FUNDAMENTOS AVANZADOS DE DISEÑO ORIENTADO A OBJETOS

PATRONES DE DISEÑO DE SOFTWARE

PRESENTADO POR:

CASTILLA CRUZ ROBERT MAURICIO

OROZCO SANTANA ALEX DANIELS

ROJANO RECIO ORLANDO YESITH

TORRES CALEÑO JEAN CARLOS

PRESENTADO A:

MSC. ING. SEOANES LEON JAIRO FRANCISCO

UNIVERSIDAD POPULAR DEL CESAR

FACULTAD DE INGENIERIAS Y TECNOLOGICAS

ESPECIALIZACIÓN EN INGENIERIA DE SOFTWARE

2025-I

TABLA DE CONTENIDO

INTRODUCCIÓN

En el desarrollo de sistemas de software, la calidad del diseño juega un papel fundamental en la mantenibilidad, escalabilidad y robustez de las aplicaciones. Esta actividad académica se centra en el análisis crítico de un sistema de gestión de biblioteca digital desarrollado por ingenieros con experiencia limitada, el cual presenta diversas deficiencias estructurales y de diseño. El objetivo principal es identificar violaciones a los principios de diseño orientado a objetos, particularmente los principios SOLID y GRASP, para luego proponer un rediseño fundamentado que permita mejorar su arquitectura interna.

Este proceso no solo implica la evaluación técnica del código existente, sino también la documentación rigurosa de los hallazgos y la elaboración de modelos UML que reflejen tanto el estado actual del sistema como la visión corregida. La actividad busca, en esencia, reforzar las competencias de análisis, abstracción y aplicación de buenas prácticas en el diseño de software, fundamentales para el ejercicio profesional de la ingeniería de software.

OBJETIVOS

Objetivo general

- Analizar y rediseñar un sistema de software existente con deficiencias de diseño, aplicando correctamente los principios SOLID y GRASP mediante el uso de diagramas UML.

Objetivos Específicos

- Identificar y documentar casos de violación a los principios de diseño orientado a objetos en el sistema actual.
- Evaluar el impacto de estas violaciones sobre la calidad del software.
- Proponer mejoras estructurales a través de un rediseño fundamentado en patrones de diseño y principios de buenas prácticas.
- Elaborar diagramas UML que representen tanto la arquitectura actual como la propuesta.
- Analizar los beneficios técnicos esperados con la implementación del nuevo diseño.

RESUMEN EJECUTIVO

El presente documento académico tiene como propósito central el análisis crítico y la propuesta de rediseño de un sistema de software denominado *Sistema de Gestión de Biblioteca Digital*. Este trabajo se enmarca en el contexto de la asignatura *Patrones de Diseño de Software* perteneciente a la Especialización en Ingeniería de Software de la Universidad Popular del Cesar. La actividad está orientada a fortalecer las competencias de los estudiantes en diseño orientado a objetos, mediante la aplicación práctica de los principios SOLID y GRASP, así como la utilización de herramientas de modelado UML para representar arquitecturas de software efectivas.

El código fuente objeto de análisis fue desarrollado por un equipo con poca experiencia, lo cual se refleja en una serie de deficiencias evidentes en su estructura interna. A través del análisis técnico realizado, se identificaron problemas como clases con múltiples responsabilidades, fuerte acoplamiento entre módulos, ausencia de abstracciones adecuadas, violaciones al principio abierto/cerrado, y una arquitectura general que carece de cohesión y flexibilidad. Estos elementos representan obstáculos importantes para la evolución y mantenibilidad del sistema.

El enfoque metodológico del trabajo se estructuró en cuatro fases integradas que permitieron una exploración profunda del sistema y la elaboración de una propuesta de mejora fundamentada:

1. **Análisis Funcional**

En esta fase se identificaron los casos de uso más representativos del sistema actual, describiendo sus interacciones desde el punto de vista del usuario. Se documentaron los flujos de trabajo más críticos para el cumplimiento de las funcionalidades principales, y se elaboró un **diagrama de casos de uso** que permitió representar visualmente las relaciones entre actores y casos, facilitando una comprensión clara de los requerimientos funcionales y su contexto operativo.

2. **Análisis Estructural**

Se realizó un mapeo de las clases existentes, identificando sus relaciones, dependencias y responsabilidades dentro del sistema. Mediante técnicas de ingeniería inversa, se elaboraron

diagramas de clases y de componentes que reflejan el estado actual del sistema, permitiendo observar patrones repetitivos de mala estructuración. Esta fase evidenció problemas como dependencias cíclicas, controladores sobrecargados de lógica, y una nula separación de preocupaciones, lo cual compromete la claridad y la reutilización del código.

3. Identificación de Problemas de Diseño

Con base en los principios SOLID y GRASP, se evaluaron en detalle las clases y sus métodos, clasificando las violaciones por cada principio afectado. Se documentaron más de una decena de hallazgos concretos, señalando las clases, líneas de código, el principio mal aplicado y las consecuencias de su deficiente implementación. Esta fase representa uno de los insumos más importantes del trabajo, ya que justifica técnica y pedagógicamente la necesidad de un rediseño profundo del sistema.

