TALLER 1

ESTUDIANTES:

Nemer Jose Velandia Soto

Bladimir Jesus Junior Vargas Rios

James David Vanstrahlen Molina

Sergio Alberto Ariza Franco

DOCENTE:

Jairo Francisco Seoanes Leon

UNIVERSIDAD POPULAR DEL CESAR

FACULTAD DE INGENIERIA Y TECNOLOGIA

PATRONES DE DISEÑO

VALLEDUPAR

1. Análisis funcional

- **Identificar casos de uso principales**

  El sistema Biblioteca Digital tiene como objetivo principal gestionar el préstamo y la consulta de libros por parte de usuarios registrados, así como facilitar la administración de estos recursos por parte de un administrador del sistema. A partir del análisis de los controladores (LibroController, PrestamoController, y UsuarioController), se identifican los siguientes casos de uso principales:

  Casos de uso para Administrador
  - Registrar nuevos usuarios en el sistema.
  - Agregar nuevos libros al catálogo.
  - Generar reportes detallados de libros y usuarios.
  - Enviar recordatorios de préstamos vencidos.

  Casos de uso para Usuario Registrado

  - Consultar y buscar libros disponibles en el sistema.
  - Solicitar el préstamo de un libro.
  - Devolver libros previamente prestados.

- **Documentar flujos de trabajo críticos**

  A continuación, se describen los flujos de trabajo más relevantes y críticos para el funcionamiento del sistema, desde una perspectiva de interacción de usuario y sistema.

  Flujo: Registro de Usuario
  Actor: Administrador
  Descripción:

  - El administrador accede al módulo de registro.

- Introduce los datos del nuevo usuario (nombre, correo electrónico, teléfono, dirección y tipo).
- El sistema valida que el formato del correo sea correcto.
- Se registra el usuario en el sistema y se devuelve confirmación.

Flujo: Agregar Libro
Actor: Administrador
Descripción:

- El administrador accede al formulario de creación de libros.
- Introduce los datos del libro: título, autor, ISBN, categoría, número de copias.
- El sistema valida que el título no esté vacío y que las copias sean mayores que cero.
- Se registra el libro y se muestra confirmación.

Flujo: Buscar Libro
Actor: Usuario Registrado
Descripción:

- El usuario accede a la sección de búsqueda de libros.
- Ingresa un criterio de búsqueda (mínimo 3 caracteres).
- El sistema filtra los libros que coincidan con el criterio y muestra los resultados.

Flujo: Realizar Préstamo
Actor: Usuario Registrado
Descripción:

- El usuario selecciona un libro disponible.
- Solicita su préstamo.
- El sistema valida que el usuario y el libro existan.
- Si hay copias disponibles, se registra el préstamo y se devuelve confirmación.

