiento entre componentes.
- Mayor adaptabilidad a nuevos requerimientos funcionales o tecnológicos.
- Estabilidad y robustez en entornos productivos.
- Facilitación del trabajo colaborativo y mantenimiento por parte de otros desarrolladores.

24. Proyecciones Futuras

Una vez consolidada esta nueva arquitectura, será posible extender el sistema sin grandes esfuerzos. Se podrán añadir funcionalidades como:

- Sistema de roles y permisos personalizados.
- Notificaciones multicanal (correo, SMS, app móvil).
- Integración con APIs de bibliotecas públicas y catálogos ISBN.
- Panel administrativo con métricas y reportes en tiempo real.
- Despliegue continuo y monitoreo automatizado.

Este rediseño sentará la base para convertir el proyecto en una plataforma robusta y escalable.

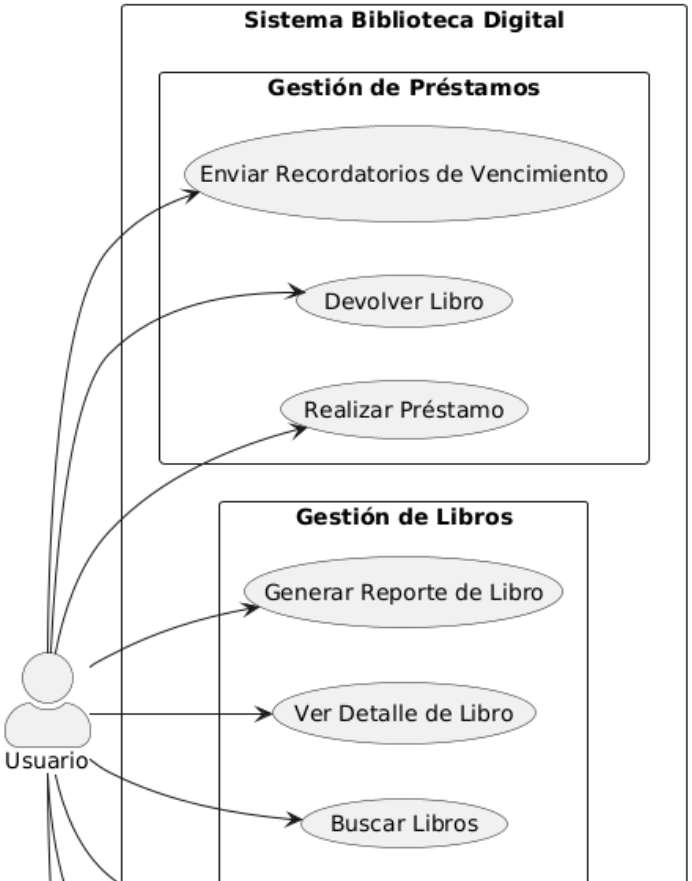
25. Conclusión

El rediseño propuesto no es solo una mejora técnica, sino un cambio en la filosofía de desarrollo del sistema. Aplicar SOLID y GRASP implica adoptar un enfoque profesional,

escalable y centrado en la calidad. Esta propuesta busca transformar el sistema de Biblioteca Digital en una solución moderna, mantenible y preparada para enfrentar los desafíos del entorno tecnológico actual y futuro.

