# Buenas prácticas de desarrollo de software: Principios SOLID

Esta sección presenta una introducción a los 5 principios SOLID los cuales ayudan a desarrollar sotfware de calidad.

Cuando se trata del diseño y desarrollo de aplicaciones, los 'Principios SOLID' son un conjunto de conceptos esenciales que debes tener en tu repertorio como fundamentos clave en la arquitectura y creación de software.

SOLID es un acrónimo creado por Michael Feathers, que se basa en los principios de la programación orientada a objetos compilados por Robert C. Martin en su artículo de 2000, titulado 'Design Principles and Design Patterns'. Estos principios sientan las bases para el desarrollo de software de alta calidad y mantenible en el mundo de la programación orientada a objetos[1)].

Los 5 principios SOLID de diseño de aplicaciones de software son:

- S – Single Responsibility Principle (SRP)
- O – Open/Closed Principle (OCP)
- L – Liskov Substitution Principle (LSP)
- I – Interface Segregation Principle (ISP)
- D – Dependency Inversion Principle (DIP)

## S - Principio de responsabilidad única

El Principio de Responsabilidad Única establece que una clase debe desempeñar una única función y, en consecuencia, tener un solo motivo para cambiar.

Para expresar este principio en términos más técnicos: Solo debería ser posible que un cambio potencial, como alteraciones en la lógica de la base de datos o en la lógica de registro, afecte la especificación de una clase.

En otras palabras, si una clase representa una entidad o un contenedor de datos, como una clase “Libro” o “Estudiante,” y contiene campos relacionados con esa entidad, solo debería requerir modificaciones cuando el modelo de datos subyacente cambie.

Es fundamental adherirse al principio de responsabilidad única por varias razones. En primer lugar, en proyectos donde múltiples equipos pueden trabajar en la misma clase por diversas razones, no seguir este principio podría resultar en módulos incompatibles.

En segundo lugar, facilita el seguimiento de versiones. Por ejemplo, si observamos cambios en un archivo en las confirmaciones de GitHub y seguimos el Principio de Responsabilidad Única, podemos inferir que esos cambios están relacionados con el almacenamiento o cuestiones vinculadas a la base de datos.

Además, este enfoque también reduce los conflictos de fusión. Los conflictos suelen surgir cuando diferentes equipos modifican el mismo archivo, pero al adherirse al Principio de Responsabilidad Única, los conflictos se minimizan, ya que los archivos solo tienen un motivo para cambiar, lo que

simplifica la resolución de conflictos.

# O - Principio de apertura y cierre

El principio de apertura y cierre establece que las clases deben ser abiertas a la extensión y cerradas a la modificación.

La 'modificación' se refiere a cambiar el código de una clase existente, mientras que la 'extensión' implica agregar nuevas funcionalidades. En resumen, este principio promueve la idea de que debemos poder introducir nuevas funciones sin necesidad de alterar el código preexistente de una clase. Esto se debe a que cada vez que modificamos el código existente, corremos el riesgo de introducir posibles errores. Por lo tanto, es aconsejable evitar tocar el código de producción que ya ha sido probado y es confiable en su mayoría.

Puede preguntarse cómo es posible agregar nueva funcionalidad sin modificar la clase original. Por lo general, esto se logra mediante el uso de interfaces y clases abstractas, que permiten extender y agregar nuevas capacidades a las clases existentes sin cambiar su implementación subyacente.

# L - Principio de sustitución de Liskov

El Principio de Sustitución de Liskov establece que las subclases deben ser intercambiables por sus clases base.

En otras palabras, si la clase B es una subclase de la clase A, deberíamos poder utilizar un objeto de la clase B en lugar de un objeto de la clase A en cualquier contexto, como un método que espera un objeto de la clase A, sin experimentar resultados inesperados.

Este comportamiento es el esperado ya que cuando aplicamos la herencia, suponemos que la clase hija hereda todas las características de la clase madre. La clase hija puede extender el comportamiento, pero nunca debe reducirlo.

Cuando una clase no cumple con este principio, puede llevar a errores inesperados y difíciles de detectar en el código.

# I - Principio de segregación de interfaces

La segregación implica la separación de elementos, y el Principio de Segregación de Interfaces se enfoca en la separación de interfaces.

Este principio defiende que es preferible tener múltiples interfaces específicas para los clientes en lugar de una única interfaz general. La idea es no obligar a los clientes a implementar funciones que no necesitan ni utilizarán.

# D - Principio de inversión de dependencia

El principio de inversión de dependencia establece que nuestras clases deben depender de interfaces o clases abstractas en lugar de depender de clases y funciones concretas.

En su artículo de 2000, Robert C. Martin resume este principio de la siguiente manera:

…“Si el Principio de Apertura y Cierre establece el objetivo de la arquitectura orientada a objetos, el Principio de Inversión de Dependencia establece el mecanismo principal”.

Estos dos principios están estrechamente relacionados, y ya hemos aplicado este patrón al discutir el Principio de Apertura y Cierre.

El objetivo es que nuestras clases estén abiertas a la extensión, por lo que reestructuramos nuestras dependencias para que dependan de interfaces en lugar de depender de clases y funciones concretas.

←Volver atras

1)
https://profile.es/blog/principios-solid-desarrollo-software-calidad/