# Test unitarios con visual studio 2017

## Definiciones

### Prueba unitaria

En programación, una prueba unitaria es una forma de probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

La idea es escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto.

### Desarrollo guiado por pruebas

Desarrollo guiado por pruebas de software, o Test-driven development (TDD) es una práctica de programación que involucra otras dos prácticas: Escribir las pruebas primero (Test First Development) y Refactorización (Refactoring). Para escribir las pruebas generalmente se utilizan las pruebas unitarias (unit test en inglés). En primer lugar, se escribe una prueba y se verifica que las pruebas fallan. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido.

### Mock (simular)

En la programación orientada a objetos se llaman objetos simulados (pseudoobjetos, mock object, objetos de pega) a los objetos que imitan el comportamiento de objetos reales de una forma controlada. Se usan para probar a otros objetos en pruebas unitarias que esperan mensajes de una clase en particular para sus métodos, al igual que los diseñadores de autos usan un crash dummy cuando simulan un accidente.

En los test de unidad, los objetos simulados se usan para simular el comportamiento de objetos complejos cuando es imposible o impracticable usar al objeto real en la prueba. De paso resuelve el problema del caso de objetos interdependientes, que para probar el primero debe ser usado un objeto no probado aún, lo que invalida la prueba: los objetos simulados son muy simples de construir y devuelven un resultado determinado y de implementación directa, independientemente de los complejos procesos o interacciones que el objeto real pueda tener.

Los objetos simulados se usan en lugar de objetos reales que tengan algunas de estas características:

- Devuelven resultados no determinísticos (por ejemplo la hora o la temperatura)
- Su estado es difícil de crear o reproducir (por ejemplo errores de conexión)
- Es lento (por ejemplo el resultado de un cálculo intensivo o una búsqueda en una BBDD)
- El objeto todavía no existe o su comportamiento puede cambiar.
- Debería incluir atributos o métodos exclusivamente para el testeo.

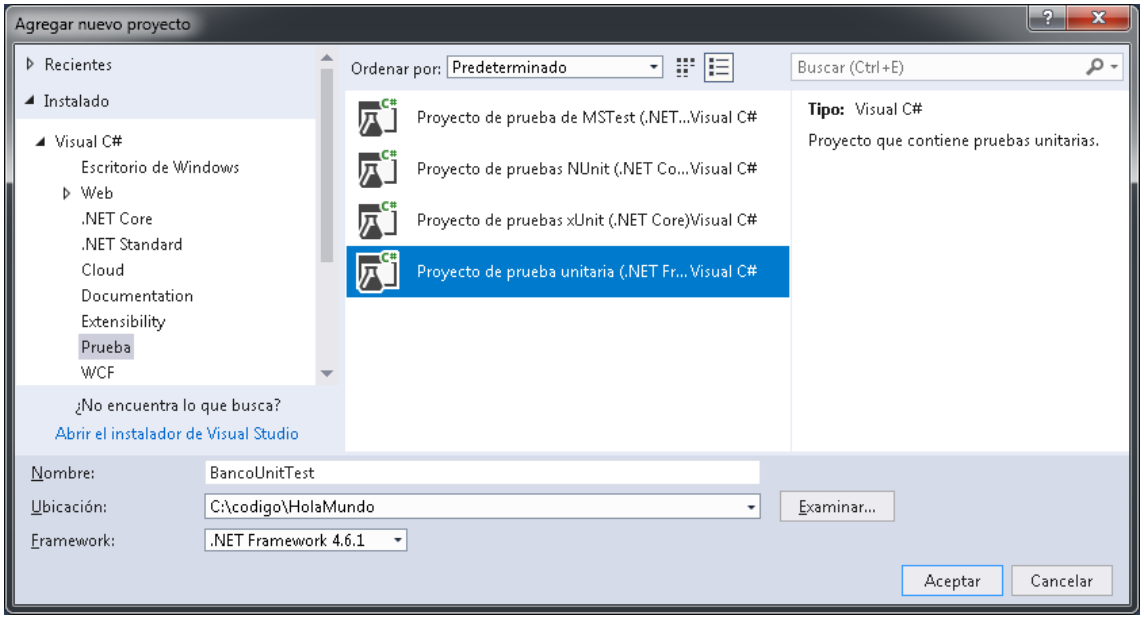
**Los objetos simulados para imitar al objeto real deben imitar su misma interfaz, aunque últimamente** .

