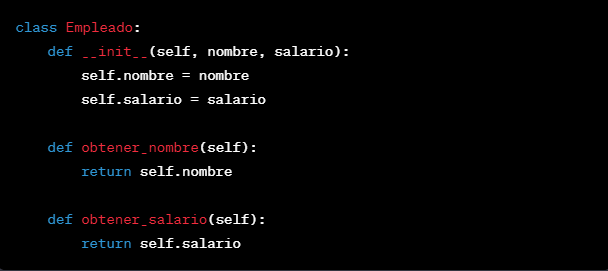
Principio de responsabilidad única

Intenta hacer a cada clase responsable de una única parte de la funcionalidad proporcionada por el software, y haz que esa responsabilidad quede totalmente encapsulada por (también puedes decir escondida dentro de) la clase.

El principal objetivo de este principio es reducir la complejidad. No hace falta que inventes un diseño sofisticado para un programa que tiene unas 200 líneas de código. Con un puñado de buenos métodos te irá bien. cuando tu programa crece y cambia constantemente. En cierto punto, las clases se vuelven tan grandes que ya no puedes recordar sus detalles.

La navegación del código se ralentiza hasta ir a paso de tortuga y tienes que recorrer clases enteras o incluso un programa completo para encontrar algo específico. El número de entidades en el programa supera tu capacidad cerebral y sientes que pierdes el control sobre el código.

Ejemplo:



Principio de abierto/cerrado

La idea fundamental de este principio es evitar que el código existente se descomponga cuando implementas nuevas funciones. La primera vez que estudié este principio me confundieron las palabras abierta y cerrada, porque suenan mutuamente excluyentes.

Pero en lo que respecta a este principio, una clase puede estar al mismo tiempo abierta (para la extensión) y cerrada (para la modificación).

Cuando una clase se ha desarrollado, probado, revisado e incluido en un framework o utilizada en una aplicación de cualquier otro modo, es arriesgado intentar juguetear con su código. En lugar de cambiar directamente el código de la clase, puedes crear una subclase y sobrescribir las partes de la clase original que quieras que se comporten de otra forma. Lograrás tu objetivo sin descomponer clientes existentes de la clase original.

Este principio no está pensado para aplicarse a todos los cambios de una clase. Si sabes que hay un error en la clase, debes arreglarlo; no crees una subclase para ello. Una clase hija no debe ser responsable de los problemas de la clase padre.

Ejemplo:

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Principio de sustitución de Liskov

Esto significa que la subclase debe permanecer compatible con el comportamiento de la superclase. Al sobrescribir un método, extiende el comportamiento base en lugar de sustituirlo con algo totalmente distinto.

El principio de sustitución es un grupo de comprobaciones que ayudan a predecir si una subclase permanece compatible con el código que pudo funcionar con objetos de la superclase.

Este concepto es fundamental al desarrollar bibliotecas y frameworks, porque otras personas utilizarán tus clases y no podrás acceder directamente ni cambiar su código. Al contrario que otros principios de diseño que están muy abiertos a la interpretación, el principio de sustitución incluye un grupo de requisitos formales para las subclases y, específicamente, para sus métodos. Repasemos esta lista en detalle.

* Los tipos de parámetros en el método de una subclase deben coincidir o ser más abstractos que los tipos de parámetros del método de la superclase.
* El tipo de retorno en el método de una subclase debe coincidir o ser un subtipo del tipo de retorno del método de la superclase.
* Un método de una subclase no debe arrojar tipos de excepciones que no se espere que arroje el método base.
* Una subclase no debe fortalecer las condiciones previas.
* Una subclase no debe debilitar las condiciones posteriores.
* Los invariantes de una superclase deben preservarse.
* Una subclase no debe cambiar los valores de campos privados de la superclase.

Ejemplo:

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

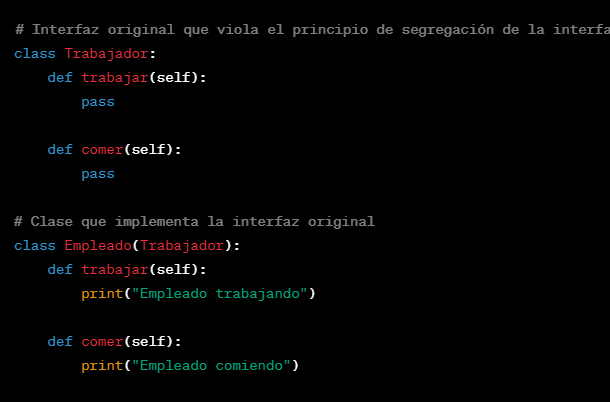
Principio de segregación de la interfaz

Intenta que tus interfaces sean lo suficientemente estrechas para que las clases del cliente no tengan que implementar comportamientos que no necesitan.

Según el principio de segregación de la interfaz, debes desintegrar las interfaces “gruesas” hasta crear otras más detalladas y específicas. Los clientes deben implementar únicamente aquellos métodos que necesitan de verdad. De lo contrario, un cambio en una interfaz “gruesa” descompondrá incluso clientes que no utilizan los métodos cambiados.

La herencia de clases permite a una clase tener una única superclase, pero no limita el número de interfaces que la clase puede implementar al mismo tiempo. Por lo tanto, no hay necesidad de amontonar toneladas de métodos no relacionados en una única interfaz. Divídela en varias interfaces más refinadas; puedes implementarlas todas en una única clase si es necesario. Sin embargo, a algunas clases no les importa que sólo implementes una de ellas.

Ejemplo:



Texto

Descripción generada automáticamente

Principio de inversión de la dependencia

Las clases de alto nivel no deben depender de clases de bajo nivel. Ambas deben depender de abstracciones. Las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones.

* Las clases de bajo nivel implementan operaciones básicas, como trabajar con un disco, transferir datos por una red, conectar con una base de datos, etc.
* Las clases de alto nivel contienen la lógica de negocio compleja que ordena a las clases de bajo nivel que hagan algo.

Algunas personas diseñan primero las clases de bajo nivel y sólo entonces comienzan a trabajar con las de alto nivel. Esto es muy habitual cuando empiezas a desarrollar un prototipo de un nuevo sistema y no estás seguro de lo que es posible a alto nivel, porque el contenido de bajo nivel aún no está implementado o claro. Con este sistema, las clases de la lógica de negocio tienden a hacerse dependientes de clases primarias de bajo nivel.

El principio de inversión de la dependencia sugiere cambiar la dirección de esta dependencia.

1. Para empezar, debes describir interfaces para operaciones de bajo nivel en las que se basarán las clases de alto nivel, preferiblemente en términos de negocio. Por ejemplo, la lógica de negocio debe invocar un método abrirInforme(archivo) en lugar de una serie de métodos abrirArchivo(x), leerBytes(n), cerrarArchivo(x). Estas interfaces cuentan como de alto nivel.
2. Ahora puedes hacer las clases de alto nivel dependientes de esas interfaces, en lugar de clases concretas de bajo nivel. Esta dependencia será mucho más débil que la original.
3. Una vez que las clases de bajo nivel implementan esas interfaces, se vuelven dependientes del nivel de la lógica de negocio, invirtiendo la dirección de la dependencia original.

El principio de inversión de la dependencia suele ir de la mano del principio de abierto/cerrado: puedes extender clases de bajo nivel para utilizarlas con distintas clases de lógica de negocio sin descomponer clases existentes.

Ejemplo:

