FUNDAMENTOS AVANZADOS DE DISEÑO ORIENTADO A OBJETOS

PATRONES DE DISEÑO DE SOFTWARE

GRUPO 05

PRESENTADO POR:

- **Olimpo Castro Thorne**
- **Juan de la Cruz Luque**
- **Jhonnier de la Cruz Guzmán**
- **Jesús Hernández**
- **Wilmer Herrera**

PRESENTADO A:
Ing. Jairo Francisco Seoanes León

UNIVERSIDAD POPULAR DEL CESAR
FACULTAD DE INGENIERÍAS Y TECNOLÓGICAS
ESPECIALIZACIÓN EN INGENIERÍA DE SOFTWARE

2025 - I

1. Introducción

El diseño orientado a objetos es clave para lograr sistemas de software mantenibles, escalables y de alta calidad. Sin embargo, es común encontrar aplicaciones con fallas estructurales que violan principios fundamentales como SOLID y GRASP, afectando su evolución y comprensión, este trabajo analiza y propone el rediseño del Sistema de Gestión de Biblioteca Digital, detectando problemas en su arquitectura original y aplicando buenas prácticas de diseño. El proceso incluye un análisis funcional y estructural, la identificación de violaciones de diseño y la propuesta de mejoras, todo documentado con apoyo de diagramas UML.

2. Problemas de Diseño Detectados

A partir del análisis del código fuente, se identificaron diversas violaciones a los principios SOLID y GRASP, reflejadas en deficiencias estructurales del sistema. Entre las más relevantes se encuentran:

- Clases con múltiples responsabilidades (SRP)
- Acoplamiento elevado entre componentes (GRASP: Low Coupling)
- Lógica de negocio en controladores (GRASP: Controller)
- Falta de abstracción para notificaciones y cálculos de multa (OCP, DIP)
- Uso inadecuado de jerarquías de herencia (LSP)

Estas fallas comprometen la mantenibilidad, escalabilidad y claridad del sistema, dificultando su evolución y prueba.

3. Enfoque de Solución y Rediseño Propuesto

Como respuesta a los hallazgos, se plantea una arquitectura modular basada en capas (presentación, aplicación, dominio e infraestructura), que promueva la separación de responsabilidades y el bajo acoplamiento. La solución incluye:

- Aplicación del patrón Strategy para manejo flexible de notificaciones y multas.
- Introducción de interfaces para cumplir con DIP e ISP.
- Refactorización de clases para aumentar la cohesión y facilitar la reutilización.
- Eliminación de lógica de negocio de controladores y entidades.

Estas acciones se respaldan con nuevos diagramas UML de clases y componentes que representan una estructura más sólida y extensible.

4. Beneficios Esperados

El rediseño propuesto genera mejoras sustanciales en la arquitectura del sistema:

- Disminución de la complejidad interna de las clases
- Mayor facilidad para realizar pruebas unitarias y mantenimiento
- Posibilidad de extender funcionalidades sin afectar el núcleo existente
- Mayor claridad, reutilización y profesionalismo en el código

OBJETIVOS

Objetivo General

Evaluar y reestructurar el diseño arquitectónico de un sistema orientado a objetos con deficiencias técnicas, mediante un proceso de análisis crítico basado en principios SOLID y GRASP, con el fin de proponer una solución más modular, flexible y mantenible.

Objetivos Específicos

- Analizar el comportamiento actual del sistema de biblioteca digital desde una perspectiva funcional y estructural.
- Identificar áreas problemáticas en el diseño que comprometan la calidad, extensibilidad o cohesión del código.
- Clasificar las violaciones encontradas según los principios SOLID y GRASP, estableciendo su impacto técnico.
- Proponer un rediseño fundamentado que aplique buenas prácticas de ingeniería de software, integrando patrones adecuados.
- Evaluar los beneficios de la solución propuesta en términos de reducción de acoplamiento, claridad de responsabilidades y facilidad de mantenimiento.

RESUMEN EJECUTIVO

Este trabajo presenta el análisis y rediseño de un sistema de gestión de biblioteca digital con fallas en su arquitectura orientada a objetos. A través de un diagnóstico estructurado, se identificaron violaciones a los principios SOLID y GRASP que afectan la calidad y mantenibilidad del software.

El análisis permitió detectar clases con múltiples responsabilidades, alto acoplamiento, lógica de negocio en controladores y ausencia de abstracciones. Como solución, se propuso una arquitectura en capas, incorporación de interfaces, aplicación del patrón Strategy y separación clara de responsabilidades.

El rediseño mejora la organización del sistema, facilita su extensión y reduce la complejidad interna. Además, refuerza el aprendizaje práctico sobre buenas prácticas de diseño, aplicables a proyectos reales.

ANALISIS FUNCIONAL

ID	Caso de Uso	Actor Principal	Breve Descripción
CU1	Registrar usuario	Usuario	El sistema permite registrar un nuevo usuario.
CU2	Validar usuario para préstamo	Sistema	Verifica si un usuario puede realizar préstamos.
CU3	Buscar libros	Usuario	Permite buscar libros por título, autor, ISBN o categoría.
CU4	Crear libro	Administrador	Agrega un nuevo libro al sistema.
CU5	Realizar préstamo	Usuario	Solicita el préstamo de un libro disponible.
CU6	Devolver libro	Usuario	Registra la devolución de un libro prestado.
CU7	Generar reporte de usuario	Administrador	Muestra un resumen de préstamos activos y datos del usuario.
CU8	Enviar recordatorios por vencimiento	Sistema (automático)	Envía recordatorios por email a usuarios con préstamos vencidos.
CU9	Consultar libros especiales o por categoría	Usuario / Admin	Lista libros por tipo (REFERENCIA, ESPECIAL) o categoría.
CU10	Consultar estadísticas de libros	Administrador	Muestra estadísticas por categoría o disponibilidad.
CU11	Validar correo o teléfono	Sistema	Validación de formato de datos de contacto.

1. Préstamo de Libro (CU5)

Flujo principal:

1. Usuario solicita préstamo de libro.
2. El sistema valida que:
    - El usuario esté activo.
    - No haya superado el límite de préstamos.
    - El libro tenga copias disponibles.
3. Se registra el préstamo con fecha actual.
4. Se calcula la fecha de devolución según tipo de usuario.
5. Se actualizan las copias disponibles del libro.
6. Se envía notificación al usuario.

2. Devolución de Libro (CU6)

Flujo principal:

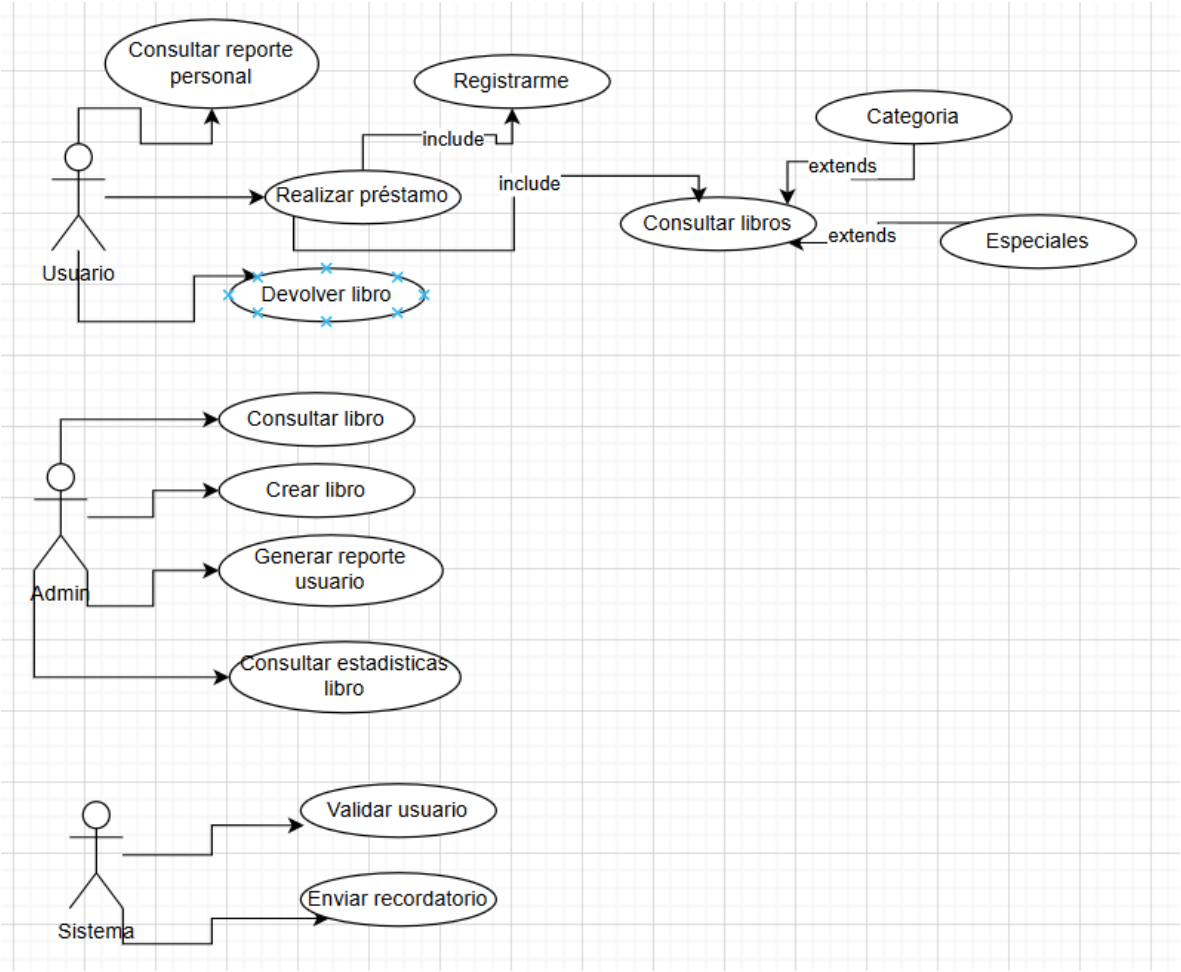
1. Usuario devuelve el libro.
2. El sistema marca el préstamo como DEVUELTO.
3. Se actualiza la disponibilidad del libro.
4. Se calcula si hay multa por retraso.
5. Si aplica multa, se envía notificación al usuario.

3. Envío de Recordatorios por Vencimiento (CU8)

Flujo principal:

1. El sistema busca todos los préstamos vencidos.
2. Por cada préstamo:
    - Se genera un mensaje de recordatorio.
    - Se envía por email al usuario.