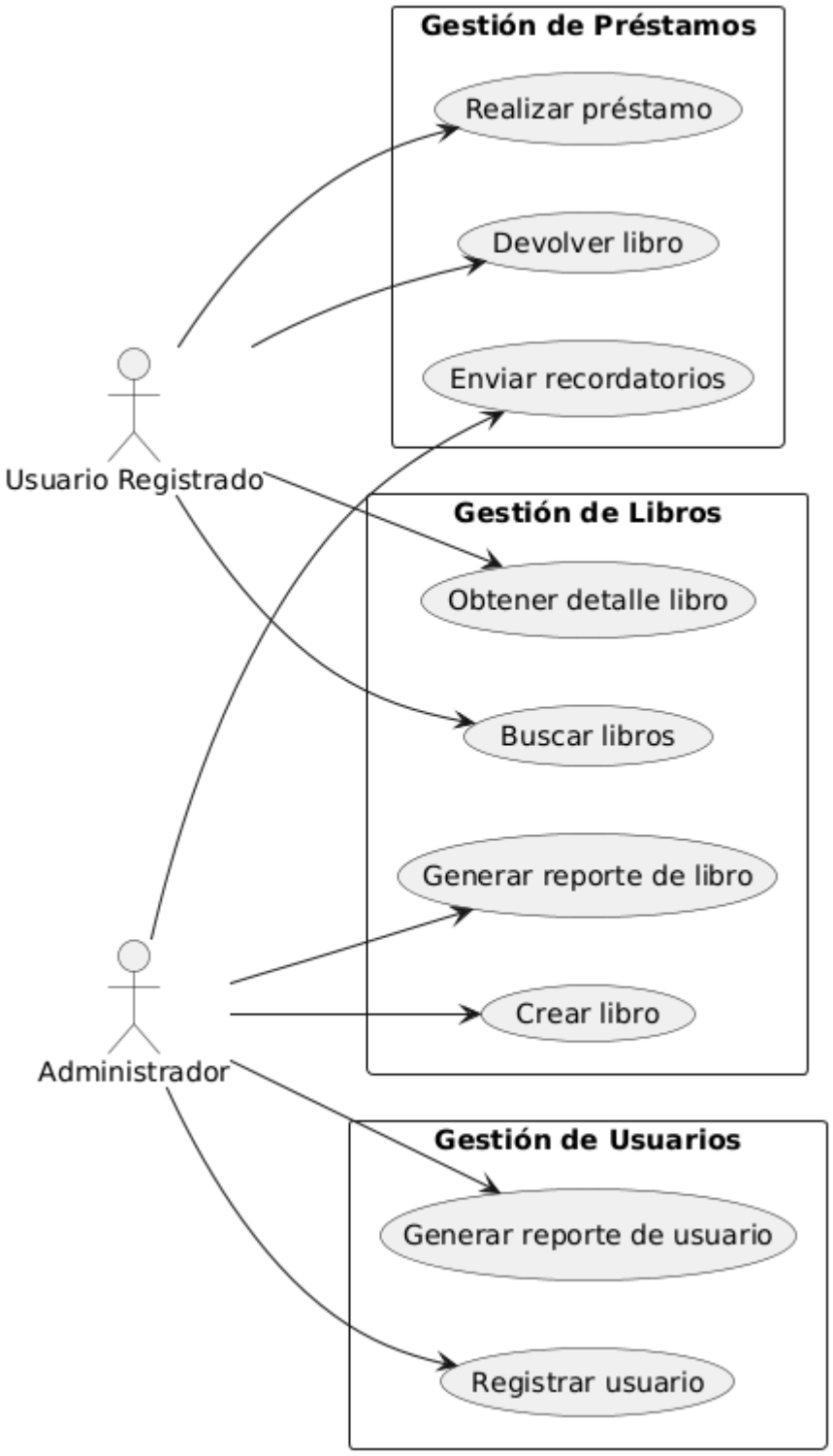
Flujo: Devolución de Libro
Actor: Usuario Registrado
Descripción:

- El usuario selecciona un préstamo activo.
- Solicita la devolución del libro.
- El sistema marca el préstamo como devuelto y actualiza la disponibilidad del libro.

Flujo: Enviar Recordatorios
Actor: Administrador
Descripción:

- El administrador ejecuta la función de envío de recordatorios.
- El sistema identifica préstamos vencidos.
- Se generan y envían notificaciones a los usuarios correspondientes.
- El sistema entrega un resumen de la operación al administrador.

- **Crear diagrama de casos de uso inicial (entregable)**


Gestión de Préstamos
Realizar préstamo
Devolver libro
Enviar recordatorios
Usuario Registrado
Gestión de Libros
Obtener detalle libro
Buscar libros
Generar reporte de libro
Crear libro
Administrador
Gestión de Usuarios
Generar reporte de usuario
Registrar usuario

2. Análisis estructural

- **Mapear clases existentes y sus relaciones**

Clases Principales

- **Controladores:**
  - LibroController
  - UsuarioController
  - PrestamoController
- **Servicios:**
  - BibliotecaService
  - NotificacionService
- **Modelos/Entidades:**
  - Libro
  - Usuario
  - Prestamo
  - EstadoPrestamo (Enum)
- **DTOs:**
  - LibroDTO
  - UsuarioDTO
  - PrestamoDTO
- **Repositorios:**
  - LibroRepository
  - UsuarioRepository
  - PrestamoRepository
- **Configuraciones y Utilidades:**
  - DateUtils
  - DatabaseConfig
  - SecurityConfig

