TALLER NO 1 - PATRONES DE DISEÑO
CASO DE ESTUDIO: ANÁLISIS DE UNA API REST PARA BIBLIOTECA CON FALLOS ARQUITECTÓNICOS Y CARENCIA DE PATRONES DE DISEÑO

GRUPO: 04

ESTUDIANTES:

ING. DANIEL CONTRERAS PEINADO

ING. ANA MARTINEZ TABORDA

ING. JERSON GREGORIO RODRÍGUEZ RAMÍREZ

ING. JEAN CARLOS PEDROZO LÁZARO

UNIVERSIDAD POPULAR DEL CESAR

FACULTAD DE INGENIERÍAS Y TECNOLOGÍAS

ESPECIALIZACIÓN INGENIERÍA DEL SOFTWARE

VALLEDUPAR

2025

Introducción

El proyecto *Sistema de Biblioteca Digital* tiene como objetivo modernizar la gestión bibliotecaria a través de una plataforma tecnológica que permita a los usuarios consultar, solicitar y devolver libros en formato digital. Al mismo tiempo, ofrece a los bibliotecarios herramientas para administrar colecciones, préstamos y usuarios de manera eficiente.

Este sistema surge como respuesta a la necesidad de adaptar los servicios bibliotecarios a las nuevas dinámicas de acceso a la información. En este contexto, se realiza un análisis del estado actual del repositorio del software con el propósito de identificar posibles fallos en su arquitectura y evaluar la correcta aplicación de patrones de diseño a nivel de clases. El objetivo es verificar la idoneidad de la arquitectura implementada y proponer mejoras que optimicen su estructura y mantenibilidad.

Planteamiento del Problema

Las bibliotecas tradicionales enfrentan limitaciones como:

- Procesos manuales que dificultan la gestión eficiente de inventarios y préstamos.
- Dificultad para que los usuarios consulten disponibilidad en tiempo real.
- Falta de trazabilidad en los historiales de préstamos y devoluciones.
- Poca claridad en el catálogo ofertado.

Estas barreras generan cuellos de botella en la atención, errores humanos frecuentes y pérdida de tiempo tanto para usuarios como para administradores.

Propuesta de solución

El análisis realizado evidenció deficiencias en la arquitectura del sistema, tales como una baja cohesión entre componentes, una alta dependencia entre clases y la ausencia de patrones de diseño que garanticen escalabilidad y mantenibilidad.

Como propuesta de mejora, se sugiere una reestructuración a nivel arquitectónico, orientada hacia una arquitectura en capas que separe claramente la lógica de

negocio, la gestión de datos y la presentación. Además, se propone revisar y refactorizar las clases existentes bajo principios SOLID, lo cual permitirá mejorar la cohesión, reducir el acoplamiento y facilitar futuras ampliaciones del sistema.

Con estas mejoras estructurales, se espera lograr un sistema más robusto, comprensible y adaptable a nuevas funcionalidades, facilitando el trabajo de mantenimiento y la evolución del software.

Objetivos del estudio

Objetivos generales:

- Analizar el estado actual del sistema.
- Identificar fallos y falencias a nivel estructural.
- Proponer mejoras y soluciones al sistema.

Objetivos específicos:

- Revisar el diseño actual del sistema.
- Determinar la estructura arquitectónica y, a nivel de clases del sistema.
- Estudiar el contenido de los diferentes archivos y carpetas del repositorio.
- Registrar los fallos observados.
- Plantear una nueva arquitectura basada en capas.
- Establecer las mejoras de las clases en base a los principios SOLID.

Resultados:

El análisis del sistema de Biblioteca Digital permitió identificar múltiples deficiencias estructurales que comprometen su calidad y mantenibilidad. Uno de los hallazgos más relevantes fue la **ausencia de una arquitectura claramente definida**, lo que genera una organización poco coherente del código y dificulta la evolución del sistema.

Se detectaron numerosas violaciones a los **principios SOLID**, especialmente al principio de responsabilidad única (SRP), ya que varias clases asumen múltiples funciones que no les corresponden. Esto contribuye a un alto acoplamiento entre

componentes, disminuye la cohesión interna y aumenta la complejidad del mantenimiento.

Asimismo, la lógica del negocio no está bien delimitada dentro de la estructura del sistema. En varias partes del código, dicha lógica se encuentra dispersa o mezclada con responsabilidades propias de otras capas (como presentación o acceso a datos), dificultando su comprensión y depuración.

Estos resultados reflejan la necesidad de replantear la arquitectura del sistema y aplicar buenas prácticas de diseño que favorezcan la modularidad, la separación de responsabilidades y la claridad del código fuente.

Análisis Crítico Consolidado

- **Casos de Uso Principales**
    - **Bibliotecario:**
        - **Registrar libros y usuarios**
        - **Gestionar préstamos**
        - **Generar reportes**
        - **Administrar cuentas**
- **Flujos de Trabajos Críticos**
    - **Préstamo de libro:** Usuario solicita, el sistema verifica disponibilidad, se registra préstamo.
    - **Registro de libro:** Bibliotecario ingresa datos, sistema valida y guarda.
    - **Registro de usuario:** Se capturan datos del lector y se crean las credenciales.

Mapear clases existentes y sus relaciones

Controllers:

Controlador	Servicio	Modelo	DTO
LibroController	**BibliotecaService**	**Libro**	**LibroDTO**
PrestamoController	**BibliotecaService**	**Prestamo**	**PrestamoDTO**
UsuarioController	**BibliotecaService**	**Usuario**	**UsuarioDTO**

Clases y relaciones:

1. **LibroController**
    - **Responsabilidad:** Gestión de libros: (crear, buscar, consultar y generar reportes)
    - **Depende de:**
        - **BibliotecaService: L**ógica del negocio
        - **LibroDTO:** Objeto de transferencia de datos
        - **Libro:** Representacion de los datos

2. **PrestamoController**
    - **Responsabilidad:** Gestionar préstamos de libros
    - **Depende de:**
        - **BibliotecaService:** Lógica del negocio
        - **PrestamoDTO:** Objeto de transferencia de datos
        - **Prestamo:** Representacion de los datos

3. **UsuarioController**
    - **Responsabilidad:** Registro de usuarios y generación de reportes.
    - **Depende de:**
        - **BibliotecaService:** para lógica de negocio
        - **UsuarioDTO:** Objeto de transferencia de datos
        - **Usuario:** Representacion de los datos

4. **BibliotecaService**
    - **Responsabilidad:** Servicio que concentra lógica de negocio (libros, préstamos, usuarios).
    - **Dependencias:**
        - Repositorios como LibroRepository, PrestamoRepository, UsuarioRepositor.

Identificador de patrones arquitectónicos utilizados

En el sistema de biblioteca se usan varios patrones arquitectónicos y patrones de diseño:

1. **Arquitectura en Capas**

   El sistema está dividido en capas definidas: presentación (controladores), lógica de negocio (servicios), acceso a datos (modelos y repositorios):
    - **Capa de presentación:** LibroController, UsuarioController, PrestamoController
    - **Capa de negocio:** BibliotecaService, NotificationService
    - **Capa de persistencia:** Libro, Usuario, Prestamo

2. **Patrón DTO (Data Transfer Object)**

Clases: LibroDTO, PrestamoDTO, UsuarioDTO

Tiene como objetivo la creación de objetos planos con una serie de atributos que puedan ser enviados o recuperados del servidor y concentrarlas en una única clase simple.

3. **Patrón de Capa de Servicios**

Clases: BibliotecaService, NotificacionService

Un servicio es una clase independiente, que no depende de la capa de presentación (controlador), donde se codifica la lógica de una manera que sea independiente y reusable.

4. **Patrón de inyección de dependencias:**

La anotación @Autowired de Spring inyecta dependencias en controladores y servicios para desacoplar implementaciones concretas, facilitando pruebas y extensibilidad.

Diagrama de clases del estado actual

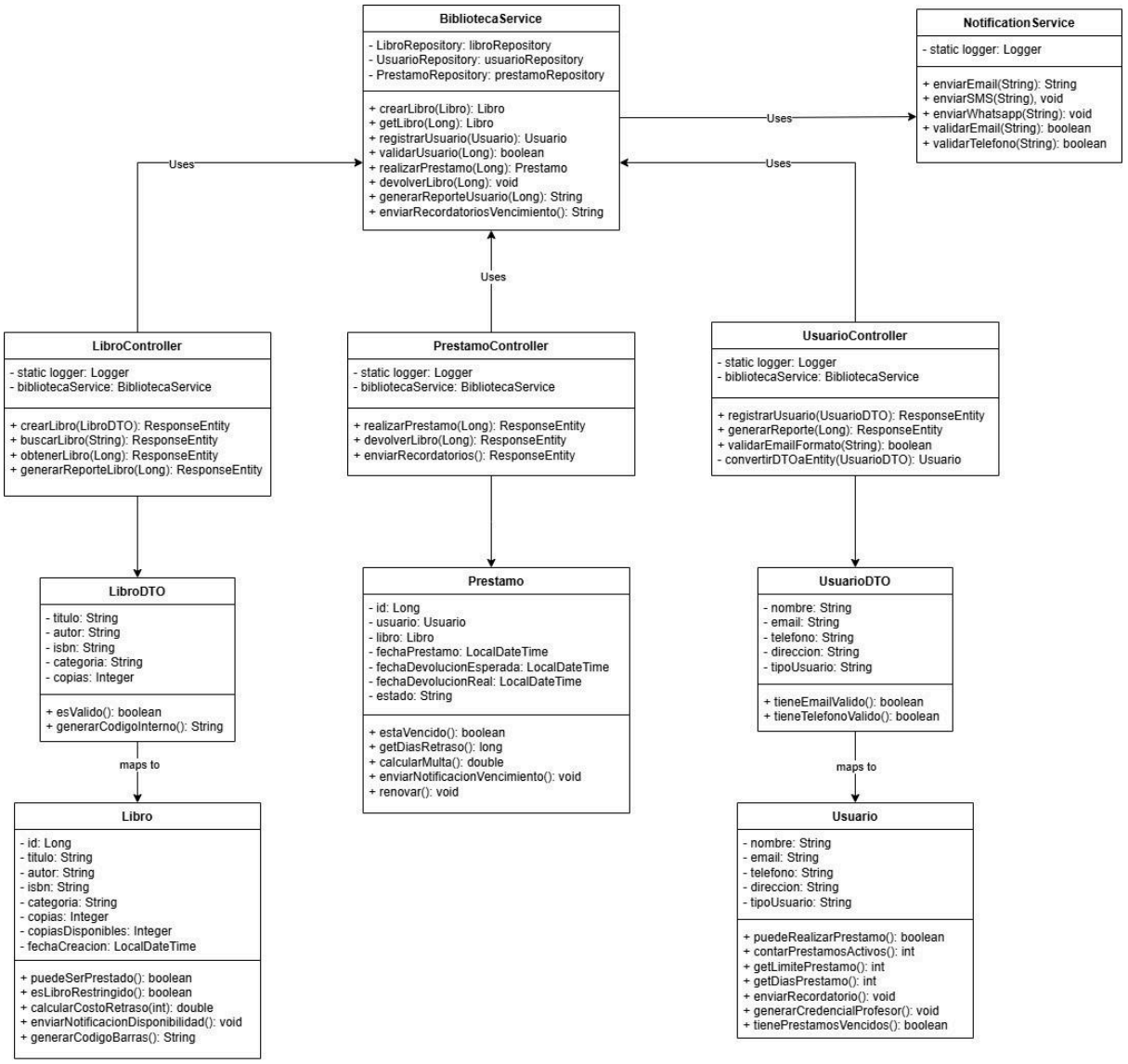
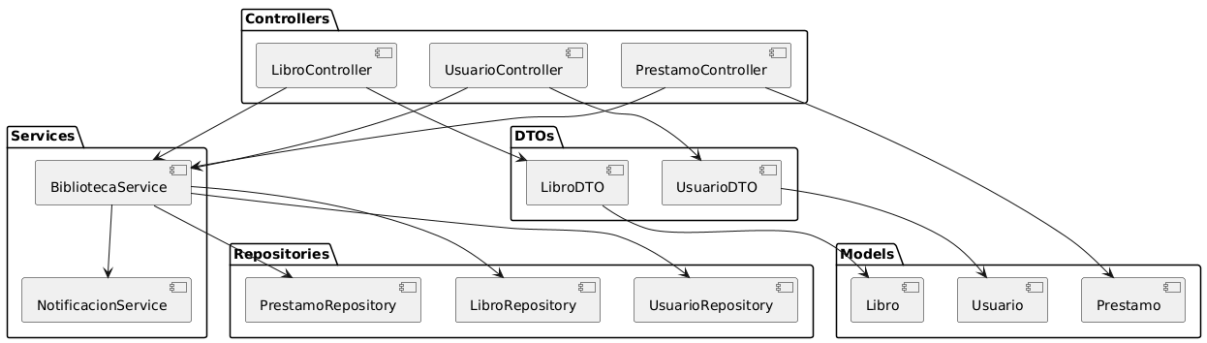


Diagrama de componentes de la arquitectura actual



Violaciones al principio SOLID:

Path:

biblioteca-digital/src/main/java/com/biblioteca/config/DatabaseConfig.java

- Linea: 21, 35
  Principio: SRP
  Comentarios: La clase que inicialmente está orientada a realizar una conexión con la base de datos, también está realizando validaciones de configuración.
- Linea: 43
  Principio: SRP
  Comentarios: La clase que inicialmente está orientada a realizar una conexión con la base de datos, también está aplicando lógica de negocio dentro del mismo.

Path:

biblioteca-digital/src/main/java/com/biblioteca/controller/LibroController.java

- Linea: 24, 29, 49
  Principio: SRP
  Comentarios: La clase que inicialmente está orientada a manejar la configuración de los libros, está realizando validaciones de negocio.
- Linea: 18
  Principio: DIP
  Comentarios: el principio DIP puede no estar presente, ya que la clase LiibroController, depende directamente de la implementación de la clase BibliotecaService, lo que se puede traducir en que, si se modifica dicha clase, puede generar errores en la otra.
- Linea: 74
  Principio: SRP
  Comentarios: La clase LibroController que está enfocada para gestionar los libros, también está realizando tareas de reportes.
- Linea: 35
  Principio: DIP

Comentarios: El controlador convierte directamente un DTO en una entidad, cosa que debería delegarse a otra clase como un mapper.