3. Se registra la operación.



Punto 2: Análisis Estructural

2.1 Mapeo de clases existentes y sus relaciones

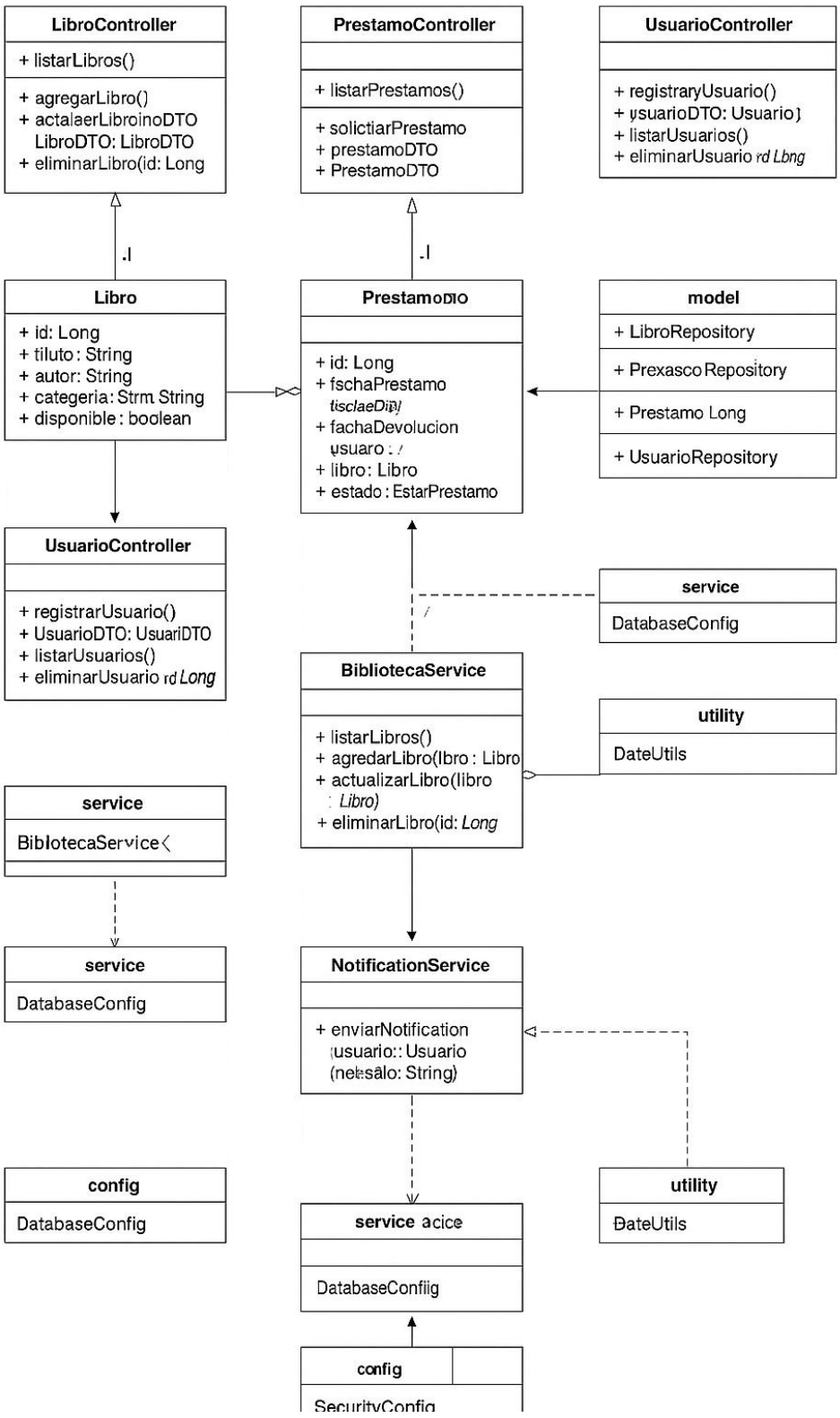
A continuación se muestra el agrupamiento de clases del sistema de biblioteca digital según su paquete y responsabilidad:

- controller/: Manejo de peticiones HTTP
  - LibroController, PrestamoController, UsuarioController
- dto/: Transferencia de datos
  - LibroDTO, PrestamoDTO, UsuarioDTO
- model/: Entidades del dominio
  - Libro, Prestamo, Usuario, EstadoPrestamo
- repository/: Acceso a datos

  - LibroRepository, PrestamoRepository, UsuarioRepository
- service/: Lógica de negocio
  - BibliotecaService, NotificacionService
- config/: Configuración
  - DatabaseConfig, SecurityConfig
- util/: Utilidades
  - DateUtils

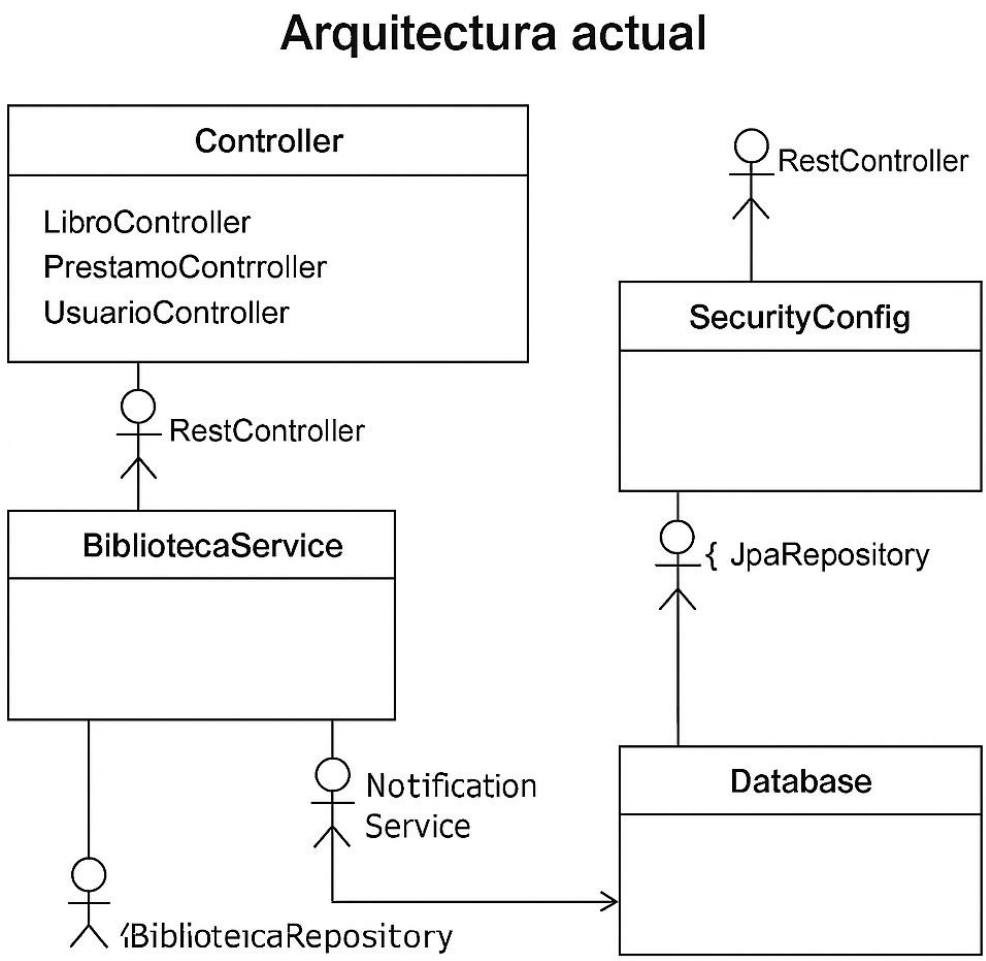
2.2 Diagrama de clases del estado actual

A continuación se presenta el diagrama de clases generado por ingeniería inversa del sistema actual:


LibroController
+ listarLibros()
+ agregarLibro()
+ actalaerLibroinoDTO
LibroDTO: LibroDTO
+ eliminarLibro(id: Long
PrestamoController
+ listarPrestamos()
+ solictiarPrestamo
+ prestamoDTO
+ PrestamoDTO
UsuarioController
+ registraryUsuario()
+ usuarioDTO: Usuario)
+ listarUsuarios()
+ eliminarUsuario rd Lbng
Libro
+ id: Long
+ tiluto : String
+ autor: String
+ categeria: Strm String
+ disponible : boolean
PrestamoDTO
+ id: Long
+ fschaPrestamo
+ fachaDevolucion
+ libro: Libro
+ estado : EstarPrestamo
model
+ LibroRepository
+ Prexasco Repository
+ Prestamo Long
+ UsuarioRepository
UsuarioController
+ registrarUsuario()
+ UsuarioDTO: UsuariDTO
+ listarUsuarios()
+ eliminarUsuario rd Long
service
DatabaseConfig
BibliotecaService
+ listarLibros()
+ agredarLibro(Ibro : Libro
+ actualizarLibro(Iibro
: Libro)
+ eliminarLibro(id: Long
utility
DateUtils
service
BiblotecaService
service
DatabaseConfig
NotificationService
+ enviarNotification
(usuario:: Usuario
(nelsâlo: String)
config
DatabaseConfig
service acice
DatabaseConfiig
utility
DateUtils
config
SecurityConfig

2.3 Diagrama de componentes del estado actual

Este diagrama representa la arquitectura de componentes tal como está diseñada actualmente en el sistema:



Punto 3: Identificación Detallada de Problemas de Diseño

Clase: DatabaseConfig.java

- Línea 19: // VIOLACIÓN SRP: Configuración con lógica de validación
- Línea 21: if (!esUrlValida(url)) {
- Línea 34: // VIOLACIÓN SRP: Clase de configuración con validación
- Línea 41: // VIOLACIÓN: Lógica de negocio en configuración
- Línea 48: if ("production".equals(ambiente)) {
- Línea 50: } else if ("staging".equals(ambiente)) {

Clase: LibroController.java

- Línea 20: // VIOLACIÓN: Controller hace validación de negocio
- Línea 23: // Validación en controller (VIOLACIÓN SRP)
- Línea 24: if (libroDTO.getTitulo() == null || libroDTO.getTitulo().trim().isEmpty()) {
- Línea 29: if (libroDTO.getCopias() == null || libroDTO.getCopias() <= 0) {
- Línea 34: // VIOLACIÓN DIP: Conversión manual DTO -> Entity
- Línea 35: Libro libro = new Libro();
- Línea 48: // VIOLACIÓN: Controller maneja lógica de negocio
- Línea 49: if (criterio == null || criterio.trim().length() < 3) {
- Línea 60: // VIOLACIÓN: No hay separación clara de responsabilidades
- Línea 72: // VIOLACIÓN SRP: Controller genera reportes
- Línea 80: StringBuilder reporte = new StringBuilder();

