

مقدمه: از آنجایی که در این سایت در مورد shim و stub صحبتی نشده دوست داشتم مطلبی در این باره بزارم. در آزمون واحد ما نیاز داریم که یک سری اشیا را moq کنیم تا بتوانیم آزمون واحد را به درستی انجام دهیم. ما در آزمون واحد نباید وابستگی به لایه‌های پایین یا بالا داشته باشیم پس باید مقلدی از object هایی که در سطوح مختلف قرار دارند بسازیم. شاید برای کسانی که با آزمون واحد کار کردند، به ویژه با فریم ورک تست Microsoft، یک سری مشکلاتی با mock کردن اشیا با استفاده از Mock داشته اند که حالا می‌خواهیم با معرفی فریم ورک‌های جدید، این مشکل را حل کنیم. برای اینکه شما آزمون واحد درستی داشته باشید باید کارهای زیر را انجام دهید:

- 1- هر objectی که نیاز به mock کردن دارد باید حتماً یا non-static باشد، یا اینترفیس داشته باشد.
- 2- شما احتیاج به یک فریم ورک تزریق وابستگی‌ها دارید که به عنوان بخشی از معماری نرم افزار یا الگوهای مناسب شیء‌گرایی مطرح است، تا عمل تزریق وابستگی‌ها را انجام دهید.
- 3- ساختارها باید برای تزریق وابستگی در اینترفیس‌های object های وابسته تغییر یابند.

Shims و Stubs:

نوع stub همانند فریم ورک mock می‌باشد که برای مقلد ساختن اینترفیس‌ها و کلاس‌های non-sealed virtual یا ویژگی‌ها، رویدادها و متدهای abstract استفاده می‌شود. نوع shim می‌تواند کارهایی که stub نمی‌تواند بکند انجام دهد یعنی برای مقلد ساختن کلاس‌های static یا متدهای non-overridable استفاده می‌شود. با مثال‌های زیر می‌توانید با کارایی بیشتر shim و stub آشنا شوید.

یک پروژه mvc ایجاد کنید و نام آن را FakingExample بگذارید. در این پروژه کلاسی با نام CartToShim به صورت زیر ایجاد کنید:

```
namespace FakingExample
{
    public class CartToShim
    {
        public int CartId { get; private set; }
        public int UserId { get; private set; }
        private List<CartItem> _cartItems = new List<CartItem>();
        public ReadOnlyCollection<CartItem> CartItems { get; private set; }
        public DateTime CreateDateTime { get; private set; }

        public CartToShim(int cartId, int userId)
        {
            CartId = cartId;
            UserId = userId;
            CreateDateTime = DateTime.Now;
            CartItems = new ReadOnlyCollection<CartItem>(_cartItems);
        }

        public void AddCartItem(int productId)
        {
            var cartItemId = DataAccessLayer.SaveCartItem(CartId, productId);
            _cartItems.Add(new CartItem(cartItemId, productId));
        }
    }
}
```

و همچنین کلاسی با نام CartItem به صورت زیر ایجاد کنید:

```
public class CartItem
{
    public int CartItemId { get; private set; }
    public int ProductId { get; private set; }

    public CartItem(int cartItemId, int productId)
    {
        CartItemId = cartItemId;
    }
}
```

```
        ProductId = productId;
    }
}
```

حالا یک پروژه unit test را با نام FakingExample.Tests اضافه کرده و نام کلاس آن را CartToShimTest بگذارید. یک reference از پروژه FakingExample تان به پروژه‌ی تستی که ساخته اید اضافه کنید. برای اینکه بتوانید کلاس‌های پروژه FakingExample را shim و یا stub کنید باید بر روی Reference پروژه تان راست کلیک کنید و گزینه Add Fakes Assembly را انتخاب کنید. وقتی این گزینه را می‌زنید، پوشه‌ای با نام Fakes در پروژه تست ایجاد شده و FakingExample.fakes در داخل آن قرار دارد همچنین در reference‌های پروژه تست، FakingExample.Fakes نیز ایجاد می‌شود. اگر بر روی فایل fakes که در reference ایجاد شده دوبار کلیک کنید می‌توانید کلاس‌های CartItem و CartToShim را مشاهده کنید که هم نوع stub شان است و هم نوع shim آنها که در تصویر زیر می‌توانید مشاهده کنید.



