Principios SOLID

Guía rápida para aprender qué son y cómo aplicarlos en tu día a día

Antonio Leiva

Principios SOLID

Guía rápida para aprender qué son y cómo aplicarlos en tu día a día

Antonio Leiva

Índice general

Principios SOLID: Qué son, cuáles, y qué beneficios aporta usarlo

Los Principios SOLID son uno de los conceptos de programación y diseño de software más populares.

Seguramente ya has intentado muchas veces aprenderlos y aplicarlos en tu día a día, pero se te siguen resistiendo.

Esto le pasa a todo el mundo, porque son conceptos abstractos difíciles de explicar y no se dan los detalles necesarios para bajarlos a tierra.

Soy Antonio Leiva, ingeniero de software y formador, y tras enseñarles a muchos de mis alumnos este principios, puedo darte las claves para que los entiendas y los apliques de una vez por todas.

¿Qué son los Principios SOLID?

Son un conjunto de principios aplicables a la Programación Orientada a Objetos que, si los usas correctamente, te ayudarán a **escribir software de calidad** en cualquier lenguaje de programación orientada a objetos. Gracias a ellos, crearás código que será más fácil de leer, testear y mantener.

Los principios en los que se basa SOLID son los siguientes:

- Principio de Responsabilidad Única (Single Responsibility Principle)
- Principio Open/Closed (Open/Closed Principle)
- Principio de Sustitución de Liskov (Liskov Substitution Principle)
- Principio de Segregación de Interfaces (Interface Segregation Principle)
- Principio de Inversión de Dependencias (Dependency Inversion Principle)

Estos principios son la base de mucha literatura que encontrarás en torno al desarrollo de software: muchas arquitecturas se basan en ellos para **proveer flexibilidad**, el testing necesita confiar en ellos para poder validar partes de código de forma independiente, y los procesos de refactorización serán mucho más sencillos si se cumplen estas reglas. Así que es muy conveniente que asimiles bien estos conceptos.

Fueron publicados por primera vez por Robert C. Martin[1], también conocido como Uncle Bob, en su libro Agile Software Development: Principles, Patterns, and Practices[2]. Una persona que te recomiendo seguir, y echarle un vistazo a su blog[3] de vez en cuando.

¿Qué beneficios aporta usar los Principios SOLID?

Las ventajas de utilizar los Principios SOLID son innumerables, ya que nos aportan todas esas características que siempre queremos ver en un software de calidad.

En cada uno de los principios nos iremos centrando en qué aportan específicamente, pero es interesante hacer un resumen general de lo que conseguiremos con ellos:

Software más flexible: mejoran la cohesión disminuyendo el acoplamiento

Los conceptos de cohesión y acoplamiento merecen un artículo a parte, pero a grandes rasgos lo que buscamos de un buen código es que sus clases puedan trabajar de forma independiente y que el cambio de uno afecte lo menos posible al resto.

Obviamente cuando dos clases se relacionan entre sí para trabajar juntas (y esto tiene que ocurrir sí o sí), va a existir un acoplamiento entre ellas.

Pero existen distintos niveles de acoplamiento, y gracias a algunos de los Principios SOLID, podemos relajar esas dependencias y hacerlas mucho más flexibles a cambios.

[1] https://twitter.com/unclebobmartin
[2] http://devexperto.com/agile-software-development
[3] http://blog.cleancoder.com/

Te van a hacer entender mucho mejor las arquitecturas

Siempre que hablo de arquitecturas, noto que hay una barrera importante para entender cómo aplicarlas y qué beneficios aportan.

Esto es porque primero hace falta entender los principios sobre los que se sustentan, y los principios SOLID son muy importantes para ello.

Simplifican la creación de tests

Todo esto está muy relacionado con los puntos anteriores: si tienes tu código desacoplado y una buena arquitectura, los tests van a ser mucho más sencillos.

En un vídeo anterior ya comentaba los7 errores que solemos cometer al escribir tests[4], y mucho del tema va por aquí.

Al final piensa que todo es como una cadena: si aplicas bien los principios, organizas mejor tu código. Esto te permite definir una arquitectura que hará que los tests sean más sencillos.

Podría decirse por tanto que los Principios SOLID son parte de la base de un código de calidad.

Los Principios SOLID están muy interrelacionados

Estos principios actúan como un todo. No es casualidad que se expliquen de forma conjunta.