Problema:

- El LibroController maneja lógica de validación, transformación de datos (DTO ↔ Entity) y generación de reportes.
- El controlador depende directamente de BibliotecaService en lugar de una interfaz
- El controlador realiza tareas que no son su responsabilidad (validación, conversión, reporte, etc.).

```
import com.biblioteca.service.BibliotecaService;
```

```
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);
```

Clase: PrestamoController.java

- Línea 21: // VIOLACIÓN: Controller maneja lógica de negocio
- Línea 22: if (usuarioId == null || libroId == null) {
- Línea 48: // VIOLACIÓN SRP: Controller ejecuta procesos batch
- Línea 53: if(rcordatorios.isEmpty()){

Problema:

- El controlador valida directamente los parámetros usuarioId y libroId (línea 21), lo que mezcla lógica de validación.
- El controlador gestiona directamente el Logger y la lógica de negocio (como respuestas de error). Esto dispersa la lógica transversal y de negocio en múltiples capas.
- El método enviarRecordatorios() mezcla lógica de negocio (determinar si hay préstamos vencidos) en el controlador.
- El controlador forma respuestas como "Recordatorios enviados exitosamente:
...", lo que mezcla lógica de presentación con lógica de aplicación.

```
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();
}
```

```
public ResponseEntity<String> enviarRecordatorios() {
    try {
        String rcordatorios=bibliotecaService.enviarRecordatoriosVencimiento();
        if(rcordatorios.isEmpty()){
            return ResponseEntity.ok("No hay prestamos VENCIDOS - No se enviaron recordatorios");
        }else {
            return ResponseEntity.ok("Recordatorios enviados exitosamente: <br>" + rcordatorios);
        }
    } catch (Exception e) {
        logger.severe("Error enviando recordatorios: " + e.getMessage());
        return ResponseEntity.internalServerError()
                .body("Error enviando recordatorios");
    }
}
```

Clase: UsuarioController.java

- Línea 21: // VIOLACIÓN: Validación en controller
- Línea 22: if (!validarEmailFormato(usuarioDTO.getEmail())) {
- Línea 26: // Conversión manual (VIOLACIÓN DIP)
- Línea 35: // VIOLACIÓN SRP: Controller genera reportes
- Línea 40: // VIOLACIÓN SRP: Controller hace validaciones
- Línea 45: // VIOLACIÓN DIP: Conversión manual sin abstracción
- Línea 47: Usuario usuario = new Usuario();

Problema:

- La validación del email se encuentra embebida en el controlador, acoplando presentación con lógica de negocio.
- El controlador convierte manualmente el DTO en la entidad Usuario, mezclando responsabilidades.
- Se viola DIP al instanciar y manejar directamente detalles concretos de Usuario.
- El controlador genera directamente un reporte, violando el principio de separación de responsabilidades.
- El método validarEmailFormato() pertenece a la lógica de negocio, no a la capa de presentación.
- Conversión DTO → Entity acoplada al modelo.
- El controlador asume múltiples responsabilidades (validación, mapeo, lógica de negocio, presentación).

```
if (!validarEmailFormato(usuarioDTO.getEmail())) {
    return ResponseEntity.badRequest().build();
}
```

Clase: LibroDTO.java

- Línea 13: // VIOLACIÓN SRP: DTO con lógica de validación
- Línea 21: // VIOLACIÓN: DTO con lógica de negocio

Problema:

- El método esValido() agrega lógica de validación dentro del DTO, violando SRP (el DTO debería solo transportar datos).
- El método generarCodigoInterno() contiene lógica de negocio (formateo y generación de código) dentro del DTO, lo que rompe cohesión y encapsulamiento de responsabilidades.

```
public boolean esValido() {
    return titulo != null && !titulo.trim().isEmpty() &&
           autor != null && !autor.trim().isEmpty() &&
           isbn != null && isbn.length() >= 10 &&
           copias != null && copias > 0;
}
```

```
public String generarCodigoInterno() {
    return categoria.substring(0, 3).toUpperCase() + "-" +
           isbn.substring(isbn.length() - 4);
}
```

Clase: UsuarioDTO.java

- Línea 13: // VIOLACIÓN SRP: DTO con validación

Problema:

- Lógica de validación

```
public boolean tieneEmailValido() {
    return email != null && email.contains("@") && email.contains(".");
}

public boolean tieneTelefonoValido() {
    return telefono != null && telefono.matches("\\d{10}");
}
```

Clase: Libro.java

- Línea 27: // VIOLACIÓN SRP: Lógica de negocio en entidad
- Línea 36: // VIOLACIÓN OCP: Lógica hardcodeada difícil de extender
- Línea 38: if (categoria.equals("NORMAL")) {
- Línea 40: } else if (categoria.equals("PREMIUM")) {
- Línea 42: } else if (categoria.equals("ESPECIAL")) {
- Línea 48: // VIOLACIÓN DIP: Dependencia directa de implementación concreta
- Línea 51: NotificacionService notificacion = new NotificacionService();
- Línea 56: // VIOLACIÓN ISP: Método que no todos los libros necesitan

Problema:

- Tiene métodos como puedeSerPrestado(), esLibroRestringido(), calcularCostoRetraso() y enviarNotificacionDisponibilidad() dentro de la entidad.
- Libro.java usaba if-else para calcular el costo de retraso según la categoría.
- Libro creaba directamente una instancia concreta de NotificacionService
- Método generarCodigoBarras() en la entidad, que no aplica a todos los tipos de libros.
- La entidad tenía demasiadas responsabilidades, y estaba acoplada a servicios externos.

```java
public boolean puedeSerPrestado() {
    return copiasDisponibles > 0 && !esLibroRestringido();
}

public boolean esLibroRestringido() {
    return categoria.equals("REFERENCIA") || categoria.equals("ESPECIAL");
}
```

```java
public double calcularCostoRetraso(int diasRetraso) {
    if (categoria.equals("NORMAL")) {
        return diasRetraso * 0.5;
    } else if (categoria.equals("PREMIUM")) {
        return diasRetraso * 1.0;
    } else if (categoria.equals("ESPECIAL")) {
        return diasRetraso * 2.0;
    }
    return 0.0;
}
```

Clase: Prestamo.java

- Línea 33: // VIOLACIÓN SRP: Prestamo maneja validaciones, cálculos Y notificaciones
- Línea 40: if (fechaDevolucionReal != null) {
- Línea 46: // VIOLACIÓN DIP: Dependencia directa de clases concretas
- Línea 49: if (diasRetraso <= 0) return 0.0;
- Línea 55: // VIOLACIÓN SRP: Prestamo envía sus propias notificaciones
- Línea 57: NotificacionService notificacion = new NotificacionService();
- Línea 64: // VIOLACIÓN: Lógica de negocio compleja en entidad
- Línea 65: if (estaVencido()) {
- Línea 66: throw new IllegalStateException("No se puede renovar un préstamo vencido");
- Línea 69: if (!usuario.puedeRealizarPrestamo()) {
- Línea 70: throw new IllegalStateException("Usuario no puede realizar más préstamos");
- Línea 77: // Envía notificación (violación SRP)
- Línea 78: NotificacionService notificacion = new NotificacionService();

Problema:
La clase Prestamo realiza varias tareas:

- Calcula vencimientos y multas.
- Envía notificaciones.
- Valida reglas de negocio.
- Ejecuta renovaciones.

- Prestamo dependía directamente de NotificacionService, lo cual rompe DIP.
- Libro.calcularCostoRetraso() tenía múltiples condicionales (if-else) basados en la categoría del libro.
- Prestamo hacía tareas que debía delegar a expertos.
- Prestamo llamaba métodos directamente de Usuario y Libro, creando un alto acoplamiento.

```
public double calcularMulta() {
    long diasRetraso = getDiasRetraso();
    if (diasRetraso <= 0) return 0.0;
```

```
return libro.calcularCostoRetraso((int) diasRetraso);
```

```
if (estaVencido()) {
    throw new IllegalStateException("No se puede renovar un préstamo vencido");
}

if (!usuario.puedeRealizarPrestamo()) {
    throw new IllegalStateException("Usuario no puede realizar más préstamos");
}
```

Clase: Usuario.java

- Línea 29: // VIOLACIÓN SRP: Usuario maneja sus propios préstamos Y validaciones Y notificaciones
- Línea 31: private List<Prestamo> prestamos = new ArrayList<>();
- Línea 43: // VIOLACIÓN OCP: Lógica hardcodeada para tipos de usuario
- Línea 45: switch (tipoUsuario) {
- Línea 54: switch (tipoUsuario) {
- Línea 62: // VIOLACIÓN DIP y SRP: Usuario se encarga de notificaciones
- Línea 64: NotificacionService notificacion = new NotificacionService();
- Línea 70: // VIOLACIÓN LSP: Método que no aplica a todos los tipos de usuario
- Línea 72: if (!tipoUsuario.equals("PROFESOR")) {

- Línea 73: throw new UnsupportedOperationException("Solo profesores pueden generar credenciales");

Problema:

- La clase Usuario maneja lógica de préstamos, validaciones, generación de credenciales, y envío de notificaciones.
- getLimitePrestamos() y getDiasPrestamo() usan switch con constantes hardcodeadas según el tipo de usuario.
- La clase Usuario instancia directamente NotificacionService.
- Método generarCredencialProfesor() no aplica a todos los tipos de usuario.
- Lógica de préstamo y negocio mezclada en la entidad.
- Alta dependencia entre clases concretas.

```
public boolean puedeRealizarPrestamo() {
    return activo && contarPrestamosActivos() < getLimitePrestamos() && !tienePrestamosVencidos();
}
```

```
public int getLimitePrestamos() {
    switch (tipoUsuario) {
        case "ESTUDIANTE": return 3;
        case "PROFESOR": return 10;
        case "ADMINISTRATIVO": return 5;
        default: return 1;
    }
}

public int getDiasPrestamo() {
    switch (tipoUsuario) {
        case "ESTUDIANTE": return 15;
        case "PROFESOR": return 30;
        case "ADMINISTRATIVO": return 20;
        default: return 7;
    }
}
```

Clase: LibroRepository.java

- Línea 18: // VIOLACIÓN ISP: Métodos específicos que no todos los clientes necesitan

- Línea 28: // Métodos de reporte (VIOLACIÓN SRP del repositorio)
- Línea 35: // Métodos de negocio (VIOLACIÓN: repositorio con lógica de negocio)

Problema:

- LibroRepository contiene muchos métodos especializados que no todos sus consumidores necesitan, mezclando búsquedas simples, consultas complejas, estadísticas y lógica de negocio.
- El repositorio está mezclando múltiples responsabilidades: acceso a datos básico, estadísticas y lógica de negocio como “libros con pocas copias”.
- Consultas avanzadas y estadísticas están dentro del repositorio principal, generando una clase inflada y poco cohesionada.

```
List<Libro> findByTituloContaining(String titulo);
List<Libro> findByAutorContaining(String autor);
List<Libro> findByIsbn(String isbn);
List<Libro> findByCategoria(String categoria);
```

Clase: UsuarioRepository.java

- Línea 16: // VIOLACIÓN: Consultas complejas de negocio en repositorio

Problema:

- El repositorio contiene lógica de negocio embebida en consultas complejas

```
@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();
```

Clase: SecurityConfig.java

- Línea 20: .requestMatchers(new AntPathRequestMatcher("/h2-console/**")).authenticated() //

Problema:

- La clase SecurityConfig está encargándose de múltiples configuraciones de seguridad directamente.