ShimDataAccessLayer را که مشاهده می‌کنید یک متد SaveCartItem دارد که به دیتابیس متصل شده و آیتم‌های کارت را ذخیره می‌کند.

حالا می‌توانیم تست خود را بنویسیم. در زیر یک نمونه از تست را مشاهده می‌کنید:

```
[TestMethod]
public void AddCartItem_GivenCartAndProduct_ThenProductShouldBeAddedToCart()
{
    //Create a context to scope and cleanup shims
    using (ShimsContext.Create())
    {
        int cartItemId = 42, cartId = 1, userId = 33, productId = 777;

        //Shim SaveCartItem rerouting it to a delegate which
        //always returns cartItemId
        Fakes.ShimDataAccessLayer.SaveCartItemInt32Int32 = (c, p) => cartItemId;

        var cart = new CartToShim(cartId, userId);
        cart.AddCartItem(productId);

        Assert.AreEqual(cartId, cart.CartItems.Count);
        var cartItem = cart.CartItems[0];
        Assert.AreEqual(cartItemId, cartItem.CartItemId);
        Assert.AreEqual(productId, cartItem.ProductId);
    }
}
```

همانطور که در بالا مشاهده می‌کنید کدهای تست ما در اسکوپ قرار گرفته اند که محدوده shim را تعیین می‌کند و پس از پایان یافتن تست، تغییرات shim به حالت قبل بر می‌گردد. متد SaveCartItemInt32Int32 را که مشاهده می‌کنید یک متد static است و نمی‌توانیم با mock و یا stub آن را تقلید کنیم. تغییر اسم متد SaveCartItem به SaveCartItemInt32Int32 به این معنی است که

متد ما دو ورودی از نوع Int32 دارد و به همین خاطر fake این متد به این صورت ایجاد شده است. مثلا اگر شما متد Save ای داشتید که یک ورودی Int و یک ورودی String داشت fake آن به صورت SaveInt32String ایجاد می‌شد. به این نکته توجه داشته باشید که حتما برای assert کردن باید assertها را در داخل اسکوپ ShimsContext قرار گرفته باشد در غیر این صورت assert شما درست کار نمی‌کند.

این یک مثال از shim بود؛ حالا می‌خواهم مثالی از یک stub را برای شما بزنم. یک اینترفیس با نام ICartSaver به صورت زیر ایجاد کنید:

```
public interface ICartSaver
{
    int SaveCartItem(int cartId, int productId);
}
```

برای shim کردن ما نیازی به اینترفیس نداشتیم اما برای استفاده از stub و یا Mock ما حتما به یک اینترفیس نیاز داریم تا بتوانیم object موردنظر را مقلد کنیم. حال باید یک کلاسی با نام CartSaver برای پیاده سازی اینترفیس خود بسازیم:

```
public class CartSaver : ICartSaver
{
    public int SaveCartItem(int cartId, int productId)
    {
        using (var conn = new SqlConnection("RandomSqlConnectionString"))
        {
            var cmd = new SqlCommand("InsCartItem", conn);
            cmd.CommandType = CommandType.StoredProcedure;
            cmd.Parameters.AddWithValue("@CartId", cartId);
            cmd.Parameters.AddWithValue("@ProductId", productId);

            conn.Open();
            return (int)cmd.ExecuteScalar();
        }
    }
}
```

حال تستی که با shim انجام دادیم را با استفاده از Stub انجام می‌دهیم:

```
[TestMethod]
public void AddCartItem_GivenCartAndProduct_ThenProductShouldBeAddedToCart()
{
    int cartItemId = 42, cartId = 1, userId = 33, productId = 777;

    //Stub ICartSaver and customize the behavior via a
    //delegate, to return cartItemId
    var cartSaver = new Fakes.StubICartSaver();
    cartSaver.SaveCartItemInt32Int32 = (c, p) => cartItemId;

    var cart = new CartToStub(cartId, userId, cartSaver);
    cart.AddCartItem(productId);

    Assert.AreEqual(cartId, cart.CartItems.Count);
    var cartItem = cart.CartItems[0];
    Assert.AreEqual(cartItemId, cartItem.CartItemId);
    Assert.AreEqual(productId, cartItem.ProductId);
}
```

امیدوارم که این مطلب برای شما مفید بوده باشد.

