## HALLAZGOS DE VIOLACIÓN DE LOS PRINCIPIOS SOLID EN LA CAPA DE SERVICIOS DEL SOFTWARE DE GESTIÓN DE BIBLIOTECA

## Presentado por

## JUAN DAVID LOPEZ

## ALVIN BRAVO CUJIA

## TUTOR:

## ING. JAIRO F. SEOANES L

## HALLAZGOS DE VIOLACIÓN DE LOS PRINCIPIOS SOLID EN LA CAPA DE SERVICIOS DEL SOFTWARE DE GESTIÓN DE BIBLIOTECA

El software de gestión de biblioteca cuenta con 1 capa de servicios que posee dos clases, estas son:

1. BibliotecaService
2. NotificacionService

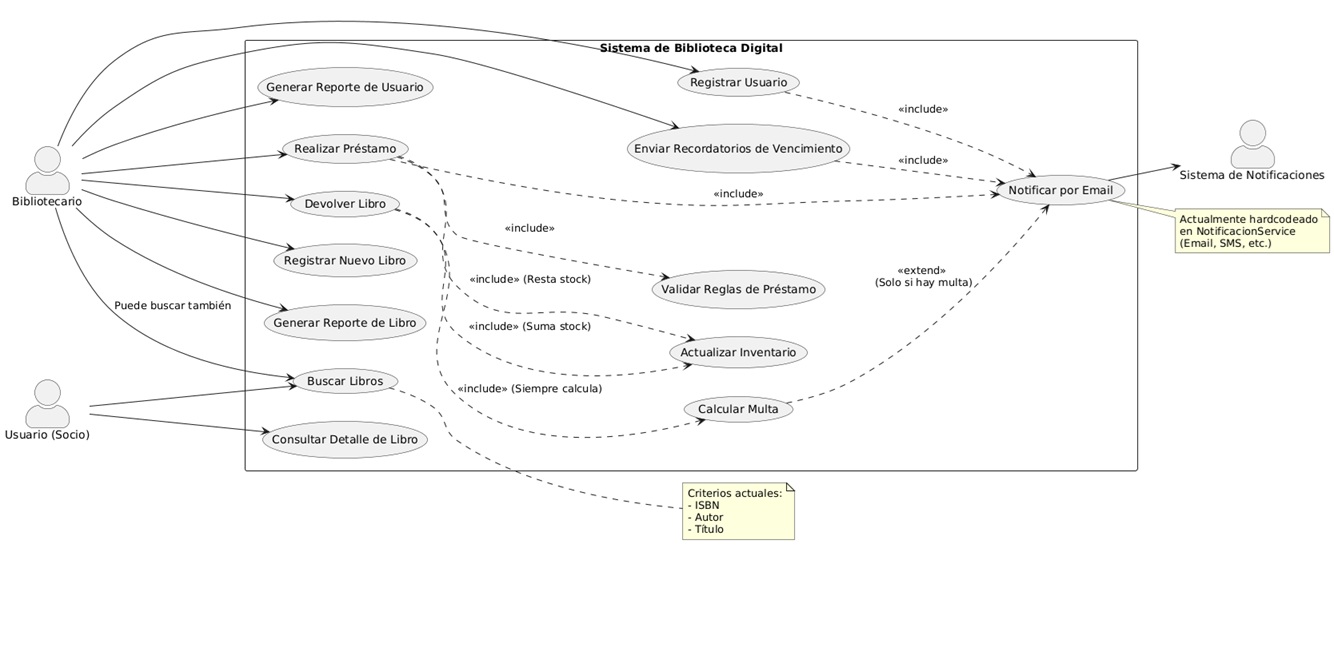
como se muestra en la imagen 1.