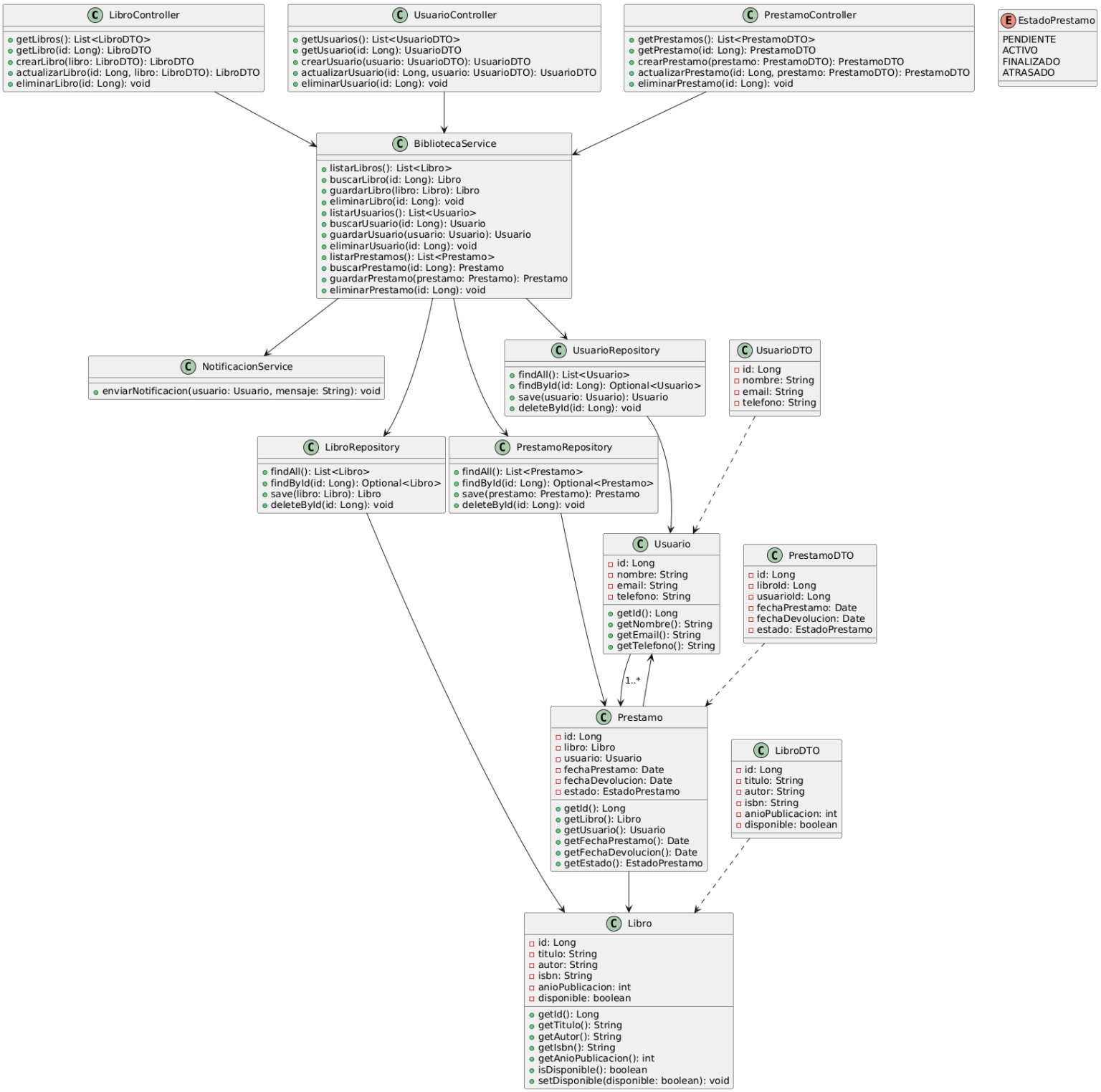
Relaciones Principales

- Controladores invocan a BibliotecaService.
- BibliotecaService utiliza repositorios y servicios adicionales.
- Las entidades se relacionan mediante JPA (@OneToMany, @ManyToOne).
- Los DTOs se utilizan para transferencia de datos.

