

**Énfasis 1: Patrones de diseño y principios SOLID**

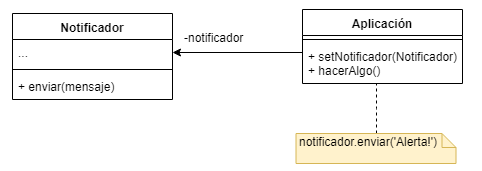
**Integrantes:**

* Juan Pablo Serna
* Alejandro Madrid
* Leonardo Vélez

**Problema:**

Suponga que usted trabaja en una empresa de desarrollo y le asignaron la tarea de modificar una librería de notificación que le permite a otros programas notificar a sus usuarios acerca de eventos importantes.

La versión inicial de la librería estaba basada en la clase Notificador, la cual tiene un método enviar. Este método puede aceptar un mensaje como parámetro de un cliente y enviar el mensaje a una lista de correos que fueron pasados al Notificador vía su constructor. Una aplicación de un tercero, que actúa como cliente, es la que debe crear y configurar el objeto Notificador una vez y luego los usa cada vez que algo importante ocurre.



En algún punto, usted se da cuenta que los usuarios de la librería esperan no solo correos de notificación. Muchos de ellos requieren recibir SMS, otros quieren recibir notificaciones vía Facebook y los clientes corporativos quizá quieran recibir notificaciones vía el sistema de mensajería empresarial. Modifique el diseño para esta situación, considerando que la librería se pueda extender para muchas más opciones de notificación de las mencionadas.

Más adelante algún cliente pregunta: ¿por qué no se pueden usar varios tipos de notificación al mismo tiempo? A usted le piden que modifique nuevamente el diseño para considerar dicha situación, tratando de reutilizar lo que se pueda y dejando el diseño flexible para poder incorporar nuevas opciones.

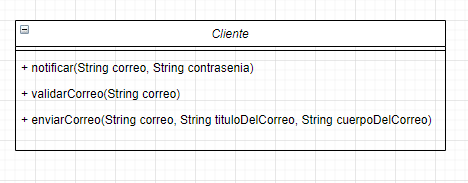
1. Análisis del problema:

Tenemos que enfocarnos en la flexibilidad y mantenibilidad del código, por consiguiente, agregar un nuevo medio de notificación no nos implique modificar las notificaciones anteriores o agregar dependencias entre notificaciones, el código debe funcionar si elimino ese medio de notificación ya que de este no dependen los otros, son aislados (desacoplados) e independientes entre ellos.

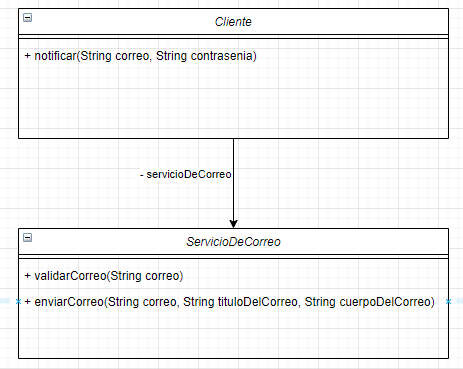
Lo haremos en el lenguaje de programación orientado a objetos (POO) llamado JAVA y es un proyecto java sencillo y básico, de solo clases sin componentes gráficos.

1. Versiones del diagrama de UML:

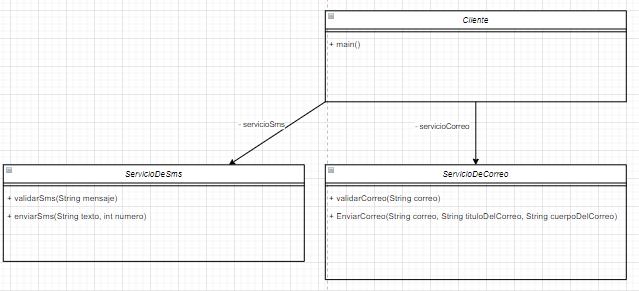
En esta primera versión el cliente es el que tiene la responsabilidad de enviar el correo, de validar, crear y configurar los objetos.



Así que decidimos mejorarlo, en vez de tener muchas responsabilidades, decidimos repartir estas responsabilidades en diferentes clases, por lo que aplicamos de los principios SOLID, el principio de única responsabilidad y en los patrones GRASP el patrón alta cohesión, donde el cliente solo tiene la responsabilidad de crear y configurar la notificación y el servicio de correo la responsabilidad de validar y enviar la notificación en este caso de correo.