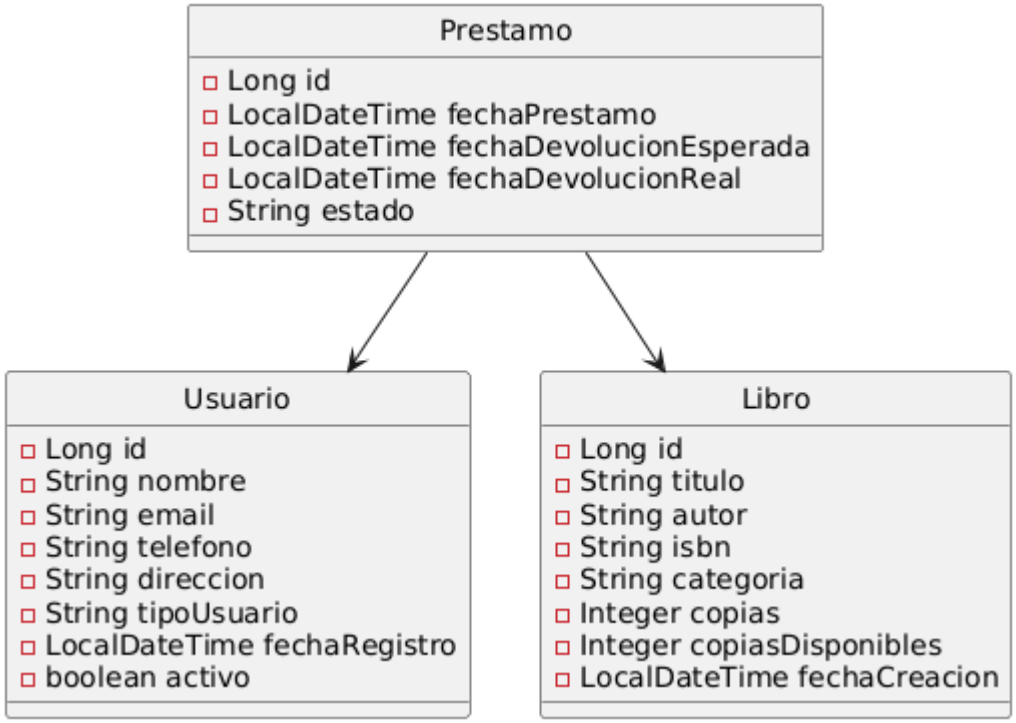
26. Análisis funcional

26.1 Diagrama de Casos de Uso Inicial (UML)

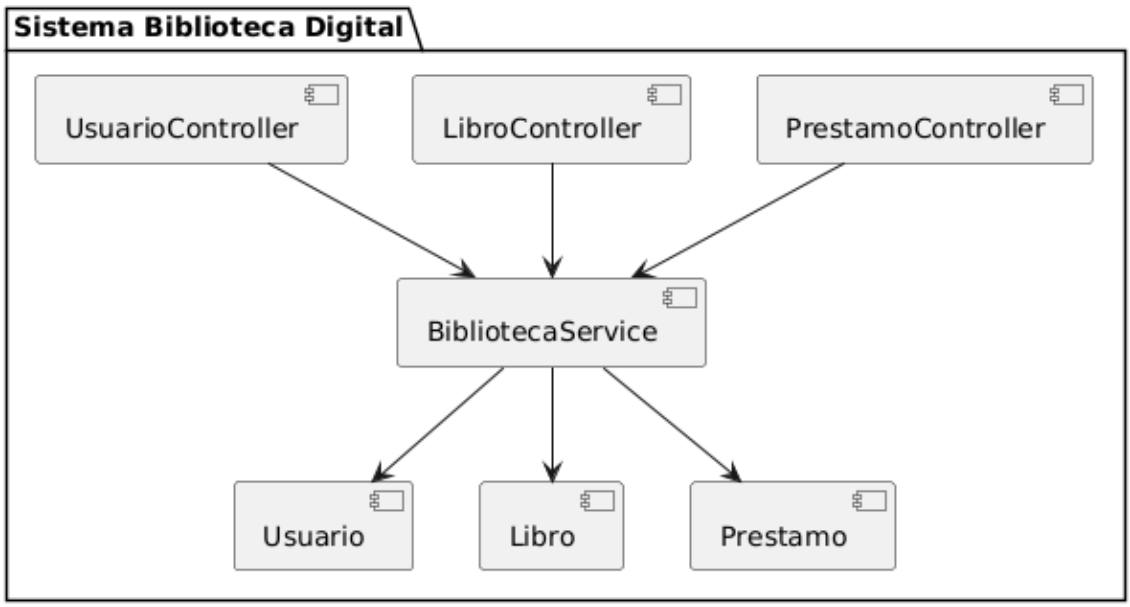


26.1.Análisis estructural

1.1. Diagrama de Clases del Estado Actual (Ingeniería Inversa)



26.1.2.Diagrama de Componentes de la Arquitectura Actual



27.Identificación de Problemas de Diseño

27.1.Identificar violaciones sistemáticamente

1.1.1. Informe de Problemas de Diseño - Sistema Biblioteca Digital.

- Violaciones al Principio de Responsabilidad Única (SRP)

Clase / Archivo	Línea Aprox.	Descripción del Problema	Consecuencias
BibliotecaService.java	**IMG 1** 13-100	Lógica de negocio de usuarios, libros y préstamos en una sola clase.	Dificulta pruebas, mantenimiento y escalabilidad.
UsuarioController.java	**IMG 2** 18-40	Validación de email y transformación manual de DTO.	La validación debería estar separada en otra capa.
LibroController.java	**IMG 3** 20-60	Validación y construcción de objetos `Libro`, además genera reportes.	Mezcla de responsabilidades y dificultad para testeo.