Path: biblioteca-digital/src/main/java/com/biblioteca/controller/PrestamoController.java

- Linea: 22
  Principio: SRP
  Comentarios: La clase PrestamoController la cual se encarga de gestionar los préstamos y las devoluciones, realiza validaciones de lógica de negocio.
- Linea: 52
  Principio: SRP
  Comentarios: La clase está ejecutando procesos batch.

Path: biblioteca-digital/src/main/java/com/biblioteca/controller/UsuarioController.java

- Linea: 22, 41
  Principio: SRP
  Comentarios: La clase UsuarioController realiza validaciones y verificaciones de lógica de negocio.
- Linea: 34
  Principio: SRP
  Comentarios: La clase UsuarioController además de registrar usuarios, también está generando reportes.
- Linea: 17, 46
  Principio: DIP
  Comentarios: El controlador convierte directamente un DTO en una entidad, cosa que debería delegarse a otra clase como un mapper.

Path: biblioteca-digital/src/main/java/com/biblioteca/dto/LibroDTO.java

- Linea: 14, 22
  Principio: SRP
  Comentarios: La clase LibroDTO realiza validaciones y verificaciones de lógica de negocio.

Path:
biblioteca-digital/src/main/java/com/biblioteca/dto/UsuarioDTO.java

- Linea: 14, 18
  Principio: SRP
  Comentarios: La clase UsuarioDTO realiza validaciones y verificaciones de lógica de negocio.

Path: biblioteca-digital/src/main/java/com/biblioteca/model/Libro.java

- Linea: 28, 32,
  Principio: SRP
  Comentarios: La clase Libro realiza validaciones y verificaciones de lógica de negocio.
- Linea: 37
  Principio: OCP
  Comentarios: Si cambian las categorías de los libros, la clase directamente colapsa y no podría ser extendida.
- Linea: 51
  Principio: DIP
  Comentarios: La clase Libro depende directamente de NotificacionService para poder enviar notificaciones, lo que quiere decir que, si la clase NotificacionService cambia posiblemente no funcione la clase Libro.
- Linea: 57
  Principio: ISP
  Comentarios: La clase Libro implementa funciones que no necesariamente todos los libros deberían de implementar.

Path:
biblioteca-digital/src/main/java/com/biblioteca/model/Prestamo.java

- Linea: 34, 40
  Principio: SRP
  Comentarios: La clase Prestamo realiza validaciones y verificaciones de lógica del negocio.
- Linea: 56
  Principio: SRP

Comentarios: La clase Prestamo envía notificaciones extendiendo sus funciones.

- Linea: 22, 26
  Principio: DIP
  Comentarios: La clase Prestamo tiene dependencias directas a las clases Libro, Usuario y NotificacionService.

Path: biblioteca-digital/src/main/java/com/biblioteca/model/Usuario.java

- Linea: 33, 37, 71
  Principio: SRP
  Comentarios: La clase Usuario realiza validaciones y verificaciones adicionales inherentes a su naturaleza.
- Linea: 63, 71, 78
  Principio: SRP
  Comentarios: La clase Usuario realiza funciones adicionales a su propósito como: enviar coreos y notificaciones, gestionar los préstamos.
- Linea: 71
  Principio: LSP y ISP
  Comentarios: La clase Usuario implementa un método que no todos los usuarios deberían de tener.
- Linea: 44, 53
  Principio: DIP
  Comentarios: La clase Usuario no se puede extender si los tipos de usuario cambian, es decir, está cerrada a extensión y abierta a modificación.

Path: biblioteca-digital/src/main/java/com/biblioteca/repository/LibroRepository.java

- Linea: 30, 33, 37
  Principio: SRP
  Comentarios: La clase LibroRepository extiende sus funcionalidades a reportes.
- Linea: 20, 23, 26
  Principio: ISP

Comentarios: La clase LibroRepository declara métodos que no todos los clientes necesitan o van a utilizar.

Path:
biblioteca-digital/src/main/java/com/biblioteca/security/SecurityConfig.java

- Linea: 17
  Principio: OCP
  Comentarios: La clase SecurityConfig su configuración es muy estática, lo que hace que sea casi que imposible extenderla.

Path:
biblioteca-digital/src/main/java/com/biblioteca/service/BibliotecaService.java

- Linea:
  Principio: SRP
  Comentarios: La clase BibliotecaService básicamente hace de todo: gestiona usuarios, gestiona prestamos, gestiona libros, gestiona reportes y envía notificaciones.
- Linea: 77, 82, 127
  Principio: SRP
  Comentarios: La clase BibliotecaService realiza validaciones y verificaciones de lógica del negocio.
- Linea: 37
  Principio: OCP
  Comentarios: La clase BibliotecaService es imposible de extender si su lógica de búsqueda es modificada.
- Linea: 54
  Principio: DIP
  Comentarios: La clase BibliotecaService depende de forma directa de las clases: NotificacionService, Prestamo.

Path:
biblioteca-digital/src/main/java/com/biblioteca/service/NotificacionService.java

- Linea: 33, 39, 43
  Principio: SRP

Comentarios: La clase NotificacionService incluye entre sus deberes validaciones de correo y teléfono, además de gestionar el log.

Path: biblioteca-digital/src/main/java/com/biblioteca/util/DateUtils.java

- Linea: 39
  Principio: SRP
  Comentarios: La clase DateUtils realiza validaciones y verificaciones de la lógica del negocio.
- Linea:
  Principio: SRP
  Comentarios: La clase DateUtils realiza la función de logs.
- Linea: 23
  Principio: OCP
  Comentarios: La clase DateUtils es imposible de extender si se cambian los tipos de usuarios.

Propuesta de Rediseño Detallado

Clase LibroController:

Se mejoró la separación de responsabilidades en la aplicación al eliminar las validaciones de negocio del método crearLibro en el controlador LibroController. En lugar de validar directamente en el controlador, estas validaciones fueron trasladadas al método correspondiente en la capa de servicio.

Se realizó una mejora en la estructura de la aplicación al remover la lógica de validación del criterio de búsqueda del método buscarLibros dentro del controlador LibroController. Esta validación fue trasladada a la capa de servicio.

