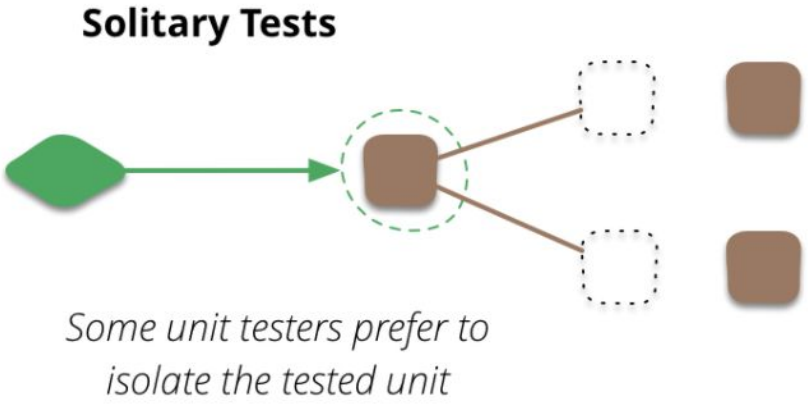
## ¿Tests Solitarios o Sociables?

Al realizar pruebas unitarias de nuestros métodos, podemos encontrarnos con que muchos de ellos necesitan de la colaboración de otros objetos. De hecho el propio paradigma de programación orientada a objetos se basa en la colaboración entre ellos.

Esto da lugar a la pregunta, ¿es realmente una prueba unitaria si estoy llamando a otros objetos? ¿debería limitarme a testear la funcionalidad del método que pertenece a la clase que estoy testeando?

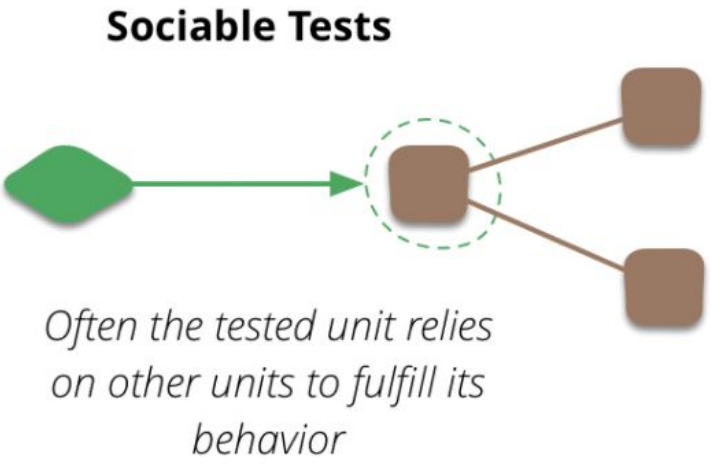
Una de las consecuencias de no aislar mis pruebas de otras clases, es que la eventual falla de una de las clases con las que yo estoy colaborando, puede hacer fallar métodos de otras clases propagando así los errores. Es por ello que muchos programadores prefieren aislar sus métodos de otras clases. Esto da lugar a los llamadas *Tests Solitarios*.

Los *Tests Solitarios* son test unitarios que no colaboran con objetos reales, sino que lo hacen con objetos ficticios que fueron "armados" para el contexto del test en particular. Profundizaremos más sobre estos **objetos ficticios** a continuación.




En cambio, los llamados *Tests Sociables*, son tests que si llaman a objetos reales a menos que la interacción con los mismos sea muy costosa y enlentezca la ejecución de las pruebas. Ejemplos de interacciones costosas pueden ser llamadas de red, como requests a una API, o interacciones con una base de datos. Así mismo, la llamada a objetos que no podemos controlar, por ejemplo APIs, puede causar una falta de determinismo en nuestras pruebas. Ya que la falla o no de las mismas está sujeta al resultado de la invocación de una API externa que no controlamos. He aquí otro beneficio de por qué aislar interacciones costosas.

Finalmente, los *Test Sociables* asumen que los objetos reales funcionarán correctamente al cumplir su objetivo.




¿Qué es mejor? ¿Realizar *Test Sociables* o *Test Solitarios*? No hay una respuesta contundente al respecto. Muchos desarrolladores en la industria prefieren uno sobre el otro y viceversa. Lo importante es entender las consecuencias de la forma que adoptemos para testear, sus beneficios y desventajas, y que seamos consistentes con dicha forma en todo nuestro código fuente.