**imagen 1.**

## 

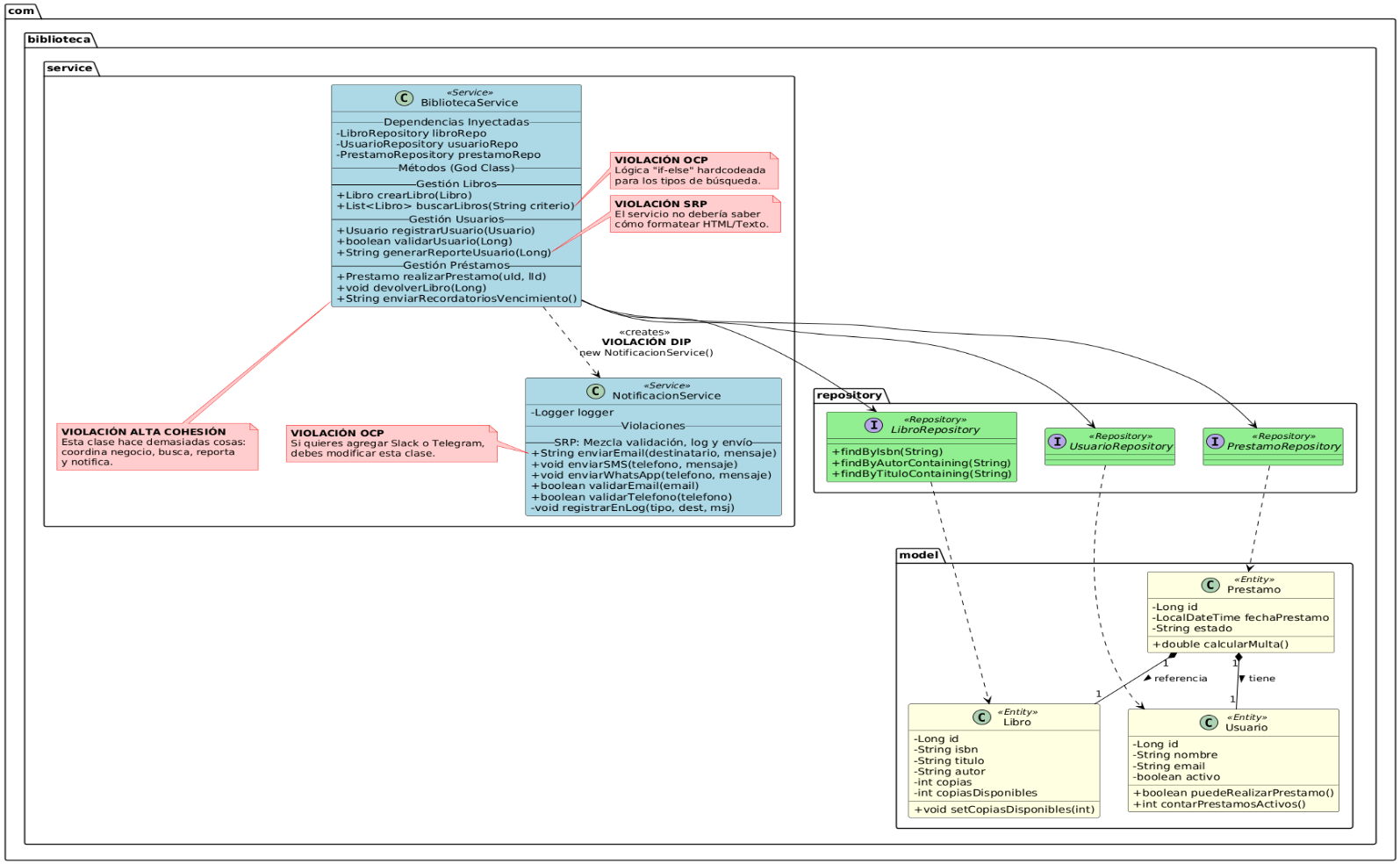
Se procede a crear el diagrama de caso de uso y de clases de todo el sistema.