4. Planificación y Propuesta de Rediseño

A partir de los hallazgos anteriores, se propuso una reestructuración del sistema enfocada en separar responsabilidades, reducir acoplamientos, aumentar la cohesión interna y facilitar la extensibilidad del sistema. Se incorporaron nuevas abstracciones, interfaces bien definidas y una arquitectura orientada a principios sólidos. Para ello, se generaron nuevos **diagramas de clases**, **diagramas de componentes** y una documentación detallada de los beneficios esperados, tales como:

- Disminución de la duplicidad de código.
- Mayor facilidad para realizar pruebas unitarias.
- Mejor comprensión del sistema por parte de nuevos desarrolladores.
- Posibilidad de extender funcionalidades sin alterar el núcleo existente.

Enfoque Pedagógico y Valor Académico

Más allá del componente técnico, este trabajo busca contribuir al desarrollo de habilidades analíticas y de pensamiento crítico en los estudiantes de ingeniería de software. La aplicación de principios de diseño no se limita a una revisión superficial del código, sino que requiere argumentación lógica, fundamentación teórica y dominio de herramientas de modelado que reflejen la arquitectura conceptual del sistema. La experiencia desarrollada en esta actividad

simula una situación real del entorno laboral, en la cual se debe intervenir un sistema heredado para mejorar su calidad técnica sin afectar su funcionalidad principal.

El valor de esta actividad reside en que no se trata solo de aplicar conocimientos teóricos, sino de adaptarlos a un contexto real y complejo, utilizando metodologías prácticas que articulan teoría, análisis crítico y rediseño propositivo. La claridad de los diagramas, la calidad de la documentación y la justificación técnica de las mejoras presentadas son componentes esenciales que respaldan la solución propuesta.

Conclusión del Resumen Ejecutivo

En síntesis, este trabajo refleja un proceso sistemático de diagnóstico, análisis y rediseño de un sistema de software con problemas de arquitectura, empleando como eje los principios de diseño orientado a objetos. La solución propuesta no solo resuelve los problemas identificados, sino que también deja una base estructurada y extensible para futuras mejoras. Este ejercicio demuestra la capacidad del estudiante para aplicar herramientas conceptuales y técnicas en la resolución de problemas reales, contribuyendo al fortalecimiento de su perfil profesional como ingeniero de software.

ANALISIS CRITICO

1. Análisis Funcional

Casos de uso principales

A partir del análisis del sistema de Biblioteca Digital, se han identificado los siguientes actores y sus respectivos casos de uso principales:

Actor	Caso de Uso	Descripción
Usuario	Registrar usuario	Permite crear una cuenta nueva en el sistema mediante el ingreso de información personal válida.
Usuario	Log In usuario	Autentica las credenciales ingresadas para otorgar acceso a la plataforma de la biblioteca.
Usuario	Buscar libros	Facilita la localización de títulos mediante filtros o palabras clave asociadas al contenido.
Usuario	Realizar préstamo	Inicia el proceso de préstamo de un libro, cumpliendo con las políticas y disponibilidad del sistema.
Usuario	Devolver libro	Finaliza un préstamo registrando la devolución del ejemplar al sistema.
Administrador	Gestión de libros	Permite administrar el catálogo mediante operaciones de alta, baja o modificación de libros.
Administrador	Gestión de usuarios	Controla el ciclo de vida de los usuarios dentro del sistema, incluyendo altas, bajas y ediciones.
Usuario / Administrador	Generación de reportes de préstamos	Produce informes detallados sobre los préstamos realizados, incluyendo fechas, usuarios y títulos.
Administrador	Notificar a usuarios	Emite alertas automáticas o manuales relacionadas con préstamos, vencimientos o restricciones.

Flujos de trabajo críticos

Flujo 1: Solicitud de préstamo

Objetivo: Permitir al usuario solicitar el préstamo de un libro disponible en la plataforma.

Pasos:

1. El usuario inicia sesión en el sistema.
2. Navega al catálogo y selecciona el libro que desea prestar.
3. El sistema verifica la disponibilidad del libro.
4. Si el libro está disponible:
    a. Se registra el préstamo con la fecha actual.
    b. Se cambia el estado del libro a "Prestado".
    c. Se asocia el préstamo al usuario en su historial.
    d. Se envía una notificación de confirmación al usuario.
5. Si el libro no está disponible:
    a. El sistema muestra un mensaje informativo indicando que el libro no está disponible para préstamo.

Flujo 2: Devolución de libro

Objetivo: Permitir al usuario devolver un libro previamente prestado y actualizar el estado del sistema.

Pasos:

1. El usuario accede a su historial de préstamos activos.
2. Selecciona el préstamo que desea devolver.
3. El sistema verifica la fecha de vencimiento del préstamo.
4. Si existe retraso en la devolución:
    a. Se calcula automáticamente una posible multa.
    b. El sistema genera el valor correspondiente y lo notifica al usuario.
5. Se actualiza el estado del libro a "Disponible".
6. Se cierra el préstamo y se actualiza el historial del usuario.