Clase: BibliotecaService.java

- Línea 23: // VIOLACIÓN SRP: Un servicio que hace TODO
- Línea 38: // VIOLACIÓN OCP: Lógica de búsqueda hardcodeada
- Línea 39: if (criterio.startsWith("ISBN:")) {

- Línea 41: } else if (criterio.startsWith("AUTOR:")) {
- Línea 53: // VIOLACIÓN DIP: Dependencia directa de NotificacionService
- Línea 54: NotificacionService notificacion = new NotificacionService();
- Línea 62: if (usuario.isEmpty()) return false;
- Línea 70: // VIOLACIÓN: Método muy largo con múltiples responsabilidades
- Línea 72: .orElseThrow(() -> new RuntimeException("Usuario no encontrado"));
- Línea 75: .orElseThrow(() -> new RuntimeException("Libro no encontrado"));
- Línea 78: if (!usuario.puedeRealizarPrestamo()) {
- Línea 79: throw new RuntimeException("Usuario no puede realizar préstamos");
- Línea 82: if (!libro.puedeSerPrestado()) {
- Línea 83: throw new RuntimeException("Libro no disponible para préstamo");
- Línea 87: Prestamo prestamo = new Prestamo();
- Línea 103: // Enviar notificaciones (VIOLACIÓN SRP)
- Línea 104: NotificacionService notificacion = new NotificacionService();
- Línea 115: .orElseThrow(() -> new RuntimeException("Préstamo no encontrado"));
- Línea 127: if (multa > 0) {
- Línea 128: // VIOLACIÓN: Lógica de multas mezclada con devolución
- Línea 129: NotificacionService notificacion = new NotificacionService();
- Línea 137: // Reportes (VIOLACIÓN SRP: no debería estar aquí)
- Línea 140: .orElseThrow(() -> new RuntimeException("Usuario no encontrado"));
- Línea 142: StringBuilder reporte = new StringBuilder();
- Línea 151: // Notificaciones masivas (VIOLACIÓN SRP)
- Línea 154: NotificacionService notificacion = new NotificacionService();
- Línea 155: StringBuilder notificacionesEnviadas = new StringBuilder();

Problema:

- Está asumiendo múltiples responsabilidades: gestión de libros, usuarios, préstamos, reportes y notificaciones.
- La búsqueda de libros está hardcodeada con condicionales.
- Creación directa de objetos (new NotificacionService()), acoplando fuertemente la lógica a implementaciones específicas.

```
public Libro crearLibro(Libro libro) {
    libro.setFechaCreacion(LocalDateTime.now());
    libro.setCopiasDisponibles(libro.getCopias());
    return libroRepository.save(libro);
}
```

```java
public boolean validarUsuario(Long usuarioId) {
    Optional<Usuario> usuario = usuarioRepository.findById(usuarioId);
    if (usuario.isEmpty()) return false;

    Usuario u = usuario.get();
    return u.isActivo() && u.contarPrestamosActivos() < u.getLimitePrestamos();
}
```

Clase: NotificacionService.java

- Línea 10: // VIOLACIÓN OCP: Difícil agregar nuevos tipos de notificación
- Línea 16: // VIOLACIÓN SRP: Lógica de logging mezclada con envío
- Línea 27: // VIOLACIÓN: Si queremos agregar WhatsApp, Slack, etc., hay que modificar esta clase
- Línea 38: // VIOLACIÓN SRP: NotificacionService también valida formatos

Problema:

- La clase NotificacionService tiene múltiples responsabilidades: enviar notificaciones(email, SMS, WhatsApp), Validar formatos de email y teléfono y registrar logs.

Clase: DateUtils.java

- Línea 11: // VIOLACIÓN SRP: Mezcla formateo, cálculos Y logging
- Línea 22: // VIOLACIÓN OCP: Lógica hardcodeada para tipos de usuario
- Línea 26: switch (tipoUsuario) {
- Línea 38: // VIOLACIÓN SRP: Validación mezclada con utilidades de fecha
- Línea 45: // VIOLACIÓN: Lógica de negocio en clase de utilidades

Problema:

- DateUtils tiene múltiples responsabilidades: formateo de fechas, cálculos, validaciones, lógica de negocio y logging.

```java
public static String formatearFecha(LocalDateTime fecha) {
    logger.info("Formateando fecha: " + fecha);
    return fecha.format(DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm"));
}

public static long calcularDiasEntre(LocalDateTime inicio, LocalDateTime fin) {
    logger.info("Calculando días entre: " + inicio + " y " + fin);
    return ChronoUnit.DAYS.between(inicio, fin);
}
```

Punto 4: Planificación del Rediseño Detallado

En esta sección se presenta la solución específica para cada una de las violaciones identificadas en el Punto 3. Cada entrada incluye la clase afectada, la línea de código referencial, el principio violado y la solución propuesta.

Clase	Línea	Principio Violado	Solución Propuesta
DatabaseConfig.java	19	Responsabilidad Única	Separar responsabilidades en clases especializadas.
DatabaseConfig.java	21	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
DatabaseConfig.java	34	Responsabilidad Única	Separar responsabilidades en clases especializadas.
DatabaseConfig.java	41	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
DatabaseConfig.java	48	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
DatabaseConfig.java	50	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
LibroController.java	20	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
LibroController.java	23	Responsabilidad Única	Separar responsabilidades en clases especializadas.
LibroController.java	24	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
LibroController.java	29	GRASP	Reubicar lógica en

			clases responsables y mejorar cohesión.
LibroController.java	34	Inversión de Dependencias	Usar interfaces e inyección de dependencias.
LibroController.java	35	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
LibroController.java	48	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
LibroController.java	49	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
LibroController.java	60	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
LibroController.java	72	Responsabilidad Única	Separar responsabilidades en clases especializadas.
LibroController.java	80	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
PrestamoController.java	21	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
PrestamoController.java	22	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
PrestamoController.java	48	Responsabilidad Única	Separar responsabilidades en clases especializadas.
PrestamoController.java	53	GRASP	Reubicar lógica en

			clases responsables y mejorar cohesión.
UsuarioController.java	21	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
UsuarioController.java	22	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
UsuarioController.java	26	Inversión de Dependencias	Usar interfaces e inyección de dependencias.
UsuarioController.java	35	Responsabilidad Única	Separar responsabilidades en clases especializadas.
UsuarioController.java	40	Responsabilidad Única	Separar responsabilidades en clases especializadas.
UsuarioController.java	45	Inversión de Dependencias	Usar interfaces e inyección de dependencias.
UsuarioController.java	47	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
LibroDTO.java	13	Responsabilidad Única	Separar responsabilidades en clases especializadas.
LibroDTO.java	21	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
UsuarioDTO.java	13	Responsabilidad Única	Separar responsabilidades en clases especializadas.
Libro.java	27	Responsabilidad Única	Separar responsabilidades

			en clases especializadas.
Libro.java	36	Abierto/Cerrado	Aplicar patrón Strategy o polimorfismo para evitar condicionales.
Libro.java	38	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Libro.java	40	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Libro.java	42	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Libro.java	48	Inversión de Dependencias	Usar interfaces e inyección de dependencias.
Libro.java	51	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Libro.java	56	Segregación de Interfaces	Segregar interfaces para que cada una tenga solo los métodos necesarios.
Prestamo.java	33	Responsabilidad Única	Separar responsabilidades en clases especializadas.
Prestamo.java	40	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Prestamo.java	46	Inversión de Dependencias	Usar interfaces e inyección de dependencias.
Prestamo.java	49	GRASP	Reubicar lógica en

			clases responsables y mejorar cohesión.
Prestamo.java	55	Responsabilidad Única	Separar responsabilidades en clases especializadas.
Prestamo.java	57	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Prestamo.java	64	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Prestamo.java	65	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Prestamo.java	66	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Prestamo.java	69	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Prestamo.java	70	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Prestamo.java	77	Responsabilidad Única	Separar responsabilidades en clases especializadas.
Prestamo.java	78	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Usuario.java	29	Responsabilidad Única	Separar responsabilidades en clases especializadas.

Usuario.java	31	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Usuario.java	43	Abierto/Cerrado	Aplicar patrón Strategy o polimorfismo para evitar condicionales.
Usuario.java	45	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Usuario.java	54	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Usuario.java	62	Responsabilidad Única	Separar responsabilidades en clases especializadas.
Usuario.java	64	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Usuario.java	70	Sustitución de Liskov	Reorganizar jerarquía para que todos los métodos sean aplicables a sus subtipos.
Usuario.java	72	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
Usuario.java	73	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
LibroRepository.java	18	Segregación de Interfaces	Segregar interfaces para que cada una tenga solo los métodos necesarios.

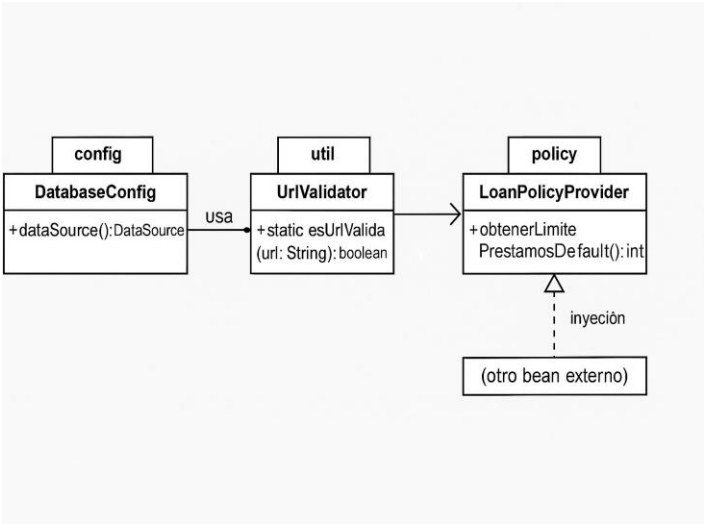
LibroRepository.java	28	Responsabilidad Única	Separar responsabilidades en clases especializadas.
LibroRepository.java	35	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
UsuarioRepository.java	16	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
SecurityConfig.java	20	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	23	Responsabilidad Única	Separar responsabilidades en clases especializadas.
BibliotecaService.java	38	Abierto/Cerrado	Aplicar patrón Strategy o polimorfismo para evitar condicionales.
BibliotecaService.java	39	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	41	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	53	Inversión de Dependencias	Usar interfaces e inyección de dependencias.
BibliotecaService.java	54	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	62	GRASP	Reubicar lógica en clases responsables y mejorar

			cohesión.
BibliotecaService.java	70	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	72	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	75	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	78	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	79	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	82	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	83	Segregación de Interfaces	Segregar interfaces para que cada una tenga solo los métodos necesarios.
BibliotecaService.java	87	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	103	Responsabilidad Única	Separar responsabilidades en clases especializadas.
BibliotecaService.java	104	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	115	GRASP	Reubicar lógica en

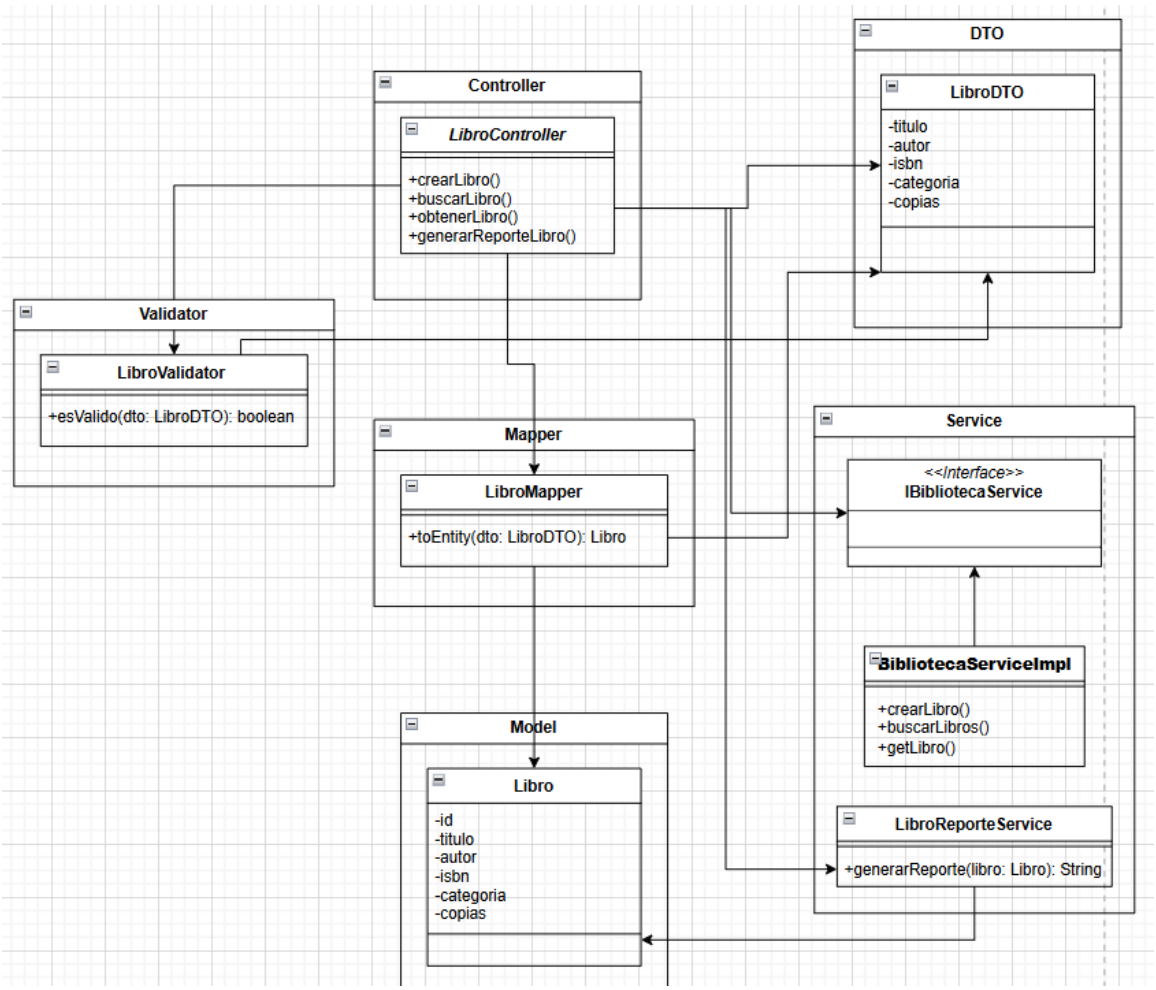
			clases responsables y mejorar cohesión.
BibliotecaService.java	127	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	128	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	129	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	137	Responsabilidad Única	Separar responsabilidades en clases especializadas.
BibliotecaService.java	140	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	142	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	151	Responsabilidad Única	Separar responsabilidades en clases especializadas.
BibliotecaService.java	154	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
BibliotecaService.java	155	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
NotificacionService.java	10	Abierto/Cerrado	Aplicar patrón Strategy o polimorfismo para evitar

			condicionales.
NotificacionService.java	16	Responsabilidad Única	Separar responsabilidades en clases especializadas.
NotificacionService.java	27	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
NotificacionService.java	38	Responsabilidad Única	Separar responsabilidades en clases especializadas.
DateUtils.java	11	Responsabilidad Única	Separar responsabilidades en clases especializadas.
DateUtils.java	22	Abierto/Cerrado	Aplicar patrón Strategy o polimorfismo para evitar condicionales.
DateUtils.java	26	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.
DateUtils.java	38	Responsabilidad Única	Separar responsabilidades en clases especializadas.
DateUtils.java	45	GRASP	Reubicar lógica en clases responsables y mejorar cohesión.

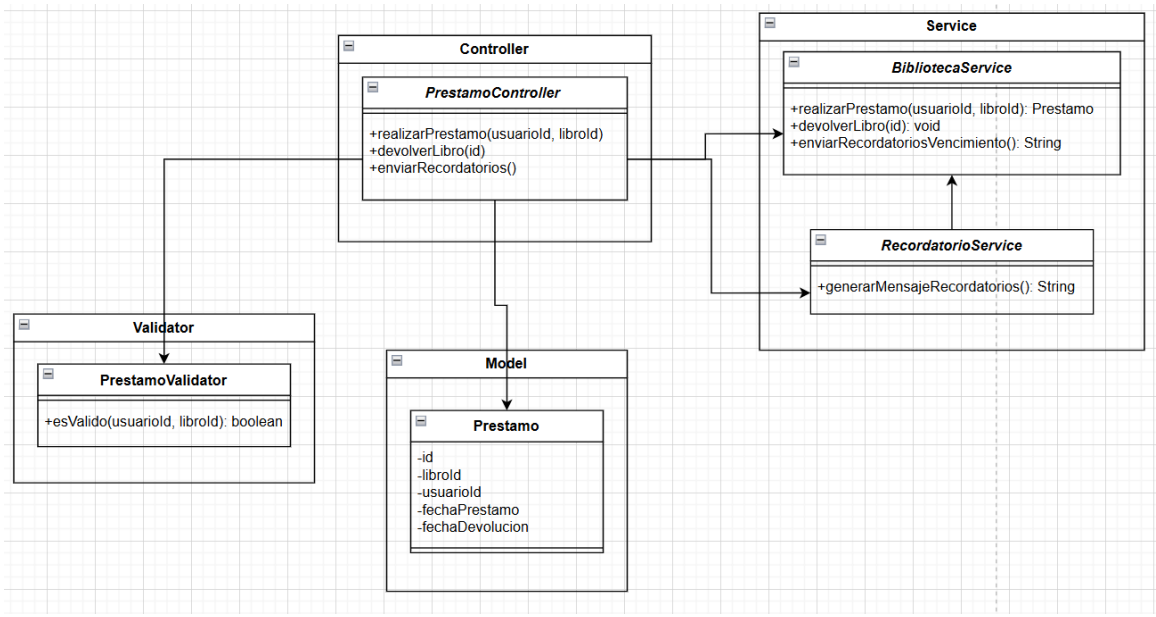
DatabaseConfig.java:



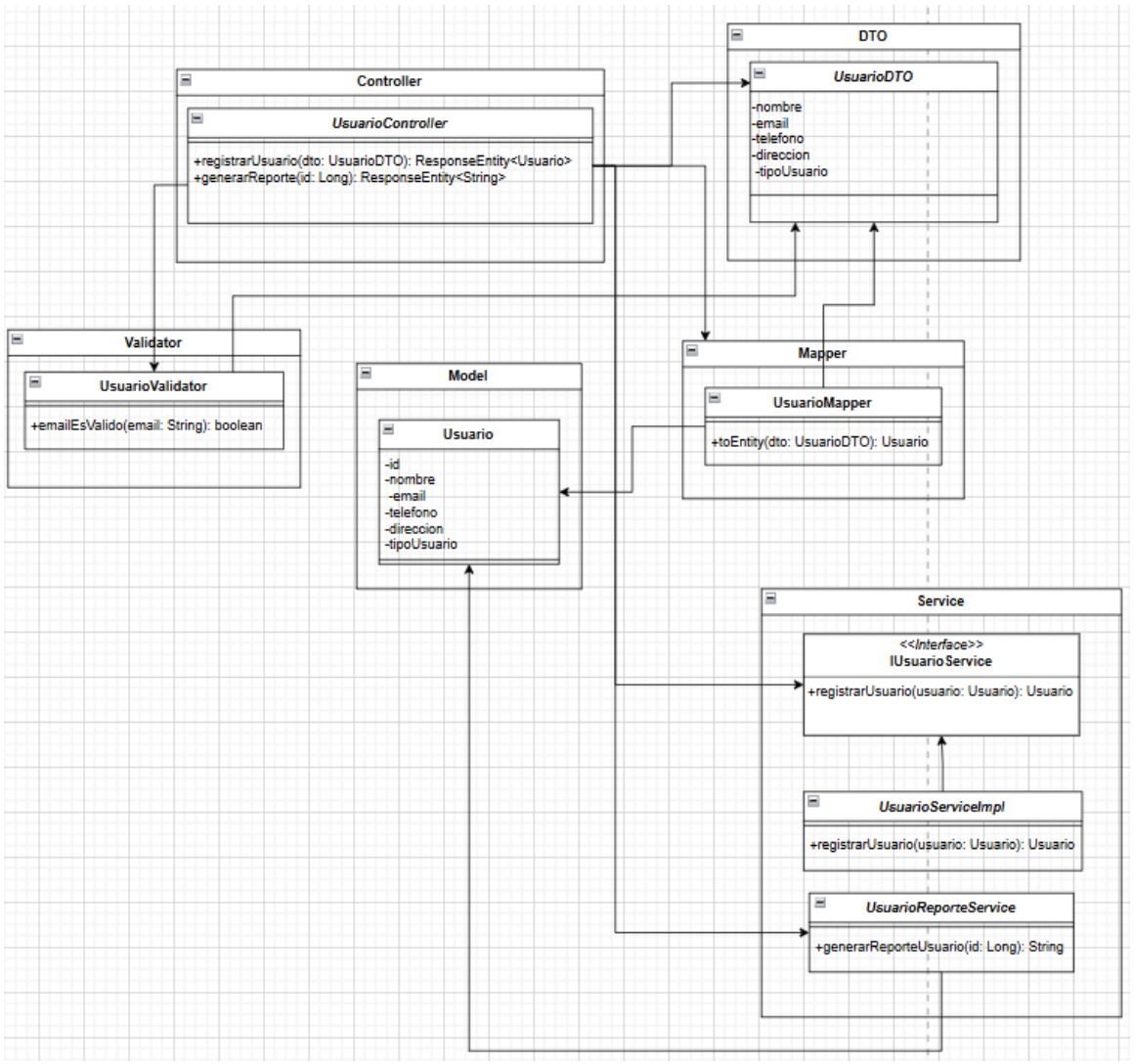
LibroController.java:


DTO
LibroDTO
-titulo
-autor
-isbn
-categoria
-copias
Controller
LibroController
+crearLibro()
+buscarLibro()
+obtenerLibro()
+generarReporteLibro()
Validator
LibroValidator
+esValido(dto: LibroDTO): boolean
Mapper
LibroMapper
+toEntity(dto: LibroDTO): Libro
Service
<<Interface>>
IBibliotecaService
BibliotecaServiceImpl
+crearLibro()
+buscarLibros()
+getLibro()
Model
Libro
-id
-titulo
-autor
-isbn
-categoria
-copias
LibroReporteService
+generarReporte(libro: Libro): String

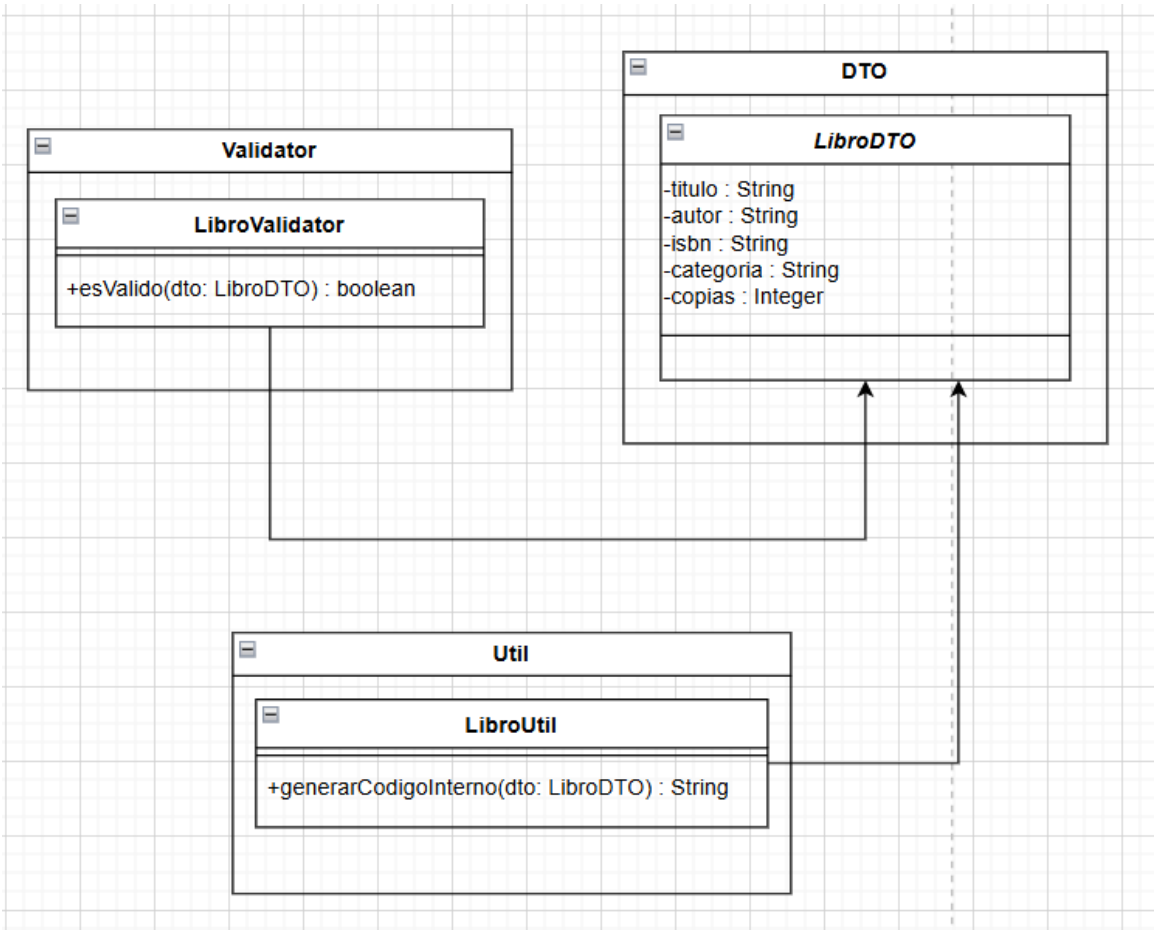
PrestamoController.java



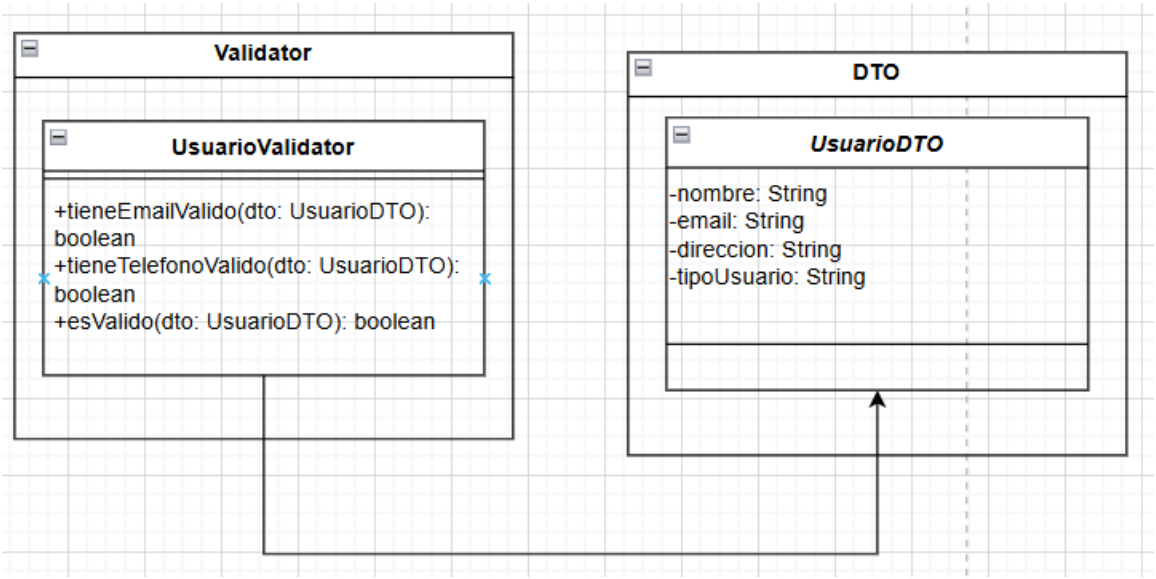
UsuarioController.java:



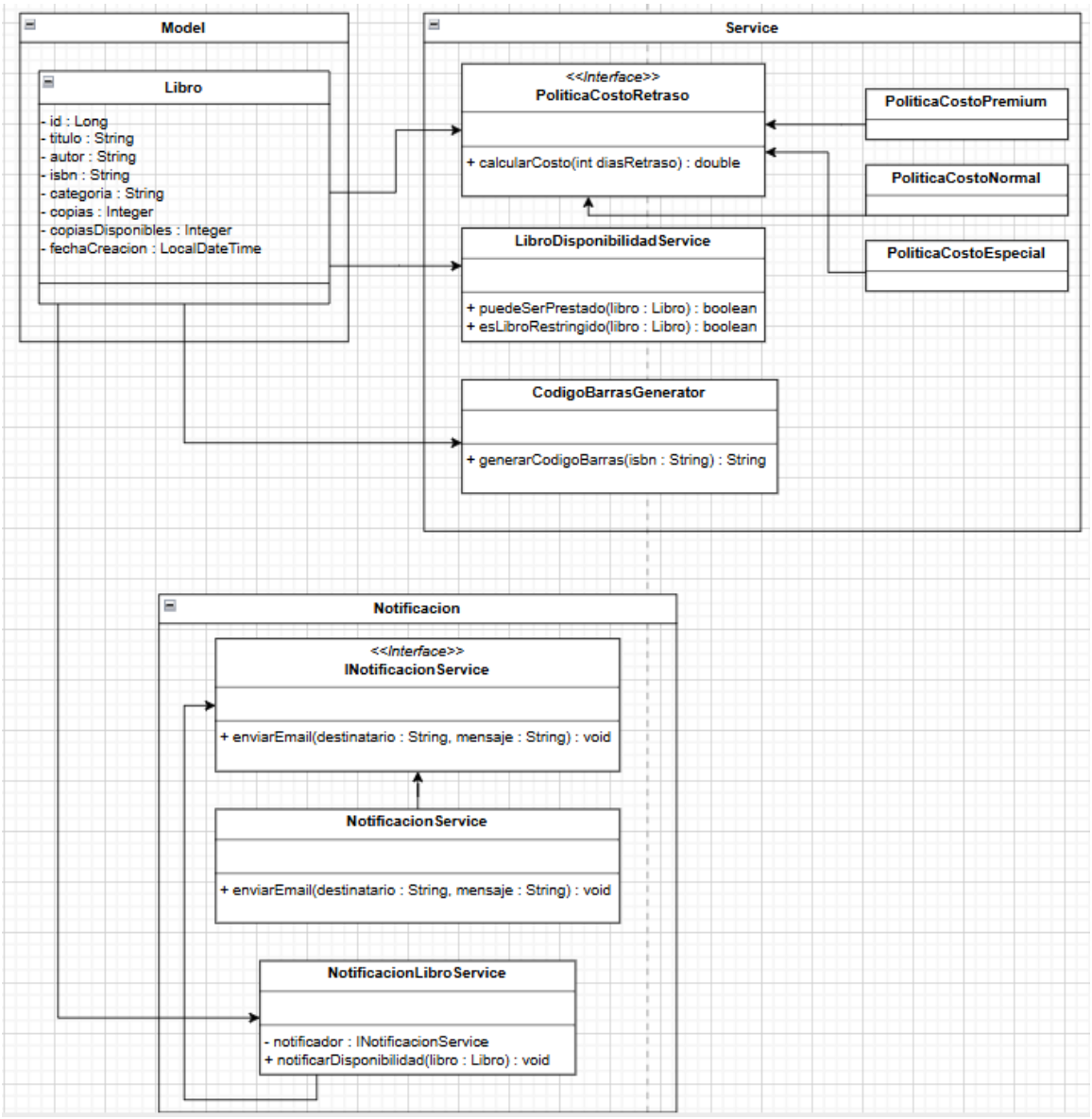
Librodto.java:



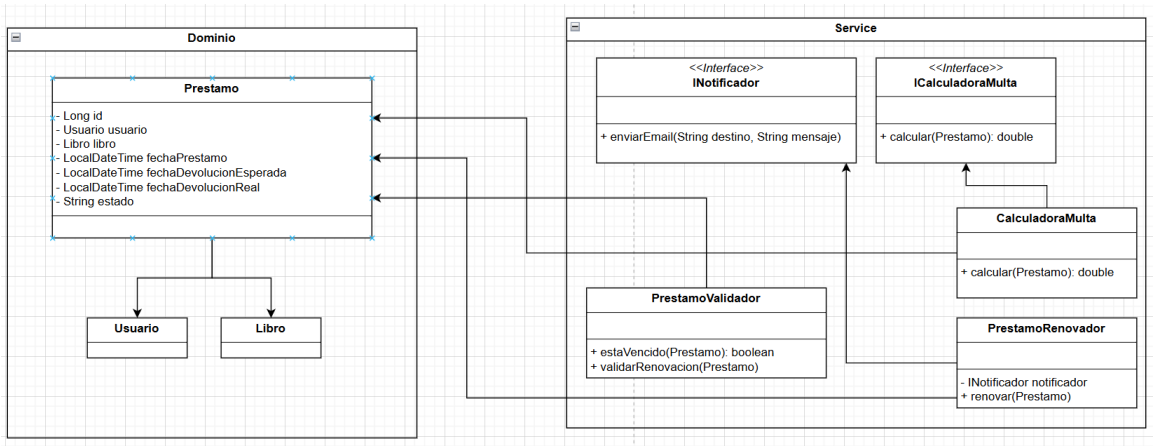
usuarioDTO.java:



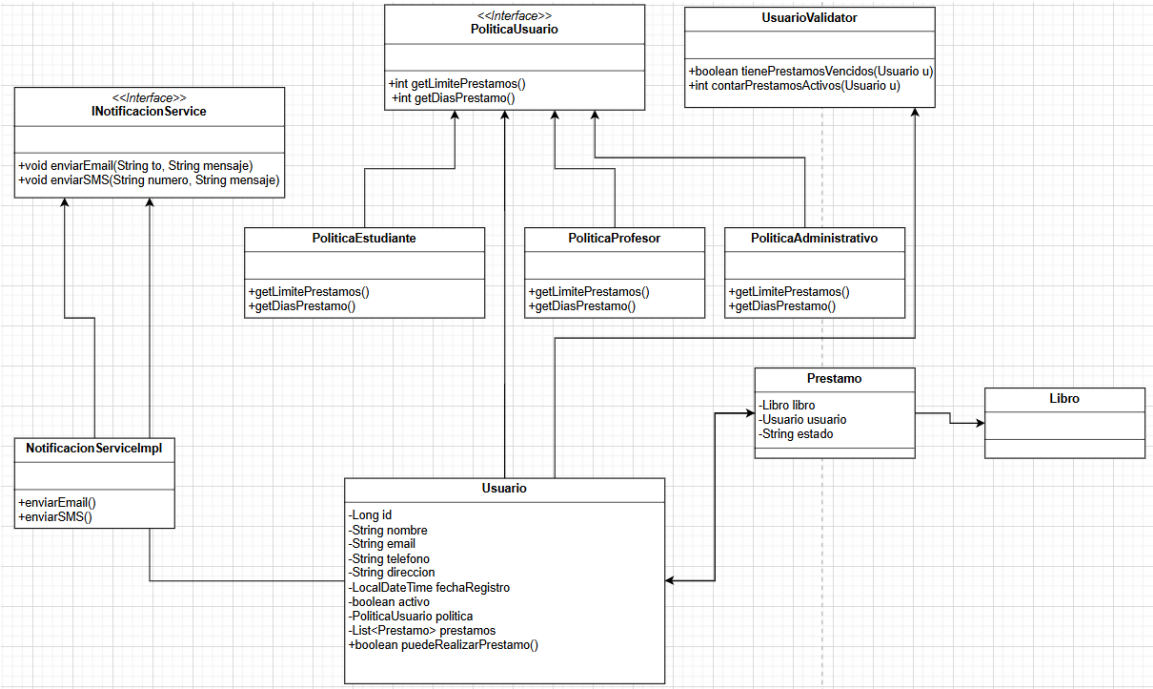
Libro.java:


Model
Libro
- id : Long
- titulo : String
- autor : String
- isbn : String
- categoria : String
- copias : Integer
- copiasDisponibles : Integer
- fechaCreacion : LocalDateTime
Service
<<Interface>>
PoliticaCostoRetraso
+ calcularCosto(int diasRetraso) : double
PoliticaCostoPremium
PoliticaCostoNormal
PoliticaCostoEspecial
LibroDisponibilidadService
+ puedeSerPrestado(libro : Libro) : boolean
+ esLibroRestringido(libro : Libro) : boolean
CodigoBarrasGenerator
+ generarCodigoBarras(isbn : String) : String
Notificacion
<<Interface>>
INotificacionService
+ enviarEmail(destinatario : String, mensaje : String) : void
NotificacionService
+ enviarEmail(destinatario : String, mensaje : String) : void
NotificacionLibroService
- notificador : INotificacionService
+ notificarDisponibilidad(libro : Libro) : void

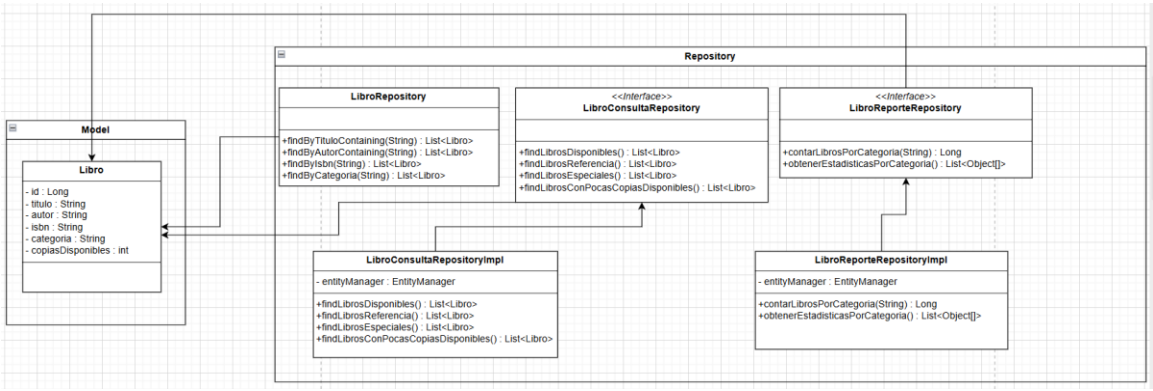
Prestamo.java:



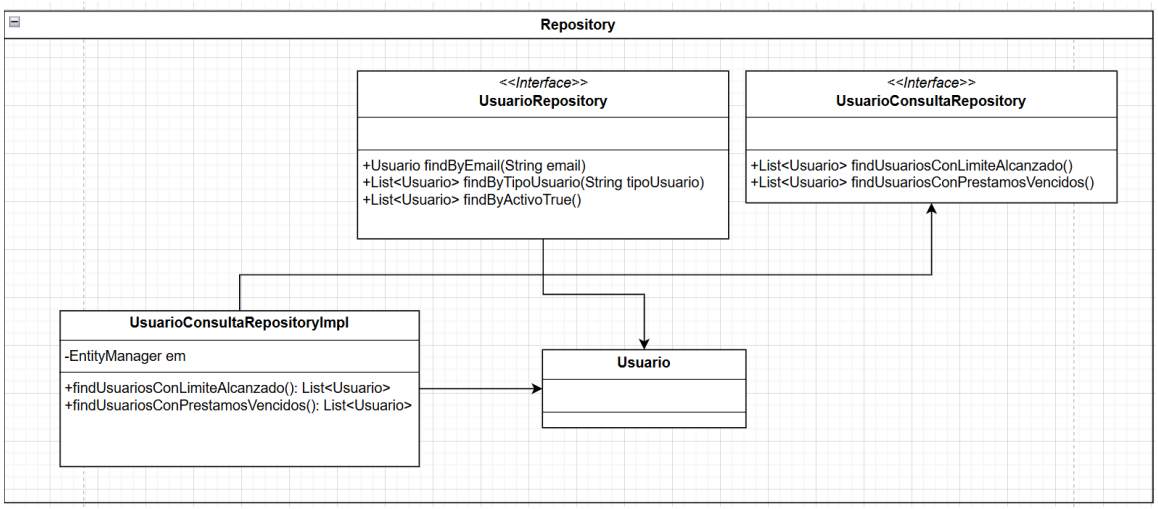
Usuario.java:



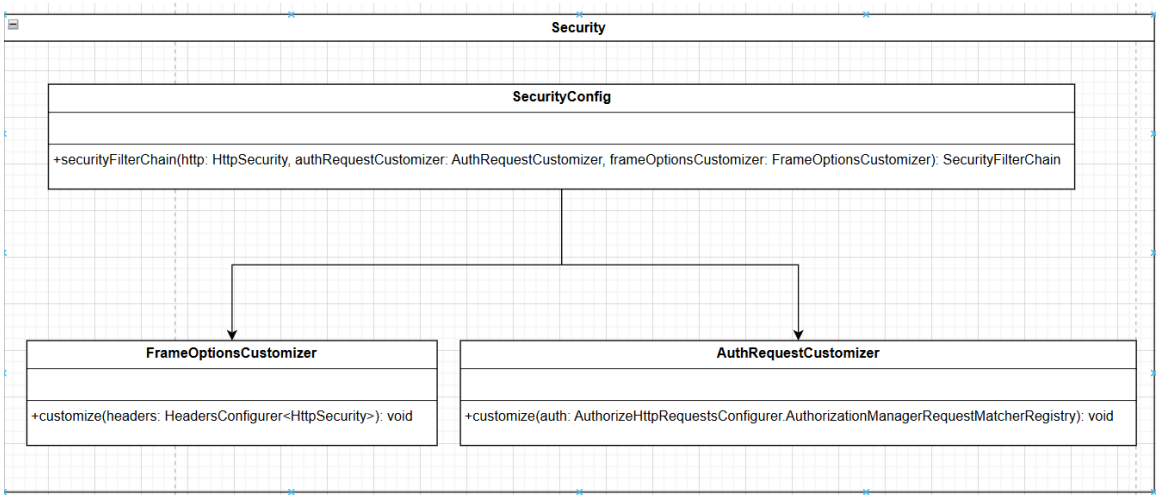
LibroRepository.java:



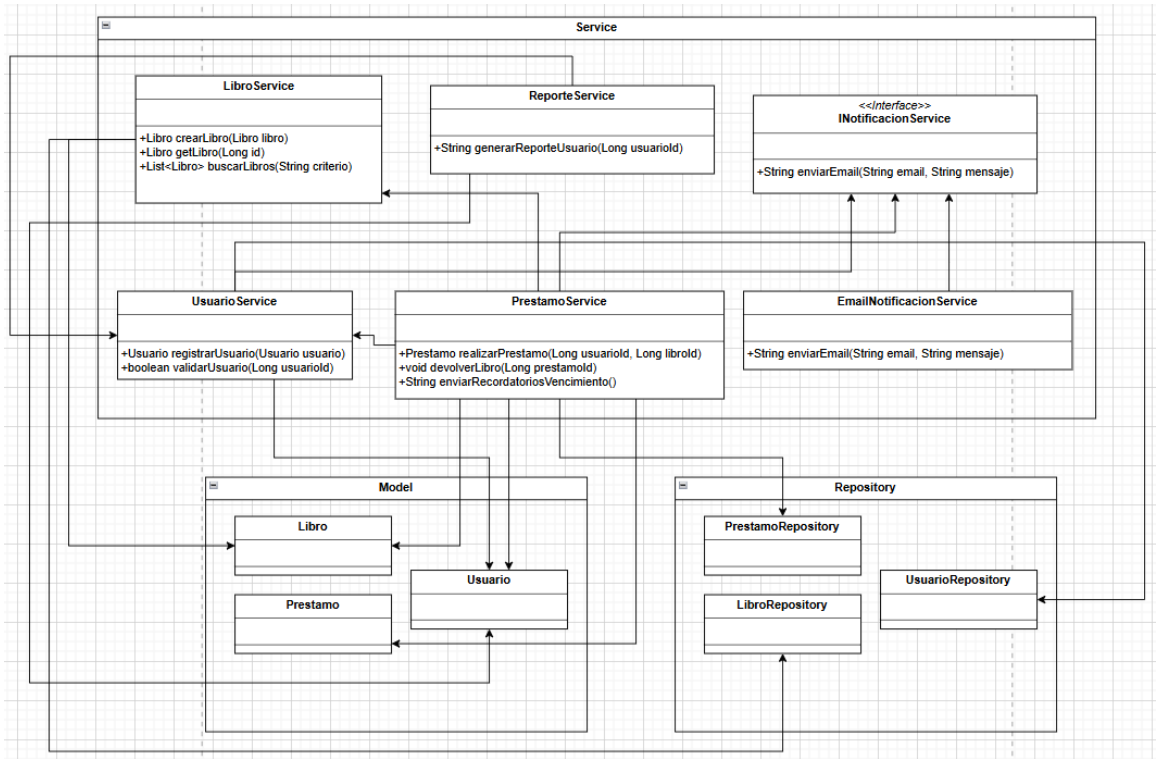
UsuarioRepository.java:



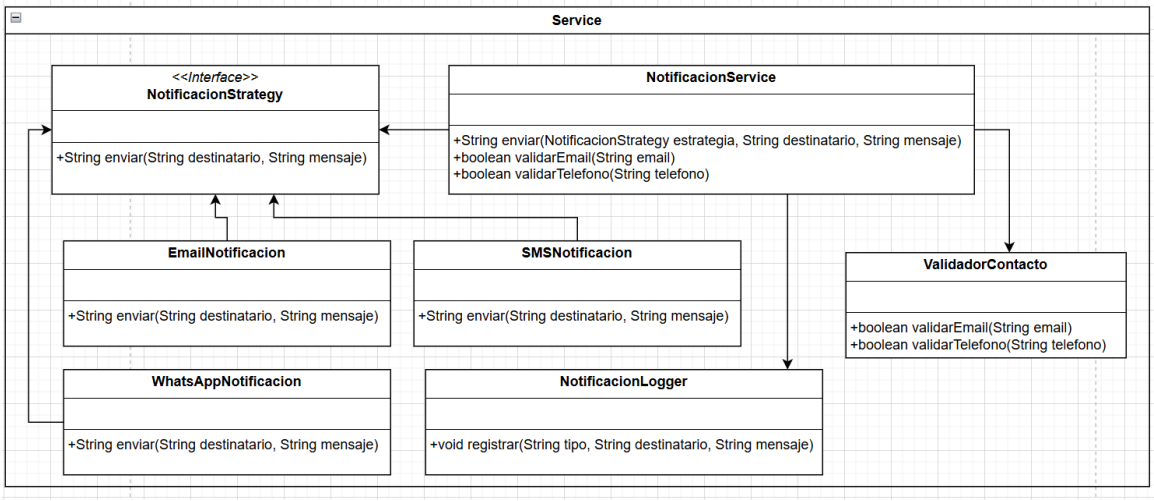
SecurityConfig.java:



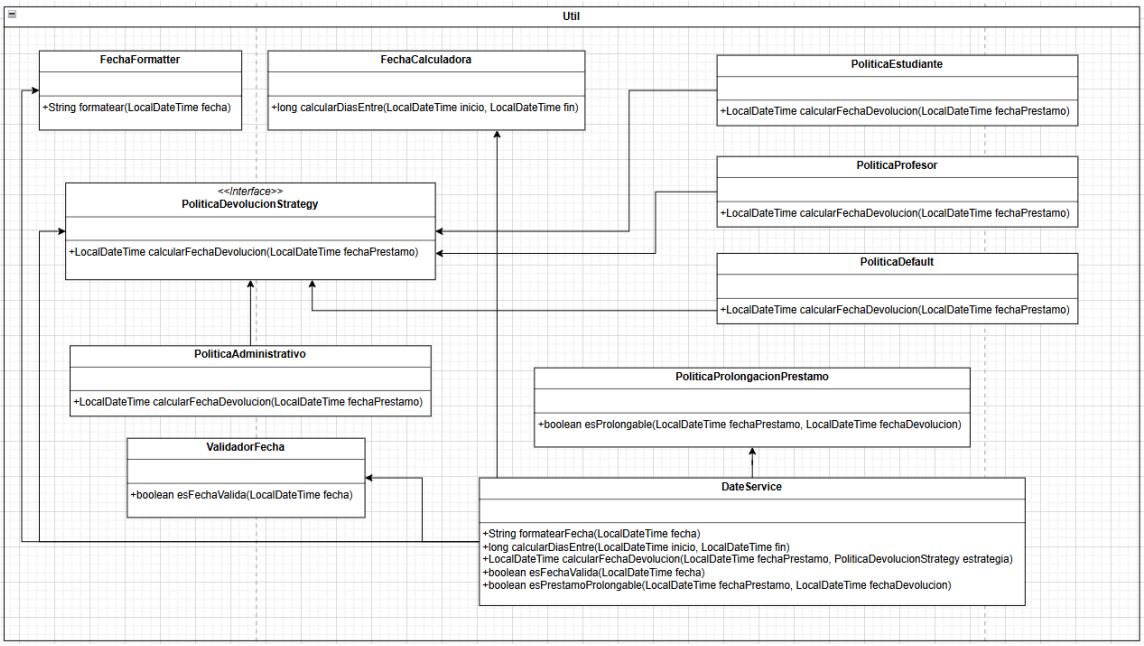
BibliotecaService.java:



NotificacionService.java:



DateUtils.java:



Valor Agregado

Este trabajo representó una experiencia práctica completa que permitió aplicar técnicas reales de análisis y mejora de un sistema existente. A través del estudio del código, la identificación de problemas concretos y la propuesta de soluciones estructuradas, se fortalecieron competencias clave como el trabajo en equipo, la lectura crítica de proyectos desarrollados por otros, y la capacidad de rediseñar sistemas para hacerlos más organizados, claros y sostenibles. Todo el proceso simuló una situación profesional, enfrentando un sistema mal estructurado y transformándolo en una solución más eficiente y fácil de mantener.

Conclusión Final

El análisis realizado evidenció que un diseño deficiente afecta directamente la calidad y evolución de un sistema. A través de la aplicación de principios SOLID y GRASP, se logró proponer una solución técnica más clara, flexible y desacoplada. Este proceso no solo mejoró la arquitectura del sistema, sino que también fortaleció la comprensión práctica del diseño orientado a objetos y su impacto real en el desarrollo de software mantenible.