PrestamoController.java	**IMG 4** 20-70	Valida y maneja lógica y mensajes en el mismo lugar.	Rompe cohesión y complica pruebas unitarias.
LibroRepository.java	**IMG 5** 15–30	Lógica de negocio en consultas personalizadas (libros disponibles, especiales, etc.).	Debe delegarse a la capa de servicios.
UsuarioRepository.java	**IMG 6** 20–30	Consulta de usuarios con lógica compleja (préstamos vencidos, límite).	Reduce cohesión y aumenta dependencia.

IMG 1: BibliotecaService.java

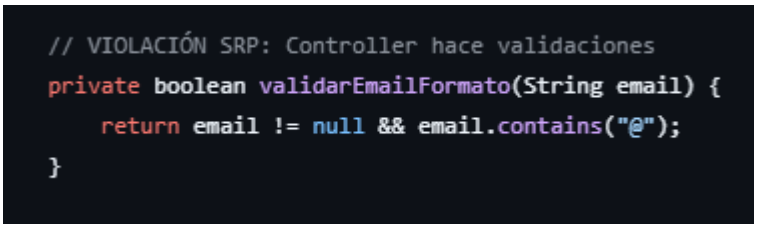
```java
// VIOLACIÓN SRP: Un servicio que hace TODO

// Gestión de libros
public Libro crearLibro(Libro libro) {
    libro.setFechaCreacion(LocalDateTime.now());
    libro.setCopiasDisponibles(libro.getCopias());
    return libroRepository.save(libro);
}

public Libro getLibro(Long id){
    Libro libro = this.libroRepository.getReferenceById(id);
    return libro;
}

public List<Libro> buscarLibros(String criterio) {
    // VIOLACIÓN OCP: Lógica de búsqueda hardcodeada
    if (criterio.startsWith("ISBN:")) {
        return libroRepository.findByIsbn(criterio.substring(5));
    } else if (criterio.startsWith("AUTOR:")) {
        return libroRepository.findByAutorContaining(criterio.substring(6));
    } else {
        return libroRepository.findByTituloContaining(criterio);
    }
}
```

IMG 2: UsuarioController.java



```java
// VIOLACIÓN SRP: Controller hace validaciones
private boolean validarEmailFormato(String email) {
    return email != null && email.contains("@");
}
```

IMG 3: LibroController.java

```java
        // VIOLACIÓN: Controller hace validación de negocio
        @PostMapping
        public ResponseEntity<Libro> crearLibro(@RequestBody LibroDTO libroDTO) {
            // Validación en controller (VIOLACIÓN SRP)
            if (libroDTO.getTitulo() == null || libroDTO.getTitulo().trim().isEmpty()) {
                logger.warning("Intento de crear libro sin título");
                return ResponseEntity.badRequest().build();
            }

            if (libroDTO.getCopias() == null || libroDTO.getCopias() <= 0) {
                logger.warning("Intento de crear libro con copias inválidas");
                return ResponseEntity.badRequest().build();
            }

            // VIOLACIÓN DIP: Conversión manual DTO -> Entity
            Libro libro = new Libro();
            libro.setTitulo(libroDTO.getTitulo());
            libro.setAutor(libroDTO.getAutor());
            libro.setIsbn(libroDTO.getIsbn());
            libro.setCategoria(libroDTO.getCategoria());
            libro.setCopias(libroDTO.getCopias());

            Libro libroCreado = bibliotecaService.crearLibro(libro);
            return ResponseEntity.ok(libroCreado);
        }
```