Y esto tiene dos consecuencias muy importantes de entender:

Unos principios no pueden existir sin los otros

Una pregunta muy recurrente en mi formación de Architect Coders cuando vemos estos Principios es que parece que al querer aplicar uno de ellos, hay otro que necesitan aplicar inevitablemente.

[4] https://devexperto.com/7-razones-por-las-que-te-cuesta-tanto-hacer-tests/

¡Claro! Ahí está la gracia. Aún no hemos hablado de los Principios a fondo, pero **por poner un ejemplo**:

Imagina que tienes una clase A que tiene un acoplamiento muy fuerte con una clase B, de tal forma que cada vez que cambia B inevitablemente tiene que cambiar A.

Esto es muy posible que esté incumpliendo el Principio de Responsabilidad Única. La forma de cumplirlo sería haciendo que cuando B cambie, A no lo haga. Y para esto, la solución puede ser aplicar el Principio de Inversión de Dependencias.

¡No pasa nada! Esto es muy normal y te pasará casi siempre.

Un mismo problema se puede resolver desde dos perspectivas distintas en función de en qué Principio nos enfoquemos cuando lo resolvamos. Pero el resultado será el mismo.

Al cumplir un Principio puede que estés incumpliendo otro

Esto es lo más difícil de aceptar: muchas veces es imposible cumplir todos los Principios a la vez.

Porque al aplicar un Principio, se puede estar dando la espalda a otro.

Pero no te obsesiones con esto: al final lo importante es entender la potencia de cada Principio.

Y ese conocimiento unido a la experiencia te irá diciendo poco a poco cuáles son las mejores decisiones a tomar.

Cómo aprender a usar los Principios SOLID

Ahora que ya tenemos claros los conceptos, puedes empezar a entenderlos uno a uno y aprender cómo aplicarlos.

Principio de Responsabilidad Única

El Principio de responsabilidad única es el primero de los cinco que componen SOLID.

El principio de Responsabilidad Única nos viene a decir que **un objeto debe realizar una única cosa**. Es muy habitual, si no prestamos atención a esto, que acabemos teniendo clases que tienen varias responsabilidades lógicas a la vez.

Si lo has visto por ahí, seguramente hayas una frase similar a esta:

> El Principio de Responsabilidad Única nos dice que un módulo tiene una única razón para cambiar

En estos artículos verás que uso de forma indiferente las palabras módulo, entidad o clase. En realidad, cuando hablamos de lenguajes orientados a objetos, esto siempre se refiere a una clase.

A mí esta definición no me gusta mucho, porque lo de "una única razón para cambiar" me suena muy etéreo, y es difícil bajarlo a tierra.

Prefiero decir que el Principio de Responsabilidad Única se cumple cuando nuestra clase solo hace una cosa.

Tampoco es fácil definir qué es "una cosa", pero ya tenemos herramientas más sencillas para detectarlo.

Por ejemplo, si cuando tienes que explicar el funcionamiento de una clase, dices que "esta clase hace esta cosa **Y** esta otra", entonces sospecha.

Pero te lo voy a poner más fácil, te voy a dar unos truquillos para mejorar la detección:

¿Cómo detectar si estamos violando el Principio de Responsabilidad Única?

La respuesta a esta pregunta es bastante subjetiva. Sin necesidad de obsesionarnos con ello, podemos detectar situaciones en las que una clase podría dividirse en varias:

- **En una misma clase están involucradas dos capas de la arquitectura**: esta puede ser difícil de ver sin experiencia previa. En toda arquitectura, por simple que sea, debería haber una capa de presentación, una de lógica de negocio y otra de persistencia. Si mezclamos responsabilidades de dos capas en una misma clase, será un buen indicio.
- **El número de métodos públicos**: Si una clase hace muchas cosas, lo más probable es que tenga muchos métodos públicos, y que tengan poco que ver entre ellos. Detecta cómo puedes agruparlos para separarlos en distintas clases. Algunos de los puntos siguientes te pueden ayudar.
- **Los métodos que usan cada uno de los campos de esa clase**: si tenemos dos campos, y uno de ellos se usa en unos cuantos métodos y otro en otros cuantos, esto puede estar indicando que cada campo con sus correspondientes métodos podrían formar una clase independiente. Normalmente esto estará más difuso y habrá métodos en común, porque seguramente esas dos nuevas clases tendrán que interactuar entre ellas.
- **Por el número de imports**: Si necesitamos importar demasiadas clases para hacer nuestro trabajo, es posible que estemos haciendo trabajo de más. También ayuda fijarse a qué paquetes pertenecen esos imports. Si vemos que se agrupan con facilidad, puede que nos esté avisando de que estamos haciendo cosas muy diferentes.
- **Nos cuesta testear la clase**: si no somos capaces de escribir tests unitarios sobre ella, o no conseguimos el grado de granularidad que nos gustaría, es momento de plantearse dividir la clase en dos.
- **Cada vez que escribes una nueva funcionalidad, esa clase se ve afectada**: si una clase se modifica a menudo, es porque está involucrada en demasiadas cosas.
- **Por el número de líneas**: a veces es tan sencillo como eso. Si una clase es demasiado grande, intenta dividirla en clases más manejables.