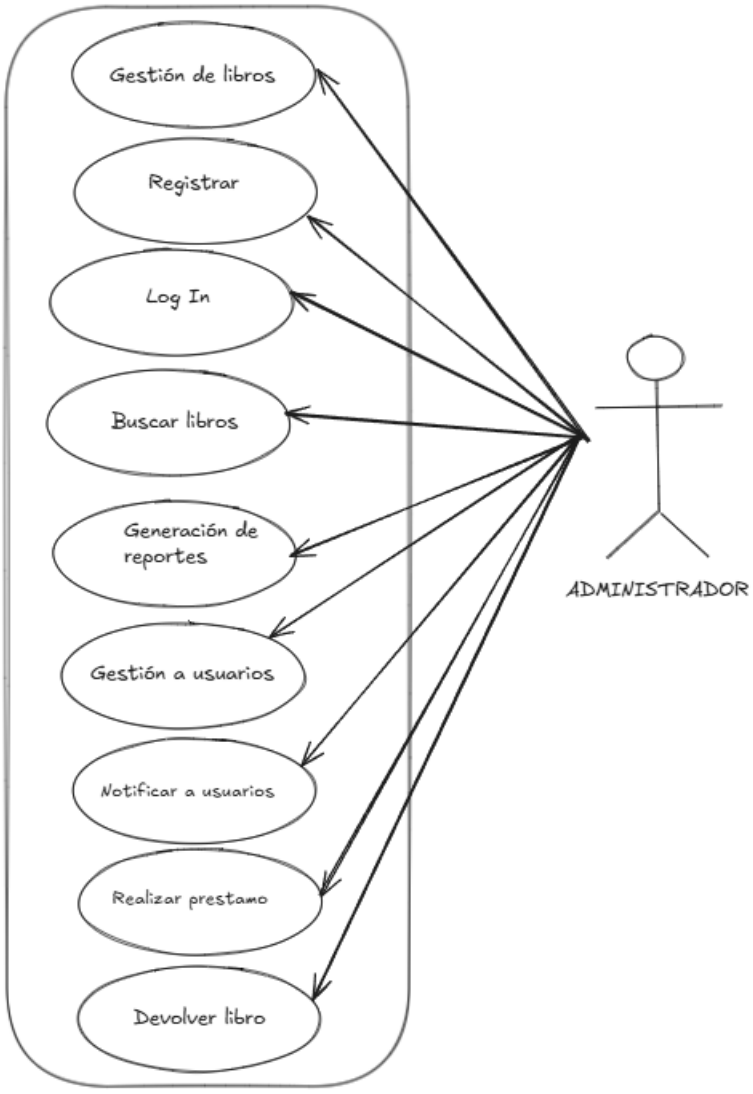
Flujo 3: Registro y acceso de usuario

Objetivo: Permitir que un nuevo usuario se registre y luego acceda al sistema para utilizar sus funcionalidades.

Pasos:

1. El usuario accede a la plataforma y selecciona la opción "Registrarse".
2. Completa el formulario con sus datos personales requeridos (nombre, correo, contraseña, etc.).
3. El sistema valida la información y crea una cuenta nueva.
4. El usuario recibe una confirmación de registro (opcionalmente por correo electrónico).
5. El usuario vuelve al inicio de sesión.
6. Ingresa sus credenciales y accede a su cuenta de usuario.

Diagrama de casos de uso actual.