## Ejemplo

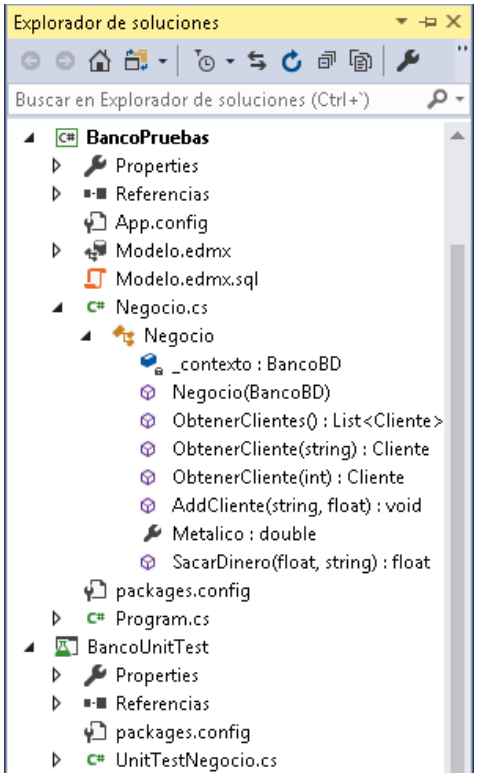
### Creación de un proyecto de pruebas

Crear un negocio tratando de usar interfaces en los elementos susceptibles de ser probados (orígenes de datos, componentes externos).

Creamos el proyecto de pruebas. Para ello, seleccionamos la solución y le damos a crear nuevo proyecto y seleccionamos proyecto de prueba.




Quedando la solución de la siguiente manera




## Estructura de una clase de pruebas

Los test unitarios se definen mediante etiquetas, una etiqueta es un atributo que le damos a nuestro código y se asigna median corchetes antes de una clase o método.

La etiqueta **TestClass** define una clase de pruebas que contendrá varias funciones de prueba etiquetadas con **TestMethod**.

Por ejemplo:

```
    [TestClass]
    public class UnitTestMiPrograma
    {
…
        [TestMethod]
        public void TestMetalico()
        {
…
```

Otras etiquetas que aparecen en una clase de prueba son:

-ClassInitialize: Se ejecuta solo una vez antes de ejecutarse los métodos de test

```
        [ClassInitialize()]
        public static void ClassInit(TestContext context)
```

-ClassCleanup: Se ejecuta solo una vez tras la ejecución de los métodos de test

```
        [ClassCleanup()]
        public static void ClassCleanup()
```

-**TestInitialize**: Se ejecuta antes de cada test, sirve para definir o preparar los datos de prueba previos al test. Cada test debe tener datos limpios no influidos por los tests previos.

```
        [TestInitialize()]
        public void Initialize()
```

-TestCleanup: Se ejecuta después de cada test

```
        [TestCleanup()]
        public void Cleanup()
```

## Estructura de un método de prueba (TestMethod)

Un test consta generalmente de tres partes:

1 – Definición del entorno de pruebas

2 – Ejecución del código de negocio

3 – Validación de los resultados

En nuestro ejemplo, definimos los datos de prueba el método Initialize porque como hemos visto se llama antes de cada test, de esta forma sabemos de qué datos dispondremos y nos aseguramos que no han sido modificados. No te preocupes si no entiendes parte del código.

```
        [TestInitialize]
        public void Initialize()
        {
            //falseo el contesto
            _mockBancoBD = new Mock<BancoBD>();

            //falseo las tablas
            //1-datos como consulta
            var datosClientes = new List<Cliente>
            {
                new Cliente { ClienteId = 8001, Numero = 9001, Nombre =
"TestUser9001", Caja = 100 },
                new Cliente { ClienteId = 8002, Numero = 9002, Nombre =
"TestUser9002", Caja = 200 },
                new Cliente { ClienteId = 8003, Numero = 9003, Nombre =
"TestUser9003", Caja = 300 },
                new Cliente { ClienteId = 8004, Numero = 9004, Nombre =
"TestUser9004", Caja = 400 },
                new Cliente { ClienteId = 8005, Numero = 9005, Nombre =
"TestUser9005", Caja = 500 },
                new Cliente { ClienteId = 8006, Numero = 9006, Nombre =
"TestUser9006", Caja = 600 }
            }.AsQueryable();
            //2-falseo la tabla
            var mockTablaClientes = new Mock<DbSet<Cliente>>();
            mockTablaClientes.As<IQueryable<Cliente>>().Setup(x =>
x.Provider).Returns(datosClientes.Provider);
            mockTablaClientes.As<IQueryable<Cliente>>().Setup(x =>
x.Expression).Returns(datosClientes.Expression);
            mockTablaClientes.As<IQueryable<Cliente>>().Setup(x =>
x.ElementType).Returns(datosClientes.ElementType);
            mockTablaClientes.As<IQueryable<Cliente>>().Setup(x =>
x.GetEnumerator()).Returns(datosClientes.GetEnumerator());
            //3-cuando se solicite la tabla cliente se devolvera un objeto
falseado
            _mockBancoBD.Setup(x =>
x.Clientes).Returns(mockTablaClientes.Object);
```

```
        //falseamos la base de datos.
        _negocio = new Negocio(_mockBancoBD.Object);
        Debug.WriteLine("TestInitialize");
    }
```

Seguido ejecutamos nuestro test, la primera parte ejecuta la lógica de la clase banco que como acabamos de ver hemos inicializado con varios clientes de prueba. En este caso buscamos el cliente con el número 9004.