**DIAGRAMA DE CASO DE USO (SIN SOLID)**

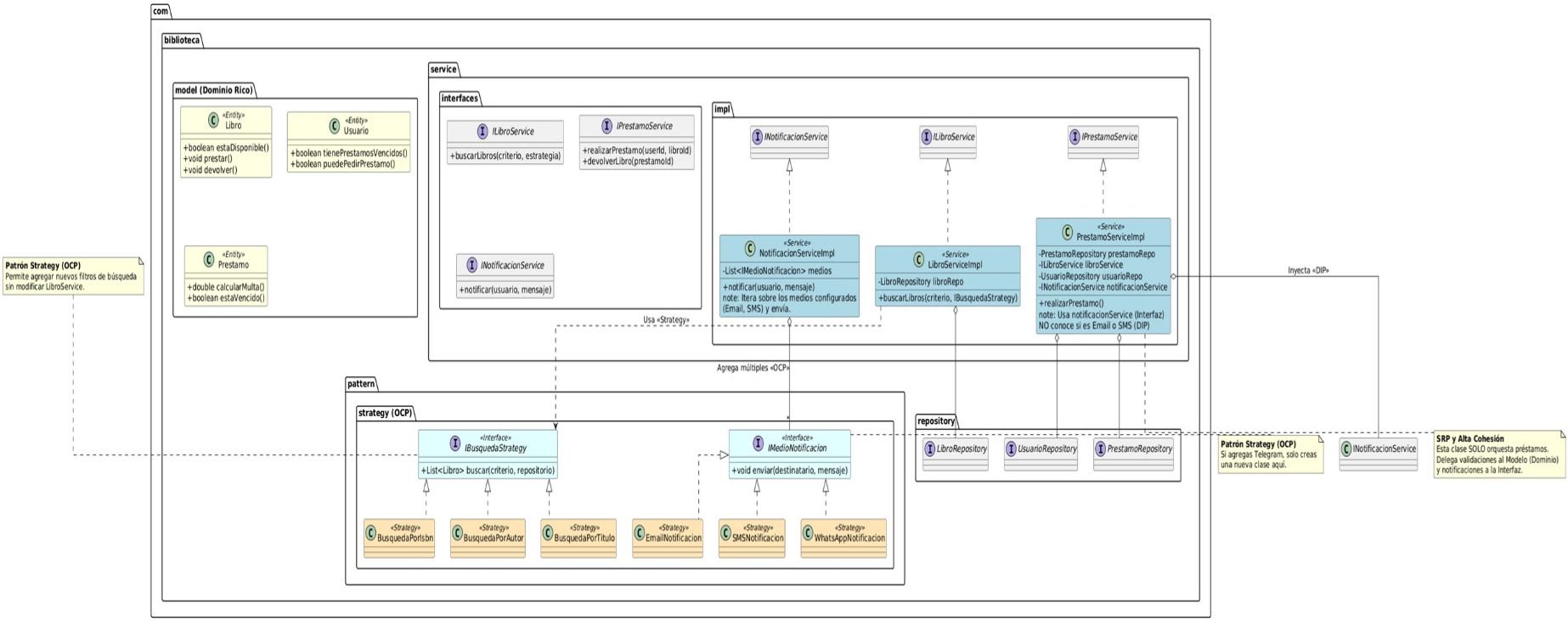


**DIAGRAMA DE CLASE (VIOLACION SOLID)**

Este Diagrama de Clases está diseñado específicamente para resaltar los "errores" arquitectónico, para evidenciar las violaciones a los principios SOLID (especialmente SRP y DIP) y GRASP (Alta Cohesión y Bajo Acoplamiento).



**DIAGRAMA DE CLASE (PROPUESTA SOLID)**



**Cambios Estructurales Clave que verás en el diagrama:**

1. **Aplicación de SRP (Single Responsibility Principle):**
   * BibliotecaService ha desaparecido. Ahora tenemos servicios especializados: LibroService, UsuarioService, PrestamoService.
   * La lógica de reportes se movió a ReporteService.
2. **Aplicación de OCP (Open/Closed Principle) con Patrón Strategy:**
   * **Búsquedas:** Creamos una interfaz IBusquedaStrategy. Si quieres buscar por "Género", creas una nueva clase BusquedaPorGenero y no tocas el código existente.
   * **Notificaciones:** Creamos una interfaz IMedioNotificacion. Puedes agregar WhatsApp o Slack sin modificar el servicio de préstamos.
3. **Aplicación de DIP (Dependency Inversion Principle):**
   * Los servicios de alto nivel (PrestamoService) ya no dependen de clases concretas (EmailService), sino de abstracciones (INotificacionService).
4. **Aplicación de ISP (Interface Segregation Principle):**
   * Interfaces pequeñas y específicas para cada servicio.

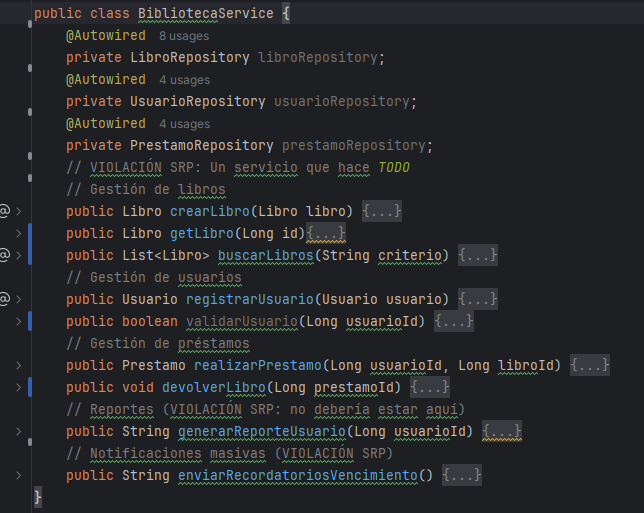
## HALLAZGOS DE VIOLACIÓN DE LOS PRINCIPIOS SOLID EN LA CAPA DE SERVICIOS DEL SOFTWARE DE GESTIÓN DE BIBLIOTECA

**1. Principio de Responsabilidad Única (SRP)**

El Principio de Responsabilidad Única (SRP) establece que "Una clase debe tener una, y solo una, razón para cambiar." En otras palabras, una clase, módulo o función debe tener una sola responsabilidad bien definida.