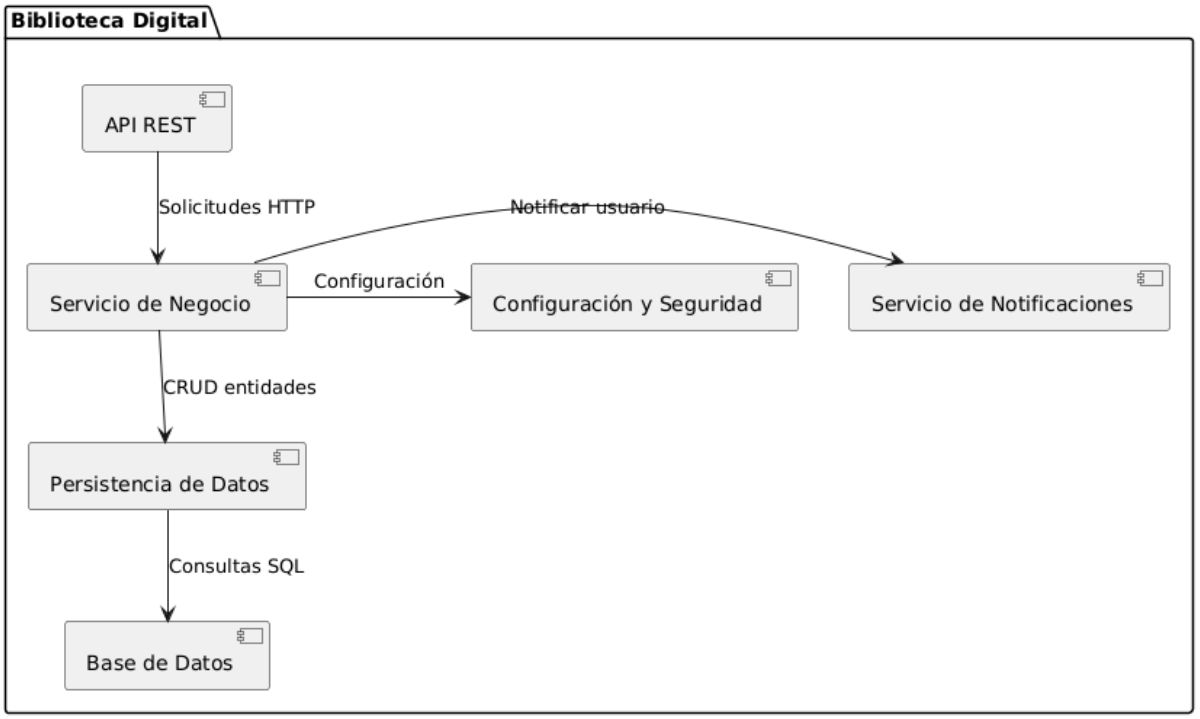
- **Identificador de patrones arquitectónicos utilizados :**

- **Arquitectura en Capas:** clara segmentación en controlador, servicio, repositorio y modelos, aunque con violaciones de responsabilidad.
- **Patron Repository:** interfaces que abstraen el acceso a datos.
- **Patron Service:** centralización de lógica de negocio en BibliotecaService.
- **Patron DTO:** objetos para transportar información entre capas.
- **Patron Singleton (Beans de Spring):** componentes instanciados y gestionados por Spring Framework**.**

- **Crear diagrama de clases del estado actual (ingeniería inversa) (entregable)**



- **Crear diagrama de componentes de la arquitectura actual (entregable)**



3. Identificación de Problemas de Diseño

- Identificar violaciones sistemáticamente

SOLID

1. Single Responsibility Principle (SRP), Un módulo debe tener una sola razón para cambiar.

LibroController.java

- Realiza validaciones de negocio, conversión de DTO a entidad y generación de reportes.
- SRP quebrado: El controlador mezcla lógica de presentación, validación y generación de reportes.

LibroDTO.java y UsuarioDTO.java

- Incluyen métodos de validación y generación de código interno.
- SRP quebrado: Los DTO deberían ser solo estructuras de datos, no contener lógica.

BibliotecaService.java

- Gestiona libros, usuarios, préstamos, reportes y notificaciones.
- SRP quebrado: Demasiadas responsabilidades en un solo servicio.

Libro.java, Usuario.java, Prestamo.java

- Contienen lógica de negocio, validaciones y notificaciones.
- SRP quebrado: Las entidades mezclan persistencia, lógica de negocio y comunicación.