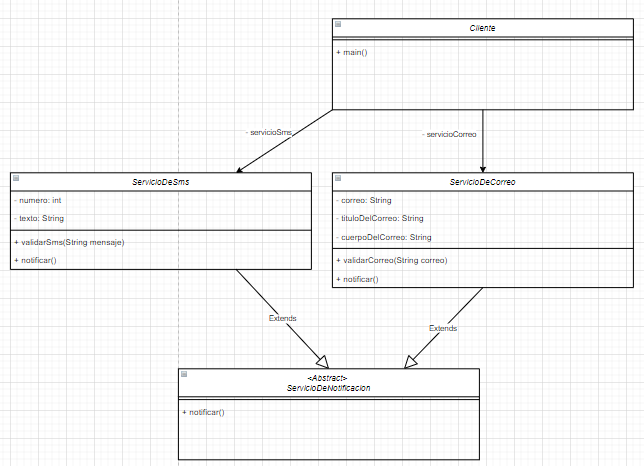


Aquí es donde entra en juego a la flexibilidad que es uno de nuestros objetivos, resulta ser que el cliente quiere enviar también notificaciones por mensajes de texto (SMS)

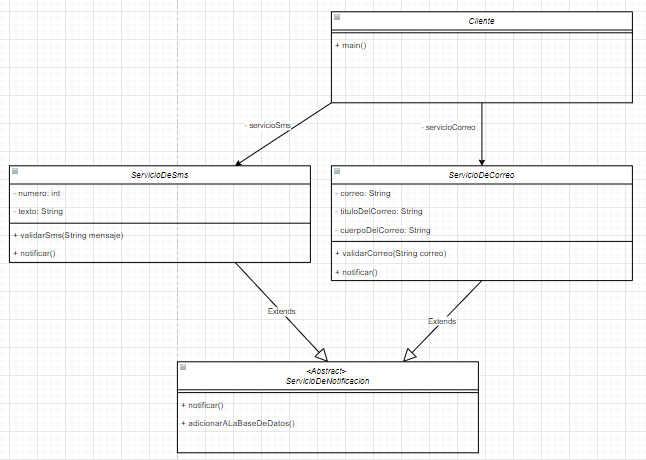


En este punto cada que queramos agregar un tipo de notificación nueva nos va tocar aplicar un condicional en la clase cliente, por lo que, si tenemos 15 diferentes tipos de notificaciones, nuestra clase cliente crecerá y se llenara de condicionales, lo que no es una buena práctica y nos resta flexibilidad y bajo acoplamiento, que es a lo que queremos llegar.

Por lo que aplicamos de los principios SOLID, el principio de abierto cerrado y en los patrones GRASP el patrón experto en información y alta cohesión, Ahora todos los tipos de notificaciones que agreguemos quedaran con un contrato y estandarizaremos más fácilmente, además de aplicar encapsulación, lo que nos ayuda para que cada clase tenga la información que necesita en sus métodos, aun nos llevamos la parte de agregar condicionales por cada tipo de notificación nueva, lo resolveremos más adelante.

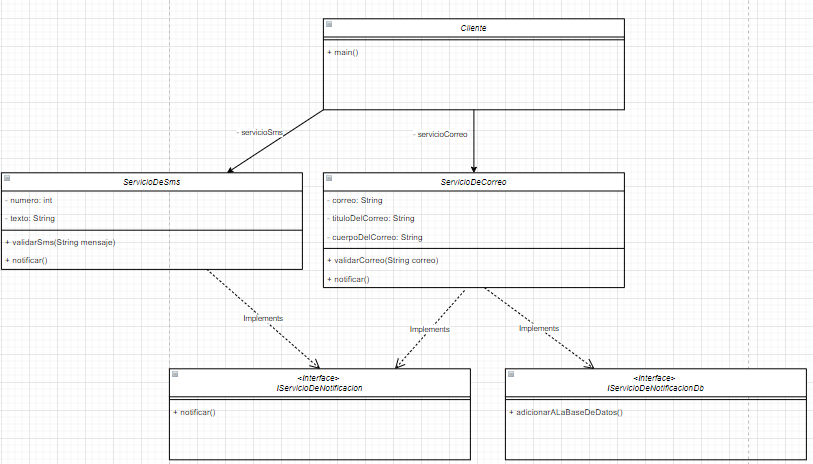


Ahora resulta que de pronto puede haber otro método que necesitemos implementar en nuestras notificaciones como por ejemplo adicionar a la base de datos el registro de que se envió la notificación, pero esto solo para notificaciones tipo email.