Se realizó una refactorización en el controlador de libros con el objetivo de cumplir el Principio de Responsabilidad Única (SRP). Inicialmente, el método encargado de generar reportes de libros contenía lógica de presentación y de negocio directamente implementada dentro del controlador, lo que generaba un acoplamiento indebido y dificultaba la reutilización y el mantenimiento del código.

```java
@RestController  Daniel Contreras +1
@RequestMapping("/api/libros")
public class LibroController {

    @Autowired
    private LibroService libroService;

    public LibroController(LibroService libroService) { this.libroService = libroService; }

    @PostMapping  Daniel Contreras
    public ResponseEntity<Libro> crearLibro(@RequestBody LibroDTO libroDTO) {
        Libro libroCreado = libroService.crearLibro(libroDTO);
        return ResponseEntity.ok(libroCreado);
    }

    @GetMapping("/buscar")  Daniel Contreras
    public ResponseEntity<List<Libro>> buscarLibros(@RequestParam String criterio) {
        List<Libro> libros = libroService.buscarLibros(criterio);
        return ResponseEntity.ok().body(libros);
    }

    @GetMapping("/{id}")  Daniel Contreras
    public ResponseEntity<Libro> obtenerLibro(@PathVariable Long id) {
        Libro libro = libroService.getLibro(id);
        return ResponseEntity.ok().body(libro);
    }

    @GetMapping("/{id}/reporte")  Daniel Contreras
    public ResponseEntity<String> generarReporteLibro(@PathVariable Long id) {
        String reporte = libroService.generarReporteLibro(id);
        return ResponseEntity.ok(reporte);
    }
}
```

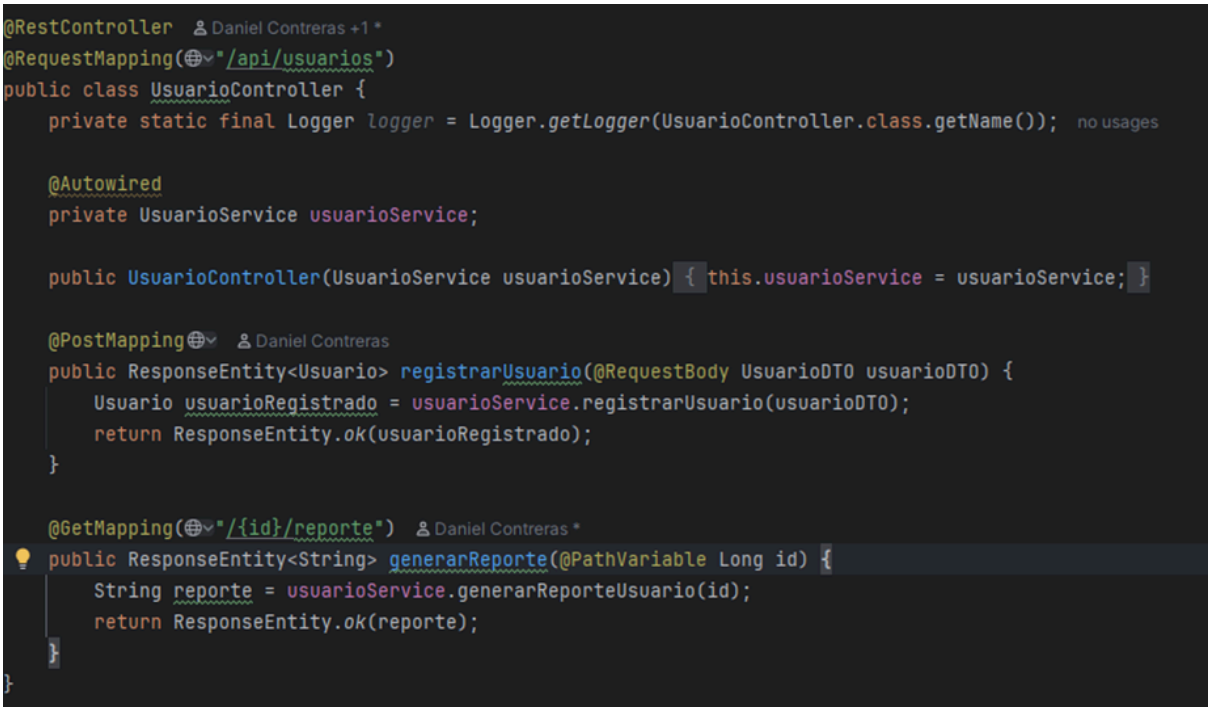
UsuarioController:

Se eliminó la validación manual del formato de email que estaba implementada directamente en el controlador UsuarioController, lo cual representaba una violación del Principio de Responsabilidad Única. Esta lógica fue extraída y centralizada en un método estático llamado esEmailValido, ubicado en una clase utilitaria, lo que permite su reutilización y mejora la organización del código.

Además, se trasladó la responsabilidad de ejecutar dicha validación desde el controlador hacia la capa de servicio. De esta forma, se asegura que la lógica de negocio, como las validaciones, resida donde corresponde: en el servicio.

Se eliminó la conversión manual de DTO a entidad en el controlador para cumplir con el principio de inversión de dependencia. Ahora, dicha conversión se delega completamente al servicio en el método de registrar usuario, centralizando así la

lógica de transformación y permitiendo una mejor abstracción y mantenimiento del código. Esto evita que el controlador tenga responsabilidades adicionales que no le corresponden.

