ACTIVIDAD 1

A la hora de diseñar la arquitectura de cualquier plataforma digital, aplicar buenos principios de desarrollo no es un capricho, sino una necesidad si queremos que el código sea mantenible, escalable y fácil de entender. En este caso, la plataforma de gestión de proyectos con inteligencia artificial no es una excepción. Para lograr un diseño sólido, utilizaremos cuatro principios fundamentales: SOLID, DRY, KISS y YAGNI. Cada uno aporta algo distinto, pero juntos permiten construir un sistema robusto y sin complicaciones innecesarias.

Aplicación de SOLID

Los principios SOLID ayudan a que el código sea modular y flexible. Empezamos con el Principio de Responsabilidad Única (SRP), que nos dice que cada clase o módulo debe hacer solo una cosa. Imagina que tenemos una clase GestorDeProyectos que se encarga tanto de crear proyectos como de enviarlos por correo y generar informes. Esto es un error común que puede llevar a problemas cuando queramos cambiar solo una parte del sistema. La solución sería dividirla en clases más pequeñas, como GestorDeTareas, Notificador e InformeGenerador, cada una con su función específica.

Luego está el Principio Abierto/Cerrado (OCP), que dice que el código debe poder ampliarse sin necesidad de modificarse. Supongamos que nuestro sistema necesita soportar distintos algoritmos para predecir retrasos en los proyectos. En lugar de modificar una clase central cada vez que añadimos uno nuevo, podemos definir una interfaz EstrategiaDePrediccion e implementar distintas clases para cada algoritmo, manteniendo el código base intacto.

El Principio de Sustitución de Liskov (LSP) nos recuerda que una subclase debe poder sustituir a su clase base sin romper el código. Si creamos una clase Usuario y luego Administrador y Empleado como subclases, estas deben comportarse de manera coherente. Si Administrador tiene un método extra que Empleado no tiene, pero intentamos usar ambos de la misma manera en el código, podríamos encontrarnos con errores inesperados.

La Segregación de Interfaces (ISP) evita que las clases dependan de métodos que no usan. Si definimos una interfaz Usuario con métodos como aprobarProyecto() y completarTarea(), un Empleado podría implementar la interfaz, pero no debería tener el método aprobarProyecto() si esa acción solo la realiza un Administrador. La solución sería dividir la interfaz en dos más específicas.

Por último, el Principio de Inversión de Dependencias (DIP) nos dice que los módulos de alto nivel no deberían depender de los de bajo nivel, sino de abstracciones. Es decir, en lugar de que GestorDeProyectos dependa directamente de una base de datos MySQL, debería depender de una abstracción RepositorioDeProyectos, permitiendo cambiar la implementación sin tocar la lógica de negocio.

Evitar la repetición con DRY

El principio DRY (Don't Repeat Yourself) es simple: si ves el mismo código en más de un sitio, algo estás haciendo mal. Copiar y pegar puede parecer una solución rápida, pero cuando

necesitas hacer un cambio, debes modificarlo en varios lugares, lo que aumenta el riesgo de errores. Para evitarlo, crearemos funciones y módulos reutilizables. Por ejemplo, si en varios sitios del sistema calculamos fechas de entrega ajustadas a la productividad del equipo, deberíamos extraer esa lógica a una función calcularFechaEntrega() en un módulo central en lugar de repetir la misma fórmula en distintos puntos del código.

Mantener la simplicidad con KISS

KISS (Keep It Simple, Stupid) nos dice que cuanto más simple sea el código, mejor. No significa escribir menos líneas de código, sino evitar estructuras complejas innecesarias. Un error típico es sobreingeniar una solución: añadir patrones de diseño cuando no hacen falta o crear abstracciones excesivas "por si acaso". Un ejemplo claro sería el abuso de fábricas para crear objetos cuando un simple constructor es suficiente. La clave es preguntarse siempre: "¿Esta solución es lo más sencilla posible sin perder funcionalidad?".

No programar lo innecesario con YAGNI

YAGNI (You Ain't Gonna Need It) es una advertencia contra la tentación de anticiparse a necesidades futuras. A veces, en un afán de "preparar el código para el futuro", se desarrollan funcionalidades que nunca se usan. Si al diseñar la plataforma decidimos incluir soporte para múltiples monedas, pero en realidad solo se trabajará en euros, estamos desperdiciando tiempo y esfuerzo. La mejor práctica es programar solo lo que realmente se necesita y añadir mejoras cuando sean necesarias, no antes.