En general no hay reglas de oro para estar 100% seguros. La práctica te irá haciendo ver cuándo es recomendable que cierto código se mueva a otra clase, pero estos indicios te ayudarán a detectar algunos casos donde tengas dudas.

Ejemplo

Un ejemplo típico es el de un objeto que necesita ser renderizado de alguna forma, por ejemplo imprimiéndose por pantalla. Podríamos tener una clase como esta:

```
class Vehicle(
    val wheelCount: Int,
    val maxSpeed: Int
) {
    fun print() {
        println("wheelCount=$wheelCount, maxSpeed=$maxSpeed")
    }
}
```

Aunque a primera vista puede parecer una clase de lo más razonable, en seguida podemos detectar que **estamos mezclando dos conceptos muy diferentes**: la lógica de negocio y la lógica de presentación. Este código nos puede dar problemas en muchas situaciones distintas:

- En el caso de que queramos presentar el resultado de distinta manera, necesitamos cambiar una clase que especifica la forma que tienen los datos. Ahora mismo estamos imprimiendo por pantalla, pero imagina que necesitas que se renderice en un HTML. Tanto la estructura (seguramente quieras que la función devuelva el HTML), como la implementación cambiarían completamente.
- Si queremos mostrar el mismo dato de dos formas distintas, no tenemos la opción si sólo tenemos un método `print()`.
- Para testear esta clase, no podemos hacerlo sin los efectos de lado que suponen el imprimir por consola.

Hay casos como este que se ven muy claros, pero muchas veces los detalles serán más sutiles y probablemente no los detectarás a la primera. **No tengas miedo de refactorizar** lo que haga falta para que se ajuste a lo que necesites.

Un solución muy simple sería crear una clase que se encargue de imprimir:

```
class VehiclePrinter {
    fun print(vehicle: Vehicle) {
        println(
            "wheelCount=${vehicle.wheelCount}, " +
                    "maxSpeed=${vehicle.maxSpeed}"
        )
    }
}
```

Si necesitases distintas variaciones para presentar la misma clase de forma diferente (por ejemplo, texto plano y HTML), siempre puedes crear una interfaz y crear implementaciones específicas. Pero ese es un tema diferente.

Otro ejemplo que nos podemos encontrar a menudo es el de objetos a los que les añadimos el método `save()`. Una vez más, la capa de lógica y la de persistencia deberían permanecer separadas. Seguramente hablaremos mucho de esto en futuros artículos.

Conclusión

El Principio de Responsabilidad Única es una **herramienta indispensable para proteger nuestro código frente a cambios**, ya que implica que sólo debería haber un motivo por el que modificar una clase.

En la práctica, muchas veces nos encontraremos con que estos límites tendrán más que ver con lo que realmente necesitemos que con complicadas técnicas de disección. **Tu código te irá dando pistas según el software evolucione**.

¿Crees que lo podrás aplicar a partir de ahora en tu día a día?

Principio Open/Closed

El principio Open/Closed fue nombrado por primera vez por **Bertrand Mayer**, un programador francés, quien lo incluyó en su libro Object Oriented Software Construction[5] allá por 1988.

Este principio nos dice que **una entidad de software debería estar abierta a extensión pero cerrada a modificación**

¿Qué quiere decir esto? Que tenemos que ser capaces de extender el comportamiento de nuestras clases sin necesidad de modificar su código.

Esto nos ayuda a seguir añadiendo funcionalidad con la seguridad de que no afectará al código existente. Nuevas funcionalidades implicarán añadir nuevas clases y métodos, pero en general no debería suponer modificar lo que ya ha sido escrito.