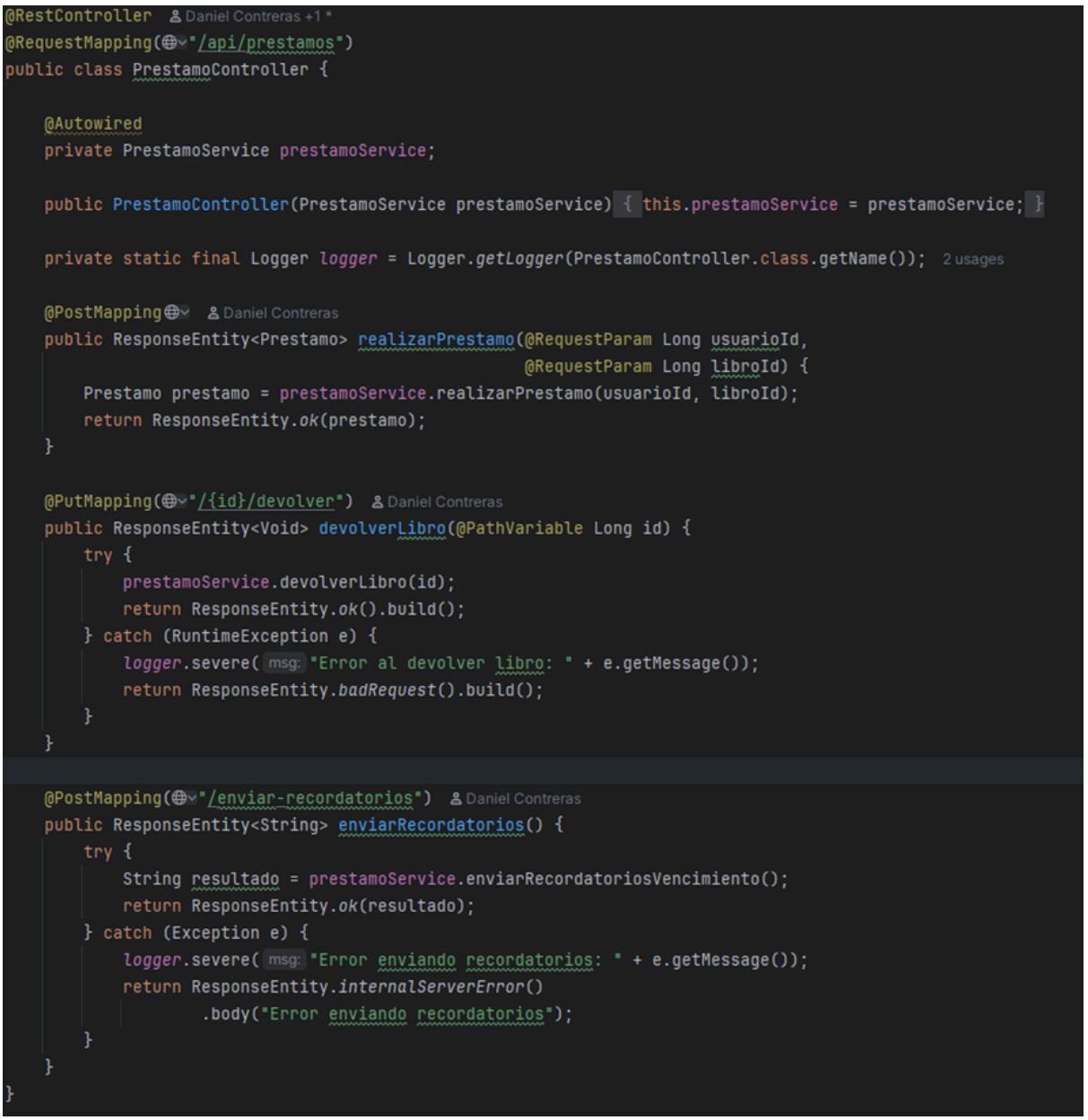


```java
@RestController  Daniel Contreras +1 *
@RequestMapping("/api/usuarios")
public class UsuarioController {
    private static final Logger logger = Logger.getLogger(UsuarioController.class.getName());  no usages

    @Autowired
    private UsuarioService usuarioService;

    public UsuarioController(UsuarioService usuarioService) { this.usuarioService = usuarioService; }

    @PostMapping  Daniel Contreras
    public ResponseEntity<Usuario> registrarUsuario(@RequestBody UsuarioDTO usuarioDTO) {
        Usuario usuarioRegistrado = usuarioService.registrarUsuario(usuarioDTO);
        return ResponseEntity.ok(usuarioRegistrado);
    }

    @GetMapping("/{id}/reporte")  Daniel Contreras *
    public ResponseEntity<String> generarReporte(@PathVariable Long id) {
        String reporte = usuarioService.generarReporteUsuario(id);
        return ResponseEntity.ok(reporte);
    }
}
```

PrestamoController:

Se eliminó la validación de parámetros usuarioId y libroId del controlador y se trasladó dicha validación al método realizarPrestamo dentro del servicio. Esto permite que el controlador mantenga su responsabilidad única.

Se refactorizó el método enviarRecordatorios() en el controlador para cumplir con el Principio de Responsabilidad Única. La lógica de negocio que consistía en verificar si había préstamos vencidos y enviar recordatorios por correo electrónico fue trasladada completamente al servicio PrestamoService.



```java
@RestController  Daniel Contreras +1 *
@RequestMapping("/api/prestamos")
public class PrestamoController {

    @Autowired
    private PrestamoService prestamoService;

    public PrestamoController(PrestamoService prestamoService) { this.prestamoService = prestamoService; }

    private static final Logger logger = Logger.getLogger(PrestamoController.class.getName());  2 usages

    @PostMapping  Daniel Contreras
    public ResponseEntity<Prestamo> realizarPrestamo(@RequestParam Long usuarioId,
                                                     @RequestParam Long libroId) {
        Prestamo prestamo = prestamoService.realizarPrestamo(usuarioId, libroId);
        return ResponseEntity.ok(prestamo);
    }

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

    @PostMapping("/enviar-recordatorios")  Daniel Contreras
    public ResponseEntity<String> enviarRecordatorios() {
        try {
            String resultado = prestamoService.enviarRecordatoriosVencimiento();
            return ResponseEntity.ok(resultado);
        } catch (Exception e) {
            logger.severe( msg: "Error enviando recordatorios: " + e.getMessage());
            return ResponseEntity.internalServerError()
                    .body("Error enviando recordatorios");
        }
    }
}
```

BibliotecaService:

Se realizó una refactorización del servicio BibliotecaService para cumplir con el Principio de Responsabilidad Única. Esta clase originalmente contenía toda la lógica de negocio relacionada con libros, usuarios, préstamos, reportes y notificaciones, lo cual generaba una gran cantidad de responsabilidades en un único componente, dificultando su mantenimiento y evolución.

Como solución, se dividió la lógica en servicios específicos según su dominio. Se creó un LibroService encargado de gestionar exclusivamente las operaciones sobre libros como creación, búsqueda y consulta. Por otro lado, se implementó un UsuarioService para manejar el registro, validación y generación de reportes de los

usuarios. Asimismo, se desarrolló un PrestamoService que concentra toda la lógica relacionada con la realización y devolución de préstamos de libros.

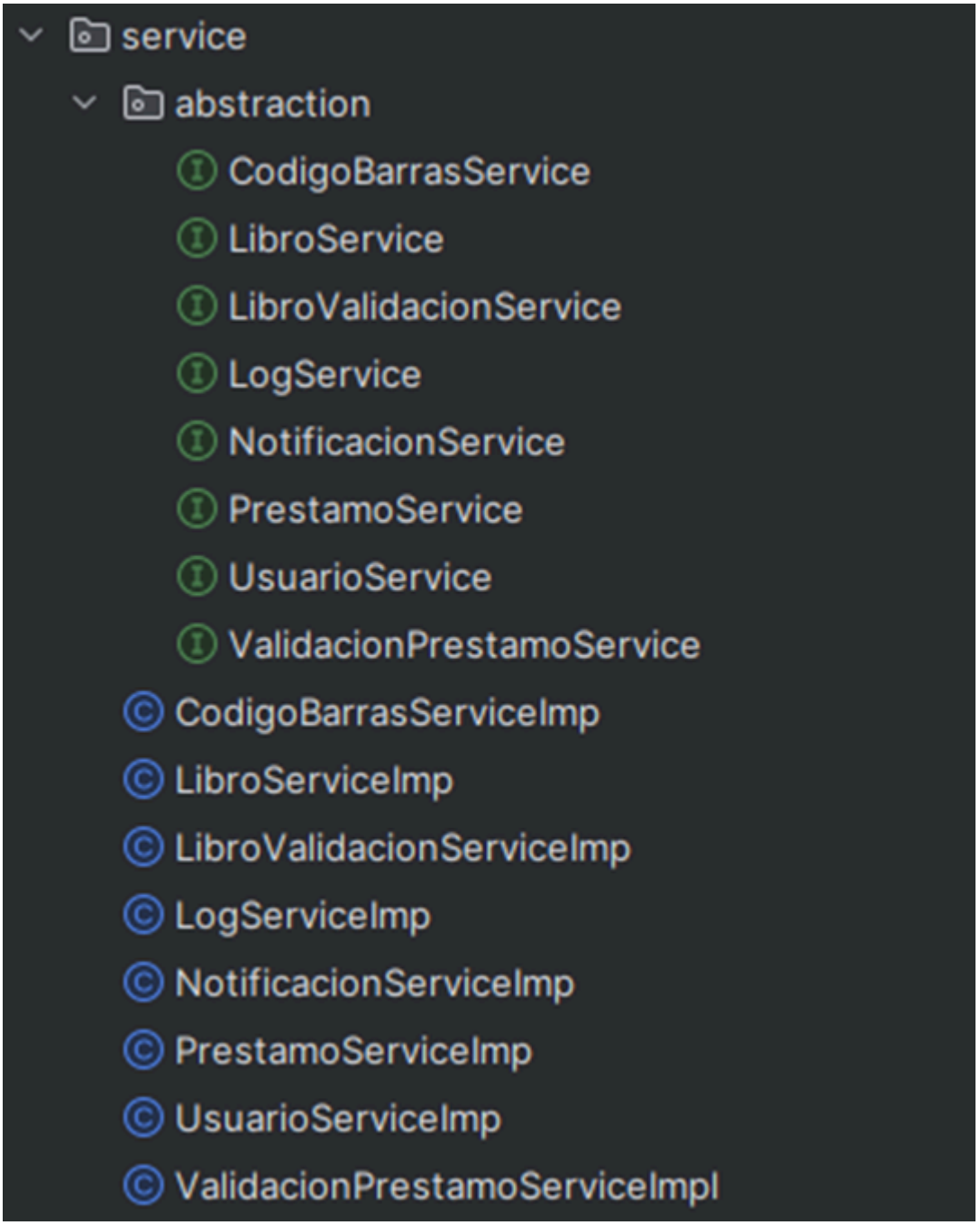
Esta separación de responsabilidades mejora la cohesión de cada clase, reduce el acoplamiento entre componentes y facilita tanto la legibilidad como las pruebas unitarias. Además, se establece una base más sólida para escalar la aplicación de manera ordenada y sostenible.

