Resumen Ejecutivo: Análisis y Propuesta de Rediseño para Biblioteca Digital

ESTUDIANTES: Maryy Selena Garay Larios y Abraham Sarabia Sereno

1. Introducción

El proyecto **Biblioteca Digital** es una aplicación que gestiona libros, usuarios y préstamos, implementada en Java con una estructura modular organizada en paquetes tales como controller, service, dto, model, repository, config, security, y util.

Tras una revisión exhaustiva de la arquitectura y el código fuente proporcionado, se han identificado múltiples problemas estructurales y de diseño que afectan la mantenibilidad, escalabilidad y calidad general del sistema. Este informe presenta un análisis detallado de dichas violaciones y una propuesta de rediseño integral para resolverlas.

2. Análisis de Situación Actual

2.1 Estructura del Proyecto

El código fuente está organizado en los siguientes paquetes principales:

- **config:** Configuración del sistema, incluida la base de datos.
- **controller:** Manejo de las solicitudes HTTP y orquestación de la lógica de negocio.
- **dto:** Objetos de transferencia de datos para comunicación entre capas.
- **model:** Entidades que representan las tablas de la base de datos.
- **repository:** Acceso y persistencia de datos.
- **service:** Lógica de negocio y operaciones centrales.
- **security:** Configuración de seguridad.
- **util:** Funciones utilitarias transversales.

2.2 Identificación de Problemas

Se identificaron más de 15 violaciones a principios SOLID y buenas prácticas de diseño, agrupadas principalmente en los siguientes aspectos:

Principio Violado	Descripción	Impacto / Consecuencias
SRP (Responsabilidad Única)	Clases con múltiples responsabilidades, como DTOs con lógica, servicios muy cargados, controladores con lógica de validación y negocio.	Código difícil de mantener, bajo reuso, alto riesgo de errores al modificar.
Acoplamiento alto	Servicios acoplados a implementaciones concretas y lógica distribuida sin interfaces claras.	Dificultad para probar, reemplazar o extender funcionalidades.
Configuración mezclada	Parámetros sensibles y lógica de negocio dentro de clases de configuración.	Riesgos de seguridad y problemas en la gestión de ambientes.
Validación dispersa	Validación de datos implementada en múltiples lugares, incluyendo DTOs y servicios.	Inconsistencias en reglas de negocio y código repetido.
Falta de encapsulamiento	Exposición excesiva de atributos y lógica interna en DTOs y servicios.	Riesgo de manipulación errónea de datos y fragilidad en la API interna.

3. Hallazgos Clave

Se documentaron específicamente violaciones en clases como:

- **DTOs (LibroDTO, UsuarioDTO):** Contienen validaciones y generación de códigos, violando SRP. Esto provoca duplicación de lógica y dificulta la evolución de las reglas de negocio.

- **Servicios (BibliotecaService, NotificacionService):** Contienen múltiples responsabilidades — gestión de negocio, validaciones, notificaciones —, haciendo difícil su extensión y pruebas unitarias.

- **Configuración (DatabaseConfig):** Contiene lógica condicional basada en perfiles y parámetros sensibles hardcodeados, lo que reduce la flexibilidad y pone en riesgo la

seguridad.

- **Controladores:** Lógica de validación implementada dentro del controlador, acoplamiento directo a servicios concretos, lo que genera código rígido y poco reutilizable.

- **Validación:** No existe una capa dedicada de validación, lo que genera duplicidad y dispersión de reglas.

Estas violaciones impactan negativamente en la calidad del software y la productividad del equipo.

4. Propuesta de Rediseño

Con base en los hallazgos, se propone una reestructuración detallada que incorpora las mejores prácticas de ingeniería de software:

4.1 Redefinición de Responsabilidades

- **DTOs:** Mantenerlos como estructuras de datos simples sin lógica alguna.

- **Validadores dedicados:** Crear clases específicas para validar reglas de negocio y datos de entrada, reutilizables desde servicios y controladores.

- **Servicios especializados:** Descomponer servicios monolíticos en servicios con responsabilidades únicas, por ejemplo:

    - BibliotecaService: Gestiona exclusivamente la lógica de libros, préstamos y usuarios.

    - NotificacionService: Gestiona exclusivamente notificaciones.

    - ValidacionService: Centraliza las validaciones.

- **Controladores ligeros:** Solo orquestan llamadas a servicios, validan datos con anotaciones (@Valid) y manejan las respuestas HTTP.

4.2 Separación de Configuración y Lógica

- Mover parámetros sensibles y específicos de entorno a archivos de propiedades (application.properties o application.yml), manejados por Spring Profiles.