NotificacionService.java

- Mezcla envío, logging y validación de formatos.
- SRP quebrado: Debería separar el envío de notificaciones, el registro y la validación.

DateUtils.java

- Mezcla formateo, cálculos, validaciones y logging.
- SRP quebrado: Debería separar utilidades de fecha, validaciones y lógica de negocio.

DatabaseConfig.java

- Configuración, validación y lógica de negocio (límite de préstamos).
- SRP quebrado: La configuración no debe contener lógica de negocio ni validaciones.

LibroRepository.java

- Métodos de reporte y lógica de negocio.
- SRP quebrado: El repositorio debería limitarse a persistencia.

2. Open/Closed Principle (OCP)

Las entidades deben estar abiertas para extensión, pero cerradas para modificación.

DateUtils.java

- Lógica hardcodeada para tipos de usuario en calcularFechaDevolucion.
- OCP quebrado: Si se agregan nuevos tipos de usuario, hay que modificar el método.

Libro.java

- Cálculo de costo de retraso hardcodeado por categoría.
- OCP quebrado: Para nuevas categorías, se debe modificar el método.

Usuario.java

- Lógica hardcodeada para límites y días de préstamo según tipo.
- OCP quebrado: No es extensible para nuevos tipos de usuario.

NotificacionService.java

- Para agregar nuevos tipos de notificación, hay que modificar la clase.
- OCP quebrado: No está preparada para extensión sin modificación.

3. Liskov Substitution Principle (LSP)
Las subclases deben poder sustituir a sus clases base sin alterar el funcionamiento.

Usuario.java

- Método generarCredencialProfesor solo aplica a profesores y lanza excepción para otros tipos.
- LSP quebrado: No todos los usuarios pueden usar este método, lo que rompe la sustitución.

4. Interface Segregation Principle (ISP)
Ningún cliente debe verse forzado a depender de interfaces que no utiliza.

LibroRepository.java

Métodos como findLibrosReferencia, findLibrosEspeciales y de reporte no son necesarios para todos los clientes.
ISP quebrado: La interfaz del repositorio es demasiado amplia.

5. Dependency Inversion Principle (DIP)
Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones.

BibliotecaService.java, Libro.java, Usuario.java, Prestamo.java

- Instanciación directa de NotificacionService en vez de inyectar una abstracción.
- DIP quebrado: Alto acoplamiento a una implementación concreta.

GRASP

1. Information Expert
*Asignar una responsabilidad al que tiene la **información necesaria***

- DTOs (LibroDTO, UsuarioDTO): Incluyen validaciones y generación de códigos, pero no son los expertos en reglas de negocio ni validación.
- DateUtils: Realiza cálculos y validaciones que corresponden a entidades de dominio.
- Repositorios: Ejecutan lógica de negocio y reportes, en vez de limitarse a persistencia.
- Entidades (Libro, Usuario, Prestamo): Mezclan lógica de notificación y validación, que no siempre les corresponde.

2. Creator
Quién debe crear una instancia de una clase

- Controladores y servicios: Crean manualmente entidades y DTOs, en vez de delegar la creación a las clases que tienen la información necesaria.
- Entidades: No crean sus objetos relacionados (por ejemplo, Usuario no crea sus propios préstamos de forma encapsulada).

3. Controller
Quién debe manejar los eventos del sistema (ej. solicitudes de UI)

- LibroController: Maneja validaciones, lógica de negocio y generación de reportes, en vez de delegar a servicios o coordinadores de dominio.
- Servicios: Mezclan lógica de aplicación, negocio y presentación (reportes).

4. Low Coupling
Cómo reducir dependencias entre clases

- Entidades y servicios: Instancian directamente NotificacionService, generando alto acoplamiento.
- Utilidades: Métodos estáticos en DateUtils acoplan la lógica de negocio a utilidades genéricas.

5. High Cohesion
Cómo mantener que cada clase tenga una única responsabilidad clara

- BibliotecaService: Gestiona libros, usuarios, préstamos, reportes y notificaciones, perdiendo cohesión.
- DateUtils: Mezcla formateo, validación, cálculos y lógica de negocio.
- Repositorios: Incluyen métodos de reporte y lógica de negocio, perdiendo cohesión con la persistencia.