**SRP-01: BibliotecaService con Múltiples Responsabilidades**

La clase BibliotecaService viola el SRP al concentrar la lógica de negocio de libros, usuarios, préstamos, devoluciones, notificaciones y reportes.



| **Razón para el Cambio (Responsabilidad)** | **Métodos en BibliotecaService** |
| --- | --- |
| Gestión de Libros | crearLibro, getLibro, buscarLibros |
| Gestión de Usuarios | registrarUsuario, validarUsuario |
| Gestión de Préstamos/Devoluciones | realizarPrestamo, devolverLibro |
| Generación de Reportes | generarReporteUsuario |
| Notificación Masiva | enviarRecordatoriosVencimiento |

**CONSECUENCIAS**

***Alto Acoplamiento*** lo que nos dice que un cambio en la lógica de préstamos obliga a modificar y recompilar la misma clase que maneja los reportes.

***Baja Cohesión:*** Los métodos dentro de la clase no están fuertemente relacionados entre sí.

***Dificultad de Prueba:*** Probar BibliotecaService requiere configurar mocks y dependencias para libros, usuarios, préstamos, y notificaciones.

**RECOMENDACIÓN**

Se puede hacer de varias maneras, pero una seria crear capa de servicios independientes asi aumentamos la cohesión.

*LibroService (para crearLibro, getLibro, buscarLibros).*

*UsuarioService (para registrarUsuario, validarUsuario).*

*PrestamoService (para realizarPrestamo, devolverLibro).*

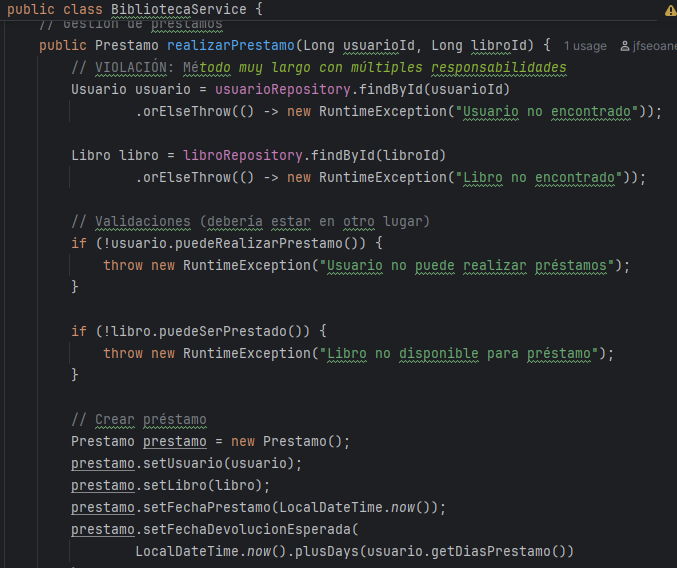
*NotificacionService (debe ser inyectado, no instanciado).*

*ReporteService (para generarReporteUsuario).*

**SRP-02: Métodos demasiado largos dentro de BibliotecaService.**

Dentro de la clase BibliotecaService() existe un metodo realizarPrestamo() que tiene más de 40 líneas y hace:

* Validación de usuario
* Validación de libro
* Construcción del préstamo
* Gestión de disponibilidad
* Persistencia del préstamo
* Envío de notificació
* Manejo de errores



Esto viola SRP porque el método tiene 6 responsabilidades distintas.

**CONSECUENCIAS**

* Difícil de entender y mantener
* Alto riesgo de errores al modificar un fragmento
* Dificultad para reutilizar lógica (como validaciones)
* Imposibilidad de testear la lógica de forma aislada

**RECOMENDACIÓN**

Separar en métodos privados específicos o mover la lógica a otros servicios:

ValidacionPrestamoService

NotificacionService

MultaService

**RECOMENDACIONES**

Métodos superiores a **20 líneas deben dividirse** siempre.

Toda lógica que se repite debe evolucionar a un servicio independiente.

Si un método cumple más de **una acción**, ya rompe el SRP.

**2. (O) Principio Abierto/Cerrado (OCP)**

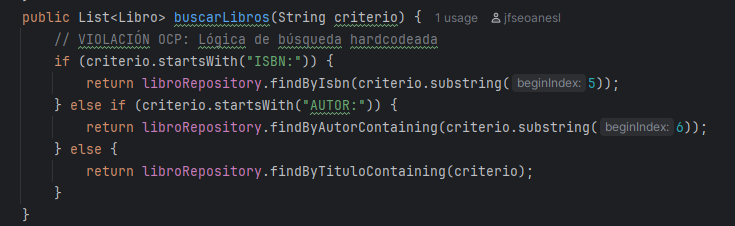
El Principio Abierto/Cerrado (OCP) establece que:

"Las clases deben estar abiertas para extensión, pero cerradas para modificación."