نظرات خوانندگان

نویسنده: سام ناصری
تاریخ: ۱۳۹۲/۰۱/۳۰ ۷:۲۳

من نویسنده خوبی نیستم و شاید بهتر باشه که در اینباره نظر ندهم. به هر روی چند نکته به نظر آمد باشد که مورد توجه شما واقع شود:

مقدمه را هنوز کامل نکردی. مقدمه خواننده را در جای پرتی از ماجرا رها میکند. اگر چهار خط آخر مقدمه را دوباره بخوانید متوجه میشوید که اگر تمام کاری که برای داشتن آزمون واحد باید انجام شود همین سه مورد باشد دیگر هرگز کسی به Fakes نیاز پیدا نمیکند، پس باید در ادامه میگفتید که این حالت مطلوب است ولی همیشه عملی نیست.

شروع و پایان مثالها مشخص نبود. مثالها بدون عنوان بودند. در شروع مثال باید مقدمه ای از مثال را مطرح میکردی و بعد مراحل مثال را توضیح میدادی.

در مثال اول باید بر بیشتر بر روی DataAccessLayer تاکید میکردی و صریح مشخص میکردی که عدم توانایی برنامه نویس در تغییر این کلاس و یا معماری سیستم گزینه IoC را کنار میگذارد و به این ترتیب مثال شما سودمندی Shim را بهتر نشان میداد.

در مثال دوم، کد CardToStub را ارائه نکردی، اگر طبق آنچه انتظار میرود، وابستگی که در CardToStub وجود دارد به اینترفیس ICartSaver است در این صورت اساساً مثال شما هیچ دلیل و انگیزشی برای Stub فراهم نمیکند. باید باز هم ذهنیت خواننده را شکل میدادی و او را متوجه این موضوع میکردی که در پیاده سازی دیگری که برنامه نویس قدرت اعمال تغییر در آن ندارد وابستگی سخت وجود دارد و به این دلیل Stub میتواند مفید واقع شود.

البته این رو به حساب اینکه من یک خواننده بسیار مبتدی هستم شاید مقاله برای دیگران بیشتر از من قابل فهم است. ولی در کل مقاله خوبی بود و برای من کاربردی بود.

نویسنده: آرش خوشبخت
تاریخ: ۱۳۹۲/۰۱/۳۰ ۱۱:۴۰

ممنونم از اینکه راهنماییم کردید تا مطالبم را درست تر بنویسم اما اون 3 موردی را که گفتم کارهایی است که برای آزمون واحد انجام می شود یعنی باید اینترفیس داشته باشیم برای مقلد ساختن و کلاس ها برای اینکه mock شوند باید non-static باشند و از این قبیل و در ادامه گفتم که اگر کلاسی ویژگی آن 3 مورد را نداشته باشد مثلاً نه اینترفیس داشته باشد و هم اینکه static باشد چیکار باید کرد.

در مورد stub گفتم که این نوع همانند فریم ورک mock می باشد و هیچ فرقی با آن ندارد یعنی شما مجبور نیستید از stub استفاده کنید می توانید به جای آن از mock استفاده کنید.

در مورد کد CardToStub همان کد آخری است فقط خطی که نام کلاس را نوشته بود نگذاشتم. در مورد اینکه برای مثال مقدمه ای باید می گذاشتم راستش من دقیقاً نمی دونم شاید هم حرف شما درست باشد ولی من فقط می خواستم طریقه نوشتن shim رو توضیح بدم یعنی در واقع حتی نیاز به ساخت پروژه و این حرفا هم نداشت. بازم متشکرم که ایرادات منو فرمودین سعی می کنم از این به بعد مطالبم رو بهتر بنویسم

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۱/۳۰ ۱۴:۷

mocking بهتره به معنای ایجاد اشیاء تقلیدی عنوان بشه تا مقلد سازی.

