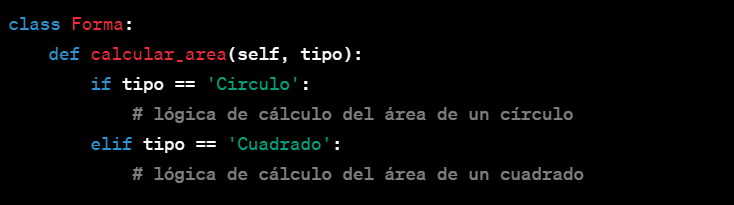
**Principios de SOLID diseño de Software**

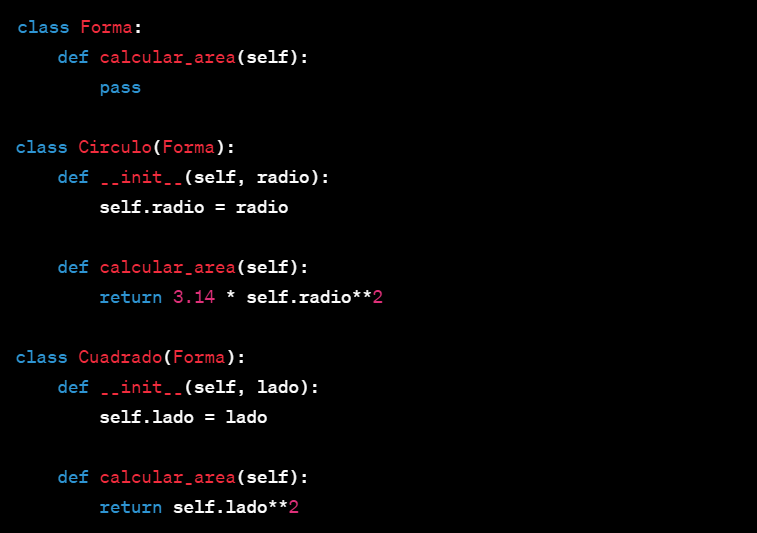
1. **Principio de Responsabilidad Única (SRP - Single Responsibility Principle):**

El principio de responsabilidad única aboga por asignar a cada clase una única responsabilidad en el software, encapsulando completamente dicha responsabilidad. Su objetivo principal es reducir la complejidad del código, especialmente cuando el programa crece y cambia con el tiempo. Evita la pérdida de control al tener clases enormes y facilita la navegación del código al mantener la encapsulación de las responsabilidades.

No se debe de hacer



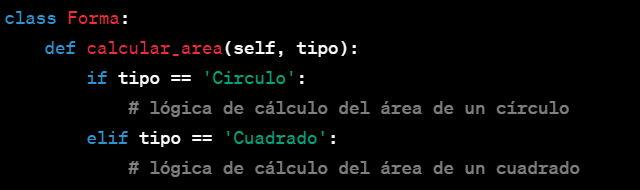
Si se debe de hacer



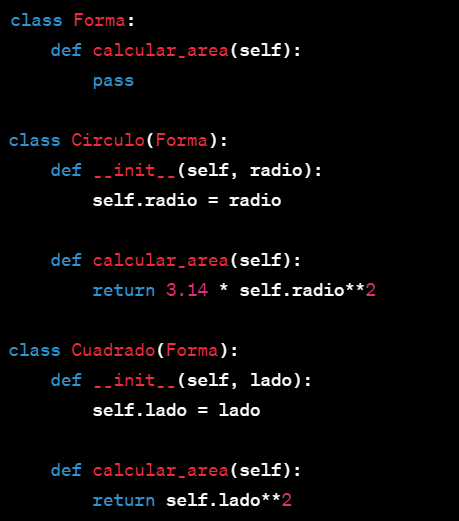
1. **Principio de Abierto/Cerrado (OCP - Open/Closed Principle):**

Este principio busca evitar que la implementación de nuevas funciones descomponga el código existente. Una clase puede ser abierta para la extensión y cerrada para la modificación. En lugar de alterar directamente el código de una clase existente, se fomenta la creación de subclases que sobrescriban las partes necesarias, permitiendo la extensión sin afectar a los clientes existentes. No obstante, no se debe aplicar a todos los cambios; correcciones de errores deben abordarse directamente.

