Clase 3 @September 8, 2022

- Recordar que completen el usuario de github y se unan a teams
- Preguntar como van con el practico 1
- Hacer una recorrida de repaso por el ejemplo integrador de la clase pasada
- Mostrar inyección de dependencias en el ejemplo anterior
- Test
- Checkeo de avance de obligatorio https://docs.google.com/spreadsheets/d/1dxk-iVQzVOh9TYzhLPPplQsv_m7n9Xt4UcChLI465Yg/edit?usp=sharing

Inyección de dependencias

¿Qué es una dependencia?

En software, cuando hablamos de que dos piezas, componentes, librerías, módulos, clases, funciones (o lo que se nos pueda ocurrir relacionado al área), son dependientes entre sí, nos estamos refiriendo a que uno requiere del otro para funcionar.

A nivel de clases, significa que una cierta **'Clase A'** tiene algún tipo de relación con una **'Clase B'**, delegándole el flujo de ejecución a la misma en cierta lógica.

Ej: **UserLogic** *depende de* **UserRepository**

```
public class UserLogic : IUserLogic
{
    public IRepositoryUser users;

    public UserLogic()
    {
        users = new UserRepository();
    }
}
```

El problema reside en que ambas piezas de código tiene la responsabilidad de la instanciación de sus dependencias.

Nuestras capas no deberían estar tan fuertemente acopladas y no deberían ser tan dependientes entre sí. Si bien el acoplamiento es a nivel de interfaz (tenemos IUserLogic y IRepository), **la tarea de creación/instanciación/"hacer el new" de los objetos debería ser asignada a alguien más. Nuestras capas no deberían preocuparse sobre la creación de sus dependencias.**

¿Por qué? ¿Qué tiene esto de malo?

1. Si queremos **reemplazar** por ejemplo nuestro BreedsBusinessLogic **por una implementación diferente**, deberemos modificar nuestro controller. Si queremos reemplazar nuestro UserRepository por otro, tenemos que modificar nuestra clase UserLogic.
2. Si la UserLogic tiene sus propias dependencias, **debemos configurarías dentro del controller**. Para un proyecto grande con muchos controllers, el código de configuración empieza a esparcirse a lo largo de toda la solución.
3. **Es muy difícil de testear, ya que las dependencias 'están hardcodeadas'.** Nuestro controller siempre llama a la misma lógica de negocio, y nuestra lógica de negocio siempre llama al mismo repositorio para interactuar con la base de datos. En una prueba unitaria, se necesitaría realizar un mock/stub las clases dependientes, para evitar probar las dependencias. Por ejemplo: si queremos probar la lógica de UserLogic sin tener que depender de la lógica de la base de datos, podemos hacer un mock de UserRepository. Sin embargo, con nuestro diseño actual, al estar las dependencias 'hardcodeadas', esto no es posible.

Una forma de resolver esto es a partir de lo que se llama, **Inyección de Dependencias**. Vamos a inyectar la dependencia de la lógica de negocio en nuestro controller, y vamos a inyectar la dependencia del repositorio de datos en nuestra lógica de negocio.

Inyectar dependencias es entonces pasarle la referencia de un objeto a un cliente, al objeto dependiente (el que tiene la dependencia). Significa simplemente que la dependencia es encajada/empujada en la clase desde afuera. Esto significa que no debemos instanciar (hacer new), dependencias, dentro de la clase.

Esto lo haremos a partir de un parámetro en el constructor, o de un setter. Por ejemplo:

```
public class UserLogic : IUserLogic
{
    public IRepositoryUser users;

    public UserLogic(IRepositoryUser users)
    {
        this.users = users;
    }
}
```

Esto es fácil lograrlo usando interfaces o clases abstractas en C#. Siempre que una clase satisfaga la interfaz,voy a poder sustituirla e inyectarla.

Luego en la clase startup

- **REGISTRO EL REPOSITORIO Y SU LÓGICA**: `services.AddScoped<Interfaz, Servicio>()`, este lo que haces es registra un servicio de tipo `Servicio` y lo va a instanciar e inyectar cuando alguien necesite una interfaz de tipo `Interfaz`.
- **REGISTRO EL CONTEXTO**: El DbContext se registra de una manera especial, utilizando el siguiente método:

```
services.AddDbContext<DbInterfaz, DbContexto>(
  o => o.UseSqlServer(Configuration.GetConnectionString("REFERENCIA_AL_ARCHIVO_DE_CONFIGURACIÓN"))
);

services.AddScoped<IRepositoryUser,UserRepository>();
```

Explicacion del ejemplo:

```
// En este caso le indicamos que cuando vea la interfaz DbContext para inyectar que instancie e inyecte el con
texto: HomeworksContext
services.AddDbContext<DbContext, HomeworksContext>(
    // Por ultimo le indicamos que la BD va a ser MSSQL y que el
    // ConnectionString lo va a extraer de una archivo de configuración (appsettings.json) y que tome el valor
 de la key "HomeworksDB"
    o => o.UseSqlServer(Configuration.GetConnectionString("HomeworksDB"))
);
```

Ventajas de ID

Logramos resolver lo que antes habíamos descrito como desventajas o problemas.