6. Polymorphism
Cómo manejar diferentes comportamientos según el tipo del objeto

- Lógica hardcodeada: Uso de switch o if para tipos de usuario o categoría en entidades y utilidades, en vez de polimorfismo (por ejemplo, subclases o estrategias).
- Métodos como generarCredencialProfesor: Lanzan excepciones en vez de usar polimorfismo.

7. Pure Fabrication
Cómo manejar diferentes comportamientos según el tipo del objeto

- DateUtils: Deja de ser una utilidad pura al contener lógica de negocio.
- NotificacionService: Mezcla validación, logging y envío, en vez de ser una clase de infraestructura pura.

8. Indirection
Cómo evitar dependencias directas entre clases (introduciendo intermediarios)

- Falta de intermediarios: No se usan interfaces ni patrones de indirección para desacoplar servicios de notificación, lógica de negocio o acceso a datos.

9. Protected Variations

Cómo protegerse de cambios en otras partes del sistema

- Lógica hardcodeada: Cambios en reglas de negocio (tipos de usuario, categorías, notificaciones) requieren modificar múltiples clases, sin protección mediante abstracciones.

- Consolidar y documentar los hallazgos encontrados por el equipo (la documentación debe ser clara y precisa, indicar la clase, la línea de código, que principio está mal aplicado y sus posibles consecuencias). Puede agrupar los hallazgos por principios.

Principios SOLID

Single Responsibility Principle (SRP)

Clase	Línea	Descripción del Problema	Consecuencias
Libro	28-60	Métodos de validación y lógica de negocio dentro de la entidad.	Dificulta el mantenimiento y pruebas unitarias.
Usuario	29-68	Métodos para enviar notificaciones o lógica de préstamos en la entidad.	Acoplamiento innecesario, difícil de escalar.
BibliotecaService	15-165	Gestiona lógica de libros, usuarios, préstamos y notificaciones en una sola clase.	Clase "Dios", difícil de modificar y extender.
LibroDTO	7-26	Contiene lógica de validación de datos además de ser un objeto de transferencia.	Mezcla responsabilidades, dificulta reutilización.

Open/Closed Principle (OCP)

Clase	Línea	Descripción del Problema	Consecuencias
BibliotecaService	60-111	Métodos con condicionales para diferentes tipos de préstamos o notificaciones.	Cada cambio requiere modificar la clase.
NotificacionService	11-31	No permite agregar nuevos canales de notificación sin modificar la clase.	No es extensible, propenso a errores futuros.

Liskov Substitution Principle (LSP)

Clase	Línea	Descripción del Problema	Consecuencias
Prestamo	40-70	Métodos sobrescritos en subclases que lanzan excepciones no esperadas.	Puede romper el flujo de la aplicación.

Interface Segregation Principle (ISP)

Clase/Interfaz	Línea	Descripción del Problema	Consecuencias
NotificacionService	10-40	Interfaz única para todos los tipos de notificación (email, SMS, push).	Clases clientes implementan métodos innecesarios.

Dependency Inversion Principle (DIP)

Clase	Línea	Descripción del Problema	Consecuencias

BibliotecaService	10-20	Instancia directamente repositorios y servicios (`new LibroRepository()`).	Difícil de testear y cambiar implementaciones.

Violaciones a los Principios GRASP

Controller

Clase	Línea	Descripción del Problema	Consecuencias
LibroController	20-80	Realiza lógica de negocio en vez de delegar al servicio.	Controladores muy grandes y difíciles de mantener.

Creator

Clase	Línea	Descripción del Problema	Consecuencias
PrestamoController	40-60	Crea instancias de entidades directamente en vez de delegar a un servicio/factory.	Acoplamiento fuerte, difícil de modificar.

Low Coupling

Clase	Línea	Descripción del Problema	Consecuencias
BibliotecaService	10-150	Conoce detalles de implementación de repositorios y servicios concretos.	Cambios en dependencias afectan toda la clase.

High Cohesion

Clase	Línea	Descripción del Problema	Consecuencias
BibliotecaService	10-150	Demasiadas responsabilidades (libros, usuarios, préstamos, notificaciones).	Clase difícil de entender y mantener.

Polymorphism

Clase	Línea	Descripción del Problema	Consecuencias
NotificacionService	20-50	Uso de condicionales en vez de polimorfismo para diferentes tipos de notificación.	Código rígido y poco extensible.