La forma de llegar a ello está muy relacionada con el punto anterior. Si las clases sólo tienen una responsabilidad, podremos añadir nuevas características que no les afectarán. **Esto no quiere decir que cumpliendo el primer principio se cumpla automáticamente el segundo**, ni viceversa. Luego verás un caso claro en el ejemplo.

El principio Open/Closed **se suele resolver utilizando polimorfismo**[6]. En vez de obligar a la clase principal a saber cómo realizar una operación, delega esta a los objetos que utiliza, de tal forma que no necesita saber explícitamente cómo llevarla a cabo. Estos objetos tendrán una interfaz común que implementarán de forma específica según sus requerimientos.

¿Cómo detectar que estamos violando el principio Open/Closed?

Una de las formas más sencillas para detectarlo es darnos cuenta de qué clases modificamos más a menudo. Si cada vez que hay un nuevo requisito o una modificación de los existentes, las mismas clases se ven afectadas, podemos empezar a entender que estamos violando este principio.

[5] http://devexperto.com/object-oriented-software-construction
[6] https://es.wikipedia.org/wiki/Polimorfismo_(inform%C3%A1tica)

Ejemplo

Siguiendo con nuestro ejemplo de vehículos, podríamos tener la necesidad de dibujarlos en pantalla. Imaginemos que tenemos una clase con un método que se encarga de dibujar un vehículo por pantalla. Por supuesto, cada vehículo tiene su propia forma de ser pintado. Nuestro vehículo tiene la siguiente forma:

```
class Vehicle(val type: VehicleType)
```

Básicamente es una clase que especifica su tipo mediante un enumerado. Podemos tener por ejemplo un `enum` con un par de tipos:

```
enum class VehicleType{
    CAR, MOTORBIKE
}
```

Y éste es el método de la clase que se encarga de pintarlos:

```
fun draw(vehicle: Vehicle) {
    when(vehicle.type){
        VehicleType.CAR -> drawCar(vehicle)
        VehicleType.MOTORBIKE -> drawMotorbike(vehicle)
    }
}
```

Mientras no necesitemos dibujar más tipos de vehículos ni veamos que este `when` se repite en varias partes de nuestro código, en mi opinión no debes sentir la necesidad de modificarlo. Incluso el hecho de que cambie la forma de dibujar un coche o una moto estaría encapsulado en sus propios métodos y no afectaría al resto del código.

Pero puede llegar un punto en el que necesitemos dibujar un nuevo tipo de vehículo, y luego otro... Esto implica crear un nuevo enumerado, un nuevo `case` y un nuevo método para implementar el dibujado. En este caso sería buena idea aplicar el principio Open/Closed.

Si lo solucionamos mediante herencia o polimorfismo[7], el paso evidente es sustituir ese enumerado por clases reales, y que cada clase sepa cómo pintarse:

[7] https://devexperto.com/herencia-vs-composicion/

```
interface Vehicle {
    fun draw()
}

class Car : Vehicle {
    override fun draw() {
        // Draw the car
    }
}

class Motorbike : Vehicle {
    override fun draw() {
        // Draw the motorbike
    }
}
```

Ahora nuestro método anterior se reduce a:

```
fun draw(vehicle: Vehicle) {
    vehicle.draw()
}
```

Añadir nuevos vehículos ahora es tan sencillo como crear la clase correspondiente que extienda de `Vehicle`:

```
class Truck: Vehicle {
    override fun draw() {
        // Draw the truck
    }
}
```

Como puedes ver, este ejemplo choca directamente con el que vimos en el Principio de Responsabilidad Única. Esta clase está guardando la información del objeto y la forma de pintarlo.

¿Implica eso que es incorrecto?

No necesariamente, tendremos que ver si el hecho de tener el método `draw` en nuestros objetos afecta negativamente la mantenibilidad y testabilidad del código. En ese caso, habría que buscar alternativas.

Aunque no la voy a presentar aquí, una alternativa para cumplir ambos sería aplicar este polimorfismo a clases que sólo tengan un método de pintado y que reciban el objeto a pintar por constructor. Tendríamos por tanto un `CarDrawer` que se encargue de pintar coches o un `MotorbikeDrawer` que dibuje motos, todos ellos implementando `draw()`, que estaría definido en una clase o interfaz padre.

¿Cuándo debemos cumplir con este principio?

Hay que decir que añadir **esta complejidad no siempre compensa**, y como el resto de principios, sólo será aplicable si realmente es necesario.