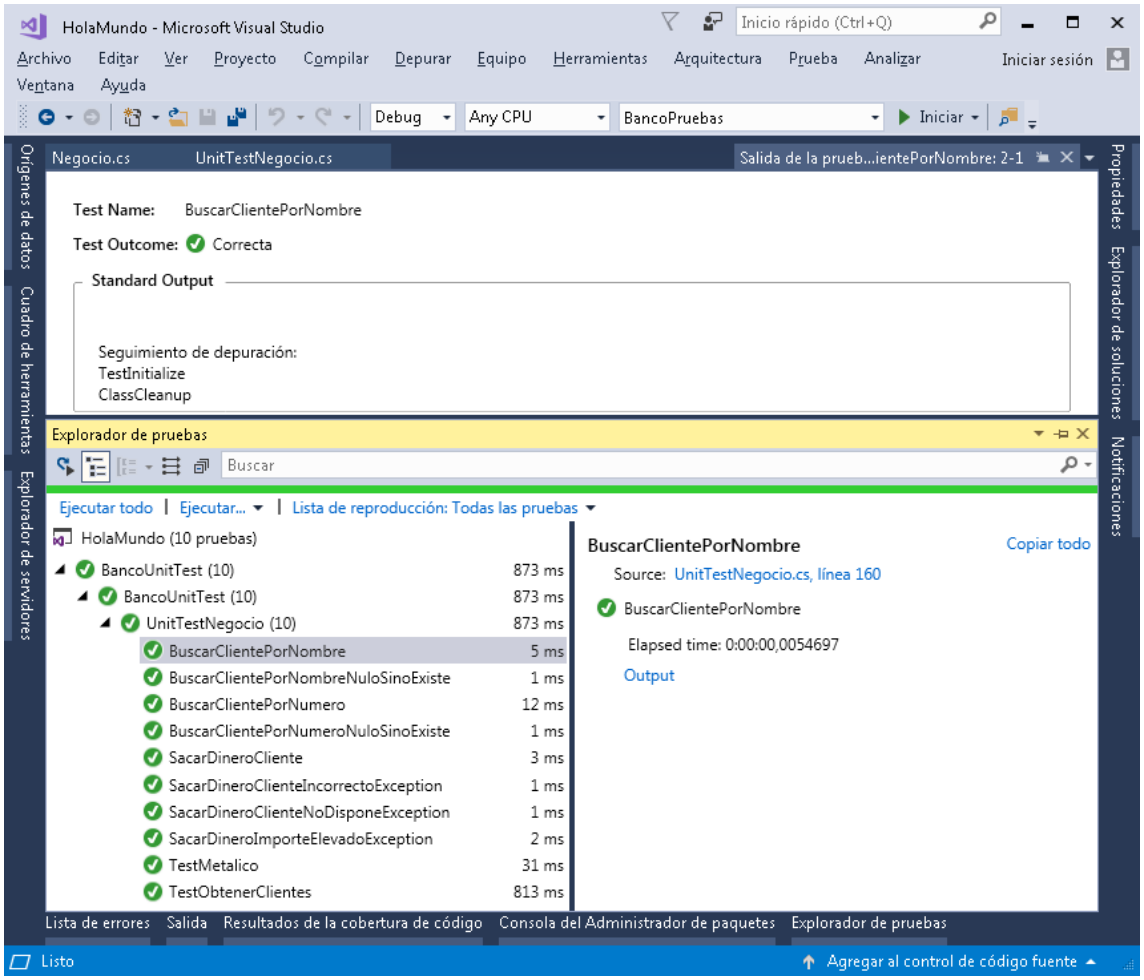
Seguido revisamos si se ha encontrado el cliente y si el cliente encontrado es el que indicamos al establecer el entorno de pruebas.

```
    [TestMethod]
    public void BuscarClientePorNumero()
    {
        //Invocamos a la logica negocio del programa
        Cliente unCliente = _negocio.ObtenerCliente(9004);

        //revisamos los resultados
        //comprobamos que el cliente no es nulo, ya que sabemos que existe
        Assert.IsNotNull(unCliente);

        //revisamos que los datos del cliente son los correctos
        Assert.AreEqual(9004, unCliente.Numero);
        Assert.AreEqual("TestUser9004", unCliente.Nombre);
        Assert.AreEqual(400, unCliente.Caja);
    }
```

El objeto con el que contamos para evaluar los resultados es Assert, y su funcionamiento principal consiste en comparar si el valor esperado coincide con el devuelto por el negocio. En el caso de no cumplirse esta igualdad el test falla.

Otro método interesante de Assert es el Fail, el cual nos permite romper el test manualmente en cualquier momento indicando mediante un mensaje el motivo.

El resultado de las pruebas se muestra en una ventana a parte indicando el resultado y el problema en caso error.




## Mocking

Normalmente suele ser necesario falsear al acceso a datos o algún componente externo, ya que no tiene sentido que realicemos pruebas sobre los datos de producción. En otros casos, podremos probar código antes de que este implementado ya que conocemos la interfaz que debemos cumplir.

El proceso por el que falseamos un objeto se llama “Mock” o “Mocking”, y con el conseguimos crear un objeto que cumpla con una interfaz o las características de la clase y que además se ajuste a nuestras necesidades. Si te fijas en el ejemplo tenemos dos propiedades, una para el objeto de negocio que queremos probar “_negocio” y otra para simular el acceso a datos” _mockBancoBD”.

El objeto de negocio incluye la lógica de nuestra aplicación, pero tiene el problema de estar asociado a una base de datos, si hacemos pruebas con la base de datos de producción seguramente rompamos algún dato. Por eso, la clase negocio cuenta con un constructor que recibe como parámetro el objeto o contexto de datos. De esta forma cuando estemos en pruebas podremos emplear una base de datos moqueada y cuando estemos en producción la base de datos de producción.