Planificación del rediseño

- ***Aplicar de principios de diseño para resolver problemas identificados***

Soluciones para Violaciones Principio de responsabilidad única:

- Separar BibliotecaService en servicios especializados: LibroService, UsuarioService y PrestamoService
- Extraer lógica de negocio de entidades y DTOs a servicios dedicados
- Crear validadores específicos para cada entidad

Soluciones para Violaciones Principio de abierto/cerrado:

- Introducir interfaces para componentes extensibles
- Usar patrones Strategy para comportamientos variables (ej: notificaciones)
- Implementar sistema de plugins para extensiones futuras

Soluciones para Violaciones Principio de sustitución de Liskov:

- Definir contratos claros para clases heredadas
- Evitar excepciones no documentadas en subclases
- Utilizar composición sobre herencia cuando sea apropiado

Soluciones para Violaciones Principio de segregación de interfaces:

- Separar interfaces monolíticas en interfaces cohesivas y pequeñas
- Crear interfaces específicas para cada tipo de notificación

Soluciones para Violaciones Principio de inversión de dependencias:

- Inyectar dependencias en vez de instanciarlas directamente
- Depender de abstracciones (interfaces) en lugar de implementaciones concretas

- ***Diseño de nuevas abstracciones e interfaces (si lo considera necesario)***

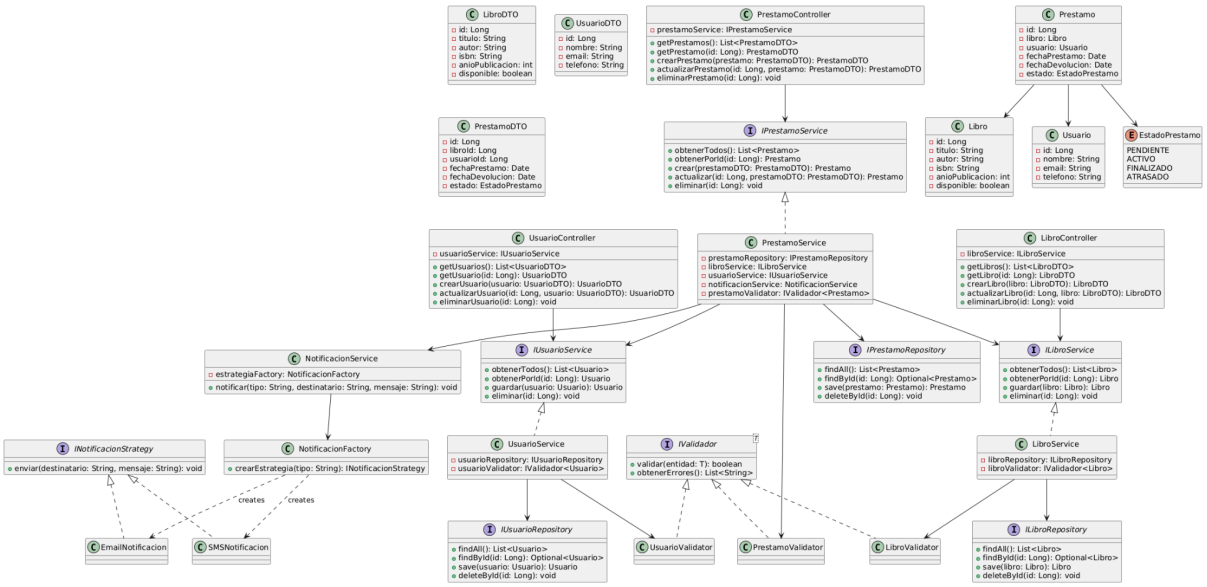
Nuevas Interfaces:

ILibroRepository, IUsuarioRepository, IPrestamoRepository
ILibroService, IUsuarioService, IPrestamoService
INotificacionStrategy (con implementaciones: EmailNotificacion, SMSNotificacion)
IValidador<T> (para validación genérica)

