**Ejemplo del Principio de Liskov con Estrategias de Envio de pedidos**

**Principio de Sustitución de Liskov.**

El Principio de Sustitución de Liskov (LSP) establece que los objetos de una clase derivada deben ser sustituibles por objetos de la clase base sin alterar el comportamiento esperado del sistema. Es decir, una clase hija debe ser capaz de realizar todo loque la clase base puede hacer, sin causar efectos no deseados o errores.

En este ejemplo, las clases *`HomeDeliveryShipping`* y *`PickupInStoreShipping`* implementan la misma interfaz *`ShippingStrategy`.* Ambas clases proporcionan implementaciones de los métodos *`calculateCost()`* y *`getDeliveryTime()`*, lo que garantiza que puedan ser utilizadas indistintamente en el sistema sin causar problemas.

**Clases e Implementaciones** -

**`*ShippingStrategy`*** es la interfaz que define los método*s `calculateCost()`* y *`getDeliveryTime()`,* asegurando que cualquier clase que la implemente cumpla con los contratos establecidos.

***`HomeDeliveryShipping` y `PickupInStoreShipping`*** son implementaciones de esta interfaz. Cada una tiene una lógica diferente, pero ambas cumplen con la firma de los métodos definidos en *`ShippingStrategy`*.

**`*ShippingService*`** es el servicio que interactúa con las estrategias de envío, utilizando la inyección de dependencias para recibir una instancia de *`ShippingStrategy*`. Este servicio encapsula la lógica de negocio relacionada con el cálculo de costos y tiempos, delegando en las estrategias concretas el comportamiento específico.

**Aplicación de Liskov en este Proyecto.**

El principio de Liskov se aplica de la siguiente manera:

1. Interfaz común: *`ShippingStrategy`* define un contrato común que debe ser seguido por todas las estrategias de envío. Esto garantiza que cualquier clase que implemente esta interfaz sea sustituible por otra sin alterar la lógica de cálculo en el servicio.
2. Sustitución segura: Las clases *`HomeDeliveryShipping`* y *`PickupInStoreShipping`* pueden ser intercambiadas sin ningún problema en el `ShippingService`, y este seguirá funcionando de la misma manera, sin depender de detalles específicos de cada clase.
3. Inyección de dependencias: Se utiliza el contenedor de servicios de Laravel para resolver la dependencia de *`ShippingStrategy*` en el controlador, lo que permite cambiar la estrategia de envío en cualquier momento sin modificar el controlador o el servicio.

**Conclusión.** Este ejemplo muestra cómo el Principio de Sustitución de Liskov puede aplicarse eficazmente en un proyecto Laravel utilizando la estrategia de diseño de patrones, y cómo la inyección de dependencias facilita el cambio dinámico de comportamientos sin alterar el funcionamiento global del sistema.

**Patrones de Diseño Aplicados a Este Ejemplo.**

**1. Strategy Pattern (Patrón de Estrategia).** El patrón de estrategia permite definir un conjunto de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Este patrón es útil para situaciones donde diferentes comportamientos pueden ser aplicados dependiendo de un contexto.

* Interface: *ShippingStrategy.* Define el contrato común que todas las estrategias deben implementar *(calculateCost* *y getDeliveryTime)*.
* Clases concretas:*HomeDeliveryShipping* y *PickupInStoreShipping* son implementaciones específicas de la estrategia de envío.
* Contexto: La clase *ShippingService* actúa como el contexto que utiliza la estrategia seleccionada para realizar cálculos de costo y tiempo.

Este patrón permite cambiar dinámicamente entre HomeDeliveryShipping y PickupInStoreShipping sin modificar el código de la lógica principal.

### **2. Dependency Injection (Inyección de Dependencias).** Es un principio fundamental que promueve la inversión de control, donde las dependencias de una clase son inyectadas desde el exterior en lugar de ser instanciadas directamente dentro de la clase.

* Controlador: *ShippingController* no crea directamente una instancia de *ShippingService.* En cambio, se la inyecta como dependencia en su constructor.
* ServiceProvider: Laravel utiliza el contenedor de dependencias para resolver e inyectar automáticamente la implementación de *ShippingStrategy* según la configuración dinámica en *ShippingStrategyProvider.*

Esto mejora la flexibilidad y testabilidad del código.

### **3. Service Layer (Capa de Servicio).** El patrón de capa de servicio organiza la lógica de negocio en una capa específica que actúa como intermediaria entre el controlador y otras partes del sistema.

* Clase: *ShippingService.* Encapsula la lógica para calcular los costos y tiempos de envío usando la estrategia seleccionada. Esto separa la lógica del negocio del controlador, haciéndolo más limpio y enfocado.

La capa de servicio centraliza y abstrae la lógica de negocio, lo que permite su reutilización en otros controladores o contextos.

**4. Factory Method (Método de Fábrica).** Es un patrón que proporciona una interfaz para crear objetos en una superclase, pero permite a las subclases alterar el tipo de objetos que se crean.

* ServiceProvider: *ShippingStrategyProvider.* Actúa como una fábrica que decide cuál estrategia de envío (*HomeDeliveryShipping o PickupInStoreShipping*) se debe instanciar en función del valor de delivery\_type.

Esto elimina la necesidad de escribir lógica condicional repetida en diferentes partes del sistema y centraliza la creación de objetos.

### **Relación con los Principios SOLID:**

* Single Responsibility Principle: Cada clase tiene una única responsabilidad:
  + *ShippingController* maneja la interacción con la vista.
  + *ShippingService* se encarga de la lógica del negocio.
  + *ShippingStrategyProvider* gestiona la resolución de la estrategia.
  + Las clases concretas(*HomeDeliveryShipping, PickupInStoreShipping*) encapsulan comportamientos específicos de envío.
* Open/Closed Principle: El sistema está abierto a nuevas estrategias de envío sin modificar el código existente. Basta con añadir una nueva implementación de *ShippingStrategy*.
* Liskov Substitution Principle: Las clases concretas pueden ser usadas indistintamente donde se espere la interfaz *ShippingStrategy.*

Este ejemplo es una demostración clara de cómo varios patrones y principios pueden trabajar juntos para construir un sistema flexible, extensible y fácil de mantener.

**Para probar el sistema.**

Una vez se tenga clonado el proyecto y abierto en un editor de texto:

* Instalar dependencias, en una terminal navegar a la dirección de la carpeta del proyecto y ejecutar el siguiente comando comando para instalar todas las dependencias de Composer: composer install
* Configurar el Archivo .env , El siguiente paso es configurar el archivo .env, que contiene las variables de entorno necesarias para la aplicación. Copiar el archivo .env.example y renombrarlo como .env**.**
* Generar la Clave de la Aplicación Laravel requiere una clave de aplicación única para funcionar correctamente. Se puede generar esta clave ejecutando el siguiente comando: php artisan key:generate
* Asegurarse de tener el proveedor *ShippingStrategyProvider* registrado en el archivo de config/app.
* Levantar el servicio con php artisan serve e ingresar a la ruta que nos devuelve.
* Ingresar la ruta en la que estamos mostrando los resultados: /shipping-test.

Se puede seleccionar la estrategia de envío que se desea utilizar, entre delivery a la casa o recogerlo directamente en la tienda.  
  
Y observar como ambas clases ejecutan los mismos métodos con resultados diferentes.