Si tienes una parte de tu código que es propensa a cambios, plantéate hacerla de forma que un nuevo cambio impacte lo menos posible en el código existente. Normalmente esto no es fácil de saber a priori, por lo que puedes preocuparte por ello cuando tengas que modificarlo, y hacer los cambios necesarios para cumplir este principio en ese momento.

Intentar hacer **un código 100% Open/Closed es prácticamente imposible**, y puede hacer que sea ilegible e incluso más difícil de mantener.

No me cansaré de repetir que las reglas SOLID son ideas muy potentes, pero hay que aplicarlas donde corresponda y sin obsesionarnos con cumplirlas en cada punto del desarrollo.

Casi siempre es más sencillo limitarse a usarlas cuando nos haya surgido la necesidad real.

Conclusión

El principio Open/Closed es una herramienta indispensable para protegernos frente a cambios en módulos o partes de código en los que esas modificaciones son frecuentes. **Tener código cerrado a modificación y abierto a extensión nos da la máxima flexibilidad con el mínimo impacto**.

Principio de sustitución de Liskov

El principio de sustitución de Liskov nos dice que **si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido**.

Esto nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento de la padre.

Este principio viene a desmentir la idea preconcebida de que las clases son una forma directa de modelar la realidad, y que hay que tener cuidado con esa modelización.

La primera en hablar de él fue Bárbara Liskov[8] (de ahí el nombre), una reconocida ingeniera de software americana.

¿Cómo detectar que estamos violando el principio de sustitución de Liskov?

Seguro que te has encontrado con esta situación muchas veces: creas una clase que extiende de otra, pero de repente uno de los métodos te sobra, y no sabes que hacer con él.

Las opciones más rápidas son bien dejarlo vacío, bien lanzar una excepción cuando se use, asegurándote de que nadie llama incorrectamente a un método que no se puede utilizar.

Si un método sobrescrito no hace nada o lanza una excepción, es muy probable que estés violando el principio de sustitución de Liskov.

Si tu código estaba usando un método que para algunas concreciones ahora lanza una excepción, ¿cómo puedes estar seguro de que todo sigue funcionando?

[8] https://en.wikipedia.org/wiki/Barbara_Liskov

Otra herramienta que te avisará fácilmente son los tests. Si los tests de la clase padre no funcionan para la hija, también estarás violando este principio.

Veremos un ejemplo con el primer caso.

Ejemplo

En la vida real tenemos claro que un elefante es un animal. Imaginemos que tenemos la clase `Animal` que representa un animal, y les damos a los animales la propiedad de andar y saltar:

```
1  open class Animal {
2      open fun walk() { ... }
3      open fun jump() { ... }
4  }
```

Y tenemos una parte del código donde recibimos un animal, y necesitamos que el animal salte:

```
1  fun jumpHole(a: Animal){
2      a.walk()
3      a.jump()
4      a.walk()
5  }
```

Ahora nos creamos un elefante. Pero claro, un elefante no puede saltar, así que decidimos lanzar una excepción para asegurarnos de detectarlos si esto ocurre:

```
1  class Elephant : Animal() {
2      override fun jump() =
3          throw Exception("Los elefantes no pueden saltar")
4
5  }
```

Ahora en todos los sitios donde estemos usando `jumpHole()`, si el animal es un elefante, tendremos una excepción.

Mal asunto, ¿no?

¿Cómo lo solucionamos?

Aquí lo que tenemos que entender es que la abstracción que hemos decidido hacer no es correcta.

Hay animales que no saltan, así que estamos dando por ciertos casos que se pueden volver en nuestra contra.

Por tanto, las clases tienen que representar esos posibles estados inequívocamente, y las funciones usar las abstracciones que necesiten.

¿Qué podríamos hacer en este caso? Plantear un tipo de animal ligero que sí que puede saltar, mientras que damos por hecho que los animales en general no pueden hacerlo:

```
1  open class Animal {
2      open fun walk() {  }
3  }
4
5  open class LightweightAnimal : Animal() {
6      open fun jump() {  }
7  }
```

Esto nos permite definir animales que sí pueden saltar y otros que no. Por ejemplo un perro y un elefante:

```
1  class Dog: LightweightAnimal()
2
3  class Elephant: Animal()
```

Y la función de `jumpHole()` solo admitiría animales que pueden saltar:

```
fun jumpHole(a: LightweightAnimal){
    a.walk()
    a.jump()
    a.walk()
}
```

Elegir las abstracciones correctas muchas veces no es fácil, pero tenemos que intentar limitar al máximo cuál es su alcance para no pedir más de lo que se necesita ni menos.