نویسنده: مرتضی
تاریخ: ۱۳۹۲/۰۹/۲۷ ۱:۲۱

سلام

(نوع stub همانند فریم ورک mock می باشد)

تعریفی که از stub تو راهنماش اومده با مطلبی که شما ذکر کردید متفاوت

Martin Fowler's article **Mocks aren't Stubs** compares and contrasts the underlying principles of Stubs and Mocks. As outlined in Martin Fowler's article, a **stub provides static canned state which results in state verification** of the system under test, whereas a **mock provides a behavior verification** of the results for the system under test and their indirect outputs as related to any other component dependencies while under test

نویسنده: آرش خوشبخت
تاریخ: ۱۳۹۲/۰۹/۲۷ ۸:۵۳

با سلام ممنون که این مطلب رو گذاشتین اما منظور من این نیست که هیچ فرقی با هم ندارند منظورم از اینه که همانطور هم بالا توضیح دادم برای مقلد سازی اینترفیس ها و abstract ها و ... به کار میره همانطور که mock برای اینطور کلاس ها و متدها استفاده می شود

در باب ضرورت نوشتن کدهای تست پذیر، توسعه کلاس‌های کوچک تک مسئولیتی و اهمیت تزریق وابستگی‌ها بارها و بارها بحث شده و مطلب نوشته شده است. این روزها کم پیش می‌آید که نرم افزاری توسعه داده شود و از پایگاه داده به جهت ذخیره و بازیابی داده‌ها استفاده نکند. با گسترش و رواج ORM ها، نوشتن کدهای دسترسی به داده‌ها سهولت یافته است و استفاده از ORM در لایه‌ی سرویس که نگهدارنده‌ی منطق تجاری برنامه است، امری اجتناب ناپذیر می‌باشد.

در این مطلب نحوه‌ی نوشتن آزمون واحد برای کلاس سرویسی که وابسته به DbContext می‌باشد، به همراه محدودیت‌ها شرح داده می‌شود.

ابتدا یک روش که در آن مستقیماً از DbContext در سرویس استفاده شده را بررسی می‌کنیم. در مثال زیر کلاس ProductService وظیفه‌ی برگرداندن لیست کالاها را به ترتیب نام دارد. در آن DbContext مستقیماً و هله سازی شده و از آن جهت انجام تراکنش‌های دیتابیس کمک گرفته شده است:

```
public class ProductService
{
    public IEnumerable<Product> GetOrderedProducts()
    {
        using (var ctx = new Entites())
        {
            return ctx.Products.OrderBy(x => x.Name).ToList();
        }
    }
}
```

برای این کلاس نمی‌توان Unit Test نوشت چرا که یک وابستگی به شی DbContext دارد و این وابستگی مستقیماً درون متد GetOrderedProducts نمونه سازی شده است. در مطالب پیشین شرح داده شد که برای تست پذیر کردن کدها باید این وابستگی‌ها را از بیرون، در اختیار کلاس مورد نظر قرار داد.

برای نوشتن تست برای کلاس ProductService حداقل دو روش در اختیار است:

- نوشتن Integration Test :

یعنی کلاس جاری را به همین شکل نگاه داریم و در تست، مستقیماً به یک پایگاه داده که به منظور تست فراهم شده وصل شویم. برای سهولت مدیریت پایگاه داده می‌توان عمل درج را در یک Transaction قرار داد و پس از پایان یافتن تست Transaction را RollBack کرد. این روش مورد بحث مطلب جاری نمی‌باشد، لطفاً برای آشنایی این دو مطلب را مطالعه بفرمایید:

[Using Entity Framework in integration tests](#)

[How We Do Database Integration Tests With Entity Framework Migrations](#)

- بهره جستن از تزریق وابستگی و نوشتن Unit Test که وابستگی به دیتابیس ندارد

یکی از قانون‌های یک آزمون واحد این است که وابستگی به منابع خارجی مثل پایگاه داده نداشته باشد. [این مطلب](#) نحوه‌ی صحیح پیاده سازی الگوی Unit of Work را شرح داده است. بعد از پیاده سازی Unit Of Work، کلاس DbContext به شرح زیر می‌شود. همانطور که مشاهده می‌کنید، اکنون DbContext یک Interface را پیاده سازی کرده است.

```
public interface IUnitOfWork
{
    IDbSet<TEntity> Set<TEntity>() where TEntity : class;
    int SaveAllChanges();
}

public class Entites : DbContext, IUnitOfWork
```

```

{
    public virtual DbSet<Product> Products { get; set; } // This is virtual because Moq needs to
    override the behaviour

    public new virtual IDbSet<TEntity> Set<TEntity>() where TEntity : class // This is virtual
    because Moq needs to override the behaviour
    {
        return base.Set<TEntity>();
    }

    public int SaveAllChanges()
    {
        return base.SaveChanges();
    }
}