Esto significa que, a medida que el sistema crece, debe ser posible agregar nuevas funcionalidades sin necesidad de modificar las clases existentes. Para lograrlo, la arquitectura debe basarse en abstracciones, polimorfismo, inyección de dependencias, y evitar condicionales rígidos que obliguen a tocar el mismo código cada vez que aparece un nuevo caso.

El propósito de OCP es evitar que los cambios introduzcan errores en funcionalidades existentes y permitir que el sistema crezca de forma estable, predecible y con bajo riesgo.

**OCP-01: LÓGICA DE BÚSQUEDA HARDCODEADA EN BUSCARLIBROS ()**

****

Dentro de BibliotecaService, el método buscarLibros() implementa una lógica basada en condicionales (if/else) que determina qué tipo de búsqueda debe ejecutarse dependiendo de un prefijo incluido en el parámetro. Esta solución es rígida porque cada vez que se necesite agregar una nueva forma de búsqueda (por editorial, año, categoría avanzada, búsqueda combinada, búsqueda difusa, etc.), será necesario modificar este mismo método, lo que rompe directamente el principio OCP.

**CONSECUENCIAS**

Fragilidad: Cada cambio afecta el comportamiento existente.

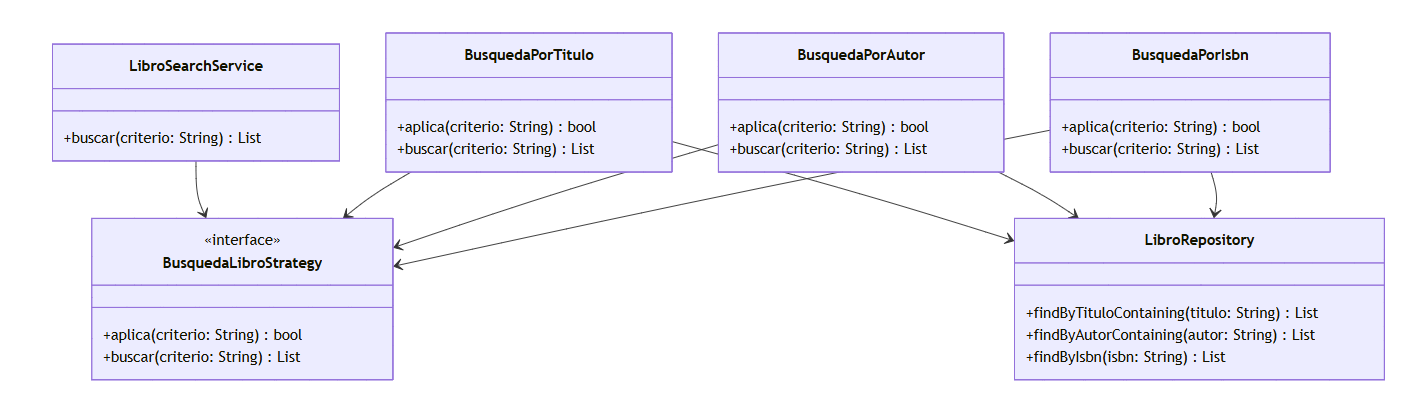
Difícil de extender: Cualquier búsqueda nueva obliga a modificar el servicio.

Acoplamiento: El servicio conoce demasiado sobre los criterios de búsqueda.

Alta probabilidad de errores al crecer el sistema.

**RECOMENDACIONES**

La mejor solución que podemos implementar es el Patrón Estrategia, porque nos permite organizar mejor la lógica y mantener el código preparado para cambios futuros sin tener que modificar lo que ya funciona. En lugar de usar un método lleno de condicionales que se vuelve rígido y difícil de mantener lo que hacemos es crear una interfaz común para todas las formas de búsqueda y luego generar estrategias concretas para cada criterio: por título, por autor, por ISBN, etc. Esto quiere decir que si mañana aparece un nuevo requisito, como buscar por categoría o por año, no tenemos que abrir ni modificar la clase principal; simplemente creamos una nueva estrategia que implemente el comportamiento deseado. Esto hace que el código sea más limpio, más organizado, mucho más fácil de probar y, sobre todo, abierto a la extensión, pero cerrado a la modificación, que es justamente lo que busca el OCP.



si agregamos una nueva busqueda:



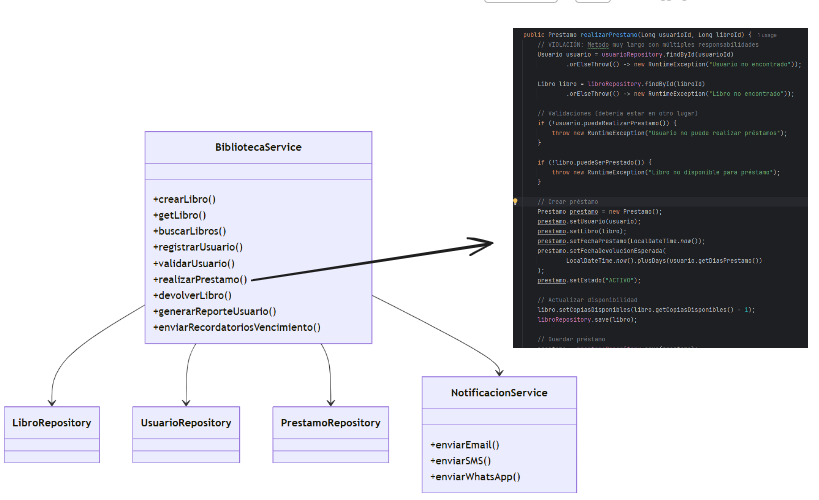
LibroSearchService no se modifica. Solo sigue iterando sobre las estrategias registradas.

Para agregar la búsqueda por categoría, se crea una nueva clase BusquedaPorCategoria que implementa BusquedaLibroStrategy.

BusquedaPorCategoria accede al repositorio (LibroRepository) usando findByCategoria().

Esto demuestra cómo el sistema está abierto para extensión (agregar BusquedaPorCategoria) pero cerrado para modificación (no tocamos LibroSearchService ni las demás estrategias).

**OCP-02 — Lógica rígida en realizarPrestamo()**

****

El método realizarPrestamo() concentra dentro de sí toda la lógica necesaria para procesar un préstamo. Esto incluye pasos como:

* Validación del usuario
* Validación del libro
* Cálculo y reglas de fechas
* Actualización de inventario
* Construcción del objeto Préstamo
* Persistencia en base de datos
* Envío de notificaciones

Este diseño implica que todas las reglas del negocio están fuertemente acopladas dentro de un único método, lo que genera un flujo rígido, monolítico y difícil de modificar.

A medida que el sistema requiere nuevos comportamientos, por ejemplo:

* nuevas modalidades de préstamo (exprés, digital, interbibliotecario, eventos),
* nuevas políticas de duración o multas,
* nuevas reglas de disponibilidad según tipo de usuario, libro o contexto,
* nuevos métodos de notificación (SMS, WhatsApp, push, etc.),
* nuevas validaciones o pasos intermedios,

El método debe ser editado directamente, agregando condicionales, duplicando lógica o insertando nuevos bloques.

Esto viola el Principio de Abierto/Cerrado (OCP) porque la función no está abierta a la extensión, ya que no permite agregar nuevas variantes del proceso sin modificarla, y no está cerrada a la modificación, porque cada cambio de negocio obliga a alterar su código interno.

**CONSECUENCIAS**

Difícil de escalar: nuevas políticas implican modificar la clase.

Método gigantesco: demasiado complejo y propenso a errores.

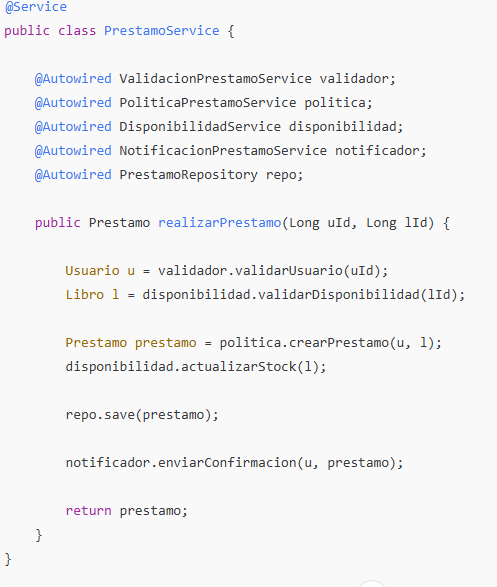
Alto riesgo al tocarlo: cambios pueden romper múltiples escenarios.

Acoplamiento innecesario entre entidades.

**RECOMENDACIONES**

Para cumplir el Principio de Abierto/Cerrado (OCP) en realizarPrestamo(), se recomienda separar las responsabilidades en servicios especializados: uno para validar al usuario y al libro (ValidacionPrestamoService), otro para aplicar las políticas de préstamo según el tipo de usuario (PoliticaPrestamoService), uno más para gestionar la disponibilidad de los libros (DisponibilidadService), otro para registrar el préstamo en el repositorio (RegistroPrestamoService) y finalmente uno encargado de las notificaciones (NotificacionPrestamoService). Además, es aconsejable implementar el patrón Estrategia, de modo que cada tipo de préstamo (estudiante, profesor, administrativo) tenga su propia estrategia con reglas específicas de duración y límite de libros. De esta manera, cualquier cambio o incorporación de un nuevo tipo de préstamo se logra extendiendo el sistema sin modificar el código existente, mejorando la mantenibilidad, reduciendo acoplamiento y respetando los principios SOLID.

**Ejemplo de cómo quedaría.**

****

1. **(L) Principio de Sustitución de Liskov (LSP)**

"Los subtipos deben ser sustituibles por sus tipos base sin alterar la corrección del programa."

Este principio aplica a las jerarquías de herencia (clases que extienden a otras o implementan interfaces).

No hay una violación visible de LSP en la clase BibliotecaService, ya que no estamos viendo ninguna jerarquía de herencia donde esta clase participe.

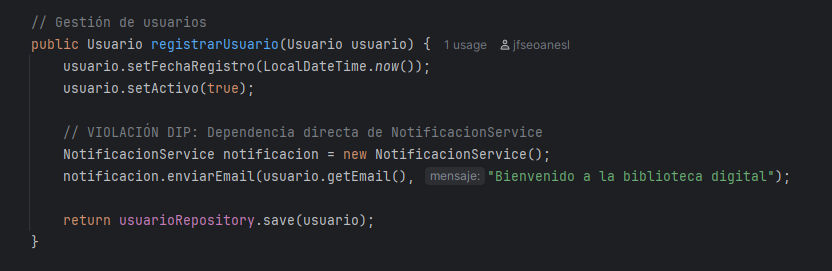
1. **PRINCIPIO DIP – Dependency Inversion Principle**

“Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones (interfaces). Además, las abstracciones no deben depender de los detalles; los detalles deben depender de las abstracciones.”

En otras palabras, un servicio de alto nivel (como BibliotecaService) no debería instanciar directamente servicios concretos o repositorios, sino depender de interfaces, lo que permite intercambiar implementaciones sin modificar el código de alto nivel. Esto reduce acoplamiento y facilita pruebas unitarias y extensión futura.

**DIP-01: BIBLIOTECASERVICE DEPENDE DIRECTAMENTE DE NOTIFICACIONSERVICE**

En la clase BibliotecaService, cada vez que se realiza un préstamo, se instancia directamente NotificacionService para enviar correos o SMS:



Esto rompe el DIP porque BibliotecaService (módulo de alto nivel) depende de una implementación concreta (NotificacionService) en lugar de depender de una abstracción, como una interfaz INotificacionService.

**CONSECUENCIAS**

**Acoplamiento fuerte:** BibliotecaService está directamente ligado a la implementación concreta.

**Difícil de testear:** Para pruebas unitarias, se tendría que instanciar NotificacionService real o usar hacks.

**Baja flexibilidad:** Cualquier cambio en el servicio de notificaciones implica cambios en la capa de negocio.

**RECOMENDACIÓN**

* Definir una interfaz INotificationService con métodos como enviarEmail() y enviarSMS().
* Hacer que BibliotecaService dependa de esa interfaz e inyectar la implementación concreta vía constructor o Spring @Autowired, usando inyección de dependencias.

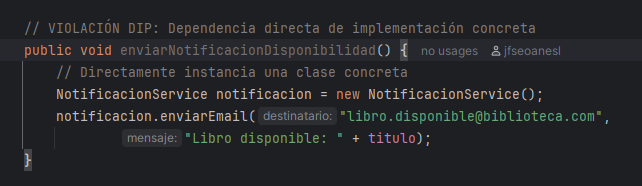
# HALLAZGO ENCONTRADO EN LA CAPA DE MODELO CLASE LIBRO

Tenemos que en la clase libro se viola el principio DIP

**DIP 1: LA CLASE LIBRO ES UN MÓDULO DE ALTO NIVEL DEPENDE DE UN MODULO DE BAJO NIVEL**

La violación del Principio de Inversión de Dependencias (DIP) en la clase Libro ocurre porque el módulo de alto nivel (la entidad de dominio Libro) depende directamente de un módulo de bajo nivel y concreto (NotificacionService), en lugar de depender de una abstracción (como una interfaz).

Específicamente, el método enviarNotificacionDisponibilidad() dentro de la clase Libro contiene la línea: NotificacionService notificacion = new NotificacionService();. Al usar la palabra clave new para crear una instancia del servicio de notificación, la entidad Libro queda fuertemente acoplada a la implementación específica de NotificacionService. Si en el futuro decidieras cambiar la forma en que se envían las notificaciones (por ejemplo, usando un SMSGateway o un servicio de colas), tendrías que modificar la clase Libro, lo cual contradice el DIP. El principio establece que los módulos de alto nivel no deben depender de los de bajo nivel, sino que ambos deben depender de abstracciones (interfaces), asegurando que los cambios en la implementación de la notificación no afecten al modelo de dominio.