Esta es la solución que obtendríamos mediante herencia aplicando el Principio de Liskov, pero también se podría haber solucionado mediante composición.

La herencia nos puede generar una jerarquía de clases muy compleja si hay muchos tipos de animales, así que en función del problema hay que plantearse cuál merece la pena usar.

Esta segunda opción es la que veremos con el Principio de segregación de interfaces.

Conclusión

El principio de Liskov **nos ayuda a utilizar la herencia de forma correcta**, y a tener mucho más cuidado a la hora de extender clases.

En la práctica nos ahorrará muchos errores derivados de nuestro afán por modelar lo que vemos en la vida real en clases siguiendo la misma lógica.

No siempre hay una modelización exacta, por lo que este principio nos ayudará a descubrir la mejor forma de hacerlo.

Principio de Segregación de Interfaces

El principio de segregación de interfaces viene a decir que **ninguna clase debería depender de métodos que no usa**. Por tanto, cuando creemos interfaces que definan comportamientos, es importante estar seguros de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. **En caso contrario, es mejor tener varias interfaces más pequeñas**.

Las interfaces nos ayudan a desacoplar módulos entre sí. Esto es así porque si tenemos una interfaz que explica el comportamiento que el módulo espera para comunicarse con otros módulos, nosotros siempre podremos crear una clase que lo implemente de modo que cumpla las condiciones.

El módulo que describe la interfaz no tiene que saber nada sobre nuestro código y, sin embargo, nosotros podemos trabajar con él sin problemas.

El problema

La problemática surge cuando esas interfaces intentan definir más cosas de las debidas, lo que se denominan *fat interfaces*.

Probablemente ocurrirá que **las clases hijas acabarán por no usar muchos de esos métodos**, y habrá que darles una implementación.

Muy habitual es lanzar una excepción, o simplemente no hacer nada.

Pero, al igual que vimos en algún ejemplo en el principio de sustitución de Liskov, **esto es peligroso**. Si lanzamos una excepción, es más que probable que el módulo que define esa interfaz use el método en algún momento, y esto hará fallar nuestro programa.

El resto de implementaciones "por defecto" que podamos dar, pueden generar efectos secundarios que no esperemos, y a los que sólo podemos responder conociendo el código fuente del módulo en cuestión, cosa que no nos interesa.

¿Cómo detectar que estamos violando el Principio de segregación de interfaces?

Como comentaba en los párrafos anteriores, si **al implementar una interfaz ves que uno o varios de los métodos no tienen sentido y te hace falta dejarlos vacíos o lanzar excepciones**, es muy probable que estés violando este principio.

Si la interfaz forma parte de tu código, divídela en varias interfaces que definan comportamientos más específicos.

Recuerda que no pasa nada porque una clase ahora necesite implementar varias interfaces. El punto importante es que use todos los métodos definidos por esas interfaces.

Ejemplo

Imagina que tienes una tienda de CDs de música, y que tienes modelados tus productos de esta manera:

```
interface Product {
    val name: String
    val stock: Int
    val numberOfDisks: Int
    val releaseDate: Int
}

class CD : Product {
    ...
}
```

El producto tiene una serie de propiedades que nuestra clase CD sobrescribirá de algún modo.

Pero ahora has decidido ampliar mercado, y empezar a vender DVDs también.

El problema es que para los DVDs necesitas almacenar también la clasificación por edades, porque tienes que asegurarte de que no vendes películas no adecuadas según la edad del cliente.

Lo más directo sería simplemente añadir la nueva propiedad a la interfaz:

```
interface Product {
    ...
    val recommendedAge: Int
}
```

¿Qué ocurre ahora con los CDs? Que se ven obligados a implementar `recommendedAge`, pero no van a saber qué hacer con ello, así que lanzarán una excepción:

```
class CD : Product {
    ...
    override val recommendedAge: Int
        get() = throw UnsupportedOperationException()
}
```

Con todos los problemas asociados que hemos visto antes.

Además, se forma una dependencia muy fea, en la que cada vez que añadimos algo a `Product`, nos vemos obligados a modificar `CD` con cosas que no necesita.

Podríamos hacer algo tal que así:

```
interface DVD : Product {
    val recommendedAge: Int
}
```

Y hacer que nuestras clases extiendan de aquí.

Esto solucionaría el problema a corto plazo, pero hay algunas cosas que pueden seguir sin funcionar demasiado bien.

