**Ejemplo de los Principios DIP y ISP con un sistema de gestión de tareas**

**1. Principio de Inversión de Dependencias (DIP)**

El **Principio de Inversión de Dependencias (DIP)** establece que las clases de alto nivel no deben depender de las clases de bajo nivel, sino de las abstracciones. Las abstracciones no deben depender de los detalles, sino los detalles de las abstracciones. Este principio ayuda a desacoplar las clases de la implementación concreta, lo que facilita el mantenimiento y la extensión del código.

**En el proyecto**, el **DIP** está siendo aplicado al inyectar interfaces en el controlador. Las interfaces TaskManagementInterface y TaskPriorityInterface se inyectan en el controlador TaskController, lo que permite que el controlador dependa solo de las abstracciones y no de las implementaciones concretas de los servicios.

**¿Cómo se demuestra?**

1. **Revisión de la Inyección de Dependencias**: Las interfaces TaskManagementInterface y TaskPriorityInterface están siendo inyectadas en el controlador, lo que desacopla el controlador de las implementaciones concretas y hace el código más flexible.
2. **Sustitución de Implementaciones**: Al estar utilizando interfaces, las implementaciones concretas pueden ser cambiadas fácilmente sin modificar el código del controlador, demostrando cómo el código está desacoplado y es flexible para cambios futuros.

La inyección de dependencias en el controlador se logra gracias a la referencia de las interfaces a sus implementaciones concretas en el **AppServiceProvider**. Dentro de este proveedor de servicios, utilizamos el método bind para asociar las interfaces TaskManagementInterface y TaskPriorityInterface con sus respectivas clases de servicio (TaskManagerService y TaskPriorityService). Esto significa que, cuando el controlador necesita una instancia de estas interfaces, Laravel se encarga de inyectar las instancias de los servicios correspondientes automáticamente.

**2. Principio de Segregación de Interfaces (ISP)**

El **Principio de Segregación de Interfaces (ISP)** establece que las interfaces deben ser específicas para las necesidades de los clientes, y no generales. Este principio sugiere que una clase no debe estar obligada a implementar métodos que no va a utilizar.

**En el proyecto**, las interfaces están bien divididas y especializadas. La interfaz TaskManagementInterface maneja las operaciones relacionadas con la gestión de tareas, mientras que TaskPriorityInterface maneja las operaciones relacionadas con las prioridades de las tareas.

**¿Cómo se demuestra?**

1. **Interfaces Especializadas**: En lugar de tener una sola interfaz que cubra múltiples responsabilidades, se han creado interfaces específicas para cada responsabilidad: gestión de tareas (TaskManagementInterface) y asignación de prioridades (TaskPriorityInterface).
2. **Evitar la Sobre Implementación**: Gracias a la separación de interfaces, si en el futuro un servicio necesita gestionar solo las prioridades de las tareas, puede implementar únicamente TaskPriorityInterface, sin necesidad de implementar métodos de gestión de tareas.

#### 3. Cambio en el Archivo .env para Selección de Servicios

En el archivo .env, se tiene la posibilidad de elegir entre dos implementaciones del servicio de prioridad, lo cual permite cambiar el comportamiento del sistema sin modificar el código fuente. Este cambio está relacionado con el **Principio de Inversión de Dependencias (DIP)**, ya que permite que el controlador se mantenga desacoplado de las implementaciones concretas, eligiendo entre diferentes implementaciones sin tener que modificar la lógica del controlador.

Por ejemplo, en el archivo .env, se puede configurar el servicio de prioridad de la siguiente manera:

# Priority service settings:

TASK\_PRIORITY\_SERVICE=App\Services\AriPriorityService

TASK\_PRIORITY\_SERVICE=App\Services\TaskPriorityService

El controlador no necesita conocer detalles sobre cuál de los dos servicios se está utilizando. Gracias a la inyección de dependencias y el archivo .env, se elige qué servicio se usará de forma centralizada, lo que cumple con el principio **DIP** de desacoplar las dependencias y hacer el código flexible y escalable.

**Diferencias entre ambos servicios:**

* **AriPriorityService**: Ofrece un sistema con más rangos de prioridad, ideal para situaciones en las que se necesitan más categorías detalladas de prioridad.
* **TaskPriorityService**: Es más simple, con tres rangos de prioridad, ideal para proyectos que requieren un sistema más directo y menos granular.

Ambos servicios implementan la misma interfaz (TaskPriorityInterface), por lo que el código que los utiliza puede cambiar entre ellos sin afectar el comportamiento general del sistema.

#### Patrones de Diseño Aplicados a Este Ejemplo

1. **Factory Method (Método de Fábrica)**: El patrón **Factory** se utiliza cuando se desea delegar la creación de objetos a un método específico, generalmente en lugar de crear objetos directamente en el código. En el código, la lógica de creación de tareas está utilizando implícitamente este patrón en el método createTask() del servicio TaskManagerService. Este servicio abstrae la lógica de creación de tareas, asegurando que se instancie correctamente sin que el controlador tenga que preocuparse por ello.
2. **Dependency Injection (Inyección de Dependencias)**: La **Inyección de Dependencias** aplicada al inyectar servicios como TaskManagerService y TaskPriorityService es una práctica que encaja con patrones como **Abstract Factory** o **Dependency Injection Container**. Este patrón mejora la modularidad y facilita el uso de clases y servicios de manera independiente, al inyectar las dependencias necesarias en lugar de crearlas directamente dentro de una clase.

**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**.**
* Asegurarse de crear la base de datos con el nombre que está en .env y generar las migraciones: php artisan migrate
* 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
* Levantar el servicio con php artisan serve e ingresar a la ruta que nos devuelve.
* Al crear una nueva tarea se puede ver cómo el sistema maneja las responsabilidades separadas al crear la tarea (gestionada por *TaskManagerService*) y asignar la prioridad automáticamente (gestionada por *TaskPriorityService*).

Con la configuración en .env TASK\_PRIORITY\_SERVICE= se puede definir que servicio se desea manejar para la prioridad de las tareas.