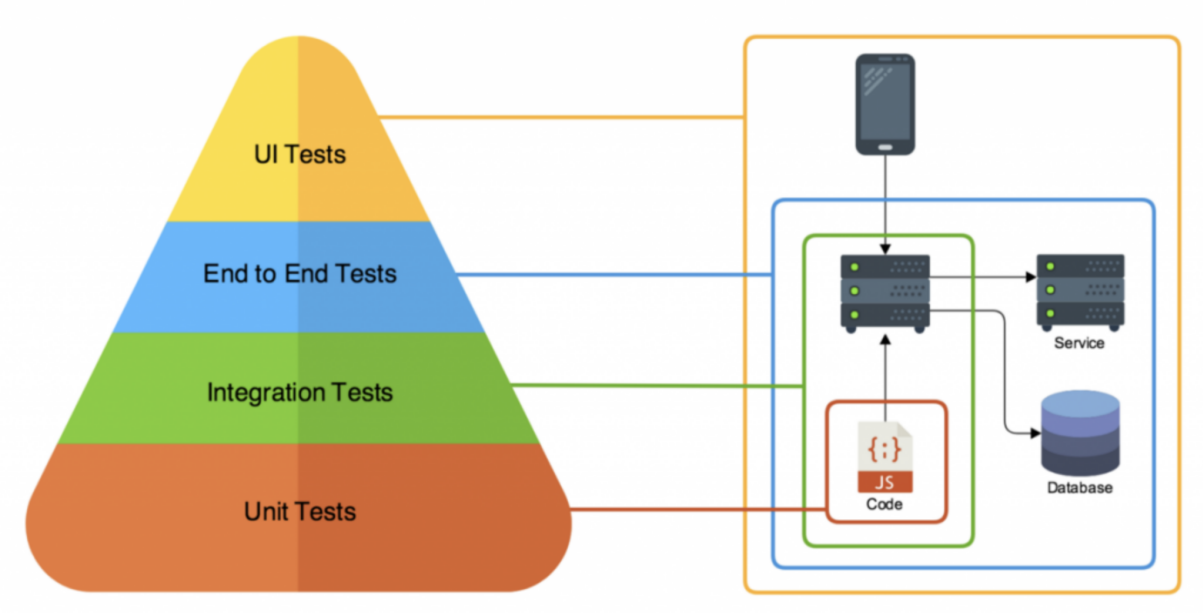
1. **Código más limpio**. El código es más fácil de leer y de usar.
2. Nuestro software termina siendo **más fácil de Testear.**
3. Es más **fácil de modificar.** Nuestros módulos son flexibles a usar otras implementaciones. Desacoplamos nuestras capas.
4. **Permite NO Violar SRP.** Permite que sea más fácil romper la funcionalidad coherente en cada interfaz. Ahora nuestra lógica de creación de objetos no va a estar relacionada a la lógica de cada módulo. Cada módulo solo usa sus dependencias, no se encarga de inicializarlas ni conocer cada una de forma particular.
5. **Permite NO Violar OCP.** Por todo lo anterior, nuestro código es abierto a la extensión y cerrado a la modificación. El acoplamiento entre módulos o clases es siempre a nivel de interfaz.

Testing

¿Qué son los Unit Test?

Es una manera de asegurarnos que nuestro código funciona como está previsto que funcione, evitando errores en el desarrollo inicial del software o a medida que este evoluciona.

Cómo dice el nombre, los unit test hacen testing de unidades de código (componentes y métodos), hay otros tipos de tests como lo pueden ser:



Las pruebas **unitarias** deberían de **probar** únicamente **código** que este al **alcance** del desarrollador y solo deben probar una cosa a la vez (vamos a tener muchas pruebas).

Para crear las pruebas unitarias en .Net Core podemos usar diferentes frameworks:

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

Vamos a estudiar **cómo podemos probar nuestro código evitando probar también sus dependencias**, asegurándonos que los errores se restringen únicamente a la sección de código que efectivamente queremos probar. Para ello, utilizaremos una herramienta que nos permitirá crear Mocks. La herramienta será Moq.

¿Qué son los Test doubles?