## ¿Qué son esos *objetos ficticios* que se usan para aislar los tests?

En primer lugar, antes comenzar a explicar lo que son estos *objetos ficticios* es importante distinguir dos conceptos centrales en los objetos. Estos son los conceptos de **estado** y **comportamiento**. El **estado** de un objeto está dado por los atributos que el mismo tiene y los valores de los mismos en determinado momento. El **comportamiento** de un objeto son los **métodos** públicos que el objeto tiene, que a su vez componen lo que se conoce como la **interfaz** del objeto.

```
[TestClass]
0 references
public class PedidoTest
{
    private const string BICICLETA = "BICICLETA";
    private Producto producto = new Producto() { Nombre = BICICLETA, Cantidad = 50 };
    private Deposito deposito = new Deposito();

    [TestInitialize]
    0 references
    public void SetUp()
    {
        this.deposito.AgregarProducto(producto);
    }

    [TestMethod]
    ✓ | 0 references
    public void LaOrdenSeLlenaSiHayStockEnElDeposito()
    {
        Producto productoAPedir = new Producto() { Nombre = BICICLETA, Cantidad = 30 };
        Pedido pedido = new Pedido(productoAPedir);

        pedido.Llenar(deposito);

        Assert.IsTrue(pedido.EstaCompletado);
        Assert.AreEqual(20, deposito.ObtenerStock(BICICLETA));
    }

    [TestMethod]
    ✓ | 0 references
    public void LaOrdenNoSeLlenaSiNoHayStockEnElDeposito()
    {
        Producto productoAPedir = new Producto() { Nombre = BICICLETA, Cantidad = 51 };
        Pedido pedido = new Pedido(productoAPedir);

        pedido.Llenar(deposito);

        Assert.IsFalse(pedido.EstaCompletado);
        Assert.AreEqual(50, deposito.ObtenerStock(BICICLETA));
    }
}
```

En el ejemplo de la imagen anterior, tenemos un depósito que tiene cierto stock de productos. Nosotros podemos hacer pedidos de productos, y el pedido se llena si el stock es suficiente.

En primera instancia agregamos los productos al depósito en un método **SetUp**, que está decorado con la notación **TestInitialize** indicando que el método se ejecuta antes de cada prueba. Esto corresponde con la fase de **setup** de la prueba, valga la redundancia.

Luego, en ambas pruebas, generamos un producto a pedir y creamos un pedido con el mismo. El pedido se llena dependiendo de si hay suficiente stock en el depósito. Para

eso llamamos al método **Llenar** correspondiente, que corresponde a la fase de **ejercitación** de la prueba o *exercise*.

Por último verificamos que la prueba haya sido correcta. Para ello en el primer caso vemos si se disminuyó stock y si el pedido está completado y en el segundo si el stock es el mismo y si el pedido no está completo. Esto corresponde a la fase de **verificación** de la prueba.

Eventualmente de ser necesario, puede existir una fase adicional en el que se limpia el contexto de la prueba, para que la siguiente se ejecute en igualdad de condiciones. Esta fase se conoce como **teardown**.

Es decir, un test bien formulado debe seguir las fases de **Setup, Exercise, Verify y Teardown** en dicho orden, pudiendo saltear la primera y la última si no aplican en el contexto de la prueba, ya que no siempre es necesario armar un contexto o limpiarlo.

En el caso de la prueba anterior estamos testeando el **Pedido** interactuando con objetos reales como **Depósito** y **Producto**, por lo que es un test *sociable*. Así mismo verificamos el éxito o no de la prueba **comprobando el estado** de los objetos **pedido** y **depósito**, preguntando por sus **atributos** directa o indirectamente.

Ahora vamos a introducir lo que se conoce como **Mock**. Los **mocks** son objetos ficticios que son utilizados para únicamente testear el comportamiento del objeto deseado, aislando al mismo de toda interacción. Al contrario de como hicimos en la prueba anterior, donde verificamos el **estado** para determinar el éxito de la prueba, los **mocks** por lo general utilizan **verificación de comportamiento** para determinar lo mismo.

```
[TestMethod]
0 references
public void LaOrdenSeLlenaSiHayStockEnElDeposito()
{
    Producto productoAPedir = new Producto() { Nombre = BICICLETA, Cantidad = 30 };
    Pedido pedido = new Pedido(productoAPedir);

    var mockDeposito = new Mock<Deposito>();

    mockDeposito.Setup(d => d.HayStock(productoAPedir.Nombre, productoAPedir.Cantidad))
        .Returns(true);

    mockDeposito.Setup(d => d.QuitarStock(productoAPedir.Nombre, productoAPedir.Cantidad))
        .Verifiable();

    pedido.Llenar(mockDeposito.Object);

    mockDeposito.Verify(d => d.HayStock(productoAPedir.Nombre, productoAPedir.Cantidad), Times.Once());
    mockDeposito.Verify(d => d.QuitarStock(productoAPedir.Nombre, productoAPedir.Cantidad), Times.Once());
    Assert.IsTrue(pedido.EstaCompletado);
}

[TestMethod]
0 references
public void LaOrdenNoSeLlenaSiNoHayStockEnElDeposito()
{
    Producto productoAPedir = new Producto() { Nombre = BICICLETA, Cantidad = 30 };
    Pedido pedido = new Pedido(productoAPedir);

    var mockDeposito = new Mock<Deposito>();

    mockDeposito.Setup(d => d.HayStock(productoAPedir.Nombre, productoAPedir.Cantidad))
        .Returns(false);

    mockDeposito.Setup(d => d.QuitarStock(productoAPedir.Nombre, productoAPedir.Cantidad))
        .Verifiable();

    pedido.Llenar(mockDeposito.Object);

    mockDeposito.Verify(d => d.HayStock(productoAPedir.Nombre, productoAPedir.Cantidad), Times.Once());
    mockDeposito.Verify(d => d.QuitarStock(productoAPedir.Nombre, productoAPedir.Cantidad), Times.Never());
    Assert.IsFalse(pedido.EstaCompletado);
}
```

