Testing-Moq-Pruebas unitarias

Que son las pruebas unitarias?

Tener pruebas **automáticas** es una gran manera de asegurarte de que el software hace lo que los desarrolladores **pretendían** que **hiciera**. Hay multiples tipos de pruebas. Algunas de ellas son **pruebas de integración**, **pruebas web** y muchas mas. Las pruebas **unitarias** son pruebas **individuales** de **componentes** de software y **métodos**.

Las pruebas **unitarias** deberían de **probar ú**nicamente **código** que este al **alcance** del desarrollador. No deberían de probar conceptos de la infraestructura, con esto me refiero a base de datos, file system, y network.

Para crear las pruebas unitarias podemos usar diferentes frameworks:

- xUnit
- NUnit
- **MSTest (**este framework es el que vamos a utilizar**)**

El sentido de las pruebas **unitarias**, como se dice mas arriba, es poder **probar** nuestro **código evitando** probar también sus **dependencias**, asegurándonos que los **errores** se restringa ú**nicamente** de código que efectivamente

queremos **probar**. Para ello, utilizaremos una herramienta que nos permitirá crear Mocks que es un tipo de test doubles. Esa herramienta se llama **Moq**.

Que son los Mocks?

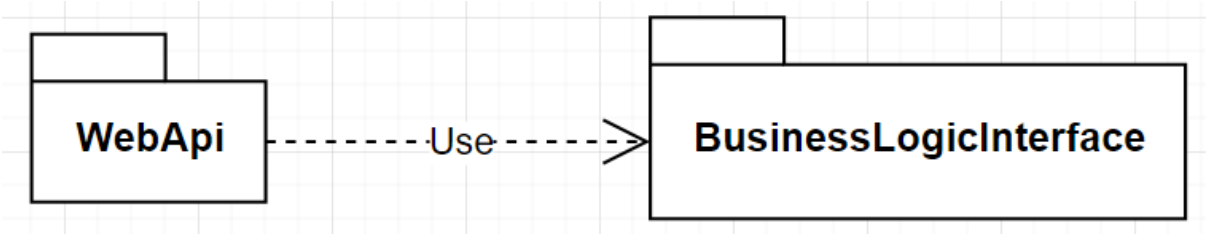
Los **mocks** son unos de los varios ***test doubles*** (objetos que no son reales respecto a nuestro dominio, y que se usan con finalidades de testing) que existen para probar nuestros sistemas. Los mas conocidos son los Mocks y los Stubs, siendo la principal diferencia en ellos, el foco de lo que se esta testeando.

Antes de hacer énfasis en tal diferencia, es importante aclarar que nos referimos a la sección del sistema a probar como **SUT**(system under test). Los **Mocks**, nos permiten verificar la interacción del **SUT** con sus **dependencias**. Los **Stubs**, nos permiten verificar el estado de los objetos que se pasan. Como queremos testear el comportamiento de nuestro código, utilizaremos los primeros.

Tipos de *Test Doubles*

Tipo	Descripción
Dummy	Son objetos que se pasan, pero nunca se usan. Por lo general, solo se utilizan para llenar listas de parámetros que tenemos que pasar si o si.
Fake	Son objetos funcionales, pero generalmente toman algún atajo que los hace inadecuados para la producción (una base de datos en la memoria es un buen ejemplo).
Stubs	Brindan respuestas predefinidas a las llamadas realizadas en el test, por lo general no responden a nada que no se use en el test.
Spies	Son Stubs pero que también registran cierta información cuando son invocados.
Mocks	Son objetos pre-programados con expectativas (son las llamadas que se espera que reciban). De todos estos objetos, los Mocks son los unicos que verifican el comportamiento. Los otros, solo verifican el estado.

Veamos a que nos referimos con dependencias y que es lo que hay que **mockear** con un diagrama. Si se acuerdan el componente **WebApi** depende (**usa**) el componente **BusinessLogicInterface**:



Entonces supongamos que queremos **probar** el método **GetAll** del controller **MovieController**, este método para realizar la operación **utiliza** la interfaz **IMovieLogic** llamando al comportamiento **GetAll**. Si la prueba **no** usa **mock** con **IMovieLogic**, se va a terminar **probando** código de la **implementacion** de ese método, en donde puede aparecer un error en forma de **bug** entonces nuestra prueba falla y no entendemos porque. A este tipo de pruebas que no solamente prueba el código si no que también las dependencias se le llaman pruebas de **integración** y estas no son las que vamos a realizar.

Las pruebas unitarias nos van ayudar a detectar problemas rápidamente del código que se esta probando ya que es el que esta mas a nuestro alcance y nos despreocupamos de los **bugs** de las **dependencias**.

Cuando hacemos pruebas unitarias, queremos probar objetos y la forma en que estos interactuan con otros objetos. Para ello existen las instancias de **Mocks**, es decir, objetos que **simulan** el **comportamiento** externo (en este caso de la interfaz **IMovieLogic**) de un cierto **objeto**.

Consecuencia

Gracias al uso de mocks y de las pruebas unitarias nos vemos forzados a **acoplarnos** a **interfaces** y no a implementaciones lo cual nos provoca un **bajo acoplamiento** entre clases y sus dependencias. Esto nos ayuda mucho también en esos componentes que dependen de un recurso externo como por ejemplo una **librería de terceros**, una **red**, un **archivo**, una **base de datos**.