```

در این حالت می‌توان به جای وهله سازی مستقیم DbContext در ProductService آن را خارج از کلاس سرویس در اختیار استفاده کننده قرار داد:

```

public class ProductService
{
    private readonly IDbSet<Product> _products;
    private readonly IUnitOfWork _uow;
    public ProductService(IUnitOfWork uow)
    {
        _uow = uow;
        _products = _uow.Set<Product>();
    }
    public IEnumerable<Product> GetOrderedProducts()
    {
        return _products.OrderBy(x => x.Name).ToList();
    }
}

```

همانطور که مشاهده می‌کنید، الان IUnitOfWork به کلاس سرویس تزریق شده و در متدها، خبری از وهله سازی یک وابستگی (DbContext) نمی‌باشد.

اکنون برای تست این سرویس می‌توان پیاده سازی دیگری را از IUnitOfWork انجام داد و در کدهای تست به سرویس مورد نظر تزریق کرد. برای سهولت این امر قصد داریم از moq به عنوان چارچوب تقلید (Mocking framework) استفاده کنیم. برای [نصب moq](#) می‌توان از [بسته‌ی نیوگت](#) آن بهره جست. پیشتر [مطلبی](#) در رابطه با چارچوب‌های تقلید در سایت نوشته شده است. با توجه به اینکه ProductService به دیتابیس وابستگی دارد، مقصود این است که این وابستگی با ایجاد یک نمونه‌ی mock از IUnitOfWork حذف شود. برای این منظور در سازنده‌ی کلاس، تعدادی کالای درون حافظه ایجاد شده و به صورت IQueryable جایگزین شده DbSet است.

اگر به تعریف کلاس Entities که همان DbContext می‌باشد دقت کنید، مشاهده می‌شود که Products و تابع Set، هر دو به صورت Virtual تعریف شده‌اند. برای تغییر رفتار DbContext نیاز است در آزمون واحد، این دو با داده‌های درون حافظه کار کنند و رفتار آنها قرار است عوض شود. این تغییر رفتار از طریق چند ریختی (Polymorphism) خواهد بود. کلاس تست در نهایت اینگونه تعریف می‌شود:

```

[TestFixture]
public class ProductServiceTest
{
    private readonly ProductService _productService;
    public ProductServiceTest()
    {
        IQueryable<Product> data = GetRoadNetworks().AsQueryable();
        var mockSet = new Mock<DbSet<Product>>();
        mockSet.As<IQueryable<Product>>().Setup(m => m.Provider).Returns(data.Provider);
        mockSet.As<IQueryable<Product>>().Setup(m => m.Expression).Returns(data.Expression);
        mockSet.As<IQueryable<Product>>().Setup(m => m.ElementType).Returns(data.ElementType);
        mockSet.As<IQueryable<Product>>().Setup(m =>
        m.GetEnumerator()).Returns(data.GetEnumerator());
        var context = new Mock<Entities>();
        context.Setup(c => c.Products).Returns(mockSet.Object);
        context.Setup(m => m.Set<Product>()).Returns(mockSet.Object);
        _productService = new ProductService(context.Object);
    }
}

```

```

}
private IEnumerable<Product> GetRoadNetworks()
{
    return new List<Product>
    {
        new Product
        {
            Id = 1,
            Name = "A"
        },
        new Product
        {
            Id = 2,
            Name = "B"
        },
        new Product
        {
            Id = 3,
            Name = "C"
        }
    };
}
[Test]
public void GetOrderedProductTest()
{
    IEnumerable<Product> products = _productService.GetOrderedProducts();
    List<string> names = products.Select(x => x.Name).ToList();
    var expected = new List<string> {"A", "B", "C"};
    CollectionAssert.AreEqual(names, expected);
}
}

```

همانطور که مشاهده می‌شود، در سازنده‌ی کلاس تست، یک منبع داده‌ی درون حافظه‌ای به صورت IQueryable تولید شده و پیاده سازی‌های تقلیدی از DbContext به همراه تابع Set و همچنین DbSet کالاهای به کمک Moq ایجاد گردیده و در اختیار ProductService قرار داده شده است.