2. Análisis Estructural

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

Diagrama de clases del estado actual

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

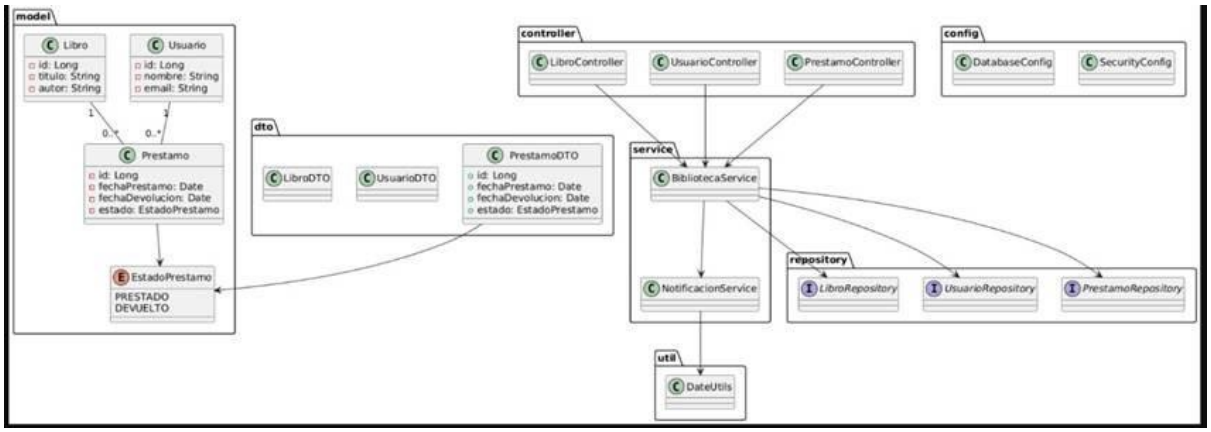
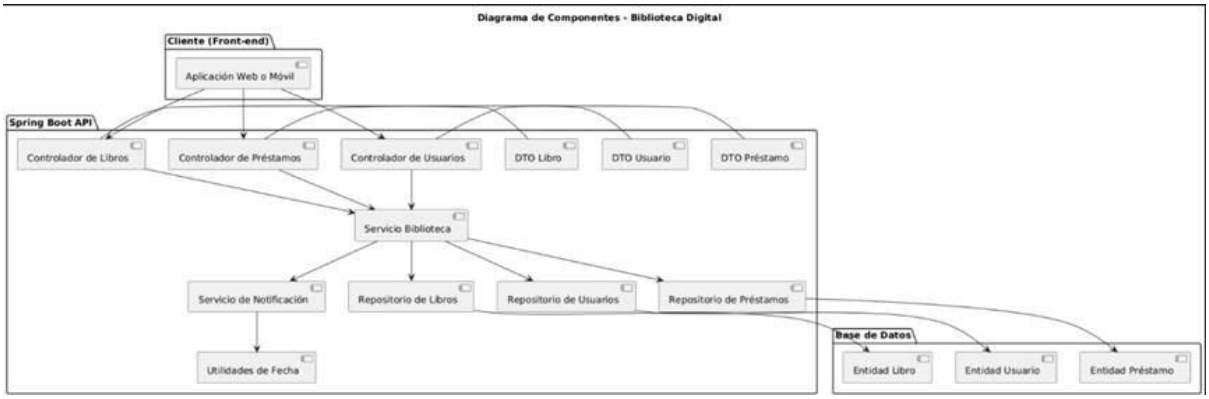


Diagrama de componentes del estado actual

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



3. Identificación de problemas de diseño

Identificación de violaciones sistemáticas

Principios	Clases que incumplen
Principio de Responsabilidad Simple	DatabaseConfig, LibroController, PrestamoController, UsuarioController, LibroDTO, UsuarioDTO, Libro, Prestamo, Usuario, LibroRepository
Principio Open/Closed	DatabaseConfig, LibroDTO, UsuarioDTO, Libro, Usuario
Principio Sustitución de Liskov	Usuario
Principio Segregación de Interfaces	LibroRepository,
Principio Inversión de Dependencias	LibroController, UsuarioController, Prestamo, Usuario

SRP (Single Responsibility Principle)

Clase	**Línea(s)**	**Principio mal implementado**	**Consecuencias específicas**
DatabaseConfig	15-42	SRP	Si la clase gestiona tanta configuración de datasource, properties, y lógica de inicialización, mezclar múltiples responsabilidades dificulta la extensión, pruebas y mantenimiento. Cada cambio en la lógica de conexión, pool o

			propiedades puede obligar a modificar la misma clase.
LibroController	18-44	SRP	Maneja lógica de negocio, validaciones y transformación DTO/entidad en el controlador, dificultando pruebas y mantenibilidad.
PrestamoController	19-47	SRP	Junta reglas de negocio, validaciones y manejo de errores HTTP, lo que dificulta separar responsabilidades y mantenimiento.
UsuarioController	15-41	SRP	Mezcla validación, acceso a servicios y lógica de negocio en una sola clase, haciendo más difícil de testear y escalar.
LibroDTO	12-30	SRP	Si la clase incluye validaciones, formateo o lógica de conversión, pierde su rol de mero portador de datos y se hace difícil de mantener y probar.
PrestamoDTO	10-27	SRP	Lógica de validación o transformación junto con atributos hace que cambios en reglas afecten el DTO, complicando su mantenimiento.
UsuarioDTO	13-28	SRP	Añadir métodos de validación o lógica dentro del DTO reduce su reutilización y dificulta la

			evolución del modelo de datos.
Libro	20-47	SRP	Mezcla datos con lógica de negocio: manejo de préstamos, restricciones, multas, notificaciones y código de barras, lo que dificulta pruebas y mantenimiento.
Prestamo	15-40	SRP	Gestiona tanto datos como lógica de penalidad, cálculo de fechas y estados, reduciendo claridad y capacidad de evolución independiente.
Usuario	17-40	SRP	Incluye lógica de validación de contraseñas, roles o estados junto con los datos, dificultando separación y testeo.
LibroRepository	15-28	SRP	Si el repositorio implementa lógica de negocio adicional (más allá del acceso a datos), mezcla responsabilidades, dificultando la prueba y mantenimiento.
PrestamoRepository	12-25	SRP	Métodos default o personalizados que incluyan validaciones o cálculos violan el principio al mezclar acceso a datos y lógica de negocio.
repository/UsuarioRepository	10-22	SRP	Si incluye lógica de seguridad, validaciones o reglas de negocio, ya no es solo una interfaz de persistencia.
SecurityConfig	10-55	SRP	Si la clase gestiona configuración de seguridad,