Si quieres usar otro sistema de notificación como

* usar WhatsApp
* usar Kafka
* mandar notificaciones a Slack
* enviar un push notification
* reemplazar el servicio por uno mock para pruebas

Tendríamos que modificar la entidad Libro, lo cual es incorrecto porque:

* el dominio no debería cambiar por detalles técnicos
* violas también el principio OCP (Open/Closed)

**CONSECUENCIAS**

* **Acoplamiento fuerte** La clase Libro queda unida al servicio de notificaciones esto resulta en que al Cambiar una tecnología → obliga a cambiar el dominio.
* **Ruptura de la arquitectura en capas,** El flujo ideal debería ser Dominio → Aplicación → Infraestructura Pero al violar el DIP pasa esto Dominio ← Infraestructura (incorrecto)
* **El código deja de ser reutilizable.**

**RECOMENDACION**

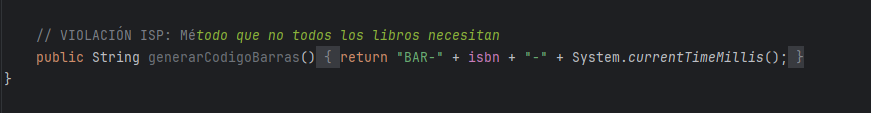
se recomienda crear una abstracción mediante una interfaz como *Notificador*, de modo que la entidad dependa de un contrato y no de una implementación concreta como *NotificacionService*.

1. **Principio ISP – Interface Segregation Principle**

El Principio ISP nos dice "Ningún cliente debe depender de métodos que no usa."

"Las interfaces deben ser pequeñas y específicas, no gigantes con métodos para todos."

**ISP-01: Método generarCodigoBarras() en la clase Libro**

****

la clase Libro viola el principio ISP debido a que implementa un método que no es necesario para todos los tipos de libros.

El método generarCodigoBarras() genera un código de barras basado en el ISBN y la hora actual. Sin embargo, no todos los libros requieren este comportamiento. Solo algunos tipos de libros pueden necesitar un código de barras, pero el modelo Libro está diseñado para representar libros en general, y esta funcionalidad puede no ser relevante para todas las implementaciones de libros, especialmente cuando se tienen libros digitales o de referencia, que no necesitan un código de barras.

El problema con la implementación de este método es que no todos los libros necesitan generar un código de barras, pero aún así, todos los objetos Libro lo tienen como método. Esto viola el ISP porque obliga a la clase Libro a exponer un método innecesario que no es utilizado por todos los tipos de libro.

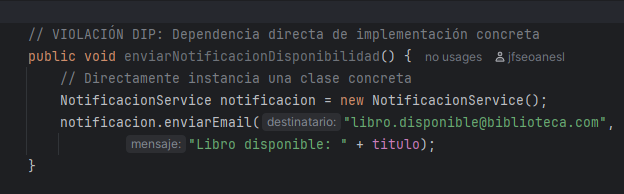
**CONSECUENCIAS**

* la clase libro se vuelve rígida y difícil de extender,si agregas un nuevo tipo de libro (por ejemplo, libro digital, manuscrito, revista), tendrás que arrastrar métodos que no necesita Cada nuevo tipo estará obligado a implementar o heredar funciones inútiles.
* los métodos que no se usan terminan generandocongestión en la clase y riesgo de que alguien los use de manera incorrecta esto ocaciona que las clases pierdan cohesión.
* Aumento del acoplamiento innecesario
* Si Libro contiene comportamientos que solo algunos objetos requieren SERA más difícil aislar responsabilidades

**RECOMENDACION**

**Crear una interfaz pequeña llamada TieneCodigoBarras con UN SOLO método.**  
Las clases que necesiten generar código la implementan.  
Las que no, simplemente no la implementan.

**VIOLACIÓN ISP-02: Método enviarNotificacionDisponibilidad() en la clase Libro**

****

El método enviarNotificacionDisponibilidad() envía una notificación cuando un libro está disponible. Esta acción es parte de una lógica que debería estar en un servicio de notificación, no en la clase Libro. La clase Libro no debe ser responsable de enviar notificaciones, ya que no todas las instancias de libros requerirán una notificación por su disponibilidad.

Esto también viola el ISP porque no todos los libros requieren enviar notificaciones. Además, el Libro depende directamente de la implementación concreta de NotificacionService, lo que también va en contra de los principios de bajo acoplamiento y diseño flexible.

**CONSECUENCIA**

Aumenta el aoplamiento El Prestamo depende directamente de NotificacionService, incluso si solo necesita enviar un email.

**RECOMENDACION**

crear interfazes pequeñas especificas