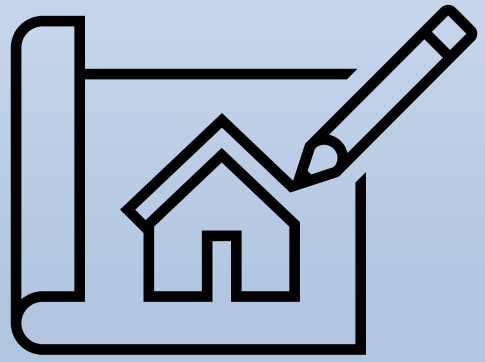
Principios del diseño de software



Fernando Díaz-Alonso Dorado

Características

Que buscar y que evitar

En el proceso de diseño del software debemos buscar y evitar determinadas características:

- Reutilización de código.
- Extensibidad

Los costos y el tiempo son dos de los parámetros más valiosos a la hora de desarrollar y medir cualquier implementación de software.

El cambio es lo único constante en la vida de un programador.

Características

Reutilización de código

La reutilización del código es una de las formas más habituales de reducir los costos de un desarrollo.

No siempre resulta tan sencillo y en ocasiones hacer que un código existente sea reutilizable, en otro nuevo contexto, puede suponer un esfuerzo adicional.

Mediante la utilización de patrones de diseño podemos aumentar la flexibilidad de los componentes y que sea más fácil su reutilización. Pero, en ocasiones el precio es complicar los componentes.

Características

Extensibilidad

Para empezar, comprendemos mejor el problema una vez que lo hemos abordado y comenzamos a resolverlo.

Por eso, cuando acabamos la primera versión de una aplicación, es uno de los momentos en los que nos encontramos en mejor posición para reescribirla desde el principio. El motivo es porque comprendemos mejor los diferentes tipos de aspectos que tiene el problema.

Características

Extensibilidad II

Al igual que evoluciona el entorno, también evolucionamos nosotros, y es muy habitual que soluciones tengan que ser reescritas porque han de actualizarse, adaptarse o mejorarse por necesidades externas.

Pero a su vez, cuando lo hacemos siempre abordamos los cambios con una perspectiva diferente al tener que abandonar las ideas iniciales.

Características

Extensibilidad III

De igual medida que el entorno evoluciona, también lo hace el problema. Tú cliente te pide algo más, o quitar y poner… Para el cliente, son cambios sencillos, siempre lo serán, aunque tengas que cambie por completo el problema inicial.

Por este motivo, cuando un equipo de desarrollo aborda las primeras versiones de una aplicación, suele prever que existan cambios en el futuro. Preparando el diseño de la arquitectura de la aplicación buscando que pueda soportarlos y sea más fácil la reutilización del código.

Principios del diseño

Entonces…

- ¿Qué es un buen diseño de software?
- ¿Cómo medimos su calidad?
- ¿Qué prácticas debemos llevar a cabo para lograrlo?
- ¿Cómo podemos hacer nuestra arquitectura flexible, estable y fácil de comprender?

Son grandes preguntas, pero, con respuestas que varían en función del tipo de aplicación a diseñar. Pero si es cierto, que existen unos principios universales del diseño de software que pueden ayudarte. Además, la mayoría de los patrones de diseño se basan en estos principios.

Encapsula

Minimizar el efecto de los cambios

El objetivo principal de encapsular es conseguir minimizar los efectos que van a provocar los cambios.

Podemos aislar las diferentes partes de un programa que varían, creando módulos independientes. De esta forma, únicamente has de afrontar los cambios en los módulos que varían y dedicas menos tiempo a volver a lograr que el programa funcione.

Por ejemplo:

- Encapsulación a nivel de método.
- Encapsulación a nivel de clases.

Interfaces

Programa interfaces y no implementes

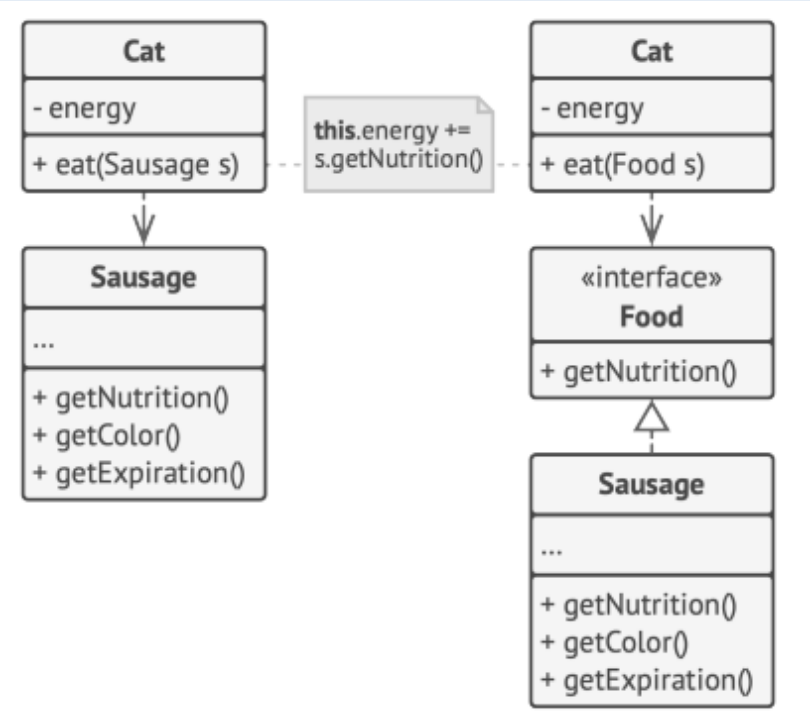
Es mejor programar una interfaz y no una implementación. Haciendo las dependencias abstractas y no de clases concretas.