Para esto utilizamos la librería Moq de C#, aunque cualquier otra opción es válida.

En este caso realizaremos un test *solitario*, que no interactúa con un objeto **Depósito** real sino con un **mock**. En la fase de **setup** de la prueba establecemos que **HayStock** devuelve **true** en la primer prueba y **false** en la segunda, mientras que **QuitarStock** lo hacemos **Verifiable**, para indicarle a la librería que sobre el verificaremos comportamiento luego.

Luego ejercitamos la prueba llamando a **Llenar**, igual que en el ejemplo anterior. Y por último verificamos el resultado. En este caso consultamos por el estado del pedido, que es marcado como completado únicamente en la primera prueba.

La diferencia con el ejemplo anterior es que en este caso verificamos si los métodos de **Depósito** se llamaron correctamente o no. Es decir, **verificamos comportamiento** y no estado. En la primer prueba sabemos que **HayStock** y **QuitarStock** deben ser llamados

una vez por igual. En la segunda sabemos que **HayStock** debe llamarse pero **QuitarStock** no, dado que según la implementación no tiene sentido quitar stock si no hay.

## Mocks, Doubles, Stubs, … ¿qué son?

Los **doubles** describen a la familia de elementos ficticios que son utilizados en las pruebas para aislar a nuestros objetos de la interacción con objetos reales. Es decir, los **doubles** engloban a los *mocks*, *stubs*, *spies*, etc.

Cualquier **double** utilizado es con el objetivo de hacer nuestros tests *solitarios*, es decir, aislados de cualquier otro elemento externo o de evitar que realice una llamada muy costosa.

Los **mocks** son los elementos que nos permiten definir elementos ficticios en la pruebas. La particularidad es que su rol en las etapas de **setup** y **verify** es diferente, dado que por lo general los **mocks** se utilizan para **verificación de comportamiento**.

Sin entrar demasiado en detalle, dado que lo mismo implicaría extenderse demasiado del alcance del curso, los *stubs*, *spies*, etc cumplen esencialmente el mismo rol que los **mocks** en el sentido que son utilizados para testear de forma controlada y aislada. Si bien pueden utilizarse para verificación de comportamiento su estilo se adecúa más a la **verificación de estado**.

## Mockist TDD vs Classic TDD

La dicotomía entre *Test Solitarios* y *Test Sociables* impactó también en la filosofía alrededor de TDD.

Los practicantes de **Mockist TDD** utilizan mocks para aislar sus pruebas de cualquier comportamiento externo.

Los practicantes de **Classic TDD** utilizan mocks para aislar sus pruebas únicamente si la misma requiere de algún tipo de interacción costosa.

## TL;DR

- Por lo general las pruebas unitarias se estructuran en fases bien definidas, conocidas como **Setup**, **Exercise**, **Verify** y **Teardown**. En las que, se inicializa lo necesario para ejecutar la prueba, se llama al método que se está testeando, se

hace algún tipo de verificación para saber si el test pasó o no, y se limpia el contexto de forma de que el siguiente test ejecute en las mismas condiciones.
- Los tests se pueden aislar, utilizando **mocks**, totalmente o parcialmente. Correspondiendo los estilos a Tests *Solitarios* o *Sociables* respectivamente.
- Los **mocks** pertenecen a la familia de **doubles** y nos permiten armar objetos ficticios, basándose en **verificación de comportamiento**.
- Los tests más enfocados en **verificar comportamiento** pueden quedar acoplados a la implementación.
- Los tests más enfocados en **verificar estado** pueden quedar más desacoplados de la implementación, pero en algunos casos requerir que se agreguen métodos a los objetos que permitan verificar estado.

## Referencias


- https://martinfowler.com/bliki/UnitTest.html
- https://martinfowler.com/articles/mocksArentStubs.html
- https://blog.cleancoder.com/uncle-bob/2014/05/10/WhenToMock.html