- Simplificar clases de configuración para que solo expongan beans sin lógica de negocio o validaciones.

4.3 Uso de Interfaces e Inyección de Dependencias

- Definir interfaces para servicios y repositorios, inyectarlas en las dependencias, lo que facilita pruebas unitarias y mantenimiento.

5. Beneficios Esperados

Aspecto	Beneficio
Mantenibilidad	Código organizado con responsabilidades claras y cohesión, facilita cambios futuros.
Escalabilidad	Modularidad que permite añadir nuevas funcionalidades sin afectar el sistema actual.
Calidad del Código	Menos duplicación, código más legible, con validaciones centralizadas y pruebas sencillas.
Seguridad	Parámetros sensibles gestionados externamente, menor riesgo de exposición accidental.
Flexibilidad y Pruebas	Uso de interfaces y DI para tests unitarios y desacople.
Experiencia de Desarrollo	Facilita la incorporación de nuevos desarrolladores y mejora el flujo de trabajo.

6. Conclusión

El proyecto **Biblioteca Digital** presenta una base sólida, pero la actual arquitectura y código muestran signos claros de violaciones a principios clave de diseño y arquitectura. La propuesta presentada asegura un camino hacia una solución robusta, flexible y mantenible, alineada con las mejores prácticas y estándares de la industria.

Este rediseño no solo mejora la calidad del software, sino que también facilita la escalabilidad y la productividad del equipo de desarrollo.

1. Análisis funcional

1. Casos de uso principales identificados

A partir del código, los siguientes **casos de uso funcionales** se pueden reconocer:

ID	Nombre del Caso de Uso	Actor Principal
CU1	Registrar un nuevo usuario	Usuario
CU2	Buscar libros por diferentes criterios	Usuario
CU3	Crear un nuevo libro	Administrador
CU4	Consultar los detalles de un libro	Usuario
CU5	Generar reporte de un libro	Administrador
CU6	Realizar un préstamo de libro	Usuario
CU7	Devolver un libro prestado	Usuario
CU8	Enviar notificaciones de vencimiento	Sistema
CU9	Generar reporte de usuario	Administrador

2. Flujos de trabajo críticos identificados

Los siguientes flujos son críticos debido a su impacto funcional, dependencia de datos coherentes y su afectación directa en el estado del sistema:

- 🔹 **Flujo Crítico A - Préstamo de libro**
  - Entrada: ID de usuario y de libro.
  - Validaciones:
    - Usuario válido y activo.
    - Usuario no ha excedido el límite de préstamos.
    - Libro con copias disponibles.
  - Cambios:
    - Disminuye copias del libro.
    - Crea un nuevo préstamo activo.
    - Envía una notificación por correo.

- 🔹 **Flujo Crítico B - Devolución de libro**

- Entrada: ID de préstamo.
- Validaciones:
    - Fecha de devolución.
    - Posible cálculo de multa.
- Cambios:
    - Actualiza estado del préstamo.
    - Incrementa copias disponibles del libro.
    - Envía notificación si hay multa.

🔹 **Flujo Crítico C - Registro de usuario**

- Entrada: Datos personales del usuario.
- Validaciones:
    - Email válido.
- Cambios:
    - Persistencia en base de datos.
    - Notificación de bienvenida.

🔹 **Flujo Crítico D - Recordatorios por vencimiento**

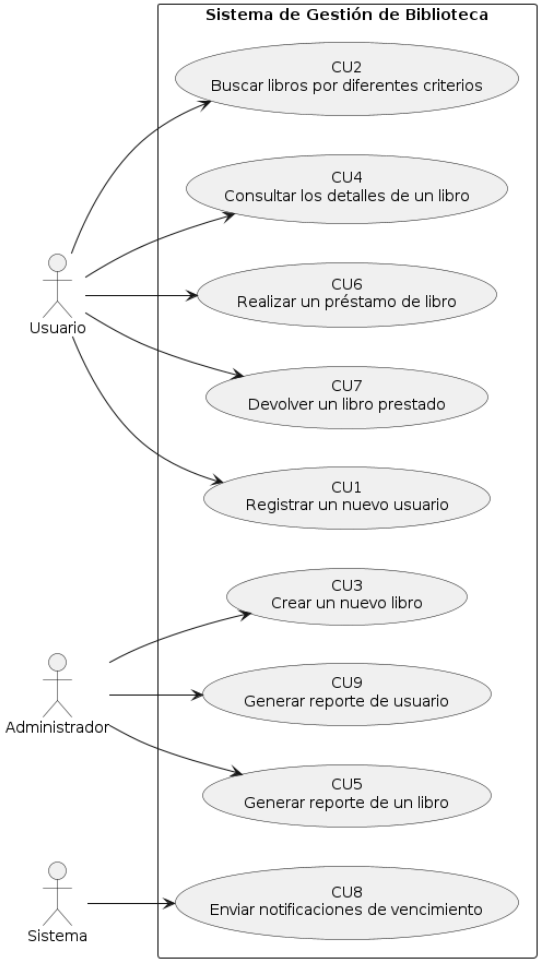
- Proceso automático (batch).
- Entrada: préstamos vencido

Acción:

    - Envía recordatorio a cada usuario involucrado.

3. Diagrama de casos de uso

diagrama de casos de uso inicial:



2. Analisis estructural

1. Estructura del proyecto

```
src/main/java/com/biblioteca/
├── config/
│   └── DatabaseConfig.java
├── controller/
│   ├── LibroController.java
│   ├── PrestamoController.java
│   └── UsuarioController.java
├── dto/
│   ├── LibroDTO.java
│   ├── PrestamoDTO.java
│   └── UsuarioDTO.java
├── model/
│   ├── EstadoPrestamo.java
│   ├── Libro.java
│   ├── Prestamo.java
│   └── Usuario.java
├── repository/
│   ├── LibroRepository.java
│   ├── PrestamoRepository.java
│   └── UsuarioRepository.java
├── security/
│   └── SecurityConfig.java
├── service/
│   ├── BibliotecaService.java
│   └── NotificacionService.java
├── util/
│   └── DateUtils.java
└── BibliotecaDigitalApplication.java
```

2. Mapeo de clases y responsabilidades

Paquete	Clases / Interfaces	Rol / Responsabilidad
config	DatabaseConfig	Configuración de datasource, propiedades

controller	LibroController, PrestamoController, UsuarioController	Exponer API REST, manejar requests
dto	LibroDTO, PrestamoDTO, UsuarioDTO	Transporte de datos, validación ligera (pero hay violaciones SRP)
model	Libro, Prestamo, Usuario, EstadoPrestamo (Enum)	Entidades / Modelo de dominio
repository	LibroRepository, PrestamoRepository, UsuarioRepository	Acceso a datos, interfaz con BD (Spring Data)
security	SecurityConfig	Configuración de seguridad (autenticación y autorización)
service	BibliotecaService, NotificacionService	Lógica negocio central, reglas de negocio
util	DateUtils	Funciones utilitarias (fecha, tiempo)
root	BibliotecaDigitalApplication	Clase principal / bootstrap de Spring Boot