La forma más sencilla de que dos clases colaboren es que una depende de la otra, pero lo más flexible es establecer la colaboración entre objetos.

1. Determina lo que necesita exactamente un objeto del otro. → Qué métodos ejecuta.
2. Describe esos métodos en una nueva interfaz o clase abstracta.
3. Haz que la clase que es una dependencia implemente la nueva interfaz.
4. Ahora, haz la segunda clase dependiente de la nueva interfaz en lugar de una clase concreta. Aún puede funcionar con los objetos de la clase original, pero ahora la conexión es más flexible.

Interfaces

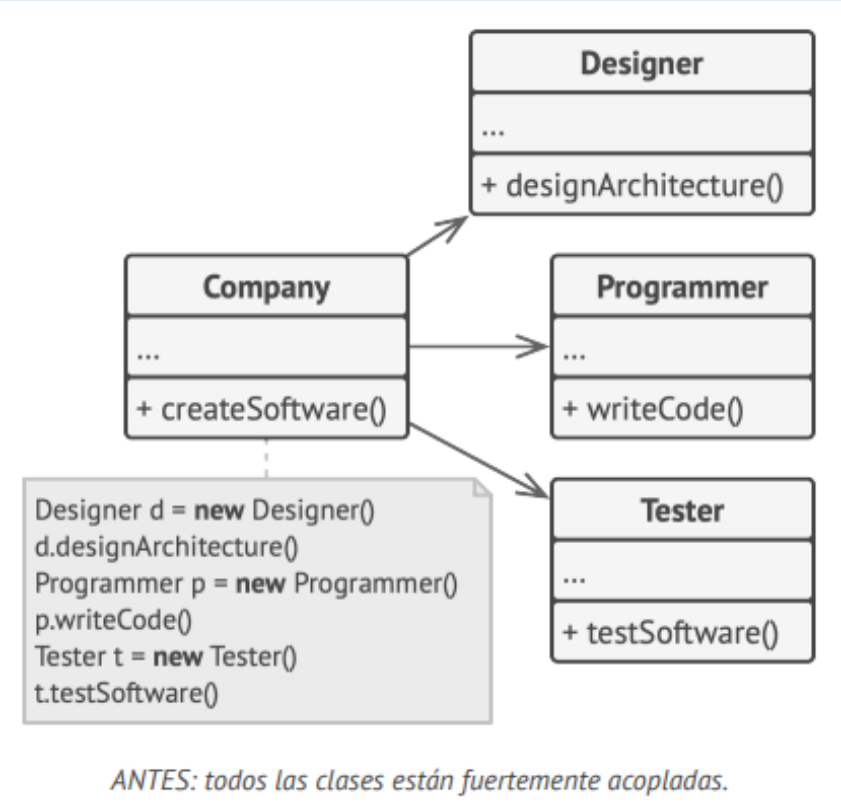
Ejemplo



Antes y después de extraer la interfaz. El código de la derecha es más flexible que el de la izquierda, pero también es más complicado.

