# Inversión de dependencia, arquitectura hexagonal y otras yerbas

La inversión de dependencia es uno de los 5 principios SOLID. Se lo suele definir como:

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

Los problemas a evitar son los siguientes:

* **Las partes más genéricas de nuestro código (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 testearla de forma aislada. Si algo en los tests falla, no tendrías forma de saber de un primer vistazo qué clase es la culpable.

Ejemplo: se requiere que la clase ElectricalPowerSwitch definida en code\_01.py pueda manejar un ventilador además de una luz.

En este código sé que ElectricalPowerSwitch depende de LightBulb.

Como primera medida creo una clase abstracta y hago que LightBulb herede de esta. A primera vista no hay cambios apreciables. Funciona todo igual y las dependencias siguen sin cambios.

La inversión de dependencia la introducimos en el code\_03.py Hacemos que ElectricalPowerSwitch deje de depender de LightBulb. En lugar de eso le pasamos por su constructor una instancia de la clase abstracta Switchable.

De esta manera es muy fácil crear la clase Fan que hereda de Switchable e instanciar 2 veces ElectricalPowerSwitch pasandole la clase correspondiente.

**Arquitectura Hexagonal**

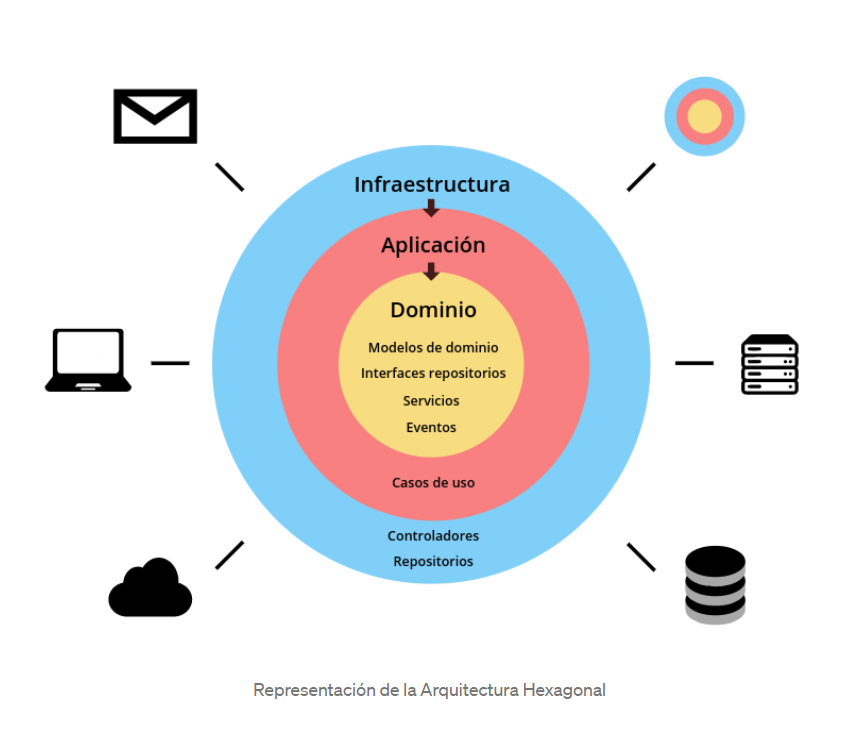
o arquitectura de puertos y adaptadores



La arquitectura hexagonal fue inventada por Alistair Cockburn para evitar los errores del diseño orientado a objetos, como por ejemplo la contaminación de la interface con la lógica de negocios. Fue publicada por primera vez en 2005.

Su objetivo es crear componentes de aplicación débilmente acoplados que se puedan conectar fácilmente a su entorno de software por medio de puertos y adaptadores. Esto hace que los componentes sean intercambiables en cualquier nivel y facilita la automatización de las pruebas.

El término "hexagonal" proviene de las convenciones gráficas que muestran el componente de la aplicación como una celda hexagonal. El propósito no era sugerir que habría seis interfaces / puertos, sino dejar suficiente espacio para representar las diferentes interfaces necesarias entre el componente y el mundo externo.



Lo que se busca con esta arquitectura es que el dominio de la aplicación no dependa de nada externo a su nivel.

Una estructura típica sería la siguiente:

system

├── main.py

│

├── app/  
│ ├── adapter/

│ ├── light\_bulb\_repository.py  
│ │ └── fan\_repository.py

│ │  
│ └── domain/  
│ ├── service.py  
│ └── switchable\_repository.py

│  
└── tests/

└── test\_service.py

En esta estructura se definió un paquete llamado domain. Dentro del mismo no se permite importar nada que no esté definido en el mismo paquete.

Esta arquitectura nos da gran flexibilidad a la hora de diseñar y agregarle mejoras al código. Podemos trabajar toda nuestra lógica de negocio o dominio sin preocuparnos por nada externo. Para nosotros es indiferente el exterior a domain. No existe.

Por ejemplo, es muy fácil implementar TDD. Definida la estructura de directorios y definido nuestro repositorio abstracto, podemos generar los test que definirán al servicio. Es muy fácil mockear la clase abstracta