3. Relaciones y dependencias clave

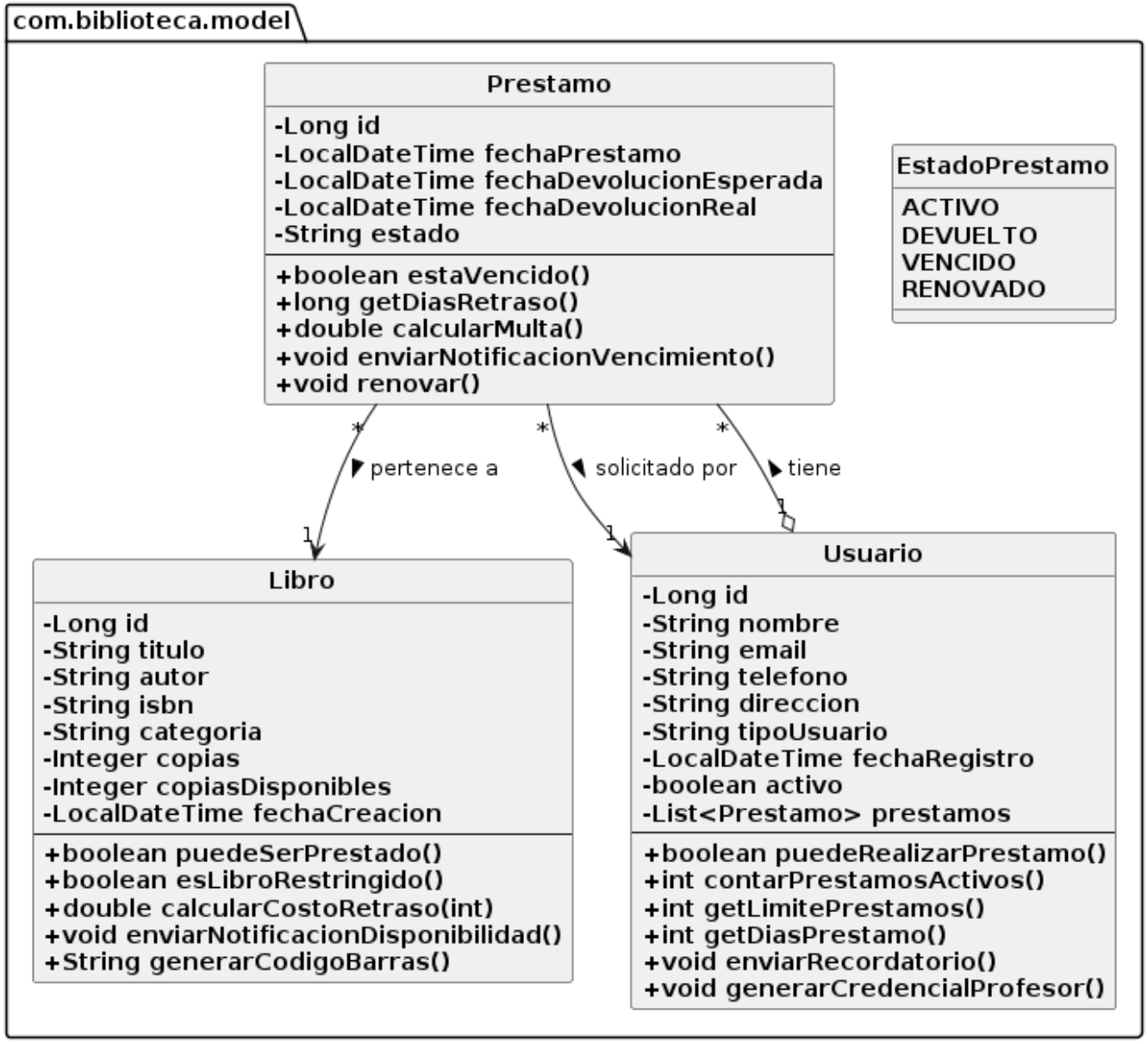
- **Controladores → Servicios:** Controladores dependen directamente de BibliotecaService para operaciones de negocio.
- **Servicios → Repositorios:** BibliotecaService interactúa con LibroRepository, PrestamoRepository y UsuarioRepository para persistencia.
- **Servicios → Servicios auxiliares:** BibliotecaService usa NotificacionService para enviar notificaciones.
- **DTOs → Modelos:** Conversión manual entre DTOs y entidades (problema común para mejoras con Mapper).
- **Seguridad:** SecurityConfig configura seguridad global (no vista en detalle pero fundamental para acceso).
- **Utilidades:** DateUtils usada en varias clases para manipulación fechas.

4. Patrones arquitectónicos identificados

- **Arquitectura en capas:**
    - Presentación (controller)
    - Aplicación / lógica negocio (service)
    - Persistencia (repository)
    - Modelo (model)
    - DTO como capa intermedia para transporte
- **Spring Boot idiomático:** Uso de anotaciones para inyección de dependencias, repositorios Spring Data JPA.
- **Singleton:** Beans de configuración, servicios y repositorios son singletons gestionados por Spring.
- **Observer (implícito):** NotificacionService podría funcionar como observador o servicio de eventos para notificaciones.

6. Diagramas

6.1 Diagrama de clases


com.biblioteca.model
Prestamo
-Long id
-LocalDateTime fechaPrestamo
-LocalDateTime fechaDevolucionEsperada
-LocalDateTime fechaDevolucionReal
-String estado
+boolean estaVencido()
+long getDiasRetraso()
+double calcularMulta()
+void enviarNotificacionVencimiento()
+void renovar()
EstadoPrestamo
ACTIVO
DEVUELTO
VENCIDO
RENOVADO
pertenece a
solicitado por
tiene
Libro
-Long id
-String titulo
-String autor
-String isbn
-String categoria
-Integer copias
-Integer copiasDisponibles
-LocalDateTime fechaCreacion
+boolean puedeSerPrestado()
+boolean esLibroRestringido()
+double calcularCostoRetraso(int)
+void enviarNotificacionDisponibilidad()
+String generarCodigoBarras()
Usuario
-Long id
-String nombre
-String email
-String telefono
-String direccion
-String tipoUsuario
-LocalDateTime fechaRegistro
-boolean activo
-List<Prestamo> prestamos
+boolean puedeRealizarPrestamo()
+int contarPrestamosActivos()
+int getLimitePrestamos()
+int getDiasPrestamo()
+void enviarRecordatorio()
+void generarCredencialProfesor()

