# Clean Architecture en iOS

Principios y buenas prácticas de diseño aplicadas en iOS

Yair Carreno

# Clean Architecture en iOS

## Principios y buenas prácticas de diseño aplicadas en iOS.

Yair Carreno

Este libro está a la venta en http://leanpub.com/clean-architecture-en-ios

Esta versión se publicó en 2020-06-06


Éste es un libro de Leanpub. Leanpub anima a los autores y publicadoras con el proceso de publicación. Lean Publishing es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener retroalimentación del lector hasta conseguir el libro adecuado.

# Índice general

# Principios de diseño

## Inversión de dependencia

La inversión de dependencia es un principio fundamental sobre el cual se sustenta muchos conceptos usados en diferentes estilos de arquitectura incluyendo *Clean Architecture*, de allí la importancia de comprender muy bien este principio.

Para entender el principio el lector debe imaginar que tiene una relación entre dos elementos como se muestra en la figura 1.1, en donde el elemento **A** depende del elemento **B**.

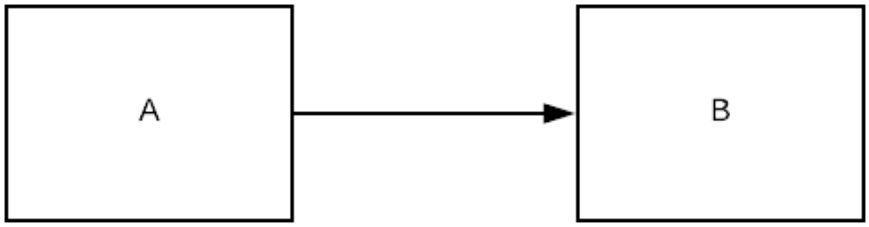


**Figura 1.1 Dependencia entre elementos**

¿Qué significa que el elemento **A** depende de **B**?

Significa que el elemento **A** está utilizando alguna funcionalidad o servicio ofrecido por **B** para cumplir con alguna tarea, es decir que el elemento **A** depende del elemento **B** para cumplir con sus funciones a cabalidad. A ese tipo de relación se le llama *dependencia*.

Ahora bien, ¿Cómo se hace para que la dependencia se invierta, es decir que el elemento **A** no dependa directamente de **B** sino que al contrario el elemento **B** dependa de **A**?

Esto se logra a través de un componente que sirve de intermediario entre el elemento **A** y **B**. Este componente es conocido en algunos lenguajes de programación como *interface*, en iOS y en particular en Swift el equivalente es un *protocol*.

La función principal de la *interface* no se limita solo a invertir la relación de dependencia, también permite separar la abstracción de la implementación, es decir desacoplar los componentes. El desacoplamiento es vital en cualquier arquitectura y es una práctica recomendada en el diseño de componentes de software.

En la figura 1.2 se muestra la nueva relación establecida con la *interface* agregada.

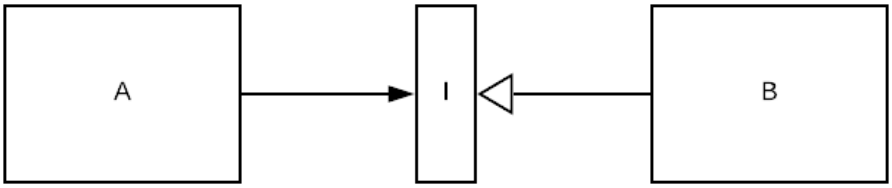


**Figura 1.2 Inversión de dependencia entre elementos**

En este caso, el elemento **A** utiliza la *interface* **I** y a su vez el elemento **B** hereda de la *interface* **I** e implementa las operaciones definidas en **I**.

En esta nueva relación al elemento **A** ya no le importa lo que ocurra con **B** ya que solo está relacionado con el elemento **I**, mientras que por otro lado el elemento **B** si depende del elemento **I** de una forma desacoplada con la libertad de implementar las operaciones definidas en **I** a su conveniencia.

¿Qué ocurriría si fuera necesario reemplazar el elemento **B** por una determinada razón, ya sea que el elemento **B** se volvió obsoleto o requiere una actualización?