در نهایت، در یک تست تلاش شده است تا منطق متد GerOrderedProducts مورد آزمون قرار گیرد. **محدودیت این روش:** با اینکه LINQ یک روش و سینتکس یکتا برای دسترسی به منابع داده‌ای مختلف را محیا می‌کند، اما این الزامی برای یکسان بودن نتایج، هنگام استفاده از Providerهای مختلف LINQ نمی‌باشد. در تست نوشته شده از LINQ To Objects برای کوئری گرفتن از منبع داده استفاده شده است؛ در صورتیکه در برنامه‌ی اصلی از LINQ To Entities استفاده می‌شود و الزامی نیست که یک کوئری LINQ در دو Provider متفاوت یک رفتار را داشته باشد.

این نکته در قسمت Limitations of EF in-memory test doubles [این مطلب](#) هم شرح داده شده است. در نهایت این پرسش به وجود می‌آید که با وجود محدودیت ذکر شده، از این روش استفاده شود یا خیر؟ پاسخ این پرسش، بسته به هر سناریو، متفاوت است.

به عنوان نمونه اگر در یک سناریو داده‌ها با یک کوئری نه چندان پیچیده از منبع داده ای گرفته می‌شود و اعمال دیگری دیگری روی نتیجه‌ی کوئری درون حافظه انجام می‌شود می‌توان این روش را قابل اعتماد قلمداد کرد. [EFTesting.zip](#) برای مطالعه‌ی بیشتر مطالب متعددی در سایت در رابطه با [تزیق وابستگی](#) و آزمون‌های واحد نوشته شده است.

نظرات خوانندگان

نویسنده: شاهین کیاست
تاریخ: ۱۸:۴۶ ۱۳۹۳/۰۹/۰۳

-نکته تکمیلی در صورتی که از AsNoTracking در کدهای لایه‌ی سرویس استفاده شده برای Mock کردن آن می‌توان به این صورت عمل کرد:

```
context.Setup(c => c..AsNoTracking()).Returns(mockSet.Object);
```

در صورت عدم درج کد بالا تست‌ها با خطای Null Exception متوقف می‌شوند. [اطلاعات بیشتر](#)

نویسنده: ح مراداف
تاریخ: ۰:۳۶ ۱۳۹۳/۱۱/۱۸

با سلام و تشکر بابت مقاله جذابتون.

درون سایت Rhino Moq معرفی شده و شما Moq رو معرفی کردید ، بنده با Moq و کدنویسی اون احساس راحتی بیشتری می‌کنم ، می‌خواستم بدونم توی عملکرد آیا با هم فرقی دارن ؟
بنده بیشتر درگیر ساخت یک تقلید از کانتکس هستم (دقیقا مشابه کاری که شما در مقاله جاری انجام داده اید)
می‌خواستم ببینم اگر Rhino امکانات خاصی در این زمینه ارائه نمیده با Moq کار کنم.
(دنبال یک فریم ورک تقید خوب هستم که همیشه با اون کار کنم و باهاش راحت باشم)

تست واحد چیست؟

تست واحد ابزاری است برای مشاهده چگونگی عملکرد یک متد که توسط خود برنامه نویس نوشته میشود. به این صورت که پارامترهای ورودی، برای یک متد ساخته شده و آن متد فراخوانی و خروجی متد بسته به حالت مطلوب بررسی میشود. چنانچه خروجی مورد نظر مطلوب باشد تست واحد با موفقیت انجام میشود.

اهمیت انجام تست واحد چیست؟

درستی یک متد، مهمترین مسئله برای بررسی است و بارها مشاهده شده، استثناهایی رخ میدهند که توان تولید را به دلیل فرسایش تکراری رخداد می‌کاهند. نوشتن تست واحد منجر به این می‌شود چنانچه بعدها تغییری در بیزنس متد ایجاد شود و ورودی و خروجی‌ها تغییر نکند، صحت این تغییر بیزنس، توسط تست بررسی میشود؛ حتی میتوان این تست‌ها را در build پروژه قرار داد و در ابتدای اجرای یک Solution تمامی تست‌ها اجرا و درستی بخش به بخش اعضا چک شوند.