Se implementó una refactorización del método de búsqueda de libros para cumplir con el Principio de Abierto/Cerrado (OCP), evitando la lógica hardcodeada que dificultaba la extensión del sistema. Para ello, se definió una interfaz de estrategia de búsqueda que abstrae la lógica para cada tipo de criterio (ISBN, autor, título).

Se crearon implementaciones concretas de esta interfaz, cada una encargada de manejar un criterio específico, como búsqueda por ISBN, por autor y por título. Posteriormente, se diseñó un servicio que mantiene un conjunto de estas estrategias y delega la ejecución de la búsqueda a la estrategia correspondiente según el criterio recibido.

Se refactorizó el método realizarPrestamo del servicio de préstamos para eliminar múltiples responsabilidades en un solo método. En particular, se extrajeron las validaciones de usuario y libro hacia un servicio específico llamado ValidacionPrestamoService, la actualización de la disponibilidad del libro se delegó al LibroService, y el envío de notificaciones se canalizó a través de la interfaz NotificacionService, en lugar de depender directamente de una clase concreta.

Se refactorizó el método devolverLibro del servicio de préstamos para eliminar responsabilidades innecesarias dentro de un único método. En esta mejora, la lógica de aumento de disponibilidad del libro se delegó al LibroService, el cálculo de la multa se trasladó a un servicio especializado CostoRetrasoService, y el envío del correo se encapsuló en el NotificacionService.

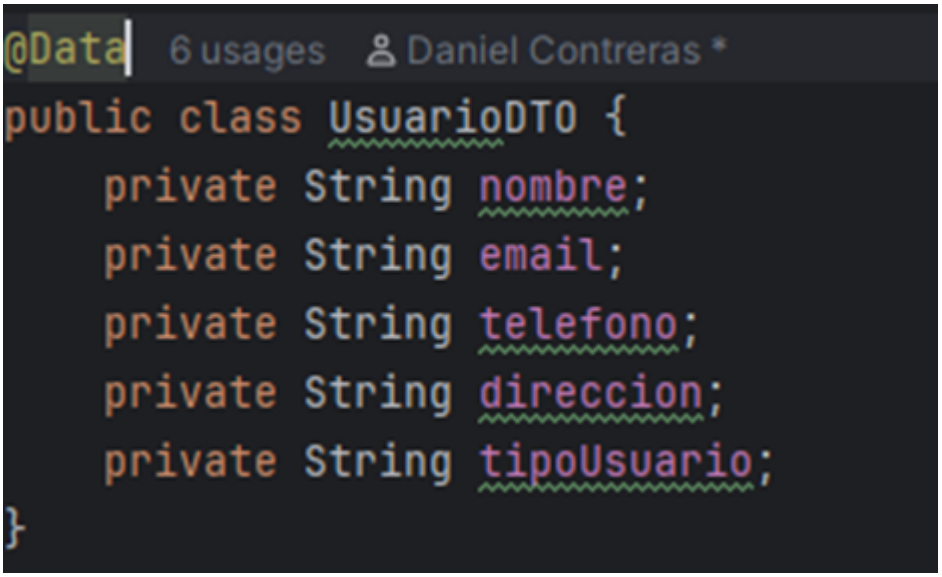


Clase LibroDTO:

Se eliminó la lógica de validación y generación de código del LibroDTO para cumplir con el principio de responsabilidad única (SRP). La validación de campos se trasladó a un servicio específico LibroValidacionService, mientras que la generación del código interno del libro fue delegada a CodigoBarrasService. De esta forma, se mantiene el DTO como un simple contenedor de datos y se evita acoplar lógica de negocio a estructuras de transporte.

UsuarioDTO:

Se procedió a eliminar ambos métodos de validación de la clase UsuarioDTO. La validación del correo electrónico fue trasladada a una clase utilitaria específica llamada EmailValidator, que expone un método estático esEmailValido(String email). De la misma manera, la validación del número telefónico fue movida a otra clase utilitaria, TelefonoValidator, que incluye el método estático esTelefonoValido(String telefono). Estas clases utilitarias son responsables exclusivamente de verificar si los valores cumplen con los formatos esperados.



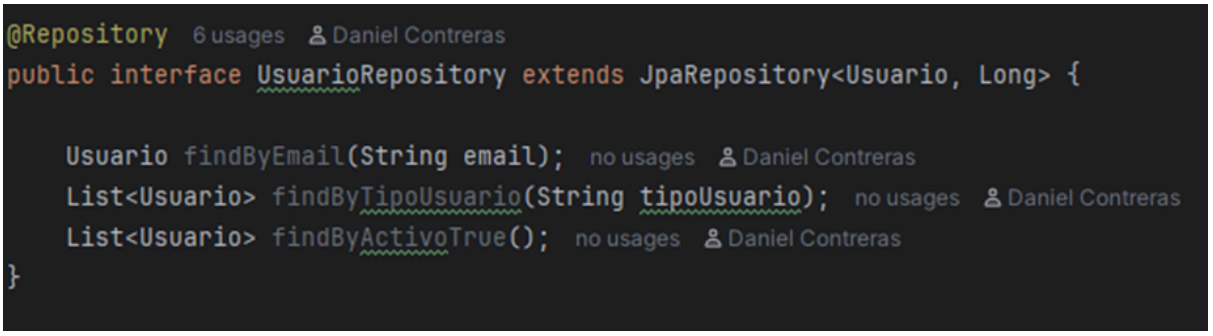
LibroRepository:

Se realizó una refactorización para mejorar la separación de responsabilidades. En primer lugar, se simplificó el repositorio LibroRepository, eliminando todas las consultas específicas con anotaciones @Query que contenían lógica de negocio. Este repositorio quedó limitado únicamente a métodos básicos de acceso a datos, como buscar por título, autor, ISBN o categoría, y se añadió el método findAll() para obtener todos los libros de forma general.