Por ejemplo, si hay otro producto que necesite categorización por edades, necesitaremos repetir parte de esta interfaz.

Además, esto no nos permitiría realizar operaciones comunes a productos que tengan esta característica.

La alternativa es segregar las interfaces, y que cada clase utilice las que necesite. Tendríamos por tanto una interfaz nueva:

```
interface AgeAware {
    val recommendedAge: Int
}
```

Y ahora nuestra clase DVD implementará las dos interfaces:

```
class CD : Product {
    ...
}

class DVD : Product, AgeAware {
    ...
}
```

La ventaja de esta solución es que ahora podemos tener código `AgeAware`, y todas las clases que implementen esta interfaz podrían participar en código común.

Imagina que no vendes sólo productos, sino también actividades, que necesitarían una interfaz diferente.

Estas actividades también podrían implementar la interfaz `AgeAware`, y podríamos tener código como el siguiente, independientemente del tipo de producto o servicio que vendamos:

```
fun checkUserCanBuy(user: User, ageAware: AgeAware)
    = user.age >= ageAware.recommendedAge
```

¿Qué hacer con código antiguo?

Si ya tienes código que utiliza *fat interfaces*, la solución puede ser **utilizar el patrón de diseño "Adapter"**. El patrón Adapter[9] nos permite convertir unas interfaces en otras, por lo que puedes usar adaptadores que conviertan la interfaz antigua en las nuevas.

[9] https://es.wikipedia.org/wiki/Adapter_(patr%C3%B3n_de_dise%C3%B1o)

Conclusión

El principio de segregación de interfaces nos ayuda a **no obligar a ninguna clase a implementar métodos que no utiliza**. Esto nos evitará problemas que nos pueden llevar a errores inesperados y a dependencias no deseadas. Además nos ayuda a reutilizar código de forma más inteligente.

Principio de Inversión de Dependencias

Si te resultó interesante el principio de segregación de interfaces, el último de los principios SOLID, el principio de inversión de dependencias, seguramente sea el que más cambie tu forma de programar una vez empieces a aplicarlo.

Este principio es una técnica básica, y será **el que más presente tengas en tu día a día si quieres hacer que tu código sea testable y mantenible**.

Gracias al principio de inversión de dependencias, podemos hacer que el código que es el núcleo de nuestra aplicación no dependa de los detalles de implementación, como pueden ser el framework que utilices, la base de datos, cómo te conectes a tu servidor…

Todos estos aspectos se especificarán mediante interfaces, y el núcleo no tendrá que conocer cuál es la implementación real para funcionar.

La definición[10] que se suele dar es:

A. Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.
B. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

Pero entiendo que sólo con esto no te quede muy claro de qué estamos hablando, así que voy a ir desgranando un poco el problema, cómo detectarlo y un ejemplo.

El problema

En la programación vista desde el modo tradicional, cuando un módulo depende de otro módulo, se crea una nueva instancia y la utiliza sin más complicaciones.

[10] https://en.wikipedia.org/wiki/Dependency_inversion_principle

Esta forma de hacer las cosas, que a primera vista parece la más sencilla y natural, nos va a traer bastantes problemas posteriormente, entre ellos:

- **Las parte más genérica de nuestro código (lo que llamaríamos el dominio o lógica de negocio) dependerá por todas partes de detalles de implementación**. Esto no es bueno, porque no podremos reutilizarlo, ya que estará acoplado al framework de turno que usemos, a la forma que tengamos de persistir los datos, etc. Si cambiamos algo de eso, tendremos que rehacer también la parte más importante de nuestro programa.
- **No quedan claras las dependencias**: si las instancias se crean dentro del módulo que las usa, es mucho más difícil detectar de qué depende nuestro módulo y, por tanto, es más difícil predecir los efectos de un cambio en uno de esos módulos. También nos costará más tener claro si estamos violando algunos otros principios, como el de Responsabilidad Única.
- **Es muy complicado hacer tests**: Si tu clase depende de otras y no tienes forma de sustituir el comportamiento de esas otras clases, no puedes testarla de forma aislada. Si algo en los tests falla, no tendrías forma de saber de un primer vistazo qué clase es la culpable.

¿Cómo detectar que estamos violando el Principio de inversión de dependencias?

Este es muy fácil: **cualquier instanciación de clases complejas o módulos es una violación de este principio**.

Además, si escribes tests te darás cuenta muy rápido, en cuanto no puedas probar esa clase con facilidad porque dependa del código de otra clase.