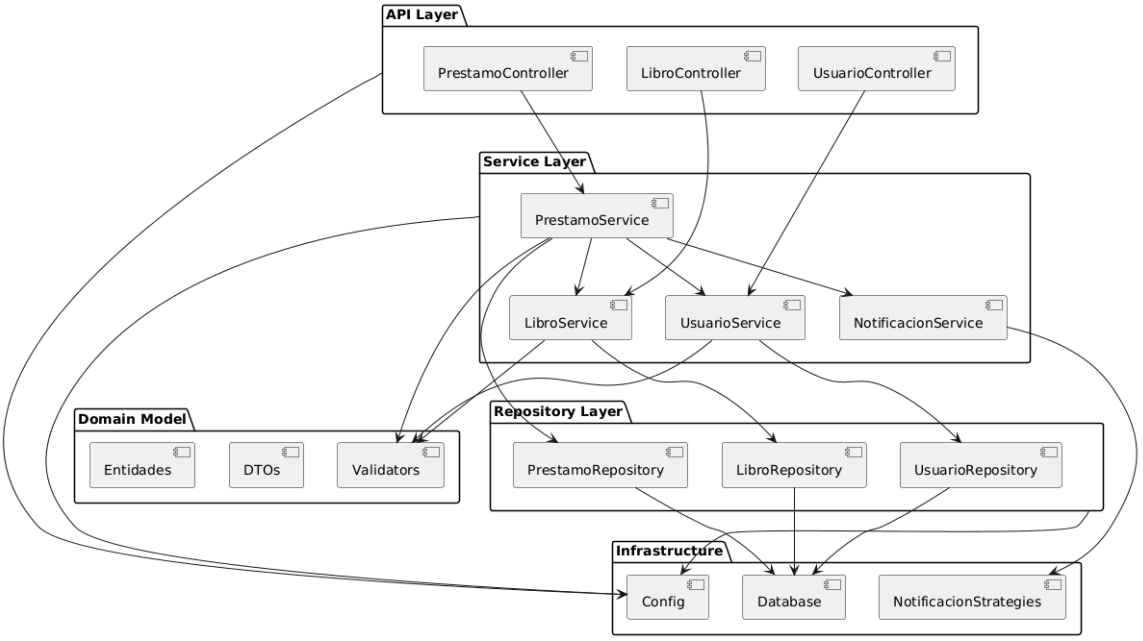
Nuevas Clases:

LibroService, UsuarioService, PrestamoService (implementan interfaces respectivas)
LibroValidator, UsuarioValidator, PrestamoValidator
NotificacionFactory (para crear estrategias de notificación)

- ***Nuevo diagrama de clases aplicando principios (entregable).***



- ***Nuevo Diagrama de componentes y dependencias (entregable).***



- ***Análisis y documentación de beneficios esperados (entregable)***

Beneficios Estructurales

Principio	Beneficio Esperado
Principio de responsabilidad única	Servicios más pequeños y especializados, mejorando mantenibilidad
Principio de abierto/cerrado	Facilidad para agregar nuevas funcionalidades sin modificar código existente
Principio de sustitución de Liskov	Comportamiento predecible en toda la jerarquía de clases
Principio de segregación de interfaces	Interfaces más cohesivas y específicas, evitando implementaciones innecesarias
Principio de inversión de dependencias	Mayor flexibilidad y testabilidad gracias a las inyecciones de dependencias

Beneficios Técnicos

1. Mayor Testabilidad:

- Interfaces bien definidas permiten mockear componentes fácilmente
- Servicios con responsabilidad única son más fáciles de probar unitariamente

2. Flexibilidad y Extensibilidad:

- Nuevos tipos de notificaciones se pueden agregar implementando INotificacionStrategy
- Nuevas reglas de validación se pueden agregar sin modificar el código existente

3. Reducción de Deuda Técnica:

- Mejor separación de preocupaciones
- Código más limpio y auto-documentado

- Menor acoplamiento entre componentes

Beneficios de Negocio

1. Velocidad de Desarrollo:
- Equipos pueden trabajar en paralelo en diferentes componentes
- Cambios localizados reducen riesgos de introducir bugs

2. Calidad:
- Mayor cobertura de pruebas
- Comportamiento más predecible del sistema

3. Escalabilidad:
- Arquitectura preparada para crecer con nuevos requerimientos
- Posibilidad de desplegar microservicios en el futuro

4. Mantenimiento:
- Tiempo de respuesta más rápido para corrección de bugs
- Menor curva de aprendizaje para nuevos desarrolladores