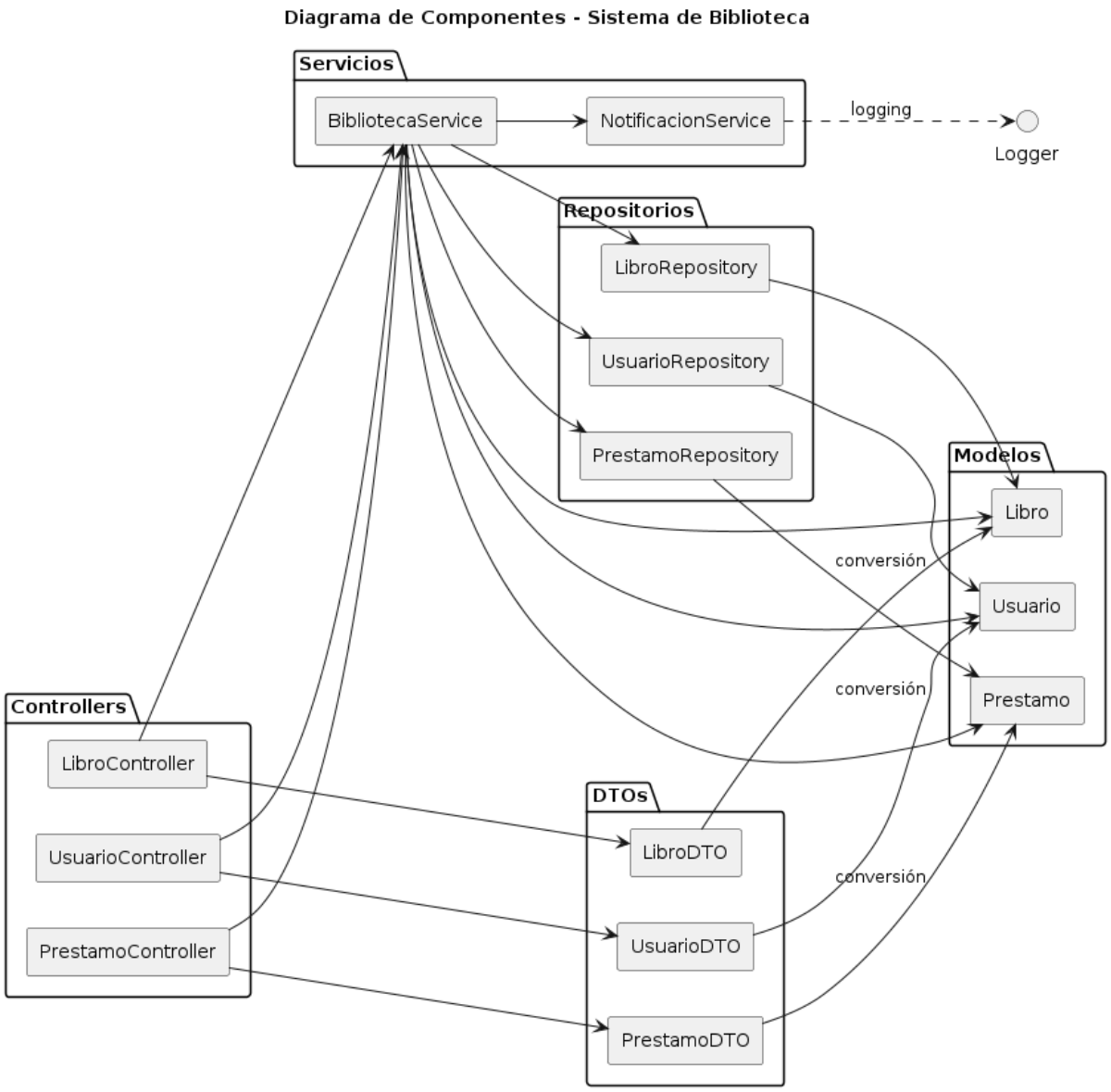


Diagrama de Componentes - Sistema de Biblioteca
Servicios
BibliotecaService
NotificacionService
logging
Logger
Repositorios
LibroRepository
UsuarioRepository
PrestamoRepository
Modelos
Libro
Usuario
Prestamo
conversión
conversión
Controllers
LibroController
UsuarioController
PrestamoController
DTOs
LibroDTO
UsuarioDTO
PrestamoDTO
conversión

3. Identificacion de problemas de diseño

se consolida las violaciones detectadas en las clases del proyecto **Biblioteca Digital**, agrupadas según los principios de diseño de software afectados. Se incluye la clase afectada, método o sección aproximada, la naturaleza de la violación y las posibles consecuencias para la calidad del software.

