Test unitarios con SOLID

# Tema 1: SOLID

En realidad, realizar test unitarios no tiene absolutamente ningún misterio ni dificultad siempre que el código se implemente correctamente, pero… ¿qué podemos considerar “correctamente”? ¿Hay alguna “norma” que podamos seguir para desarrollar un código “correcto” que sea fácil de testear?

Pues la realidad es que existen varias prácticas reconocidas “correctas”, de las cuales la más importantes son sin lugar a dudas las definidas en los principios SOLID.

## S (Single Responsability Principle):

Cada módulo/paquete/clase/método debe tener una y solo una única responsabilidad sobre el total de la funcionalidad total del software. El nivel de atomicidad de la responsabilidad se puede determinar por lo extenso de la definición del objeto: Un método debe ser lo más atómico, realizando acciones muy concretas y devolviendo el control lo antes posible a su invocador. Una clase debe contener métodos que se limiten a cumplir con el objetivo de dicha clase, siendo lo ideal que exista un único método público (aunque dicho caso “ideal” es algo que, aunque se debe buscar, no es una norma y dependerá del caso). Un paquete debe tener clases que se limiten al comportamiento especificado en el paquete. Un módulo debe encargase de una parte concreta del total del software.

Pongamos algunos ejemplos para comprender mejor esto:

* Un software con el **módulo** “clientes” solo debe implementar la funcionalidad relacionada con los clientes. Las partes transversales del software pueden pertenecer a otro módulo llamado “shared”, “común”, o “core”.

NOTA: Los distintos módulos pueden tener dependencias entre sí, de modo que un fragmento de código se considera “transversal” solo cuando no pertenece de forma clara a un único módulo. Aunque el código relacionado a clientes podría ser usado en muchos otros módulos, el concepto de cliente tiene la identidad suficiente como para tener el suyo propio.

* Un **paquete** “com.santander.clientes.domain” solo debe contener clases que definan las reglas del dominio de los clientes (en el contexto de lo que se entiende como “cliente” por el santander). Se puede considerar “reglas del dominio” las reglas de negocio, el modelo, y las interfaces con las que se trabaje. Un paquete “com.santander.clientes.infrastructure” solo debe contener clases externas a las reglas del dominio, como son los frameworks o las implementaciones de los DAO y/o Repository.
* Una **interfaz** “com.santander.clientes.domain.ClienteDao” solo puede definir (no implementar) los métodos necesarios para consultar, modificar, borrar y persistir clientes.
* Una **clase** “com.santander.clientes.domain.NumeroDeTelefono” será una clase que defina los atributos, las normas, y los métodos necesarios para trabajar con uno.
* Una **clase** “com.santander.clientes.infrastructure.OracleClienteDao” solo puede ser una implementación de la interfaz ClienteDao, con la implementación necesaria de los métodos para trabajar con una BBDD Oracle (Y cualquier otro método que necesite para realizar su función NO deberá ser público, aunque como en todo existen excepciones).
* Una **clase** “com.santander.clientes.infrastructure.MongoDbClienteDao” solo puede ser una implementación de la interfaz ClienteDao, con la implementación necesaria de los métodos para trabajar con una BBDD MongoDB (Y cualquier otro método que necesite para realizar su función NO debería ser público, aunque como en todo existen excepciones).
* Un **método** Cliente.setNombre no deberá realizar otra tarea más que asignar un nombre (y si las reglas de negocio especifican alguna validación necesaria, sería necesario realizarla en otro método que sea invocado desde este).
* Un **método** Cliente.validarNombre no deberá realizar otra tarea más que validar un nombre.
* Un **método** ClienteDao.buscarCliente no deberá realizar otra tarea más que buscar y devolver un cliente (y si hace falta alguna transformación intermedia, se deberá invocar a otro método que lo haga, que como dicha operación no forma parte del trabajo de ClienteDao, deberá ser realizado por una clase distinta. PE: ResultSetToClienteMapper.buildCliente).

Este es uno de los principios más difíciles de aplicar por la interdependencia obvia entre los módulos/clases/métodos que existe en todo software y por lo subjetivo del concepto “responsabilidad única”, pero a la vez es uno de los más importantes para mantener bajo el acoplamiento a la vez que se consigue una alta cohesión.

## O (Open/Closed Principle)

Las clases deben estar abiertas para extensión, pero nunca para modificación. Esto no implica que no se puedan corregir bugs, sino que **todo comportamiento nuevo** **debe poder ser implementado sin modificar el código antiguo** en **una clase que extienda de la antigua** (opción viable si no se definieron interfaces), o **en una** **clase completamente nueva que implemente cierta interfaz** (Esta forma sería la más deseable. Si no se hace así, nos podemos ver obligados a sobrescribir el comportamiento de métodos que, aunque puede ser a penas admisible, ofusca el código dificultando su lectura). Algunos patrones de diseño relacionados con este principio son el AbstractFactory y el Strategy.