No se debe de hacer



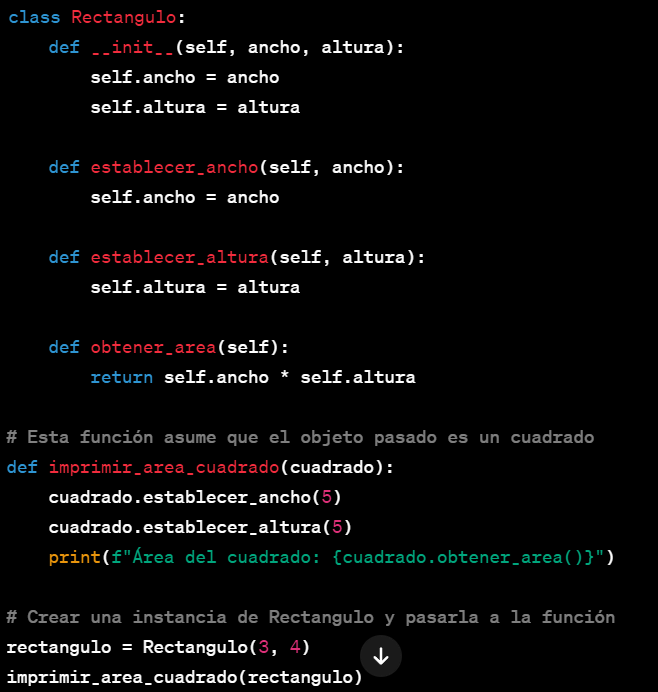
Si se debe de hacer



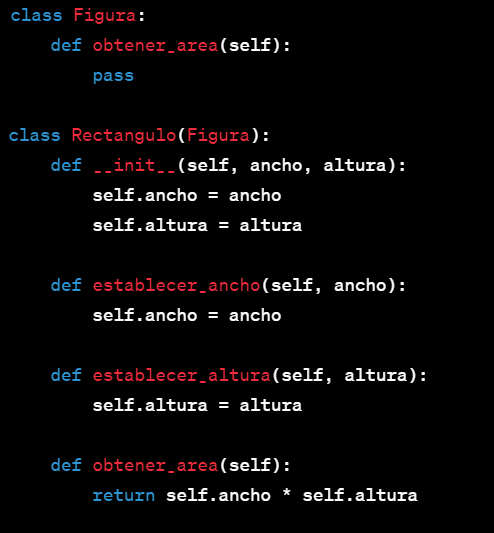
1. **Principio de Sustitución de Liskov (LSP - Liskov Substitution Principle):**

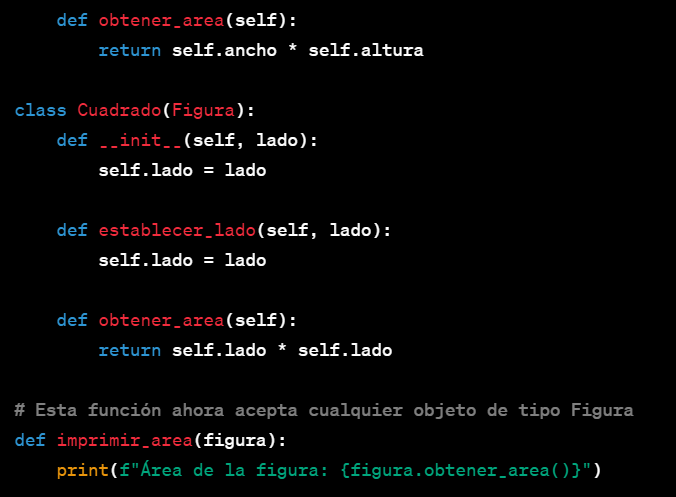
Este principio establece que una subclase debe ser compatible con el comportamiento de la superclase. Al sobrescribir métodos, se debe extender el comportamiento base en lugar de reemplazarlo completamente. Impone requisitos formales a las subclases, como la coincidencia de tipos de parámetros, el tipo de retorno y la preservación de invariantes, asegurando que las subclases no introduzcan cambios incompatibles.

No se debe de hacer



Si se debe de hacer

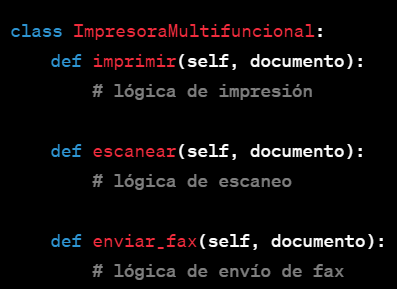




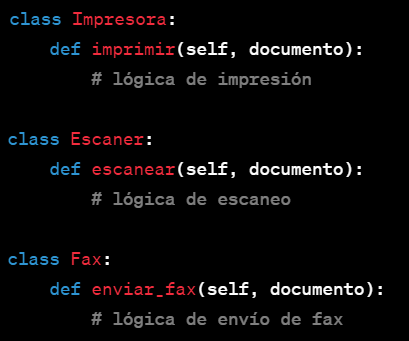
1. **Principio de Segregación de la Interfaz (ISP - Interface Segregation Principle):**

Busca crear interfaces estrechas para que las clases del cliente no implementen funcionalidades innecesarias. Propugna dividir interfaces gruesas en otras más detalladas y específicas. La herencia de clases permite implementar varias interfaces a la vez, evitando la necesidad de agrupar métodos no relacionados en una única interfaz. Clientes implementarán solo los métodos que necesiten.

No se debe de hacer



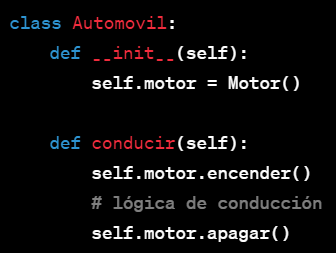
Si se debe de hacer



1. **Principio de Inversión de la Dependencia (DIP - Dependency Inversion Principle):**

Este principio sostiene que las clases de alto nivel no deben depender de las clases de bajo nivel. Ambas deben depender de abstracciones, y los detalles deben depender de las abstracciones. Se propone describir interfaces en términos de operaciones de bajo nivel, haciendo que las clases de alto nivel dependan de estas interfaces, invirtiendo así la dependencia original. Esto se combina con el principio de abierto/cerrado, permitiendo la extensión sin descomponer clases existentes.

No se debe de hacer



Si se debe de hacer