1. Violaciones al Principio de Responsabilidad Única (SRP)

Clase	Método / Línea Aproximada	Violación detectada	Consecuencias
LibroDTO.java	Métodos esValido(), generarCodigoInterno()	DTO contiene lógica de validación y generación de código interno, mezclando responsabilidad.	Dificulta mantenimiento, testeo y extensibilidad; baja cohesión.
UsuarioDTO.java	Métodos tieneEmailValido(), tieneTelefonoValido	Validaciones de datos incluidas dentro del DTO.	Rompe el patrón DTO, reduce claridad y modularidad.
DatabaseConfig.java	Métodos dataSource() y obtenerLimitePrestamosDefaul	Configuración mezclada con validaciones y lógica de negocio según ambiente.	Dificulta la mantenibilidad y puede inducir errores en ambientes diferentes.

BibliotecaService.java	Toda la clase	Servicio realiza lógica de negocio, acceso a repositorios y manejo de notificaciones en conjunto.	Baja cohesión, código difícil de mantener y probar.
NotificacionService.java	Métodos principales	Mezcla validación y envío de notificaciones en un solo servicio.	Difícil extensión y mantenimiento, violación de SRP.
UsuarioController, libroController y prestamoController	métodos principales	Mezcla de validaciones, generación de reportes y conversiones manuales y préstamoController ejecuta procesos batch	Difícil extensión y mantenimiento, violación de SRP
libroRepository	métodos como contar libros por categoría y obtener estadísticas por categorías	métodos de reporte en el repository	violacion de SRP

2. Violaciones de Cohesión

Clase	Método / Línea Aproximada	Violación detectada	Consecuencias
BibliotecaService.java	Toda la clase	Multiples responsabilidades combinadas (negocio, persistencia y notificaciones).	Código difícil de entender y mantener.

NotificacionService.java	Métodos principales	Lógica de validación y envío combinadas.	Baja cohesión, difícil de extender.
LibroDTO.java	Métodos esValido() y generarCodigoInterno()	Lógica innecesaria en un objeto que debería ser sólo transporte de datos.	Reducción de modularidad y claridad.

3. Violaciones al Encapsulamiento

Clase	Método / Línea Aproximada	Violación detectada	Consecuencias
LibroDTO.java	Métodos validación y generación	Acceso directo a atributos para lógica interna, sin métodos getters/setters.	Posible inconsistencia y fragilidad en el mantenimiento.
UsuarioDTO.java	Métodos tieneEmailValido(), tieneTelefonoValido()	Validación interna sin encapsulación estricta.	Riesgo de inconsistencias y código difícil de modificar.

4. Violaciones al Acoplamiento

Clase	Método / Línea Aproximada	Violación detectada	Consecuencias
BibliotecaService.java	Toda la clase	Uso directo de implementaciones concretas, sin interfaces o inyección adecuada.	Baja flexibilidad, difícil testeo y evolución.

5. Violaciones en Configuración y Seguridad

Clase	Método / Línea Aproximada	Violación detectada	Consecuencias

DatabaseConfig.java	Métodos dataSource() y obtenerLimitePrestamosDefault()	Hardcodeo de parámetros sensibles y lógica de negocio en la configuración.	Riesgo de seguridad, difícil mantenimiento y configuración.

El análisis realizado evidencia múltiples violaciones principalmente en la aplicación del Principio de Responsabilidad Única (SRP), así como problemas significativos de cohesión, encapsulamiento y acoplamiento.

La solución recomendada es refactorizar para separar responsabilidades en clases o servicios especializados, mejorar la encapsulación de los datos y aplicar interfaces para reducir el acoplamiento. Además, es crucial externalizar configuraciones sensibles y lógica de negocio de las clases de configuración.

4. Planificación del rediseño

5. Observaciones de diseño y principios SOLID

Principio / Aspecto	Estado Actual	Observaciones y Recomendaciones
SRP (Responsabilidad única)	Varias violaciones: DTOs tienen lógica de validación y negocio; Configuración tiene validación y lógica	Mover validaciones a servicios o clases validator, no en DTO; separar lógica de negocio de configuración
DIP (Inversión de dependencias)	Controladores y servicios dependen de implementación concreta (repositorios, otros servicios)	Mejor usar interfaces para servicios y repositorios para facilitar pruebas y desacoplamiento
Encapsulamiento	Conversión DTO <-> entidad manual dispersa, duplicada	Usar mappers automáticos (MapStruct, ModelMapper) para evitar errores y mejorar mantenimiento
Acoplamiento	Bajo-moderado, con dependencias claras entre capas	Correcto, pero la lógica en config y DTOs crea acoplamientos innecesarios

Cohesión	Servicios y repositorios cohesionados	Buena cohesión en servicios; DTOs mezclan responsabilidades
Seguridad	SecurityConfig bien separado	Correcto; pero revisar configuraciones específicas no detalladas

.