Posteriormente, toda la lógica que antes estaba implementada con consultas específicas en el repositorio se trasladó al servicio LibroService. En este servicio se implementaron métodos que utilizan la lógica estándar para filtrar y contar libros, trabajando directamente sobre listas obtenidas desde el repositorio. Por ejemplo, se implementaron métodos para obtener libros disponibles, libros de referencia, libros especiales, conteo por categoría y libros con pocas copias disponibles, todos ellos usando flujos (Streams) y operaciones de filtrado y agrupamiento en memoria.

UsuarioRepository:

Se ha trasladado la lógica de negocio que antes residía en consultas complejas dentro del repositorio UsuarioRepository hacia el servicio UsuarioService. En lugar de usar consultas JPQL específicas, ahora se recuperan todos los usuarios mediante el método findAll() y se aplican las validaciones y filtros correspondientes en Java, utilizando streams para obtener los usuarios que han alcanzado su límite de préstamos activos o que tienen préstamos vencidos.



Libro:

En la clase Libro se identificaron y eliminaron varias violaciones a los principios SOLID, trasladando su lógica correspondiente a servicios dedicados en la capa de negocio (LibroService, CodigoBarrasService, LibroValidacionService, CostoRetrasoService, LibroNotificacionService), lo cual permitió que la entidad quedara como una simple representación de datos persistentes.

Primero, se eliminó la lógica de validación puedeSerPrestado() y esLibroRestringido() ya que implicaban decisiones de negocio que deben estar desacopladas de la entidad. Esta lógica fue movida al servicio LibroValidacionService, el cual ahora centraliza las reglas sobre si un libro puede o no ser prestado.

Luego, se abordó la violación del principio OCP, presente en el método calcularCostoRetraso, el cual tenía lógica hardcodeada para cada tipo de categoría. Esta lógica fue refactorizada y transferida al servicio CostoRetrasoService, permitiendo que el cálculo de multas por retraso sea extensible y más fácil de mantener.

Además, se detectó una clara violación del principio de inversión de dependencias (DIP) en el método enviarNotificacionDisponibilidad, ya que la entidad instanciaba directamente una implementación concreta (NotificacionServiceImp). Este método fue eliminado de la entidad, y su funcionalidad ahora está correctamente gestionada en el servicio LibroNotificacionService, donde se inyecta la interfaz NotificacionService.

Por último, el método generarCodigoBarras, que también era una responsabilidad ajena a la entidad (y que no todos los libros necesitan), fue eliminado y migrado a un servicio especializado CodigoBarrasService, evitando así la violación del principio de segregación de interfaces.

```java
@Entity  Daniel Contreras *
@Table(name = "libros")
@Data
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class Libro {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String titulo;
    private String autor;
    private String isbn;
    private String categoria;
    private Integer copias;
    private Integer copiasDisponibles;
    private LocalDateTime fechaCreacion;
}
```

Prestamo:

La clase Prestamo presentaba múltiples violaciones a los principios SOLID, especialmente al principio de responsabilidad única (SRP) y al principio de inversión de dependencias (DIP). Para resolver esto, se realizó una refactorización exhaustiva que trasladó toda la lógica de negocio hacia la capa de servicios correspondiente, dejando a la entidad enfocada únicamente en representar el estado de un préstamo.
Inicialmente, la entidad contenía métodos como estaVencido, getDiasRetraso, calcularMulta, enviarNotificacionVencimiento y renovar. Todos estos métodos contenían lógica propia de validación, cálculos de negocio, notificaciones e incluso reglas de renovación, lo cual hacía que la clase estuviera fuertemente acoplada y fuera difícil de mantener o extender.
El método estaVencido, que evaluaba si un préstamo estaba vencido, y getDiasRetraso, que calculaba el retraso en días, fueron movidos a un nuevo servicio PrestamoValidacionService, el cual se encarga ahora de encapsular toda la lógica relacionada con la validez y estado temporal de los préstamos.

El cálculo de la multa, que anteriormente dependía directamente del método calcularCostoRetraso de la entidad Libro, fue movido a un servicio MultaService o integrado dentro de un servicio de PrestamoService, rompiendo así el alto acoplamiento entre Prestamo y Libro, y corrigiendo la violación al principio de inversión de dependencias (DIP).

También se eliminó el método enviarNotificacionVencimiento, que violaba el SRP y dependía de una clase concreta NotificacionServiceImp. Su lógica fue transferida a un servicio de notificaciones, como PrestamoNotificacionService, el cual se encarga de enviar los correos apropiados utilizando una interfaz inyectada, lo cual permite cumplir con el DIP y facilita el testing.

Finalmente, el método renovar, que contenía lógica de negocio compleja y reglas sobre el estado del préstamo y del usuario, también fue externalizado a PrestamoService. Esto no solo redujo la complejidad de la entidad, sino que además permitió centralizar y testear la lógica de renovación de manera más eficiente y desacoplada.

```java
@Entity  Daniel Contreras *
@Table(name = "prestamos")
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
@Data
public class Prestamo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "usuario_id")
    private Usuario usuario;

    @ManyToOne
    @JoinColumn(name = "libro_id")
    private Libro libro;

    private LocalDateTime fechaPrestamo;
    private LocalDateTime fechaDevolucionEsperada;
    private LocalDateTime fechaDevolucionReal;
    private String estado; // ACTIVO, DEVUELTO, VENCIDO
}
```

Usuario:

La clase Usuario fue refactorizada para cumplir con los principios SOLID, especialmente el Principio de Responsabilidad Única (SRP) y el Principio de Inversión de Dependencias (DIP). Anteriormente, esta entidad contenía múltiples responsabilidades que no le correspondían, como validaciones, cálculos de lógica de negocio y envío de notificaciones. Entre estas lógicas se incluían métodos para verificar si el usuario podía realizar préstamos, contar préstamos activos, determinar límites y días de préstamo según su tipo, enviar recordatorios y generar credenciales para profesores.

Toda esta lógica fue extraída y movida al servicio UsuarioService, donde se implementaron métodos como puedeRealizarPrestamo, contarPrestamosActivos,

getLimitePrestamosPorTipo, getDiasPrestamoPorTipo, enviarRecordatorio y generarCredencialProfesor. De esta forma, se mantiene la entidad Usuario como una representación limpia del modelo de dominio, sin mezclas innecesarias de lógica de negocio o detalles de implementación.