Interfaces

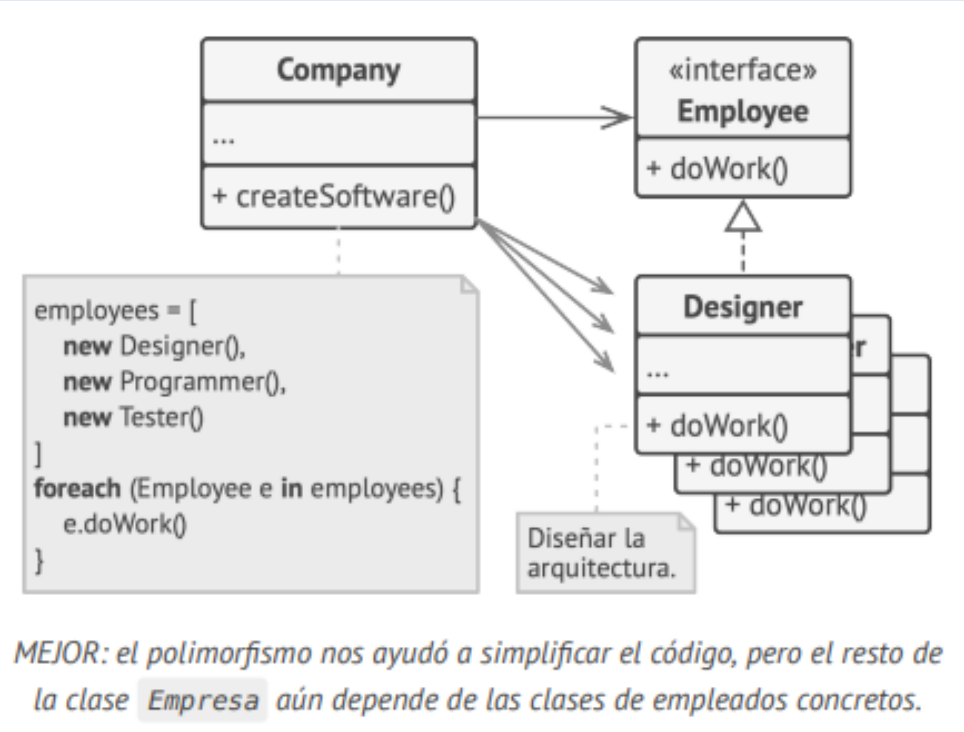
Ejemplo Company I



ANTES: todos las clases están fuertemente acopladas.

Interfaces

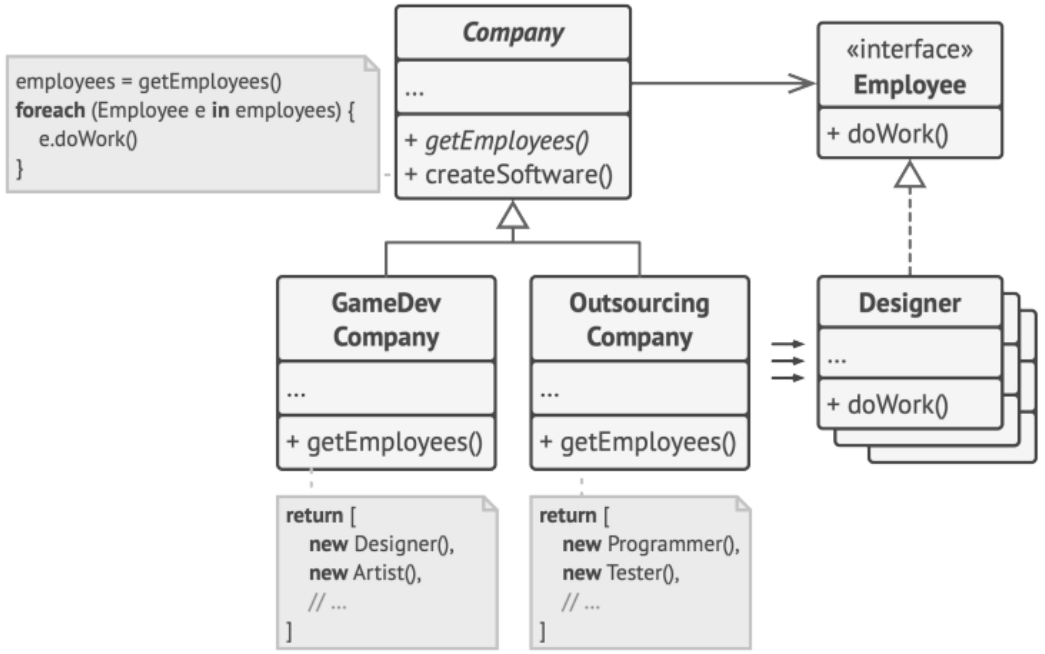
Ejemplo Company II



MEJOR: el polimorfismo nos ayudó a simplificar el código, pero el resto de la clase `Empresa` *aún depende de las clases de empleados concretos.*

Interfaces

Ejemplo Company III



DESPUÉS: el método primario de la clase `Empresa` es independiente de clases de empleado concretos. Los objetos de empleado se crean en subclases de empresas concretas.

Composición

Favorece la composición sobre la herencia

A pesar de que la herencia es una de las formas más obvias y sencillas de reutilizar código entre clases. La herencia tiene sus contras, que generalmente son evidentes hasta que ya es tarde:

- **Una subclase no puede reducir la interfaz de la superclase.**
- **Al sobrescribir método debes asegurarte de que el nuevo comportamiento sea compatible con la base.**
- **La herencia rompe la encapsulación de la superclase.**
- **Las subclases están fuertemente acopladas a superclases.**
- **Intentar reutilizar código mediante la herencia puede conducir a la creación de jerarquías de herencia paralelas.**

Composición

Composición vs herencia

La composición se basa en una relación de *“tiene un/a”* en vez de la herencia que es una relación *“es un/a”*.

Herencia → Un coche es un medio de transporte.