Propuesta de Rediseño Detallada para Biblioteca Digital

1. Principios Guía para el Rediseño

- **SRP (Principio de Responsabilidad Única):** Cada clase o componente debe tener una sola responsabilidad bien definida.
- **Alta Cohesión:** Las funcionalidades relacionadas deben estar juntas en la misma clase o módulo.
- **Bajo Acoplamiento:** Minimizar dependencias directas entre clases mediante el uso de interfaces y la inyección de dependencias.
- **Encapsulamiento Estricto:** Proteger los datos internos y ofrecer sólo la interfaz necesaria.
- **Separación de Configuración y Lógica:** Mantener la configuración libre de lógica de negocio.
- **Configuración Externa y Segura:** Parámetros sensibles y lógicos configurables fuera del código fuente.

2. Propuesta de Cambios por Paquete y Clases

corregir las violaciones SOLID detectadas y alinearlas a la arquitectura modular propuesta:

1. Módulo libro

- SRP / Modelo vs Lógica de Negocio

- Mueve la lógica puedeSerPrestado, esLibroRestringido, calcularCostoRetraso, generarCodigoBarras a un servicio específico: LibroService.

- Deja la entidad Libro solo con atributos y getters/setters.

- OCP

- Sustituye la lógica condicional de calcularCostoRetraso por una estrategia o fábrica de tarifas, por ejemplo usando un enum con comportamiento o una interfaz CostoRetrasoStrategy.

- DIP / ISP

- Eliminar enviarNotificacionDisponibilidad() y delegar envío al NotificacionService.

- Si solo algunos libros requieren código de barras, extrae esa funcionalidad a BarcodeService.

2. Módulo usuario

- SRP

- Elimina validaciones (puedeRealizarPrestamo, contarPrestamosActivos, getLimitePrestamos, getDiasPrestamo) a un UsuarioService.

- Elimina enviarRecordatorio y generarCredencialProfesor, reubicándolos en un servicio acorde (RecordatorioService, CredencialService).

- OCP

- Usa polimorfismo para definir diferentes tipos de usuarios y sus límites en préstamos. Ejemplo: Estudiante, Profesor extienden una interfaz UsuarioType.

- ISP / LSP

- Evita lanzar UnsupportedOperationException: si la operación no aplica, usa un servicio externo.

- Interface diferenciada para generar credencial solo para tipos de usuario que lo requieran.

3. Módulo prestamo

- SRP / DIP

- Extrae lógica de cálculo de multa a MultaService.

- Da responsabilidades de envío de notificaciones a NotificacionService, no dentro de la entidad Prestamo.
- Resposabilidad
- Convierte estado: String en un campo fuertemente tipado EstadoPrestamo o un enum con lógica asociada.

4. BibliotecaService (servicio orquestador)

- SRP
- Rompe este gran servicio en partes:
- LibroService
- UsuarioService
- PrestamoService
- ReporteService
- NotificacionService (ya existe)
- DIP
- Inyecta NotificacionService en vez de crearlo internamente.
- Usa interfaces para Repository y NotificacionService con implementación inyectada, facilitando testing.
- OCP
- Externaliza la lógica de búsqueda en LibroSpecification o usa Specification de JPA.

5. Controllers

- SRP / DIP
- Mueve validaciones de negocio a servicios.
- Utiliza mapeadores (MapStruct o conversiones manuales) en una capa mapper, no en el controlador.
- Clean Architecture

- Los DTOs deberían ser requests y responses. Convierte en servicios (o mappers) antes de procesar, no en el controlador.

6. NotificacionService

- SRP / OCP
- Separa los canales de notificación:
- Interfaces: EmailSender, SmsSender, WhatsAppSender, con implementaciones concretas.
- NotificacionService depende de interfaces, no de lógica concreta.
- Permite agregar nuevos canales sin modificar la clase original.
- Single Responsibility
- Extrae la lógica de logging y validación a LoggerService o ValidationService, manteniendo cada clase con una única responsabilidad.

7. Repositories

- SRP / ISP
- Mantén solo consultas CRUD básicas en los repositorios.
- Mueve consultas complejas o de negocio a CustomRepository o ReporteService.
- SRP
- Elimina consultas como findLibrosConPocasCopiasDisponibles() o findUsuariosConPrestamosVencidos() desde el repositorio y pásalas a un servicio especializado (ReporteService).