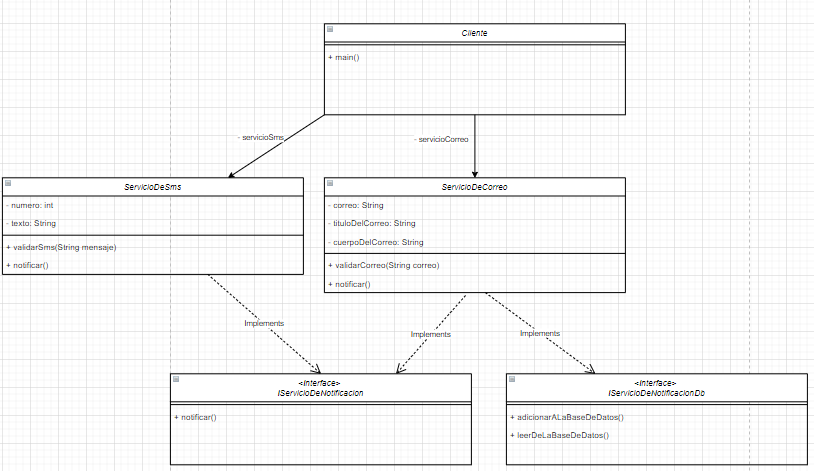


El problema aquí radica en que la clase ServicioDeSms obligatoriamente va a tener que implementar el método adicionarALaBaseDeDatos() así no lo use y esto no está bien porque va a ser una implementación vacía, sin código que funcione. Por lo que posteriormente aplicamos de los principios SOLID el principio de sustitución de liskov y en los patrones GRASP el patrón controlador y polimorfismo, para resolver este problema.

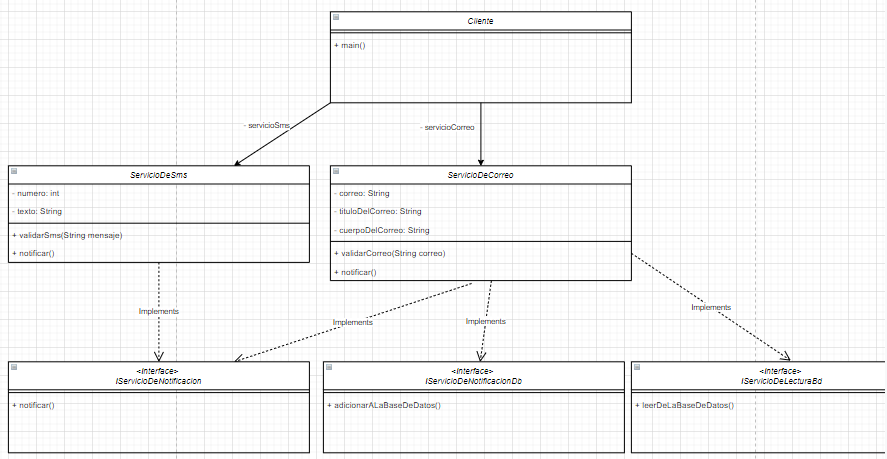
Vale aclarar que ya no podemos aplicar una herencia múltiple, porque en nuestro caso que utilizamos el lenguaje de programación java, entonces no nos permite implementarlo, por ello cambiamos de clase abstracta a interface, “tratando de simular una herencia múltiple”.



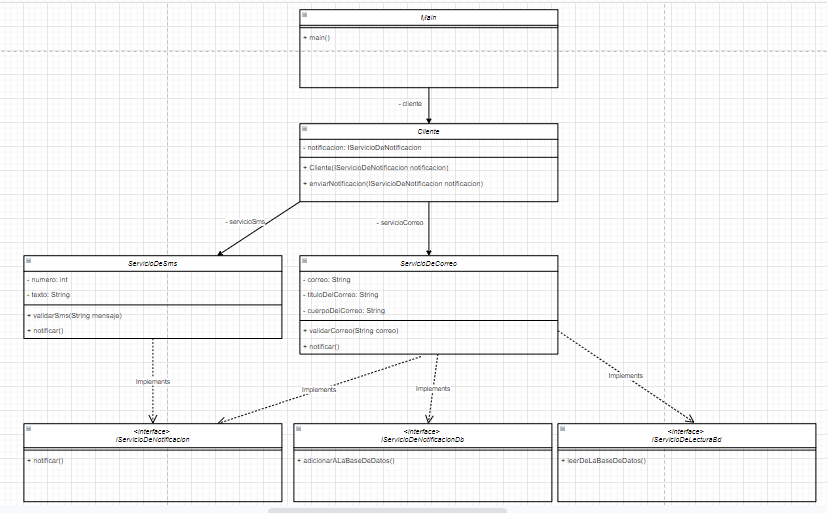
Ahora queremos implementar otro método en la interface IServicioDeNotificacionDb, el cual se va llamar leerDeLaBaseDeDatos