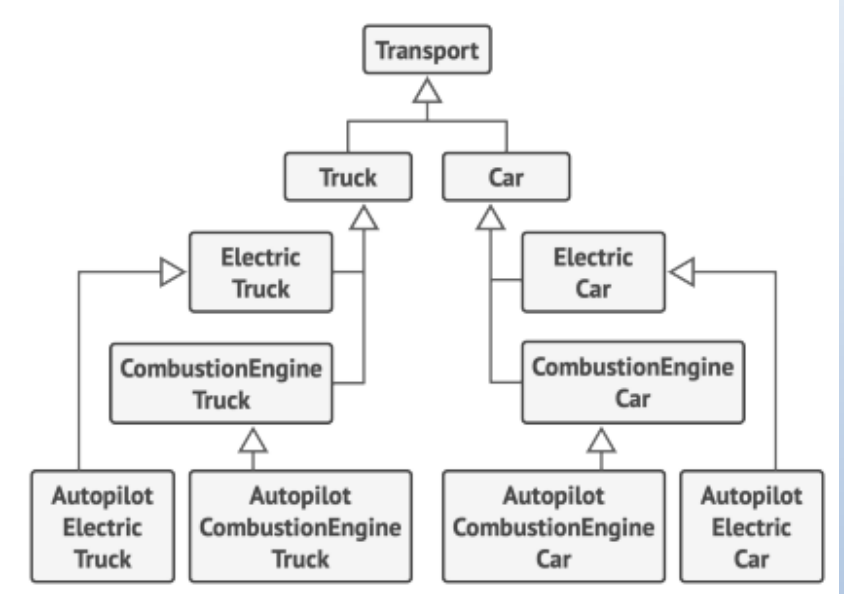
Composición → Un coche tiene un motor.

También podemos extender este principio a una agregación:

Agregación –> Un coche tiene un conductor. (Pero el conductor puede ir en coche o caminando)

Composición

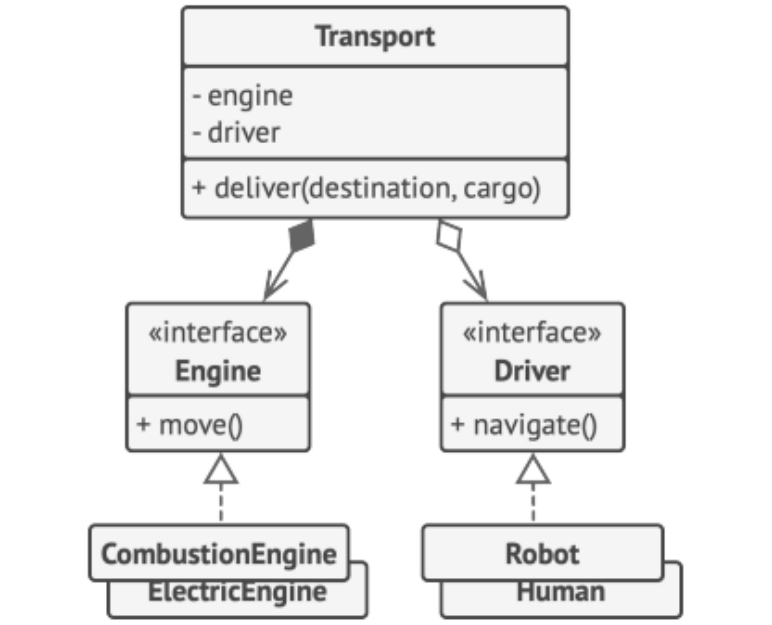
Ejemplo herencia



HERENCIA: extender una clase en varias dimensiones (tipo de carga × tipo de motor × tipo de navegación) puede provocar una explosión combinatoria de subclases.

Composición

Ejemplo composición



COMPOSICIÓN: diferentes "dimensiones" de funcionalidad extraídas para ponerse dentro de sus propias jerarquías de clases..

Principios SOLID

5 principios SOLID

Los principios SOLID es una regla mnemotécnica para cincos principios de diseño ideados para hacer que los diseños de software sean más comprensibles, flexibles y fáciles de mantener.

Los principios SOLID mal implementados implican un coste elevado y complicar el diseño innecesariamente.

Podemos decir que existen muy pocos o ningún diseño que cumpla los 5 principios a la vez. Aspirar a implementar alguno es bueno, pero no es dogma de fe. **Si no es útil no lo hagas.**

Principios SOLID

S → Single Responsability Principle

El principio de responsabilidad única es muy sencillo: **Una clase solamente debe tener una razón para cambiar.**

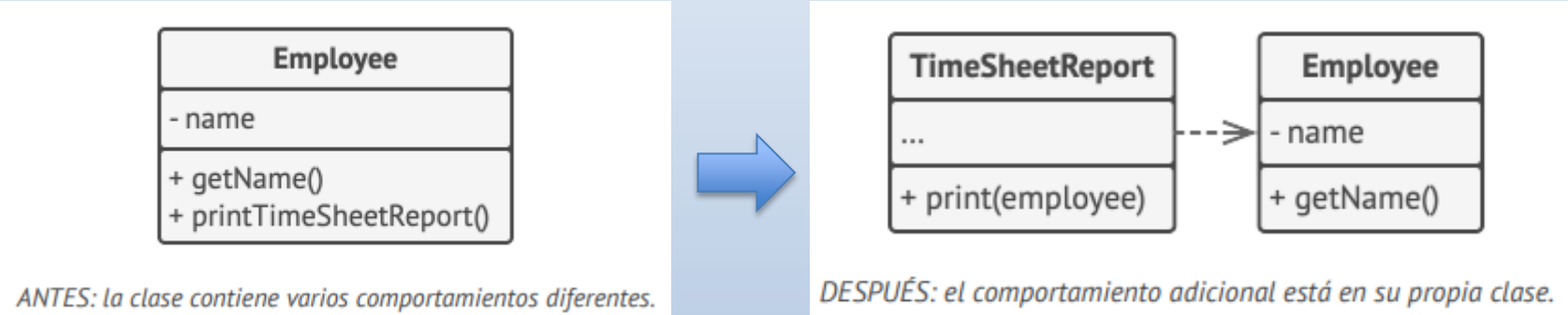
Esto quiere decir que una clase debe responsabilizarse únicamente de una parte de la funcionalidad proporcionada por el software. Con lo que reducimos su complejidad.