IMG 4: PrestamoController.java

```java
        // VIOLACIÓN: Controller maneja lógica de negocio
        if (usuarioId == null || libroId == null) {
            logger.warning("Parámetros inválidos para préstamo");
            return ResponseEntity.badRequest().build();
        }

        try {
            Prestamo prestamo = bibliotecaService.realizarPrestamo(usuarioId, libroId);
            logger.info("Préstamo realizado exitosamente: " + prestamo.getId());
            return ResponseEntity.ok(prestamo);
        } catch (RuntimeException e) {
            logger.severe("Error al realizar préstamo: " + e.getMessage());
            return ResponseEntity.badRequest().build();
        }
    }

    @PutMapping("/{id}/devolver")
    public ResponseEntity<Void> devolverLibro(@PathVariable Long id) {
        try {
            bibliotecaService.devolverLibro(id);
            return ResponseEntity.ok().build();
        } catch (RuntimeException e) {
            logger.severe("Error al devolver libro: " + e.getMessage());
            return ResponseEntity.badRequest().build();
        }
    }
```

IMG 5: LibroRepository.java

```
    // VIOLACIÓN ISP: Métodos específicos que no todos los clientes necesitan
    @Query("SELECT l FROM Libro l WHERE l.copiasDisponibles > 0")
    List<Libro> findLibrosDisponibles();

    @Query("SELECT l FROM Libro l WHERE l.categoria = 'REFERENCIA'")
    List<Libro> findLibrosReferencia();

    @Query("SELECT l FROM Libro l WHERE l.categoria = 'ESPECIAL'")
    List<Libro> findLibrosEspeciales();

    // Métodos de reporte (VIOLACIÓN SRP del repositorio)
    @Query("SELECT COUNT(l) FROM Libro l WHERE l.categoria = ?1")
    Long contarLibrosPorCategoria(String categoria);

    @Query("SELECT l.categoria, COUNT(l) FROM Libro l GROUP BY l.categoria")
    List<Object[]> obtenerEstadisticasPorCategoria();

    // Métodos de negocio (VIOLACIÓN: repositorio con lógica de negocio)
    @Query("SELECT l FROM Libro l WHERE l.copiasDisponibles < 2")
    List<Libro> findLibrosConPocasCopiasDisponibles();
}
```

IMG 6: UsuarioRepository.java

```
    // VIOLACIÓN: Consultas complejas de negocio en repositorio
    //@Query("SELECT u FROM Usuario u WHERE SIZE(u.prestamos) >= u.limitePrestamos")
    @Query("SELECT u FROM Usuario u WHERE SIZE(u.prestamos) >= 5")
    List<Usuario> findUsuariosConLimiteAlcanzado();

    @Query("SELECT u FROM Usuario u JOIN u.prestamos p WHERE p.estado = 'VENCIDO'")
    List<Usuario> findUsuariosConPrestamosVencidos();
}
```

27.Violaciones al Principio de Inversión de Dependencias (DIP)

Clase / Archivo	Línea Aprox.	Descripción del Problema	Consecuencias
UsuarioController.java	**IMG 7** 28-36	Conversión manual de DTO a entidad.	Debería delegarse a una clase Mapper o usar MapStruct.
LibroController.java	**IMG 8** 31-43	Construcción manual de `Libro` en el controller.	Alta dependencia con el modelo, rompe DIP.

IMG 7: UsuarioController.java: Conversión manual de DTO a entidad.

```
44
45          // VIOLACIÓN DIP: Conversión manual sin abstracción
46          private Usuario convertirDTOaEntity(UsuarioDTO dto) {
47              Usuario usuario = new Usuario();
48              usuario.setNombre(dto.getNombre());
49              usuario.setEmail(dto.getEmail());
50              usuario.setTelefono(dto.getTelefono());
51              usuario.setDireccion(dto.getDireccion());
52              usuario.setTipoUsuario(dto.getTipoUsuario());
53              return usuario;
54          }
55      }
```

IMG 8: LibroController.java: Construcción manual de `Libro`

```java
// VIOLACIÓN: Controller hace validación de negocio
@PostMapping
public ResponseEntity<Libro> crearLibro(@RequestBody LibroDTO libroDTO) {
    // Validación en controller (VIOLACIÓN SRP)
    if (libroDTO.getTitulo() == null || libroDTO.getTitulo().trim().isEmpty()) {
        logger.warning("Intento de crear libro sin título");
        return ResponseEntity.badRequest().build();
    }

    if (libroDTO.getCopias() == null || libroDTO.getCopias() <= 0) {
        logger.warning("Intento de crear libro con copias inválidas");
        return ResponseEntity.badRequest().build();
    }

    // VIOLACIÓN DIP: Conversión manual DTO -> Entity
    Libro libro = new Libro();
    libro.setTitulo(libroDTO.getTitulo());
    libro.setAutor(libroDTO.getAutor());
    libro.setIsbn(libroDTO.getIsbn());
    libro.setCategoria(libroDTO.getCategoria());
    libro.setCopias(libroDTO.getCopias());

    Libro libroCreado = bibliotecaService.crearLibro(libro);
    return ResponseEntity.ok(libroCreado);
}
```

28.Violaciones al Principio Abierto/Cerrado (OCP)

Clase / Archivo	Línea Aprox.	Descripción del Problema	Consecuencias
BibliotecaService.java	Global	No permite extender funcionalidades sin modificar la clase.	Afecta estabilidad con cada cambio.
PrestamoController.java	Global	Lógica de recordatorios está hardcoded.	No se puede extender modularmente.

29.Violaciones al Principio de Segregación de Interfaces (ISP)

Clase / Archivo	Línea Aprox.	Descripción del Problema	Consecuencias
LibroRepository.java	**IMG 9 10–30**	Demasiados métodos específicos sobre una misma interfaz genérica.	Los clientes cargan métodos que no necesitan.

IMG 9: LibroRepository.java: Demasiados métodos específicos sobre una misma interfaz genérica.

```java
@Repository
public interface LibroRepository extends JpaRepository<Libro, Long> {

    // Métodos básicos de búsqueda
    List<Libro> findByTituloContaining(String titulo);
    List<Libro> findByAutorContaining(String autor);
    List<Libro> findByIsbn(String isbn);
    List<Libro> findByCategoria(String categoria);
```

30.Violaciones a Principios GRASP

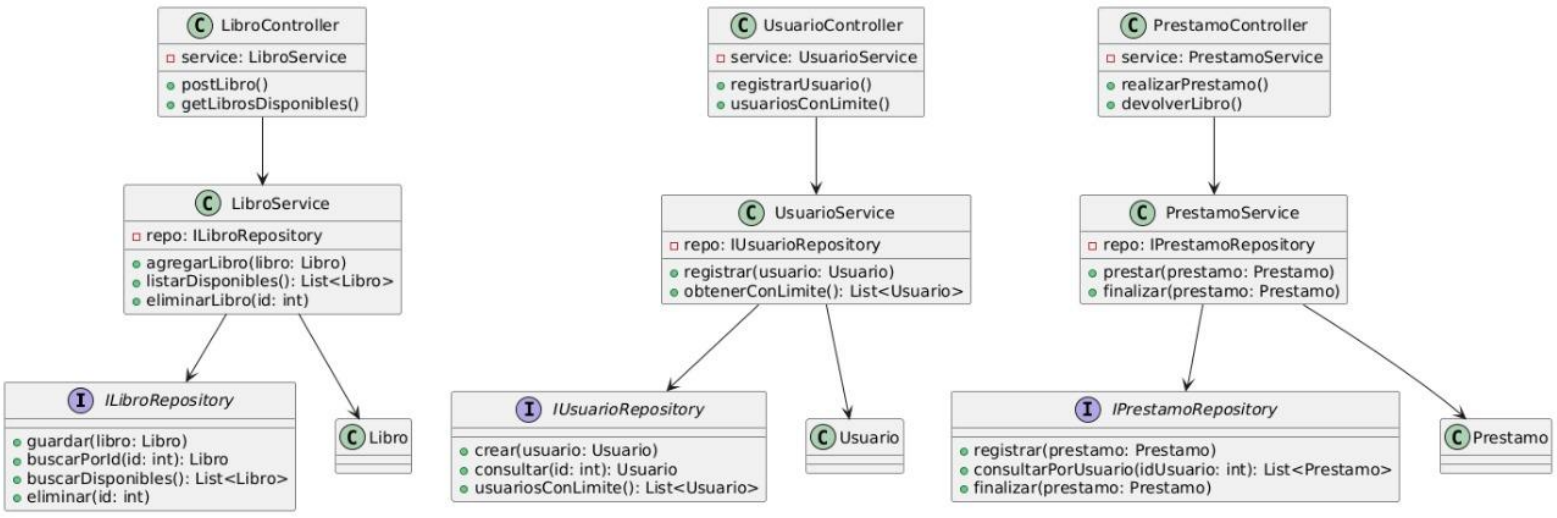
- Alta cohesión: BibliotecaService maneja demasiadas responsabilidades de distintos dominios.