En ese caso se lleva a cabo la modificación agregando un nuevo elemento **C** con la única condición de que debe conformar el elemento **I** y por lo tanto implementar las operaciones definidas en **I**.

En la figura 1.3 se muestra la nueva relación:

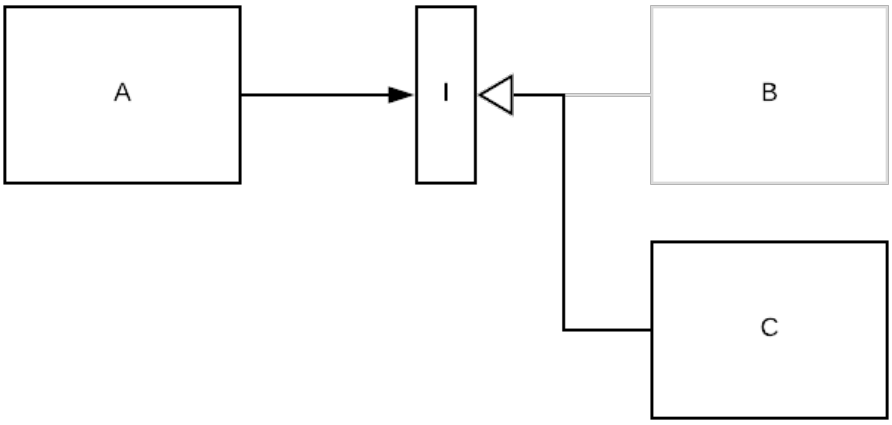


**Figura 1.3 Desacoplamiento e inversión de dependencia entre elementos**

Se puede notar que no hubo necesidad de tocar el elemento **A** gracias al desacoplamiento y la inversión de la dependencia.

En estilos de arquitectura tales como *Clean Architecture* o *Hexagonal Architecture* se utiliza este

principio de diseño que tiene sus orígenes en el patrón *Separated Interface*[1]. Es aplicado en las fronteras de los niveles de la aplicación, tema que se revisará en el siguiente capítulo llamado Niveles en una aplicación.

Los elementos tipo **I** es decir las *interfaces* son conocidos como *puertos* y a los componentes que los implementan como por ejemplo en este caso **B** y **C**, se les conoce como *Adapters*.

# Inversión de control

Existen dos tipos de relaciones entre los elementos presentados en la figura 1.4

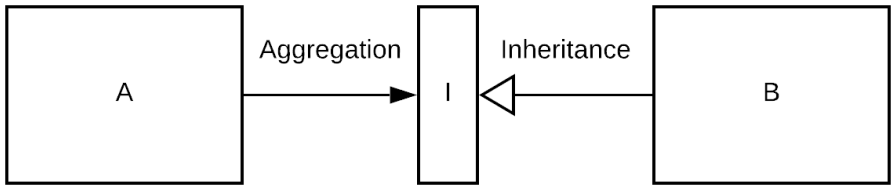


**Figura 1.4 Relaciones de agregación vs herencia**

La relación entre el elemento **A** y el elemento **I** es una relación de *agregación* mientras que la relación entre el elemento **B** y el elemento **I** es de *herencia*.

### Relaciones de asociación, agregación y composición

En el apéndice A se describe cómo distinguir cuando una relación entre elementos es de asociación, agregación o composición.

Existe un mecanismo conocido como *delegación*[2] que permite usar dicha relación de *agregación* para separar adecuadamente las responsabilidades de un elemento y encomendar ciertas responsabilidades a otro elemento.

Dicho mecanismo es lo que se conoce como *inversión de control*, ya que un elemento puede delegarle la ejecución de una tarea a otro elemento, en otras palabras le entrega el control de una tarea a otro elemento.

Para entender mejor el concepto se muestra la figura 1.5 con un ejemplo de pseudocódigo:

[1]Separated Interface: Patterns of enterprise application architecture, page 476 by Martin Fowler.
[2]Delegation: Design Patterns - Elements of Reusable Object-Oriented Software, page 20 by GoF.

