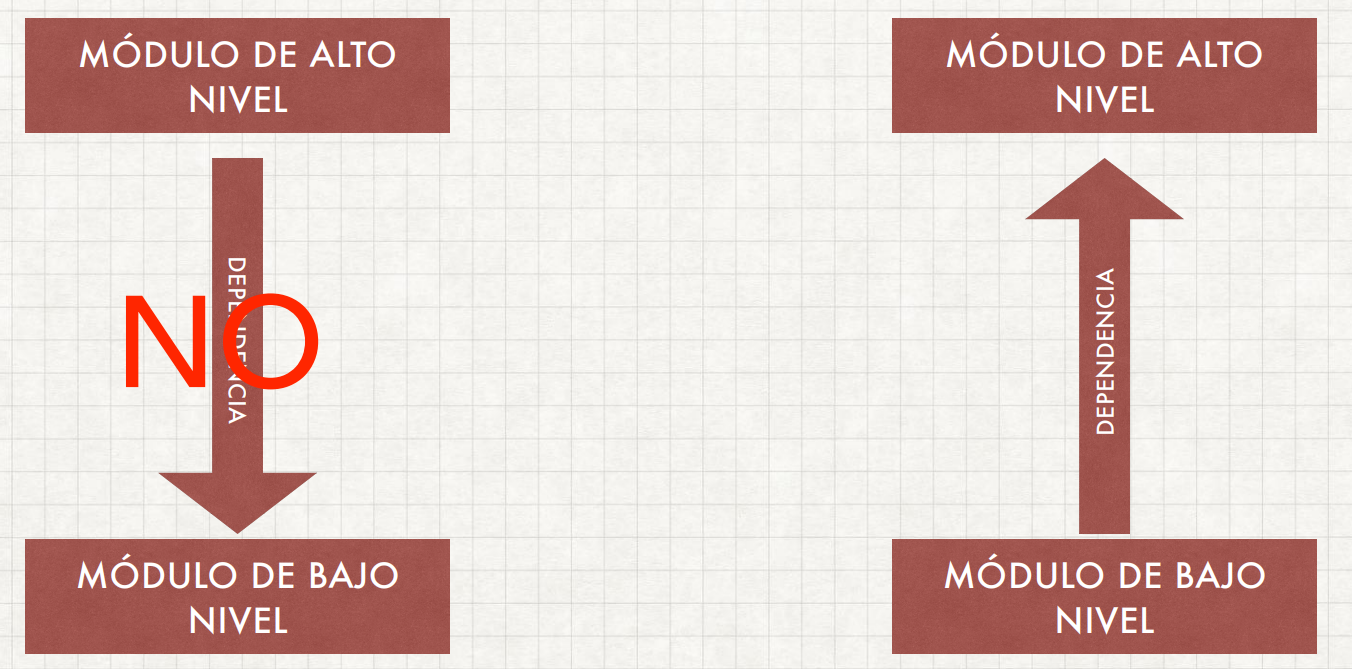
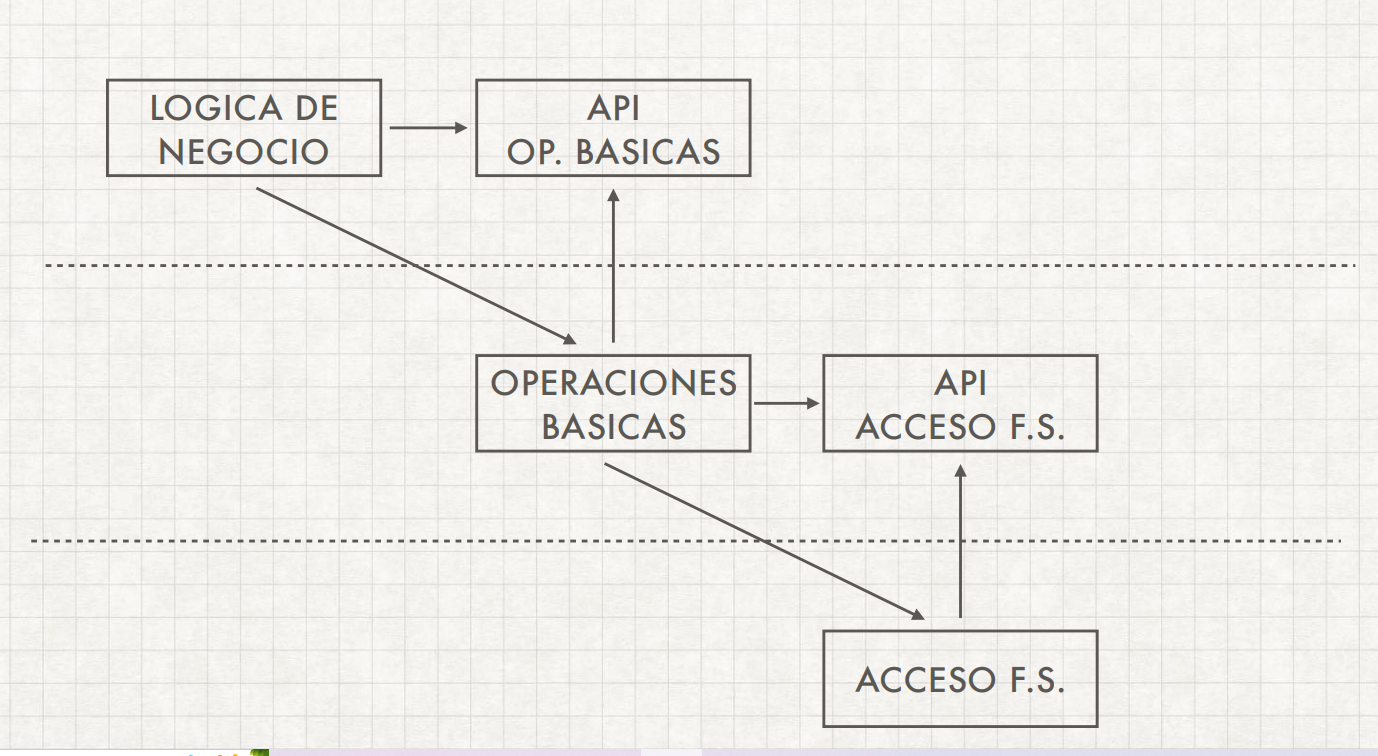
**1.5. Dependency Inversion Principle:**