Ahora tenemos el problema de que el servicio de correo no tiene implementación de ese método leerDeLaBaseDeDatos por lo que es código sin funcionar o una implementación vacía, por lo tanto, aplicamos de los principios SOLID, el principio segregación de interfaces lo que nos permite dar más flexibilidad a nuestro código y reducir el acoplamiento de esta clase con la interfaz y en los patrones GRASP el patrón polimorfismo,

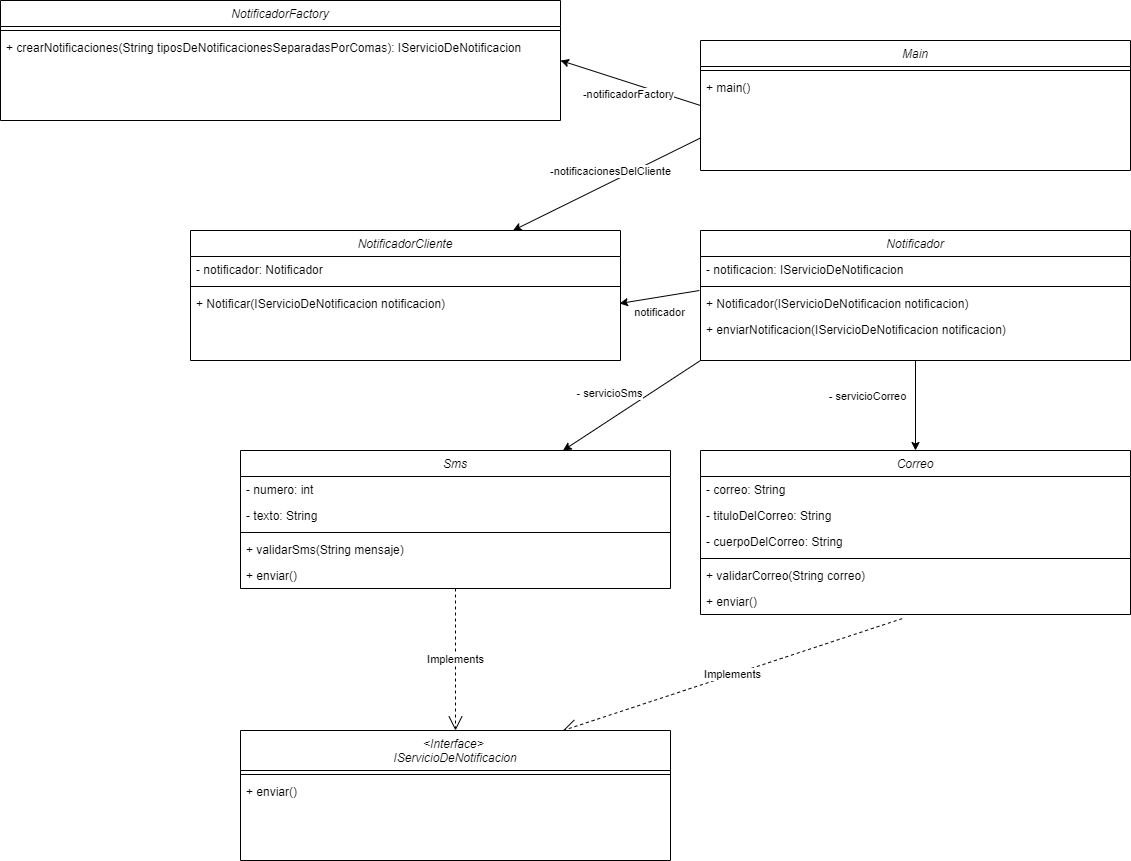


Nuestro problema ahora es hay mucho código en nuestra clase cliente y nuestros módulos de alto nivel dependen de los de bajo nivel y no es así, debería ser que los de bajo nivel deben dependan de la abstracción, es decir, vamos a separar y aislar ese IServicioDeNotificacion de la clase Cliente y lo separaremos en una clase nueva, así la clase Main solo quedara con la responsabilidad de configurar y crear las notificaciones.



Ahora hagamos un pequeño de refactoring sencillo para mejorar la reutilización de código y nombramiento de clases y métodos, basados en los patrones GRASP el patrón creador, finalmente quedara así:

Paquete: Librerianotificaciones.release:



Ahora simplemente tengo que crear las notificaciones en la clase Main y entregársela a NotificadorCliente y el hará el trabajo de enviarla, sin tener que saber la implementación la clase Main.

1. Video: Adjunto en carpeta documentación
2. Enlace al repositorio:

Repositorio GitHub:

<https://github.com/JuanPabloSerna/LibreriaDeNotificaciones>