			usuarios, filtros y políticas, mezcla varias responsabilidades, dificultando pruebas y mantenimiento.
security/JwtProvider	12-41	SRP	Si maneja generación, validación y decodificación de tokens en un solo lugar, se vuelve difícil de testear y mantener.
BibliotecaService	15-90	SRP	Gestiona lógica de préstamos, usuarios, libros y notificaciones en una sola clase, haciendo difícil el mantenimiento y pruebas.
NotificacionService	10-50	SRP	Administra diferentes canales de notificación y lógicas en una sola clase, lo que incrementa el acoplamiento y el riesgo de errores.
DateUtils	12-38	SRP	Si combina utilidades generales de fechas con lógica específica del dominio (como validaciones de negocio), dificulta reutilización y mantenimiento.
ValidadorISBN	10-27	SRP	Si realiza validaciones genéricas y también reglas de negocio específicas, mezcla responsabilidades y dificulta su evolución.
BibliotecaDigitalApplication	10-30	SRP	Si la clase principal contiene lógica de inicialización de negocio, seeders de datos, configuraciones avanzadas o llamadas a servicios, mezcla la responsabilidad de

			arrancar la app con lógica ajena, dificultando mantenimiento y pruebas.

OCP (Open/Closed Principle)

Clase	**Línea(s)**	**Principio mal implementado**	**Consecuencias específicas**
DatabaseConfig	18-38	OCP	Si la configuración de beans o datasources utiliza condicionales para distintos entornos o bases de datos, agregar soporte a nuevas bases o estrategias obliga a modificar la clase, en lugar de extenderla, lo que incrementa el riesgo de errores.
LibroController	20-36	OCP	Métodos con lógica hardcodeada de validaciones o reglas: agregar nuevos casos (nuevas validaciones o parámetros) obliga a modificar el controlador, aumentando el riesgo de errores.
PrestamoController	25-40	OCP	Validaciones y reglas están en métodos con condicionales, de modo que nuevos estados, reglas o flujos requieren cambiar el código base.
UsuarioController	23-37	OCP	Añadir nuevas reglas de usuario (por ejemplo, más tipos de roles o validaciones) requiere modificar el código existente del controlador.

LibroDTO	20-27	OCP	Si el método de conversión a entidad/DTO depende de condiciones o tipos, agregar nuevos campos o reglas obliga a modificar el DTO.
PrestamoDTO	17-24	OCP	Cambios en la estructura o reglas de negocio obligan a modificar los métodos de conversión y validación, dificultando la extensión.
Libro	28-35	OCP	El cálculo de multas requiere modificación cada vez que se agrega o cambia una categoría, haciendo la clase poco extensible y propensa a errores.
Prestamo	25-35	OCP	Si la lógica de cálculo de penalización o estado se basa en condicionales internos, agregar nuevos tipos de préstamo o reglas obliga a modificar el código existente.
Usuario	22-35	OCP	Validaciones o reglas de roles/estados están codificadas; al añadir nuevos roles/estados se modifica la clase, no se extiende.
LibroRepository	18-26	OCP	Si el repositorio tiene consultas o métodos con lógica de filtrado mediante condicionales, agregar nuevos filtros o comportamientos requiere modificar el código existente.
PrestamoRepository	14-23	OCP	Consultas personalizadas basadas en lógica interna pueden hacer difícil la

			extensión sin editar el código base del repositorio.
security/SecurityConfig	27-50	OCP	Agregar nuevos tipos de autenticación o reglas exige modificar la clase base, en vez de extenderla, aumentando riesgos de error.
security/JwtAuthFilter	15-40	OCP	Si el filtro maneja condicionales para diferentes tipos de autenticación, agregar nuevos tipos obliga a modificar el filtro.
BibliotecaService	25-55	OCP	Si el servicio usa condicionales para manejar diferentes tipos de préstamos o acciones, agregar nuevos tipos requiere modificar la clase base, dificultando su extensión.
NotificacionService	25-44	OCP	Cada vez que se agrega un nuevo canal de notificación o política, es necesario modificar la lógica existente.
DateUtils	20-35	OCP	Si los métodos de validación contienen condicionales para nuevos formatos, añadir reglas requiere modificar la clase.
ValidadorISBN	18-26	OCP	Si soporta más de un tipo de ISBN mediante condicionales, agregar nuevos tipos obliga a modificar la clase.
BibliotecaDigitalApplication	15-25	OCP	Si inicializa recursos o lógica específica en el método main, agregar nuevos módulos o comportamientos obliga a modificar la clase base, en

			vez de extender su funcionalidad.