- Bajo acoplamiento: Controladores dependen directamente de entidades de dominio (Libro, Usuario).
- Experto en información: Controladores ejecutan lógica que debería estar en los servicios expertos.
- LibroRepository.java: Realiza cálculos, estadísticas y filtrados que deberían ir en el servicio.
- UsuarioRepository.java: Contiene lógica como `findUsuariosConLimiteAlcanzado`, que pertenece al servicio.

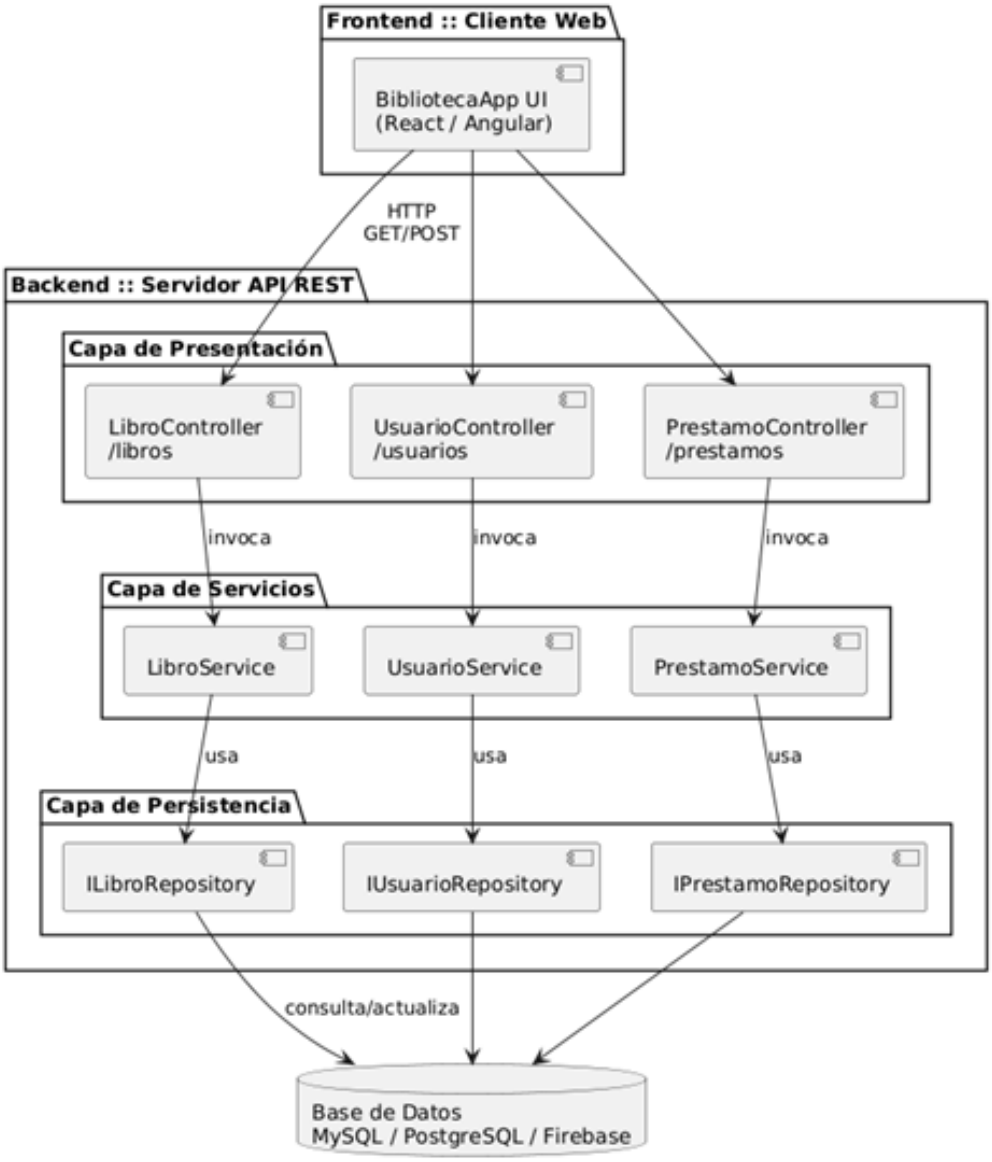
31. Recomendaciones Generales

- Dividir `BibliotecaService` en `LibroService`, `UsuarioService` y `PrestamoService`.
- Usar librerías como MapStruct para convertir DTOs a entidades.
- Crear validadores dedicados para entrada de datos.
- Aplicar roles y control de acceso a endpoints.