شروع تست واحد:

یک پروژه‌ی ساده را داریم برای تعریف حساب‌های بانکی شامل نام مشتری، مبلغ سپرده، وضعیت و 3 متد واریز به حساب و برداشت از حساب و تغییر وضعیت حساب که به صورت زیر است:

```
/// <summary>
/// حساب بانکی
/// </summary>
public class Account
{
    /// <summary>
    /// مشتری
    /// </summary>
    public string Customer { get; set; }
    /// <summary>
    /// موجودی حساب
    /// </summary>
    public float Balance { get; set; }
    /// <summary>
    /// وضعیت
    /// </summary>
    public bool Active { get; set; }

    public Account(string customer, float balance)
    {
        Customer = customer;
        Balance = balance;
        Active = true;
    }
    /// <summary>
    /// افزایش موجودی / واریز به حساب
    /// </summary>
    /// <param name="amount">مبلغ واریز</param>
    public void Credit(float amount)
    {
        if (!Active)
            throw new Exception("این حساب مسدود است");
        if (amount < 0)
            throw new ArgumentOutOfRangeException("amount");
        Balance += amount;
    }
    /// <summary>
    /// کاهش موجودی / برداشت از حساب
    /// </summary>
    /// <param name="amount">مبلغ برداشت</param>
    public void Debit(float amount)
    {

```

```

        if (!Active)
            throw new Exception("این حساب مسدود است.");
        if (amount < 0)
            throw new ArgumentOutOfRangeException("amount");
        if (Balance < amount)
            throw new ArgumentOutOfRangeException("amount");
        Balance -= amount;
    }
    /// <summary>
    /// انسداد / رفع انسداد
    /// </summary>
    public void ChangeStateAccount()
    {
        Active = !Active;
    }
}

```

تابع اصلی نیز به صورت زیر است:

```

class Program
{
    static void Main(string[] args)
    {
        var account = new Account("Ali", 1000);

        account.Credit(4000);
        account.Debit(2000);
        Console.WriteLine("Current balance is ${0}", account.Balance);
        Console.ReadKey();
    }
}

```

به Solution، یک پروژه از نوع تست واحد اضافه میکنیم.

در این پروژه ابتدا Reference ایی از پروژه‌ای که مورد تست هست میگیریم. سپس در کلاس تست مربوطه شروع به نوشتن متدی برای انواع تست متدهای پروژه اصلی میکنیم.

توجه داشته باشید که Data Annotation های بالای کلاس تست و متدهای تست، در تعیین نوع نگاه کامپایلر به این بلوک‌ها موثر است و باید این مسئله به درستی رعایت شود. همچنین در صورت نیاز میتوان از کلاس Startup برای شروع تست استفاده کرد که عمدتاً برای تعریف آن از نام ClassInit استفاده میشود و در بالای آن از [ClassInitialize] استفاده میشود. در Library تست واحد میتوان به دو صورت چگونگی صحت عملکرد یک تست را بررسی کرد: با استفاده از Assert و با استفاده از ExpectedException، که در زیر به هر دو صورت آن میپردازیم.

```

[TestClass]
public class UnitTest
{
    /// <summary>
    /// تعریف حساب جدید و بررسی تمامی فرآیندهای معمول روی حساب
    /// </summary>
    [TestMethod]
    public void Create_New_Account_And_Check_The_Process()
    {
        //Arrange
        var account = new Account("Hassan", 4000);
        var account2 = new Account("Ali", 10000);
        //Act
        account.Credit(5000);
        account2.Debit(3000);
        account.ChangeStateAccount();
        account2.Active = false;
        account2.ChangeStateAccount();
        //Assert
        Assert.AreEqual(account.Balance, 9000);
        Assert.AreEqual(account2.Balance, 7000);
        Assert.IsTrue(account2.Active);
        Assert.AreEqual(account.Active, false);
    }
}

```

همانطور که مشاهده میشود ابتدا در قسمت Arrange، خوراک تست آماده میشود. سپس در قسمت Act، فعالیتهایی که زیر ذره

