**Ejemplo del Principio Open/Close y Patrones de Diseño Aplicados**

Este documento detalla un ejemplo práctico donde se aplica el Principio Open/Close y varios patrones de diseño, en un escenario de notificación. Se implementan notificaciones a través de diferentes canales, como correo electrónico, SMS y notificaciones push. La estructura sigue el principio Open/Close, permitiendo extender las funcionalidades sin modificar las clases existentes.

**Principio Open/Close.**

El Principio Open/Close es uno de los principios fundamentales de la programación orientada a objetos (OOP). Este principio establece que las clases deben estar abiertas para su extensión, pero cerradas para su modificación.

En este ejemplo, tenemos una interfaz común para todas las notificaciones (*NotificationInterface*) y clases concretas para cada tipo de notificación (*EmailNotification, SmsNotification, PushNotification*.). Al agregar un nuevo tipo de notificación, simplemente extendemos la interfaz y creamos una nueva clase que implemente el método de envío, sin modificar las clases existentes.

**Patrones de Diseño Aplicados:**

**Patrón Singleton.**

El Patrón Singleton es utilizado al registrar el *NotificationService* en el contenedor de servicios de Laravel. Este patrón asegura que solo haya una instancia del servicio en todo el ciclo de vida de la aplicación.

En el ejemplo, cuando registras el *NotificationService* en el contenedor de servicios usando singleton, aseguras que solo se cree una única instancia de *NotificationService* a lo largo de la ejecución de la aplicación, independientemente de cuántas veces se solicite el servicio. Esto es útil cuando no necesitas que se creen múltiples instancias del mismo servicio y quieres compartir el estado o la configuración a lo largo de la aplicación.

**Patrón Dependency Injection (Inyección de Dependencias).**

Aunque este patrón se utiliza principalmente en el contexto de Laravel, también es uno de los patrones de diseño aplicados en este ejemplo.

La inyección de dependencias permite que las clases obtengan sus dependencias (en este caso, las estrategias de notificación) a través del constructor en lugar de crearlas directamente dentro de la clase. Laravel se encarga de inyectar las dependencias en el *NotificationController* a través de su contenedor de servicios, lo que promueve la desacoplamiento y mejora la testabilidad.

**Patrón Strategy.**

El patrón de diseño Strategy se aplica en este caso porque hemos separado la lógica de notificación en diferentes clases de notificación concretas (*EmailNotification*, *SmsNotification*, *PushNotification*). El *NotificationService* utiliza estos 'estrategas' para enviar las notificaciones sin necesidad de conocer la implementación específica de cada tipo. Esta separación permite cambiar o agregar nuevas estrategias sin alterar el código cliente.

**Patrón Composite.**

El patrón Composite también se ve reflejado en este ejemplo, ya que el *NotificationService* actúa como un 'composite'. Este patrón permite tratar a un grupo de objetos (en este caso, los diferentes tipos de notificaciones) de manera uniforme. El *NotificationService* maneja todos los *notifiers* (Email, SMS, Push) como un conjunto de objetos que implementan la interfaz *NotificationInterface*, enviando notificaciones de manera agregada a través de todos ellos.

**Patrón Factory.**

El patrón Factory es evidente en la forma en que las notificaciones son creadas y gestionadas. En lugar de instanciar directamente las clases de notificación en el controlador, utilizamos el contenedor de servicios de Laravel para crear instancias de las clases concretas. Esto permite crear objetos de manera flexible y desacoplada, facilitando la extensión del sistema para admitir nuevos tipos de notificación en el futuro.

**Conclusión.**

En este ejemplo, hemos implementado el principio Open/Close y varios patrones de diseño, lo que ha resultado en un sistema extensible y mantenible. Gracias al uso de interfaces y clases concretas para cada tipo de notificación, el sistema permite extensiones de nuevas funcionalidades sin necesidad de modificar las clases existentes. Los patrones Strategy, Composite y Factory han ayudado a estructurar este sistema de manera eficiente, demostrando cómo se pueden aplicar los principios de diseño en un proyecto real.

**Probar el sistema.**

* Primero levantar el servicio con php artisan serve.
* Luego ingresar a la ruta que nos devuelve
* Ingresar la ruta en la que estamos mostrando los resultados:/send-notification

Para cambiar los datos que se estan ingresando se debe ir al controlador, para ingresar el mensaje que queremos enviar y los receptores a los que queremos enviarlo.