32. Diagrama de clases aplicando principios.



32.2 Diagrama de componentes y dependencias



33. Análisis y Documentación de Beneficios Esperados del Rediseño

Contexto del Rediseño

El rediseño del sistema de gestión de biblioteca digital se realizó con base en un análisis exhaustivo de la versión inicial, donde se identificaron múltiples violaciones a principios SOLID y GRASP. Se propuso una nueva arquitectura modular, orientada a capas, basada en la separación de responsabilidades y el bajo acoplamiento entre componentes.

Principios Aplicados

- SRP (Responsabilidad Única): cada clase y servicio se enfoca en una única responsabilidad (ej. UsuarioService, LibroService).
- OCP (Abierto/Cerrado): la arquitectura permite agregar nuevas funcionalidades sin modificar el código existente.
- DIP (Inversión de Dependencias): los servicios dependen de interfaces, no de clases concretas, facilitando pruebas y mantenimiento.
- ISP (Segregación de Interfaces): se evita que las interfaces incluyan métodos innecesarios para sus implementaciones.
- GRASP: se aplicaron Expert, Bajo Acoplamiento y Alta Cohesión para distribuir las responsabilidades correctamente.

Beneficios Esperados

- Mejora en la mantenibilidad: el sistema es más fácil de entender, modificar y extender.
- Reutilización de componentes: los servicios y DTOs pueden ser reutilizados en otras aplicaciones o interfaces.

- Mayor facilidad para pruebas unitarias y de integración.
- Separación clara entre presentación, lógica de aplicación, dominio y persistencia.
- Facilidad para aplicar políticas de seguridad y control de acceso en la capa de presentación.
- Simplificación del mantenimiento técnico al evitar clases con múltiples responsabilidades.
- Mejor preparación para escalar el sistema a múltiples módulos o microservicios en el futuro.
- Reducción del acoplamiento entre clases y mayor flexibilidad ante cambios de requerimientos.

Entregables Relacionados

- Diagrama de Clases UML rediseñado (PlantUML): Diagrama_Clases_SOLID.puml
- Diagrama de Componentes y dependencias (PlantUML): Diagrama_Componentes_Ampliado.puml
- Documento de análisis de problemas previos: Informe_Problemas_Diseno_Biblioteca_Actualizado.docx

34. Conclusión

Después de todo este proceso, queda claro que rediseñar el sistema de Biblioteca Digital fue mucho más que solo organizar el código. Fue aprender a trabajar de una manera distinta, con más orden, más claridad y sobre todo con una visión a futuro. Al principio teníamos un sistema que funcionaba, sí, pero era difícil de mantener, con partes mezcladas, responsabilidades mal repartidas y muchas cosas hechas sin seguir principios sólidos.

Aplicar SOLID y GRASP nos permitió ver el sistema con otros ojos. Nos ayudó a entender la importancia de dividir bien las tareas, de que cada clase haga lo que le toca, de evitar depender directamente de otras partes del sistema. Aprendimos a construir pensando en el futuro, en que cualquier cambio debería ser más fácil de hacer, más seguro y menos propenso a errores.

El resultado fue un sistema más limpio, más modular, más fácil de probar y de escalar. Pero más allá del código, lo que ganamos fue experiencia. Entendimos por qué vale la pena tomarse el tiempo de diseñar bien, por qué seguir buenas prácticas no es solo una exigencia académica, sino una herramienta real que mejora el trabajo diario de cualquier equipo de desarrollo.

Este proyecto nos demostró que siempre se puede mejorar, que nunca es tarde para arreglar lo que no está bien y que construir software de calidad no es solo saber programar, sino saber diseñar. Nos vamos con la satisfacción de haber hecho un cambio real, con la motivación de seguir aprendiendo y con una base sólida para seguir creciendo.