Ejemplo de flujo tras refactorización

1. LibroController recibe LibroDTO, lo convierte con LibroMapper y llama LibroService.crearLibro().
2. LibroService aplica lógica de negocio, guarda entidad y llama a NotificacionService.emailSender().send(...) si es necesario.

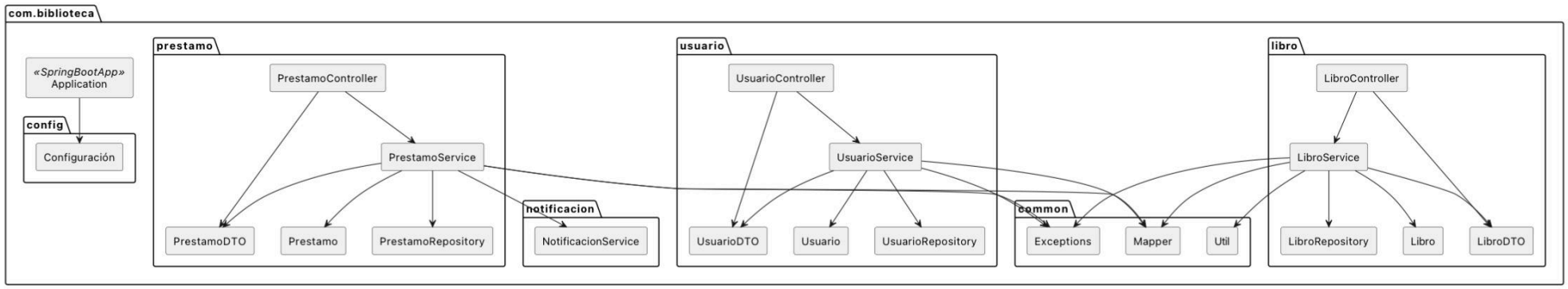
3. PrestamoController delega en PrestamoService, que a su vez usa MultaService, NotificacionService e inyecta repositorios.

Principio Violado	Solución propuesta
SRP	Modularizar lógica en clases específicas
OCP	Sustituir condicionales por estrategia/polimorfismo
DIP	Introducir interfaces, eliminar new dentro de clases
ISP	Interfaces más específicas, eliminar métodos no necesarios
LSP	Evitar lanzar excepciones por tipo incompatible

Nota: Mejoras en el código en el repo: https://github.com/ASARABIAS/ingsoft-2025-I/tree/master/Unidad%201/Taller1/biblioteca-digital-mejorado

3. Propuesta de Arquitectura Modular

Arquitectura Modular - Sistema de Biblioteca



6. Beneficios Esperados

Área de Mejora	Beneficio Esperado
Separación de responsabilidades (SRP)	Clases más pequeñas y enfocadas; mantenimiento más simple y seguro.
Modularización del dominio	Reutilización de componentes; posibilidad de escalar o reemplazar módulos sin afectar otros.
Sustitución de lógica condicional por estrategias (OCP)	Permite añadir nuevos comportamientos (por ejemplo, tarifas o notificaciones) sin modificar el código existente.
Inyección de dependencias (DIP)	Mayor testabilidad mediante el uso de mocks; menor acoplamiento a implementaciones concretas.
Interfaces específicas (ISP)	Evita la exposición de métodos innecesarios; interfaces más fáciles de entender, implementar y mantener.
Mejora de testabilidad	Pruebas unitarias más precisas y rápidas; menor dependencia de pruebas end-to-end para validar cambios en componentes individuales.
Claridad en el diseño	Arquitectura más comprensible para nuevos desarrolladores; mejor documentación y comunicación visual mediante diagramas claros.

Escalabilidad del sistema	Cada módulo puede escalarse de forma independiente (horizontal o verticalmente), lo que facilita la ampliación del sistema según la demanda.
Facilidad para aplicar seguridad	Servicios desacoplados permiten implementar validaciones o filtros de seguridad sin afectar la lógica principal del negocio.
Notificaciones desacopladas	Facilidad para agregar nuevos canales de notificación (como email, SMS o WhatsApp) sin modificar la estructura básica del servicio de notificaciones.
Reutilización de servicios y lógica	Servicios como el cálculo de multas, validación de usuarios o gestión de préstamos pueden ser reutilizados en distintos contextos, reduciendo duplicación de código.
Reducción del tiempo de desarrollo	Al estar basado en componentes modulares y pruebas unitarias bien definidas, se reduce el tiempo requerido para desarrollar, probar y desplegar nuevos módulos.
Menor riesgo de errores	La separación de responsabilidades y la reducción de dependencias monolíticas disminuyen la probabilidad de introducir errores en el sistema.