Son objetos que no son reales respecto a nuestro dominio, y que se usan con finalidades de testing. Existen algunos 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.

Nosotros vamos a ver los Mocks pero ustedes podrían llegar a usar los Dummies y Stubs.

¿Qué son los Mocks?

Los mocks son unos de los varios "test doubles" que existen para probar nuestros sistemas. Los más conocidos son los Mocks y los Stubs, siendo la principal diferencia en ellos, el foco de lo que se está testeando.

Antes de hacer énfasis en tal diferencia, es importante aclarar que nos referiremos 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.

Para comenzar a utilizar Moq, comenzaremos probando nuestro paquete de controllers de la WebApi. Para esto:

Primero vamos a hacer unos pequeños cambios en nuestra Api para que retorne un OrderBasicInfoModel cuando se cree una Orden.

```
namespace WebAPI.Models.Out;

public class OrderBasicInfoModel
{
    public int Id { get; set; }

    public DateTime DeliveryDateTime { get; set; }
}
```

1- Crearemos un nuevo proyecto de MSTests (WebApi.Tests)

```
dotnet new mstest -n WebApi.Test
```

Agregamos el proyecto a la solución:

```
dotnet sln add Tacos.WebApi
```

2- Instalamos a este paquete Moq.

```
cd WebApi.Test
dotnet add package Moq
```

3- Luego al proyecto de tests le agregaremos las referencias a WebApi, Domain y BusinessLogic.Interface

```
dotnet add reference ../WebApi
dotnet add reference ../Domain
dotnet add reference ../BusinessLogic
```

Una vez que estos pasos estén prontos, podemos comenzar a realizar nuestro primer test.

4- Creamos entonces la clase OrdersControllerTest, y en ella escribimos el primer TestMethod.

5- **Probando el POST**

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Domain;
using IBusinessLogic;
using WebAPI;
using Moq;

namespace WebApi.Test;

[TestClass]
public class OrdersControllerTest
{
    [TestMethod]
    public void CreateValidOrderTest()
    {
        Order order = new Order()
        {
            Name = "Papas fritas",
            Address = "Juan Gomez 3878",
            PutchaseNumber = 1,
            Price = 100,
        };//1

      var mock = new Mock<IOrderService>(MockBehavior.Strict); // 2
      var controller = new OrderController(mock.Object);//3
      var result = controller.Add(order); // 4
      var createdResult = result as CreatedAtRouteResult; // 5
      var model = createdResult.Value as OrderBasicInfoModel; // 6

    }
}
```

Veremos que pasa en el test:

1. Crea un objeto de Order que usaremos para el mock. Este retorna data que no nos importa, es solo para testing
2. Creamos el mock. La notacion es `new Mock<A>` siendo A la interfaz que queremos mockear. El parámetro (`MockBehhavior.Strict`) es una parámetro de configuracion del mock. `.Strict` hace que se tire una excepción cuando se llama un método que no fue mockeado, mientras que `.Loose` retorna un valor por defecto si se llama un método no mockeado.
3. Se crea el controlador (Order `Controller`) con el objeto mockeado.
4. Se ejecuta el metodo Post del controlador
5. Debido a que la clase retorna un `CreatedAtRouteResult` , como se puede ver en la implementación, casteamos el resultado a esto
6. Mediante `.Value` obtenemos el resultado de la request

Sin embargo, nos falta definir el comportamiento que debe tener el mock del nuestro `IOrderService` . Esto es lo que llamamos **expectativas** y lo que vamos asegurarnos que se cumpla al final de la prueba. Recordemos, los mocks simulan el comportamiento de nuestros objetos, siendo ese comportamiento lo que vamos a especificar a partir de expectativas. Para ello, usamos el método **Setup**.

¿Cómo saber qué expectativas asignar?

Esto va en función del método de prueba. Las expectativas se corresponden al caso de uso particular que estamos probando dentro de nuestro método de prueba. Si esperamos probar el `Add()` de

nuestro `OrderController`, y queremos mockear la clase `OrderService`, entonces las expectativas se corresponden a las llamadas que hace `OrderController` sobre `OrderService`.

Veamos el método a probar, el `Add` de un order:

```
[HttpPost]
public IActionResult Add([FromBody] Order order)
{
    Order createdOrder = _orderService.Create(order);
    return CreatedAtRoute("AddOrder", new OrderBasicInfoModel()
        {
                Id = createdOrder.Id,
                DeliveryDateTime = createdOrder.DeliveryDateTima
        });
}
```

La línea que queremos mockear es la de:

```
     _orderService.Create(order);
```

Entonces:

1. Primero vamos a decirle que esperamos que sobre nuestro Mock que se llame a la función Create().
2. Luego vamos a indicarle que esperamos que tal función retorne un order con id que definimos en otro lado.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Domain;
using IBusinessLogic;
using Microsoft.AspNetCore.Mvc;
using Moq;
using WebAPI.Controllers;
using Models.Out;

namespace WebApi.Test;

[TestClass]
public class OrdersControllerTest
{
    [TestMethod]
    public void CreateValidOrderTest()
    {
        Order order = new Order()
        {
            Id = 1,
            Name = "Papas fritas",
            Address = "Juan Gomez 3878",
            PutchaseNumber = 1,
            Price = 100,
            DeliveryDateTima = DateTime.Now.AddHours(1)
        };

        OrderBasicInfoModel createdOrder = new OrderBasicInfoModel()
        {
            Id = order.Id,
            DeliveryDateTime = order.DeliveryDateTima
        };

        var mock = new Mock<IOrderService>(MockBehavior.Strict);
```

```
        mock.Setup(o => o.Create(It.IsAny<Order>())).Returns(order); //1
      var controller = new OrderController(mock.Object);
      var result = controller.Add(order);
      var createdResult = result as CreatedAtRouteResult;
        Console.WriteLine(createdResult.Value);
      var model = createdResult.Value as OrderBasicInfoModel;

        mock.VerifyAll();//2
        Assert.IsTrue(createdOrder.Id == model.Id && createdOrder.DeliveryDateTime == model.DeliveryDateTime);//3
    }
}
```

Veamos que le agregamos al metodo de test:

1. Seteamos el mock. Cuando decimos setear, queremos decir que le definimos el comportamiento que queremos en el test de un método de un mock. El método setup recibe una función inline de LINQ, la cual recibe el objeto a mockear. Es decir, en este caso `o` es un objeto de tipo `IOrderService`. Aquí estamos definiendo que para el método `Create`, cuando reciba cualquier parámetro de tipo `Order` (`It.IsAny<Order>()`). En este caso, se retorna el `order`. Así como devolvimos el estudiante porque estamos simulando un caso sin errores también podemos devolver excepciones con *Throw* para simular que sucedió algo inesperado en la capa que se esta haciendo uso.
2. También debemos verificar que se hicieron las llamadas pertinentes. Para esto usamos el método `VerifyAll` del mock. Este revisa que fueron llamadas todas las funciones que mockeamos.
3. Verificamos que los datos obtenidos sean correctos. Para esto hacemos asserts (aquí estamos probando estado) para ver que los objetos usados son consistentes de acuerdo al resultado esperado. Para que este Assert funcione hay que hacer un *override* del método *Equals* en la clase OrderBasicInfoModel para que se ejecute. Otra opción seria verificar estado por estado sin redefinir el método *Equals*.

Corremos los tests utilizando `dotnet test` y veremos que pasa 😎

Mockeando excepciones

Como dijimos antes tambien podemos mockear excepciones. Ahora veamos como probar cuando nuestro `Add()` del Controller nos devuelve una **BadRequest**.

Particularmente, en el caso que hemos visto antes nuestro Controller retornaba `CreatedAtRoute` para dicha situación. Ahora, nos interesa probar el caso en el que nuestro Controller retorna una BadRequest. Particularmente, esto se da cuando el método `Add()` recibe `null`. Para probar este caso entonces, seteamos dichas expectativas y probemos.

```
[TestMethod]
    public void CreateInvalidOrderBadRequestTest()
    {
        var mock = new Mock<IOrderService>(MockBehavior.Strict);
        mock.Setup(m => m.Create(null)).Throws(new ArgumentException());
        var controller = new OrderController(mock.Object);

        var result = controller.Add(null);

        mock.VerifyAll();
        Assert.IsInstanceOfType(result, typeof(BadRequestResult));
    }
```

```
[HttpPost]
    public IActionResult Add([FromBody] Order order)
    {
        try
        {
            Order createdOrder = _orderService.Create(order);
            return CreatedAtRoute("AddOrder", new OrderBasicInfoModel()
            {
                Id = createdOrder.Id,
                DeliveryDateTime = createdOrder.DeliveryDateTima
            });
        }
        catch (ArgumentException ex)
        {
            return BadRequest();
        }
    }
```

Lo que hicimos fue indicar que cuando se invoque `Add` con el párametro en `null`, se lance `ArgumentException`. En consecuencia, cuando nuestro controller llame a este mock, se lanzara `ArgumentException` causando que nuestro controller la capture y retorne `BadRequesResult`.

Finalmente entonces, verificamos que las expectativas se hayan cumplido (con el `VerifyAll()`), y luego que el resultado obtenido sea un `BadRequestResult`, usando el método de `Assert` `IsInstanceOfType`.