Te estarás preguntando entonces cómo vas a hacer para darle a tu módulo todo lo que necesita para trabajar. Tendrás que utilizar alguna de las alternativas que existen para suministrarle esas dependencias.

Aunque hay varias, las que más se suelen utilizar son **mediante constructor y mediante `setters`** (funciones que lo único que hacen es asignar un valor).

¿Y entonces auién se encarga de proveer las dependencias? Lo más habitual es utilizar un **inyector de dependencias**: un módulo que se encarga de instanciar los objetos que se necesiten y pasárselos a las nuevas instancias de otros objetos.

Se puede hacer una inyección muy sencilla a mano, o usar alguna de las muchas librerías que existen si necesitamos algo más complejo.

En cualquier caso esto se escapa un poco del objeto de este artículo.

Si quieres ver un caso particular y algo más sobre inyección, puedes leer este artículo sobre inyección de dependencias en Android con Hilt[11].

Ejemplo

Imaginemos que tenemos una cesta de la compra que lo que hace es almacenar la información y llamar al método de pago para que ejecute la operación. Nuestro código sería algo así:

```
1  class Shopping { ... }
2
3  class ShoppingBasket {
4      fun buy(shopping: Shopping?) {
5          val db = SqlDatabase()
6          db.save(shopping)
7          val creditCard = CreditCard()
8          creditCard.pay(shopping)
9      }
10 }
11
12 class SqlDatabase {
13     fun save(shopping: Shopping?) {
14         // Saves data in SQL database
15     }
16 }
17
18 class CreditCard {
19     fun pay(shopping: Shopping?) {
20         // Performs payment using a credit card
```

[11] https://devexperto.com/dagger-hilt/

```
    }
}
```

Aquí estamos incumpliendo todas las reglas que impusimos al principio. Una clase de más alto nivel, como es la cesta de la compra, está dependiendo de otras de bajo nivel, como cuál es el mecanismo para almacenar la información o para realizar el método de pago. Se encarga de crear instancias de esos objetos y después utilizarlas.

Piensa ahora qué pasa si quieres añadir métodos de pago, o enviar la información a un servidor en vez de guardarla en una base de datos local. No hay forma de hacer todo esto sin desmontar toda la lógica. ¿Cómo lo solucionamos?

Primer paso, dejar de depender de concreciones. Vamos a crear interfaces que definan el comportamiento que debe dar una clase para poder funcionar como mecanismo de persistencia o como método de pago:

```
interface Persistence {
    fun save(shopping: Shopping?)
}

class SqlDatabase : Persistence {
    override fun save(shopping: Shopping?) {
        // Saves data in SQL database
    }
}

interface PaymentMethod {
    fun pay(shopping: Shopping?)
}

class CreditCard : PaymentMethod {
    override fun pay(shopping: Shopping?) {
        // Performs payment using a credit card
    }
}
```

¿Ves la diferencia? Ahora ya no dependemos de la implementación particular que decidamos. Pero aún tenemos que seguir instanciándolo en `ShoppingBasket`.

Nuestro segundo paso es invertir las dependencias. Vamos a hacer que estos objetos se pasen por constructor:

```
class ShoppingBasket(
    private val persistence: Persistence,
    private val paymentMethod: PaymentMethod
) {
    fun buy(shopping: Shopping?) {
        persistence.save(shopping)
        paymentMethod.pay(shopping)
    }
}
```

¿Y si ahora queremos pagar por Paypal y guardarlo en servidor? Definimos las concreciones específicas para este caso, y se las pasamos por constructor a la cesta de la compra:

```
class Server : Persistence {
    override fun save(shopping: Shopping?) {
        // Saves data in a server
    }
}

class Paypal : PaymentMethod {
    override fun pay(shopping: Shopping?) {
        // Performs payment using Paypal account
    }
}
```

Ya hemos conseguido nuestro objetivo. Además, si ahora queremos testear `ShoppingBasket`, podemos crear Test Doubles[12] para las dependencias, de forma que nos permita probar la clase de forma aislada.

[12] https://en.wikipedia.org/wiki/Test_double

Conclusión

Como ves, este mecanismo nos obliga a organizar nuestro código de una manera muy distinta a como estamos acostumbrados, y en contra de lo que la lógica dicta inicialmente, pero a la larga **compensa por la flexibilidad que otorga** a la arquitectura de nuestra aplicación.

Y con esto terminamos la serie de artículos sobre principios SOLID.