Si una clase hace demasiadas cosas, debes cambiarla cada vez que una de esas cosas cambia.

La solución es dividir las clases para que hagan únicamente una cosa.

Principios SOLID

S → Single Responsability Principle



Principios SOLID

O → Open / Close Principle

El principio de abierto / cerrado es que **las clases deben estar abiertas a la extensión pero cerradas a la modificación**.

Este principio pretende evitar que el código existente se descomponga cuando se implementan nuevas funciones.

Cuando la clase está abierta si puede extenderla, crear una subclase y hacer lo que quieras con ella. Al mismo tiempo, la clase está cerrada (o completa) si está lista al 100% para que otras clases la utilicen; su interfaz está claramente definida y no se cambiará en el futuro.

Principios SOLID

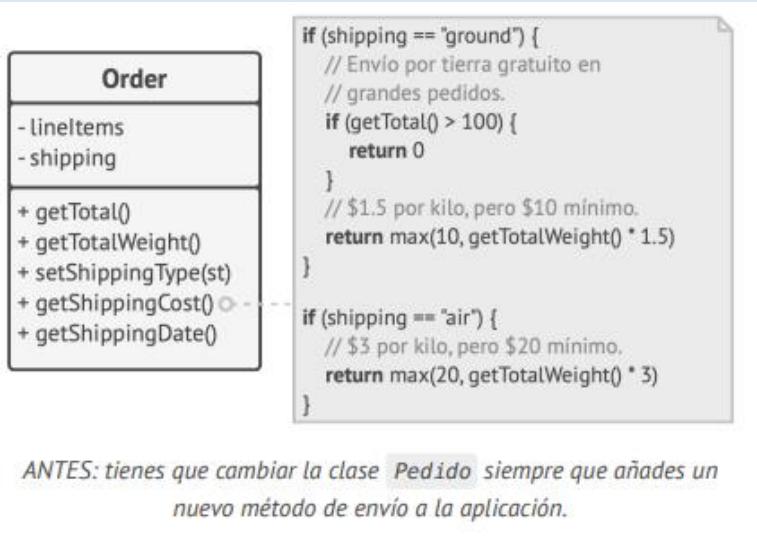
O → Open / Close Principle

Hay que entender una cuestión importante que con el lenguaje podemos confundir y es que Open / Close puede sonar a excluyentes entre sí y no es el caso. Porque una clase pude estar abierta para la extensión y cerrada a la modificación a la vez.

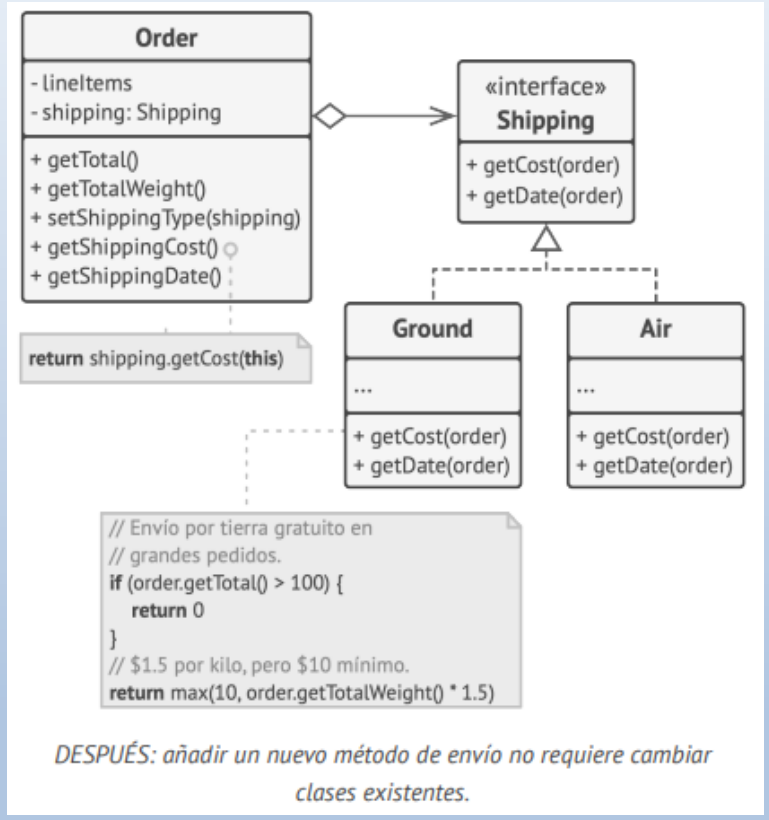
Este principio no está pensado para aplicarse a todos los cambios de una clase. ***Si existe un error debes corregirlo, una subclase no debe ser responsable de los errores de una superclase.***

Principios SOLID

O → Open / Close Principle



ANTES: tienes que cambiar la clase `Pedido` siempre que añades un nuevo método de envío a la aplicación.