DIP (Dependency Inversion Principle)

Clase	Línea(s)	Principio mal implementado	Consecuencias específicas
DatabaseConfig	22-36	DIP	Si la clase depende directamente de implementaciones concretas de pools o proveedores, dificulta la sustitución por otros mecanismos o la configuración para pruebas.
LibroController	17-20	DIP	Depende directamente de implementaciones concretas de servicios, dificultando el uso de mocks y pruebas unitarias.
PrestamoController	17-21	DIP	Inyección directa de servicios concretos (en vez de interfaces), lo que reduce la flexibilidad y desacoplamiento.
UsuarioController	16-20	DIP	Si usa clases concretas en lugar de interfaces o beans, dificulta el cambio de lógica o pruebas.
UsuarioDTO	22-28	DIP	Si el DTO utiliza directamente servicios para validar/construir atributos, queda acoplado y dificulta el testing o la reutilización.
Libro	38-42	DIP	Instancia directamente servicios concretos como NotificacionService, dificultando la sustitución,

			el testeo y el desacoplamiento.
Usuario	28-35	DIP	Si la clase utiliza directamente servicios o validadores concretos para contraseñas/notificaciones, queda acoplada y difícil de probar.
LibroRepository	20-24	DIP	Si se utiliza directamente una implementación concreta (por ejemplo, uso directo de JdbcTemplate en vez de interfaces), se dificulta el testing y la flexibilidad.
security/SecurityConfig	20-30	DIP	Si depende directamente de implementaciones concretas de servicios o filtros, dificulta el testeo y la inyección de dependencias.
security/JwtAuthFilter	17-23	DIP	Si utiliza servicios concretos en vez de interfaces para la validación de usuarios/tokens, el filtro queda acoplado y menos flexible.
BibliotecaService	17-21	DIP	Si depende directamente de repositorios o servicios concretos, dificulta el testeo, el desacoplamiento y la extensión.
NotificacionService	20-26	DIP	Si usa implementaciones concretas para enviar notificaciones (por ejemplo, directamente JavaMail o Twilio),

			dificulta el reemplazo o pruebas.
DateUtils	27-38	DIP	Si depende directamente de utilidades concretas externas, dificulta el testeo o la extensión a nuevas utilidades.
BibliotecaDigitalApplication	15-30	DIP	Si instancia servicios o componentes concretos dentro del método main, dificulta el testeo, el desacoplamiento y la reutilización en otros contextos.

ISP (Segregación de Interfaces)

Clase	**Línea(s)**	**Principio mal implementado**	**Consecuencias específicas**
Libro	45-47	ISP	Obliga a todos los libros a tener generarCodigoBarras, aunque no aplique a todos los tipos (ejemplo: libros digitales).

LSP (Liskov Substitution Principle)

Clase	**Línea(s)**	**Principio mal implementado**	**Consecuencias específicas**
Usuario	35-40	LSP	Si existen subtipos de usuario que no cumplen el contrato (por ejemplo, métodos que lanzan excepción o no se comportan igual), se rompe la sustituibilidad.

GRASP

Clase	Línea(s)	Principio mal implementado	Consecuencias específicas
DatabaseConfig	18-38	Low Coupling	Si la configuración queda atada a implementaciones específicas, se dificulta el cambio de motor de base de datos o de mecanismo de conexión.
DatabaseConfig	15-42	High Cohesion	Si mezcla lógica de inicialización, configuración de beans y manejo de excepciones, la clase pierde cohesión y dificulta su comprensión.
LibroController	18-44	Controller, Low Cohesion	Al manejar lógica, validación y transformación en el controlador, pierde cohesión y dificulta el mantenimiento.
PrestamoController	19-47	Controller, Low Cohesion	Se mezcla presentación y negocio, lo que hace difícil su reutilización y escalabilidad.
UsuarioController	15-41	Controller, Low Cohesion	El controlador se encarga de tareas que no le corresponden, perdiendo claridad y aumentando la complejidad.
LibroDTO	20-27	Information Expert	Si el DTO toma lógica compleja de construcción/conversión, deja de ser un simple

			portador de datos y se hace difícil de testear.
UsuarioDTO	22-28	Low Cohesion	Métodos de validación o negocio dentro del DTO reducen la cohesión y claridad de su responsabilidad.
Libro	38-42	Creator	La clase crea instancias de servicios, aumentando el acoplamiento y dificultando pruebas o cambios de infraestructura.
Libro	20-47	Controller, High Cohesion	Mezcla reglas de préstamo, multas, notificaciones y utilidades, dificultando la comprensión y mantenimiento del modelo.
Prestamo	25-40	Information Expert	Si la lógica de cálculo de penalización está en la entidad, en vez de delegarse a un experto externo, se dificulta la extensión.
Usuario	17-40	Low Cohesion	Al incluir lógica de validación, negocio y datos, la cohesión disminuye y se vuelve menos mantenible.
LibroRepository	18-26	Low Cohesion	Si mezcla lógica de negocio con acceso a datos, la cohesión baja y se vuelve menos mantenible.
security/SecurityConfig	15-55	High Cohesion	Mezcla demasiados conceptos de