Este principio es el “más laxo”, y hay que usarlo con sabiduría. Si un nuevo comportamiento necesita una modificación mínima y solo rompería este principio, habría que valorar el costo/beneficio de aplicarlo. Lo más importante de este principio es programar con la idea de que puede ser necesario hacerlo, para así tener el código preparado cuando llegue el caso.

## L: (Liskov Substitution Principle)

Las clases derivadas deben poder ser sustituidas por sus clases base sin afectar el comportamiento del programa. Este principio, aunque confuso, en realidad ya está bastante resuelto en su mayoría con las limitaciones que impone el compilador de Java. En resumen, quiere decir que “se debe respetar el contrato”. Si en una clase A un método acepta por parámetro números enteros y devuelve texto sin posibilidad de lanzar excepciones, entonces toda clase que extienda de A debe poseer ese método que acepte por parámetro números enteros y devuelva texto sin lanzar excepciones. Además, hay que tener en cuenta “la intencionalidad”, ya que forma parte del contrato, pero eso no quiere decir los métodos no puedan ser sobrescritos. Por ejemplo, un método “sumar” no debe restar, pero un método “operar” puede realizar distintas operaciones según su implementación, porque esto no rompe el contrato (siempre que tampoco se modifiquen parámetros, tipo de resultado, y tipo de los errores). **Como java ya lo limita bastante, al final solo hay que prestar atención a “la intención del método en su clase base” para que esta se siga cumpliendo en las clases derivadas.** Como extra, aclarar que aunque este principio habla de clases base y clases derivadas, es algo derivado de la época donde se realizó el estudio, aplicando en realidad a todos los tipos que de alguna forma son heredables y/o pueden heredar: Interfaces, Clases, Clases Abstractas…

Este es uno de los pocos principios innegables y con extremadamente raras excepciones que sean realmente admisibles. Para más información: [Principio de sustitución de Liskov](https://es.wikipedia.org/wiki/Principio_de_sustituci%C3%B3n_de_Liskov)

## I (Interface Segregation Principle)

Las interfaces deben ser específicas para cada cliente y no deben contener métodos que no sean utilizados por el cliente. En realidad, esta regla **es solo un refuerzo de Single Responsability Principle, haciendo a la vez de puente con el siguiente principio.** Sirvepara evitar que por el principio de responsabilidad única se cuelen casos excesivamente genéricos, como por ejemplo las típicas clases Utils que son un cajón desastre donde hay de todo un poco, pero que puede considerarse que no rompen el principio de responsabilidad única. Para ello, te dice que **el cliente de una clase es solo uno, y que los únicos métodos que deben ser públicos en una clase son exclusivamente los que van a ser usados por su único cliente**. Hay que tener en cuenta aquí que con cliente único puede referirse a una clase base o interfaz, de modo que toda clase que extienda dicha clase base también será un cliente válido.

Al contrario de lo que pasa con el principio de responsabilidad única, este principio es realmente fácil de aplicar (sobre todo cuando se aplica a la vez que el siguiente), siendo la pata más importante para mantener un bajo acoplamiento con una alta cohesión.

## D (Dependency Inversion Principle)

El protagonista principal cuando se trata de posibilitar el testing. Sin este principio, simplemente es imposible realizar test unitarios con una cobertura mínimamente decente.

Lo que dice este principio es que **los módulos más importantes** **no deben depender de los módulos menos importantes**. Consideramos como módulos más importantes aquellos módulos/clases/interfaces que implementan/definen las reglas de negocio. Como módulos menos importantes consideramos a aquellos módulos/clases/interfaces que sirven a los objetivos de negocio sin implementar o definir sus reglas. Arquitecturas por capas como la [clean architecture](https://www.google.es/search?q=clean+architecture) están basadas en este principio.

Para conseguir este objetivo, se definen interfaces que pertenecen a las capas de negocio, que se verán implementadas en las capas que sirven para interactuar con el exterior (usuarios del software, bases de datos, ficheros de datos, colas de eventos, etc). Un ejemplo clásico de esto es el patrón Repository, que define una interfaz con los métodos necesarios para persistir elementos de negocio, pero cuya implementación se realiza en capas superiores.

Para el correcto uso del principio de inversión de dependencias es fundamental hablar de una técnica que no forma parte de los principios SOLID, pero es igual de importante…

## EXTRA: Dependency Injection

O inyección de dependencias. Esta técnica se resume en no instanciar las dependencias de una clase en dicha misma clase, inyectándolas desde el exterior ya sea a través de frameworks (sprint), constructor (como argumentos), o simples Factory (Modificando la implementación devuelta por el factory por medio de un fichero de configuración). Un buen ejemplo de su enorme utilidad es el testing, donde no es raro tener que mockear partes del código para poder realizar los tests unitarios sin tener que preocuparnos del comportamiento de otras partes del software que en un test concreto no se están testeando.

Esto, cuando se suma al principio de inversión de dependencias, hace que podamos realizar tests muy personalizados de una forma extremadamente sencilla, sin siquiera ser necesario hacer uso de librerías como Mockito (aunque dicha librería podría facilitar aún más dicho mockeo dependiendo del caso).