DESPUÉS: añadir un nuevo método de envío no requiere cambiar clases existentes.

Florida Universitària

Principios SOLID

L → Liskov Substituion Principle

El principio de sustitución de Liskov nos recuerda que **al extender una clase debes tener la capacidad de pasar objetos de las subclases en lugar de objetos de la clase padre, sin descomponer el código del cliente**.

Mediante este principio cualquier subclase debe ser siempre compatible con el comportamiento de la superclase. Por lo tanto, *si sobrescribimos un método, debemos extender el comportamiento en vez de sustituirlo por algo diferente.*

Este principio es el concepto fundamental en el que se basan las librerías y los frameworks.

Principios SOLID

L → Liskov Substituion Principle

Este principio posee un grupo de requisitos formales para las subclases y para sus métodos:

- *Los tipos de parámetros en el método de una subclase debe coincidir o se más abstractos que los tipos de parámetros del método de la super clase.*
- *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. (En muchos lenguajes ya está implementado en su propia estructura)*

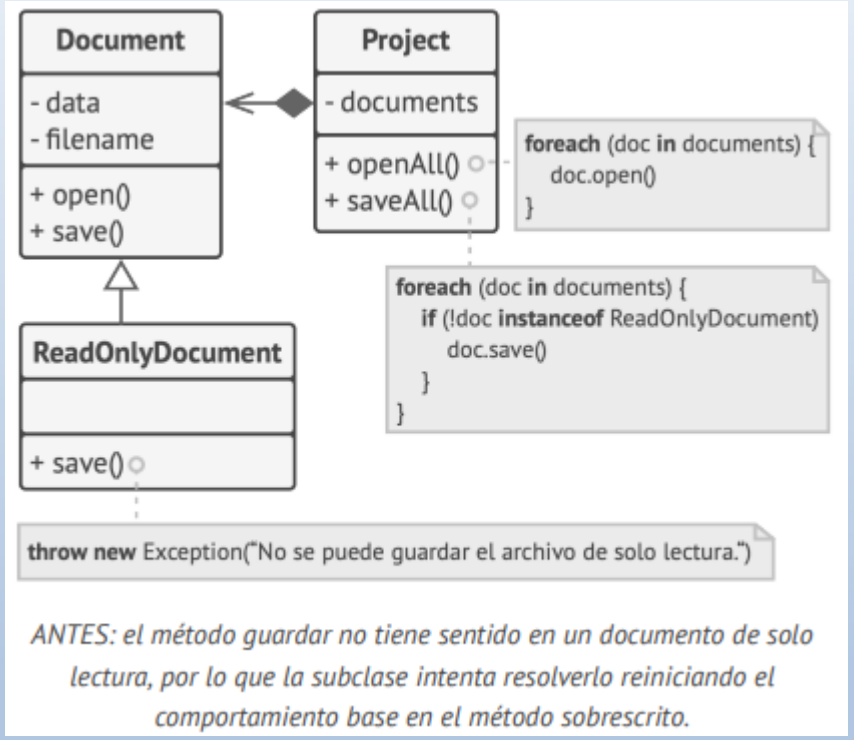
Principios SOLID

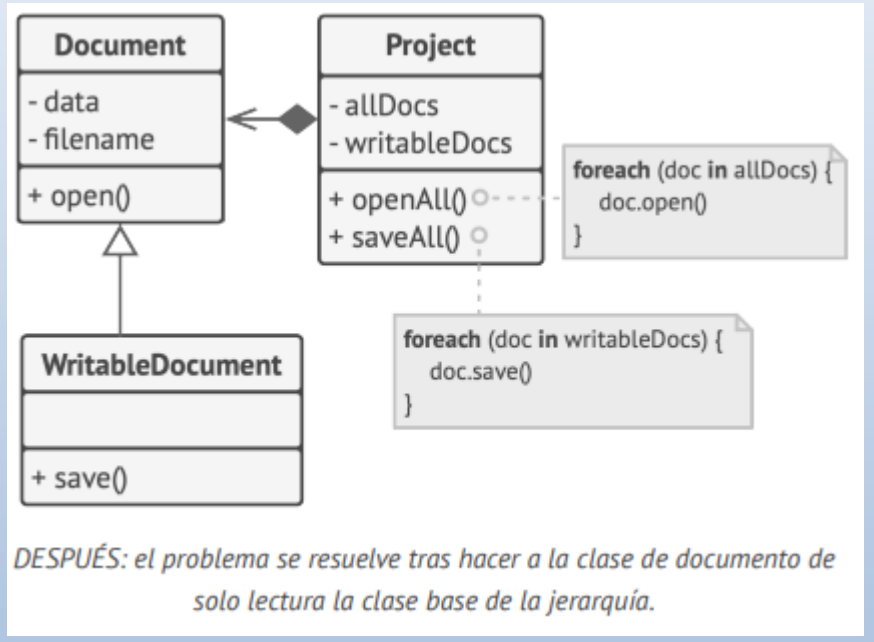
L → Liskov Substituion Principle

- *Una subclase no debe fortalecer las condiciones previas. No puede ser más restrictivo.*
- *Una subclase no debe debilitar las condiciones posteriores.*
- *Las invariantes de una superclase deben preservarse. (La más complicada y más vulnerada)*
- *Una subclase no debe cambiar los valores de campos privados de la superclase.*

Principios SOLID

L → Liskov Substituion Principle





Principios SOLID

I → Interface Segregation Principle

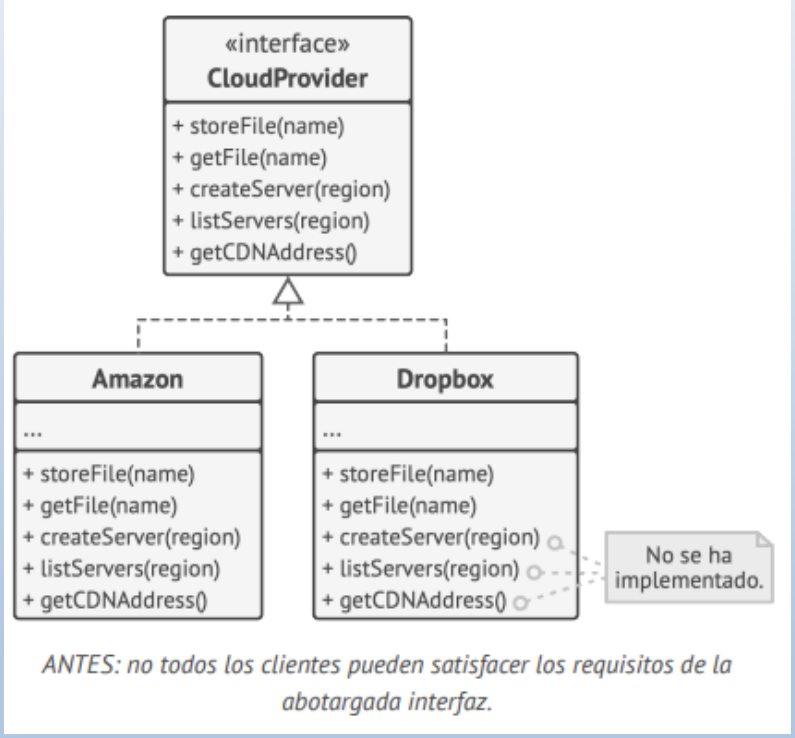
El principio de segregación de la interfaz **busca que las interfaces sean los suficientemente básicas para que una clase de un cliente no deba implementar métodos que no necesita.**

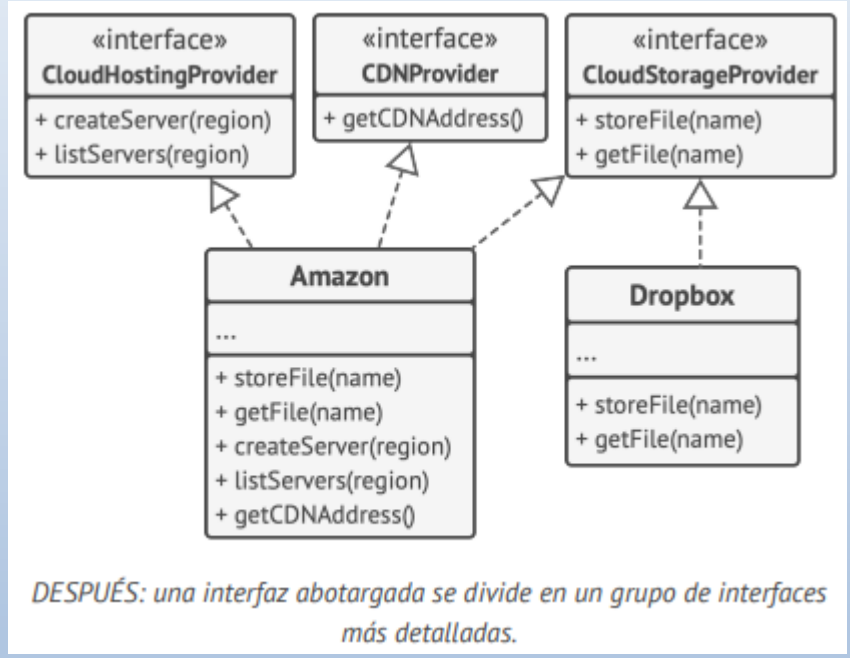
Una interfaz gruesa debe ser desintegrada y crear otras más detalladas y específicas. De manera que un cliente únicamente utilice los métodos que necesita de verdad.