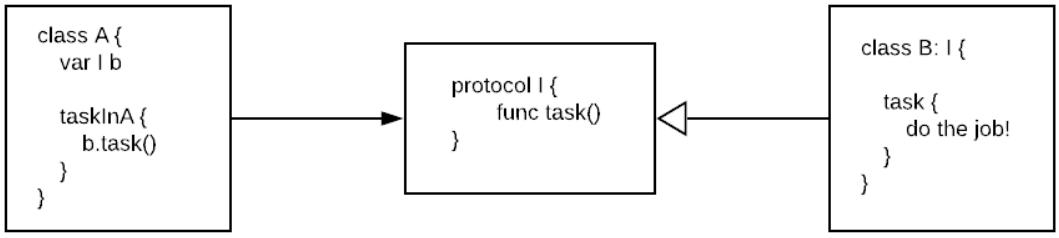


**Figura 1.5 Ejemplo de pseudocódigo en inversión de control**

El elemento **A** a través de agregación conoce la abstracción de **B** más no su implementación. **A** contiene una tarea llamada *taskInA* en la cual delega parte de la operación al elemento **B** a través de la tarea *task*.

El elemento **A** desconoce cómo la tarea delegada es ejecutada en el elemento **B**, solo sabe que el elemento **B** tiene la capacidad de realizar dicha tarea.

La aplicación de *inversión de control* aporta al cumplimiento del principio *Single Responsibility*[3] ya que permite que cada elemento se encargue solo de las funciones para las cuales fueron diseñados y tenga una única razón de cambio.

# Inyección de dependencia

Inyección de dependencia[4] es un patrón de diseño que aplica el mecanismo de *inversión de control* de tal forma que un elemento **A** puede conocer un elemento **B** a través de una relación de *agregación* sin que la responsabilidad de la creación de la instancia del elemento **B** quede del lado del elemento **A**.

En la figura 1.6 se puede notar que el elemento **A** conoce de la existencia de un elemento de tipo **B** que conforma la interfaz **I**, sin embargo el elemento **A** desconoce la forma en que el elemento **B** se instancia o se crea.

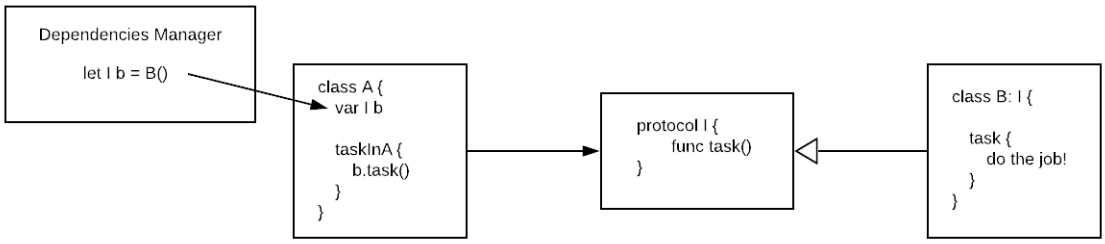


**Figura 1.6 Inyección de dependencias**

[3]The Single Responsibility Principle: *Clean Architecture*, page 61 by Robert C. Martin.
[4]Dependency Injection: *Clean Code* Elements of Reusable Object-Oriented Software, chapter 11 by Robert C. Martin.

Así que, ¿Quién crea la instancia del elemento **B**?

Esta tarea es delegada a una entidad o sistema externo comúnmente conocido como *Gestor de inyección de dependencias*. Un gestor de inyección de dependencias se puede implementar en la aplicación a través de:

- El patrón *Service Locator*.
- A través de la integración de librerías diseñadas para dicha función de inyectar objetos[5].
- A través de la implementación manual haciendo inyección por constructor, por método o por propiedad.

Para concluir y como se mencionó al comienzo, estos tres principios y conceptos recientemente explicados se utilizan en gran medida en estilos de arquitecturas tales como *Clean Architecture* o *Hexagonal Architecture*, de allí la importancia de comprender dichos principios a cabalidad, reconocerlos cuando se aplican y entender sus objetivos.

[5] Resolver by Michael Long, Dagger by Google.