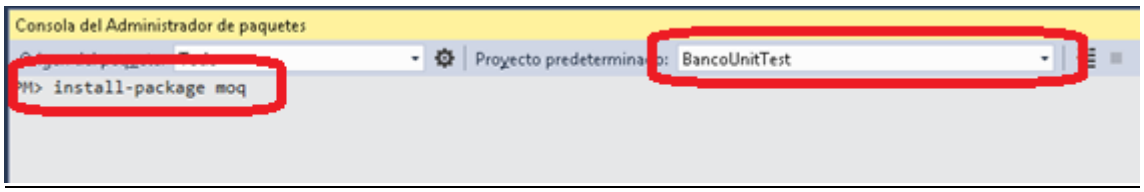
```
[TestClass]
public class UnitTestNegocio
```

```
{
    private Negocio _negocio;
    private Mock<BancoBD> _mockBancoBD;

   …
    public void Initialize()
    {
          _mockBancoBD = new Mock<BancoBD>();
   …
          _negocio = new Negocio(_mockBancoBD.Object);
```

La herramienta para moquear objetos no es propia del entorno de desarrollo y debemos descargarla de nuget, para ello teclearemos lo siguiente en la consola de Administración de paquetes.

**Install-Package Moq**

**(OJO, instalar el mock en el proyecto de pruebas, en la consola de nuget hay un desplegable)**




**Propiedades y métodos útiles de framework Moq, SON TODAS IMPORTANTES**

- objMoq.Object -> Objeto moqueado, en este caso un objeto de tipo IDatos
- objMoq.Verify -> Permite evaluar la forma en la que el código que se prueba accede a el objeto mockeado. Es decir, controla el número de veces que se accede a una propiedad o método mockeado durante el test.
- objMoq.Setup(linq).Returns(valor) - > Permite asignar a métodos y propiedades del objeto el valor a devolver cuando se invoquen.
- objMoq.As<tipoInterfaz>() -> Añade soporte para la interfaz indicada al objeto moqueado.
- Objeto it -> permite definir parámetros o valores esperados de un determinado tipo. Normalmente se emplea como parámetro de la función “Verify”.

```
It.IsAny<int>() -> es un parámetro de función, cualquier valor entero en este caso
```

En nuestro caso definiríamos el comportamiento del objeto BancoBD, que hereda de DbContext, de la siguiente manera, fíjate en el uso del método Setup:

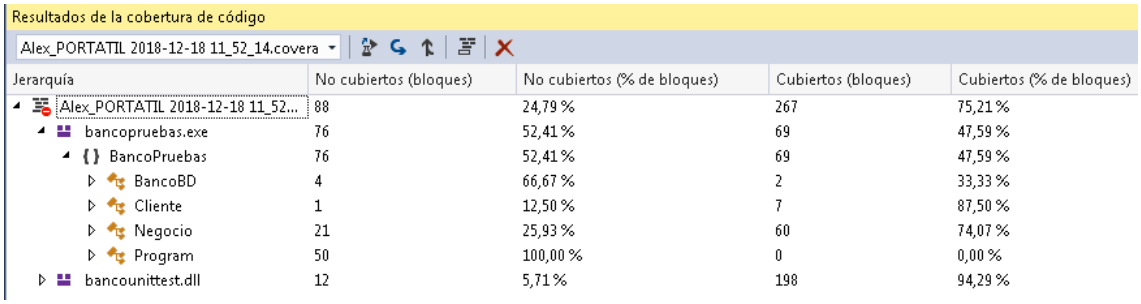
```
[TestInitialize]
public void Initialize()
{
    //falseo el contesto
    _mockBancoBD = new Mock<BancoBD>();

    //falseo las tablas
    //1-datos como consulta
    var datosClientes = new List<Cliente>
```

```
            {
                new Cliente { ClienteId = 8001, Numero = 9001, Nombre =
"TestUser9001", Caja = 100 },
                new Cliente { ClienteId = 8002, Numero = 9002, Nombre =
"TestUser9002", Caja = 200 },
                new Cliente { ClienteId = 8003, Numero = 9003, Nombre =
"TestUser9003", Caja = 300 },
                new Cliente { ClienteId = 8004, Numero = 9004, Nombre =
"TestUser9004", Caja = 400 },
                new Cliente { ClienteId = 8005, Numero = 9005, Nombre =
"TestUser9005", Caja = 500 },
                new Cliente { ClienteId = 8006, Numero = 9006, Nombre =
"TestUser9006", Caja = 600 }
            }.AsQueryable();
            //2-falseo la tabla
            var mockTablaClientes = new Mock<DbSet<Cliente>>();
            mockTablaClientes.As<IQueryable<Cliente>>().Setup(x =>
x.Provider).Returns(datosClientes.Provider);
            mockTablaClientes.As<IQueryable<Cliente>>().Setup(x =>
x.Expression).Returns(datosClientes.Expression);
            mockTablaClientes.As<IQueryable<Cliente>>().Setup(x =>
x.ElementType).Returns(datosClientes.ElementType);
            mockTablaClientes.As<IQueryable<Cliente>>().Setup(x =>
x.GetEnumerator()).Returns(datosClientes.GetEnumerator());
            //3-cuando se solicite la tabla cliente se devolvera un objeto
falseado
            _mockBancoBD.Setup(x =>
x.Clientes).Returns(mockTablaClientes.Object);

            //falseamos la base de datos.
            _negocio = new Negocio(_mockBancoBD.Object);
            Debug.WriteLine("TestInitialize");
        }
```

Aunque pueda parecer complejo el código anterior en realizad es bastante intuitivo, al final lo que queremos emular es una tabla de la base de datos para lo cual en primer lugar emulamos el contexto, en segundo lugar emulamos la tabla o DbSet y por último asociamos la tabla emulada al contexto emulado.

## Cobertura de código

Otro aspecto interesante asociado a las pruebas unitarias es la cobertura de código o “code coverage”. Mediante esta funcionalidad podemos comprobar que porcentaje de nuestra aplicación está cubierta por pruebas y que partes no lo están, es decir, podemos conocer fácilmente que bloques de la lógica no disponen de pruebas. El acceso a esta funcionalidad se encuentra en “Prueba -> Analizador cobertura de código”.




| Jerarquía | No cubiertos (bloques) | No cubiertos (% de bloques) | Cubiertos (bloques) | Cubiertos (% de bloques) |
|---|---|---|---|---|
| Alex_PORTATIL 2018-12-18 11_52... | 88 | 24,79 % | 267 | 75,21 % |
| bancopruebas.exe | 76 | 52,41 % | 69 | 47,59 % |
| BancoPruebas | 76 | 52,41 % | 69 | 47,59 % |
| BancoBD | 4 | 66,67 % | 2 | 33,33 % |
| Cliente | 1 | 12,50 % | 7 | 87,50 % |
| Negocio | 21 | 25,93 % | 60 | 74,07 % |
| Program | 50 | 100,00 % | 0 | 0,00 % |
| bancounittest.dll | 12 | 5,71 % | 198 | 94,29 % |

## Enlaces

http://es.wikipedia.org/wiki/Prueba_unitaria

http://es.wikipedia.org/wiki/Objeto_simulado

http://si.ua.es/es/documentacion/c-sharp/documentos/pruebas/07pruebasunitarias.pdf

http://www.nuget.org/packages/Moq/

http://willemdejong.net/blog/using-moq-in-a-visual-studio-test-project

Mocking de contexto

https://docs.microsoft.com/es-es/ef/ef6/fundamentals/testing/mocking