Asimismo, se eliminaron las dependencias directas de clases concretas como NotificacionServiceImp, que ahora se inyectan en el servicio mediante su interfaz correspondiente, cumpliendo con DIP y facilitando la inyección de dependencias y el testeo. También se eliminó el método tienePrestamosVencidos de la entidad, ya que esta lógica se encuentra mejor ubicada dentro del servicio, que tiene acceso a los préstamos asociados al usuario. Por último, se solucionó la violación del Principio de Sustitución de Liskov (LSP) al extraer el método generarCredencialProfesor, que no aplicaba a todos los tipos de usuario.

```java
@Entity  Daniel Contreras +1 *
@Table(name = "usuarios")
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
@Data
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;
    private String email;
    private String telefono;
    private String direccion;

    @OneToMany(mappedBy = "usuario", cascade = CascadeType.ALL)
    private List<Prestamo> prestamos = new ArrayList<>();
}
```

Beneficios Esperados

El rediseño de la aplicación de la biblioteca digital, basado en los principios SOLID, representa una transformación estructural profunda que promueve una arquitectura limpia, desacoplada y orientada al mantenimiento a largo plazo. Este cambio no solo responde a buenas prácticas de diseño de software, sino que busca garantizar escalabilidad, facilidad de prueba, comprensión clara de responsabilidades y la reducción del riesgo de errores en futuras modificaciones.

Los principios SOLID —Responsabilidad Única (SRP), Abierto/Cerrado (OCP), Sustitución de Liskov (LSP), Segregación de Interfaces (ISP) e Inversión de Dependencias (DIP)— han sido aplicados de manera transversal en controladores, servicios, entidades y repositorios. A continuación, se analiza su implementación y los beneficios esperados.

Principio de Responsabilidad Única (SRP)

Este principio se aplicó con mayor énfasis en el rediseño. Los controladores (como LibroController, UsuarioController y PrestamoController) fueron limpiados de lógica de validación y negocio, delegando estas tareas a servicios especializados. De igual forma, entidades como Libro, Usuario y Prestamo fueron depuradas, extrayendo métodos de negocio para colocarlos en servicios correspondientes, tales como LibroService, PrestamoService y UsuarioService.

Beneficios esperados:

- Mayor facilidad para mantener y extender el sistema.
- Controladores más enfocados en su rol principal: recibir peticiones y coordinar servicios.
- Entidades de dominio más puras, facilitando su comprensión y reutilización.

Principio de Abierto/Cerrado (OCP)

Se identificaron y corrigieron varios casos de lógica hardcodeada, como en la búsqueda de libros por distintos criterios y en el cálculo de multas por retraso. Estos fueron refactorizados utilizando patrones como el patrón Estrategia, permitiendo que nuevas funcionalidades puedan ser añadidas sin modificar el código existente.

Beneficios esperados:

- Extensión del sistema sin impacto negativo en funcionalidades actuales.
- Reducción del riesgo de introducir errores al modificar funcionalidades existentes.
- Mayor adaptabilidad ante cambios de requisitos.

Principio de Sustitución de Liskov (LSP)

Uno de los problemas detectados fue el uso inapropiado de métodos que no aplicaban a todos los tipos de usuarios, como generarCredencialProfesor. Esta lógica fue extraída, respetando así la coherencia de las subclases o roles del sistema.

Beneficios esperados:

- Evita errores en tiempo de ejecución derivados de comportamientos no esperados.
- Mejora la seguridad del diseño orientado a objetos.
- Claridad sobre qué operaciones son válidas para cada tipo de entidad o rol.

Principio de Segregación de Interfaces (ISP)

Aunque no se trata de interfaces en sentido estricto en todos los casos, se observó una mejora en el diseño al mover funcionalidades como la generación de código de barras (generarCodigoBarras) a servicios independientes. Esto evita que clases como Libro tengan métodos que no todos los libros necesitan, respetando así el principio.

Beneficios esperados:

- Clases con contratos más simples y claros.
- Menor riesgo de tener métodos innecesarios o incorrectos en algunas instancias.
- Interfaces especializadas, alineadas con las necesidades reales de cada componente.

Principio de Inversión de Dependencias (DIP)

Un cambio crucial fue eliminar las instancias directas de clases concretas como NotificacionServiceImpl dentro de entidades o controladores, utilizando en su lugar interfaces inyectadas. Esto promueve una arquitectura más flexible y desacoplada, especialmente útil para pruebas unitarias y mocking.

Beneficios esperados:

- Mayor facilidad para testear los servicios.
- Posibilidad de sustituir implementaciones sin modificar consumidores.
- Alineación con frameworks de inversión de control (IoC) y dependencia (DI).

Diagrama de Clases Rediseñado

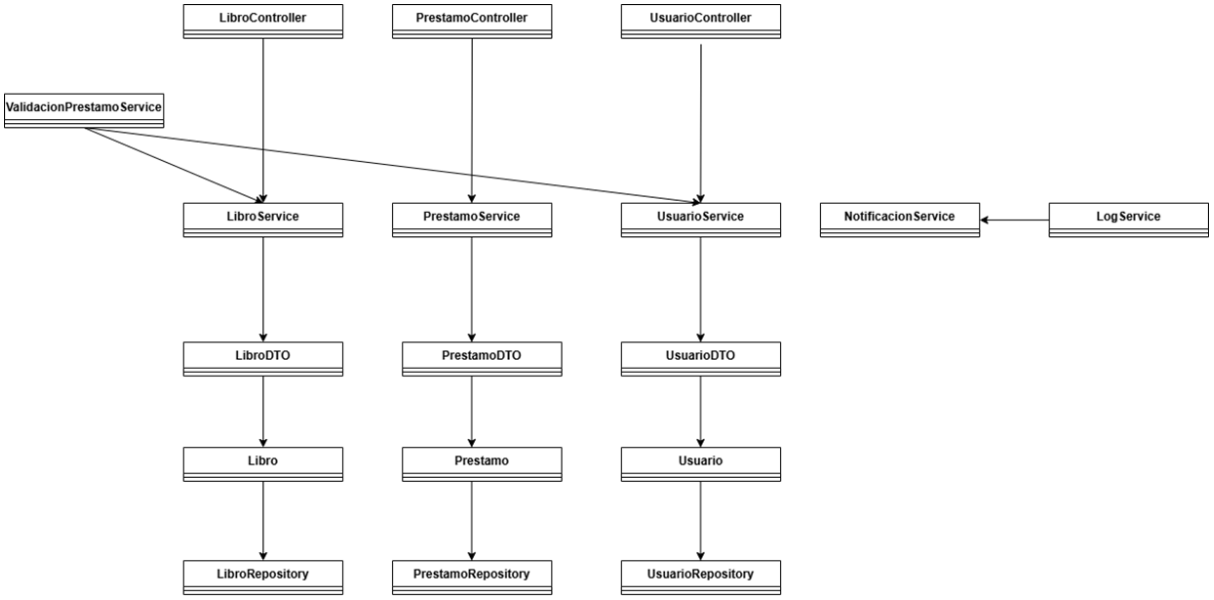


Diagrama de Componentes Rediseñado