**Ejemplo de los Principios DIP y ISP con un sistema de gestion 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.

**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.

**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 *TaskManager*. Este servicio puede tener un método que abstrae la lógica de creación de la tarea y se asegura de que se instancie correctamente sin que el controlador tenga que preocuparse por la creación de la tarea en sí.

**2. Dependency Injection (Inyección de Dependencias).** La inyección de dependencias aplicada al inyectar servicios como *TaskManager* y *TaskPriorityService* es una práctica que encaja con patrones como Abstract Factory o Dependency Injection Container.

* Este patrón se refiere a inyectar las dependencias necesarias (como servicios o repositorios) en lugar de crearlas directamente dentro de una clase. Esto mejora la modularidad y facilita el uso de clases y servicios de manera independiente.

**Para probar el sistema.**

* Primero asegurarse de crear la base de datos con el nombre que esta en .env y generar las migraciones.
* Levantar el servicio con php artisan serve e ingresar a la ruta que nos devuelve.
* Ingresar la ruta de la vista index: /tasks