La herencia permite a una clase únicamente tener una superclase. Pero no limita cuantas interfaces de la clase puede implementar al mismo tiempo. Por lo tanto, podemos dividir las interfaces. Implementando en la clase únicamente las que necesites y no todas.

Principios SOLID

I → Interface Segregation Principle





Principios SOLID

D → Dependency Inversión Principle

El principio de inversión de dependencia **nos dice que las clases de alto nivel no deben depender de las clases de bajo nivel. Ambas deben depender de abstracciones. Las abstracciones no deben depender de detalles. Y los detalles deben depender de abstracciones.**

Es necesario explicar los niveles de clases:

- **Clases de alto nivel:** Implementan operaciones básicas, como trabajar con un disco, trasferir datos por una red, conectar con una base de datos, etc…
- **Clases de bajo nivel:** Contienen la lógica de negocio compleja que ordena a las clases de bajo nivel que hagan algo.

Principios SOLID

D → Dependency Inversión Principle

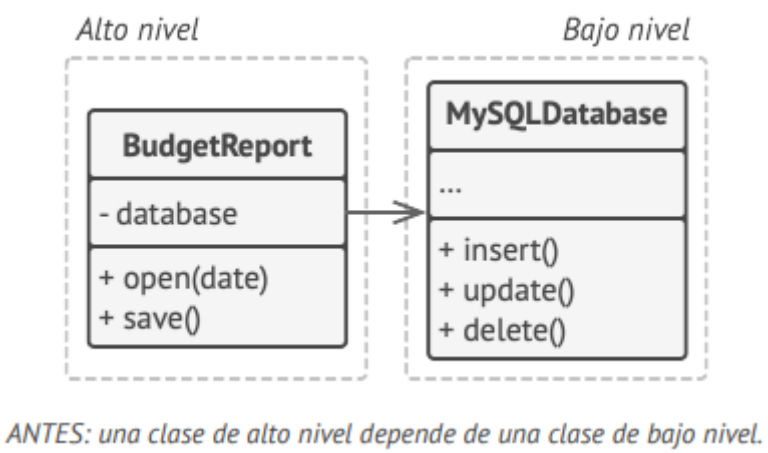
El principio de la inversión de dependencia sugiere:

1. *Debes escribir interfaces para operaciones de bajo nivel en las que se basarán las clases 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 de nivel de la lógica de negocio, invirtiendo la dirección de la dependencia original.*

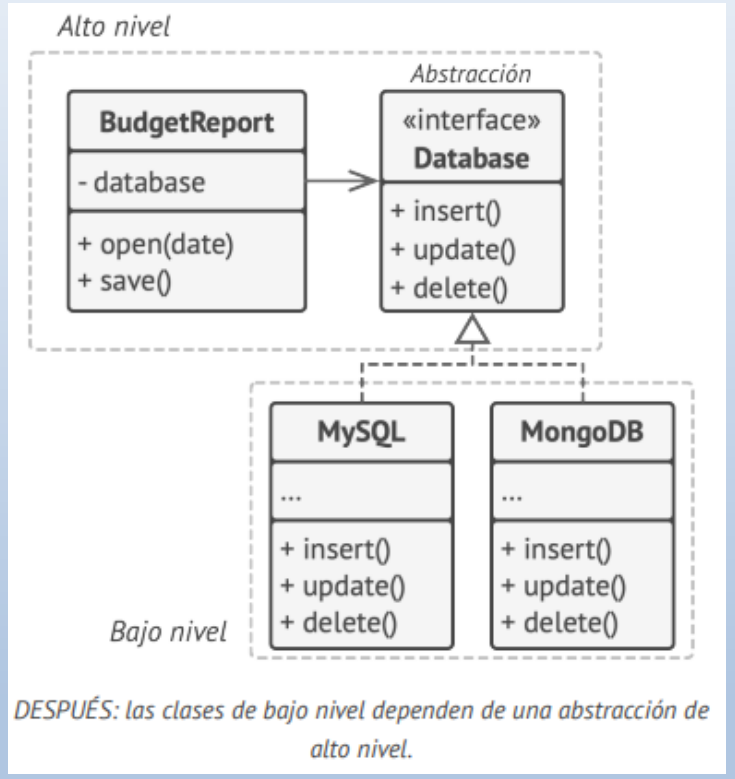
Este principio suele ir de la mano con el principio O (Open / close)

Principios SOLID

D → Dependency Inversión Principle



ANTES: una clase de alto nivel depende de una clase de bajo nivel.



DESPUÉS: las clases de bajo nivel dependen de una abstracción de alto nivel.

Principios SOLID

Resumen

Estos principios son una guía y como se ha comentado es muy complicado aplicarlos todos a la vez, porque incluso en ocasiones unos se contradicen con otros.

Es recomendable conocerlos y aplicarlos según nuestra aplicación los necesite o se nos requieran.

Principios del diseño de software

Bibliografía

- Patrones de Diseño – Alexander Shvets