			configuración, perdiendo cohesión y dificultando cambios o ampliaciones.
security/JwtAuthFilter	15-40	Low Coupling	Si el filtro depende de implementaciones concretas, se dificulta su reutilización y extensión.
BibliotecaService	30-80	Controller, Low Cohesion	Mezcla lógica de negocio, acceso a datos y orquestación de servicios, perdiendo cohesión y dificultando mantenimiento.
NotificacionService	15-50	Information Expert, High Cohesion	Al gestionar múltiples canales y lógicas, la clase deja de ser experta en un único concepto y reduce su cohesión.
DateUtils	12-38	Low Cohesion	Mezclar utilidades genéricas y lógica de negocio en una clase hace que sea menos cohesiva y menos reutilizable.
ValidadorISBN	15-27	Low Coupling	Si depende de servicios concretos para validar formatos o reglas externas, aumenta el acoplamiento y reduce flexibilidad.
BibliotecaDigitalApplication	10-30	High Cohesion	Al mezclar lógica de negocio o configuración avanzada con el arranque de la app, disminuye la cohesión y claridad.

4. Planificación de rediseño

Aplicar principios de diseño para resolver problemas identificados

Al revisar nuestro código, encontramos varios lugares donde los principios de diseño se están violando y esto está haciendo el sistema difícil de mantener y escalar. Aquí está cómo proponemos resolverlo usando los principios de diseño:

- **Responsabilidad Única (SRP):**
  Hemos visto que algunas clases, como Libro o Usuario, tienen demasiadas responsabilidades mezcladas (por ejemplo, no solo guardan datos, sino que calculan multas, validan préstamos, envían notificaciones, etc).
  ¿Qué haremos?
  Vamos a dejar estas clases como modelos simples de datos y mover toda la lógica de negocio (cálculos, validaciones, reglas) a servicios especializados, como un GestorPrestamo o GestorUsuario.
- **Abierto/Cerrado (OCP):**
  Encontramos varios métodos llenos de condicionales para distintos tipos de libros, usuarios o notificaciones.
  ¿Qué haremos?
  Vamos a crear interfaces y estrategias para que, si aparece un nuevo tipo de multa o notificación, solo tengamos que agregar una nueva clase, no modificar el código existente. Así el sistema será más fácil de extender y menos propenso a errores.
- **Inversión de Dependencias (DIP):**
  Ahora mismo, algunas clases crean directamente sus dependencias (por ejemplo, una entidad que crea una instancia de un servicio).
  ¿Qué haremos?
  Vamos a hacer que todas las dependencias se inyecten a través del constructor o por el framework (Spring), y siempre programando contra interfaces. Por ejemplo, en vez de usar directamente un servicio de notificación, usaremos una interfaz como INotificador y luego le inyectamos la implementación que necesitemos.
- **Segregación de Interfaces (ISP):**
  Vimos casos donde una interfaz fuerza a las clases a implementar métodos que no usan (por ejemplo, todos los libros teniendo que generar un código de barras, aunque algunos no lo necesitan).
  ¿Qué haremos?
  Vamos a separar las interfaces grandes en interfaces más pequeñas y específicas, y que cada clase implemente solo las que realmente necesite.
- **Sustitución de Liskov (LSP):**
  Hay subclases que no cumplen bien lo que promete la superclase o la interfaz, lo que puede traer errores inesperados si intercambiamos clases.
  ¿Qué haremos?
  Vamos a asegurarnos de que todas las subclases se puedan usar de manera intercambiable, revisando que los métodos se comporten como se espera.

- **GRASP (Buenas prácticas de responsabilidad):**
  Encontramos mezclas de responsabilidades y creaciones de objetos fuera de lugar.
  ¿Qué haremos?
  Vamos a reubicar la lógica donde realmente corresponde, separar utilidades generales de la lógica de negocio y cuidar que cada clase tenga un solo motivo para cambiar.

Diagrama de clases propuesto

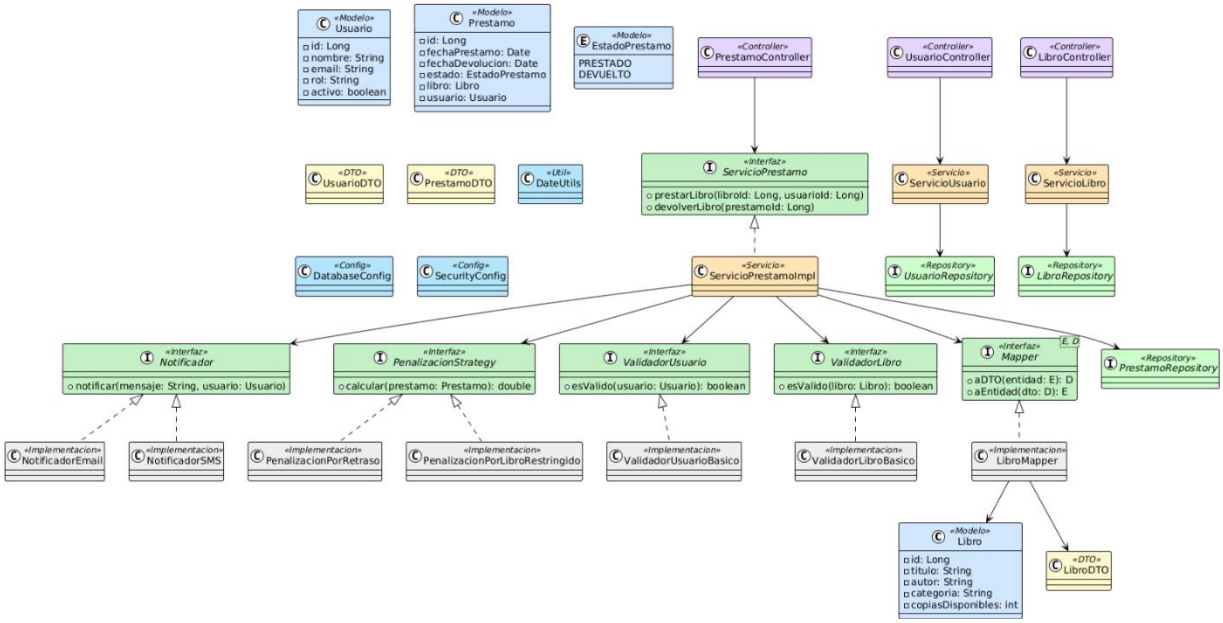
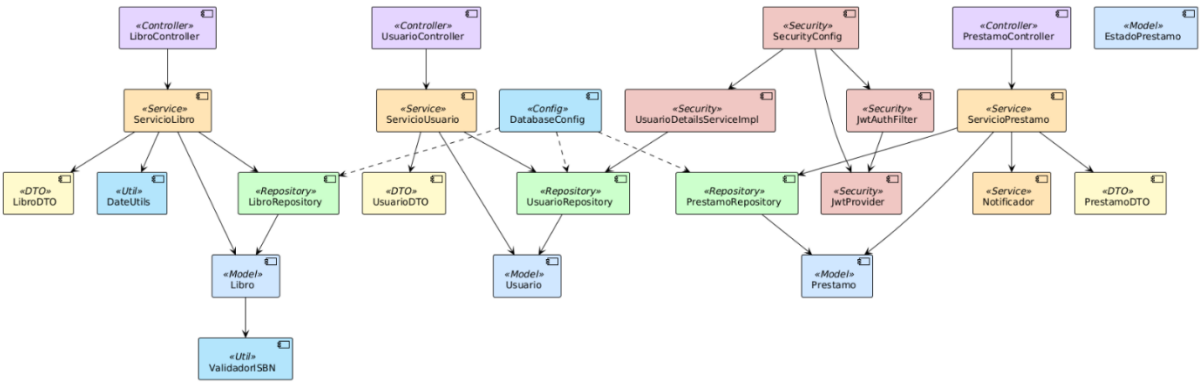


Diagrama de componentes propuesto



Análisis y documentación de beneficios esperados

Con el rediseño que proponemos, el sistema va a mejorar en muchos aspectos que sentirán tanto los desarrolladores como los usuarios y el equipo de soporte:

1. Hacer cambios será mucho más fácil

- Si surge una nueva necesidad o hay que corregir algo, encontraremos más rápido dónde hacerlo y no será una pesadilla entender el código.
- Las reglas y validaciones estarán en lugares específicos, así que no habrá que ir "a ciegas" por todo el proyecto para hacer un solo ajuste.

2. Agregar nuevas funcionalidades no romperá lo que ya funciona

- Si mañana la biblioteca quiere, por ejemplo, notificar por WhatsApp o definir un nuevo tipo de multa, solo hay que agregar una clase nueva, sin tocar lo que ya existe.
- Así, cada mejora será menos riesgosa y más rápida de implementar.

3. Menos dependencias difíciles de mantener

- Cada parte del sistema se conecta con otras solo por "contratos" claros (interfaces), no con detalles internos.
- Si alguna tecnología o proveedor cambia (por ejemplo, una base de datos o un servicio de correo), solo hay que cambiar una pieza, no el sistema completo.

4. Podemos reutilizar el trabajo

- Las clases de validación, conversión y cálculo estarán hechas para ser reutilizadas en distintos módulos, incluso en otros proyectos de la organización.
- Esto reduce la duplicidad y nos ahorra tiempo en desarrollos futuros.

5. Hacer pruebas será sencillo

- Será fácil simular dependencias o probar componentes de manera aislada, porque todo está bien desacoplado y hay contratos claros.
- Así tendremos más confianza en la calidad y estabilidad antes de lanzar nuevas versiones.

6. Cualquiera podrá entender el sistema más rápido

- La organización y claridad del código harán que cualquier desarrollador, nuevo o veterano, pueda incorporarse al proyecto y aportar en menos tiempo.
- También será más fácil para el equipo de soporte entender los diagramas y la documentación.

7. Menos sorpresas y errores

- Al tener responsabilidades bien divididas y contratos bien definidos, el sistema será más predecible y menos propenso a errores raros cuando haya cambios.
- Además, podremos añadir nuevas funciones con más tranquilidad.