- Hi-level modules should not depend on low-level modules. Both modules should depend on abstractions.

- Abstractions should not depend upon details. Details should depend upon abstractions.





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](https://en.wikipedia.org/wiki/Dependency_inversion_principle) 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.

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](https://devexperto.com/principio-responsabilidad-unica).
* **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.

## **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:

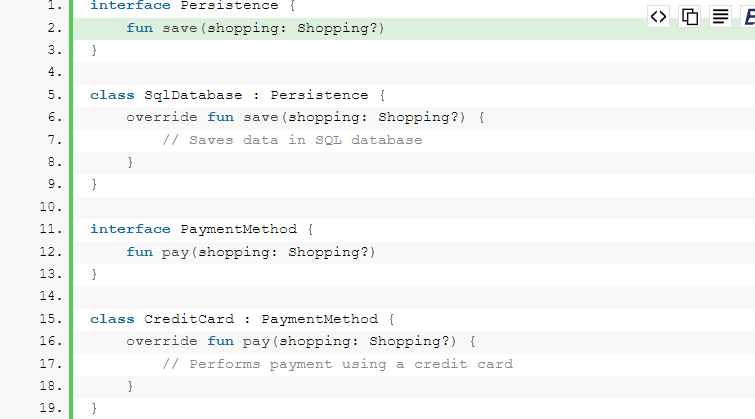


Aquí estamos incumpliendo todas las reglas que impusimos al principio. Una clase de más alto nivel, como es la cesta de la compra, está dependiendo de otras de bajo nivel, como cuál es el mecanismo para almacenar la información o para realizar el método de pago. Se encarga de crear instancias de esos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

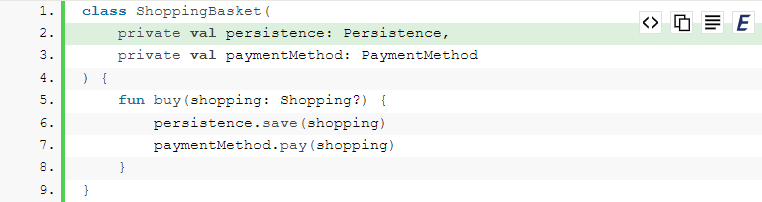
**¿Cómo lo solucionamos?**

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:

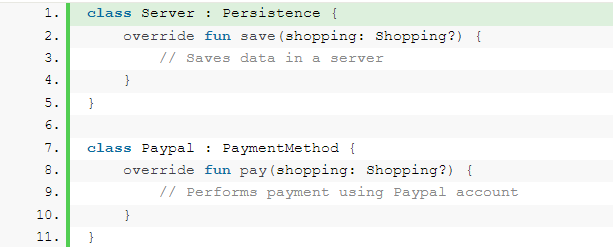


¿Ves la diferencia? Ahora ya no dependemos de la implementación particular que decidamos. Pero aún tenemos que seguir instanciándolo en ShoppingBasket.

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



¿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:



Ya hemos conseguido nuestro objetivo. Además, si ahora queremos testear ShoppingBasket, podemos crear [Test Doubles](https://en.wikipedia.org/wiki/Test_double) para las dependencias, de forma que nos permita probar la clase de forma aislada.

## **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.