بین تست هستند صورت می‌پذیرند و سپس در قسمت Assert درستی مقادیر با مقادیر مورد انتظار ما مطابقت داده میشوند. برای بررسی خطاهای تعیین شده هنگام نوشتن یک متد نیز میتوان به صورت زیر عمل کرد:

```
/// <summary>
/// زمانی که کاربر بخواهد به یک حساب مسدود واریز کند باید جلوی آن گرفته شود.
/// </summary>
[TestMethod]
[ExpectedException(typeof (Exception))]
public void When_Deactive_Account_Wants_To_add_Credit_Should_Throw_Exception()
{
    //Arrange
    var account = new Account("Hassan", 4000) {Active = false};
    //Act
    account.Credit(4000);
    //Assert
    //Assert is handled with ExpectedException
}

[TestMethod]
[ExpectedException(typeof (ArgumentOutOfRangeException))]
public void
When_Customer_Wants_To_Debit_More_Than_Balance_Should_Throw_ArgumentOutOfRangeException()
{
    //Arrange
    var account = new Account("Hassan", 4000);
    //Act
    account.Debit(5000);
    //Assert
    //Assert is handled with ArgumentOutOfRangeException
}
```

همانطور که مشخص است نام متد تست باید کامل و شفاف به صورتی انتخاب شود که بیانگر رخداد درون متد تست باشد. در این متدها Assert مورد انتظار با DataAnnotation که پیش از این توضیح داده شد کنترل گردیده است و بدین صورت کار میکند که وقتی Act انجام میشود، متد بررسی می‌کند تا آن Assert رخ بدهد.

استفاده از Library Moq در تست واحد

ابتدا باید به این توضیح بپردازیم که این کتابخانه چه کاری میکند و چه امکانی را برای انجام تست واحد فراهم میکند. در پروژه‌های بزرگ و زمانی که ارتباطات بین لایه‌ای زیادی موجود است و اصول SOLID رعایت میشود، شما در یک لایه برای ارائه فعالیت‌ها و خدمات متدهایتان با Interface های لایه‌های دیگر در ارتباط هستید و برای نوشتن تست واحد متدهایتان، مشکلی بزرگ دارید که نمیتوانید به این لایه‌ها دسترسی داشته باشید و ماهیت تست واحد را زیر سوال میبرید. Library Moq این امکان را به شما میدهد که از این Interface ها یک تصویر مجازی بسازید و همانند Snap Shot با آن کار کنید؛ بدون اینکه در لایه‌های دیگر بروید و ماهیت تست واحد را زیر سوال ببرید.

برای استفاده از متدهایی که در این Interface ها موجود است شما باید یک شیء از نوع Mock<> از آنها بسازید و سپس با استفاده از متد Setup به صورت مجازی متد مورد نظر را فراخوانی کنید و مقدار بازگشتی مورد انتظار را با Return معرفی کنید، سپس از آن استفاده کنید.

همچنین برای دسترسی به خود شیء از Property ایی با نام Objeet از موجودیت mock شده استفاده میکنیم. برای شناسایی بهتر اینکه از چه اینترفیس هایی باید Mock<> بسازید، میتوانید به متد سازنده کلاسی که معرف لایه ایست که برای آن تست واحد مینویسید، مراجعه کنید.

نحوه اجرای یک تست واحد با استفاده از Moq با توجه به توضیحات بالا به صورت زیر است:

پروژه مورد بررسی لایه Service برای تعریف واحدهای سازمانی است که با الگوریتم DDD و CQRS پیاده سازی شده است. ابتدا به Constructor خود لایه سرویس نگاه میکنیم تا بتوانید شناسایی کنید از چه Interface هایی باید Mock<> کنیم.

```
public class OrganizationalService : ICommandHandler<CreateUnitTypeCommand>,
    ICommandHandler<DeleteUnitTypeCommand>,
{
    private readonly IUnitOfWork _unitOfWork;
    private readonly IUnitTypeRepository _unitTypeRepository;
    private readonly IOrganizationUnitRepository _organizationUnitRepository;
    private readonly IOrganizationUnitDomainService _organizationUnitDomainService;
```

```

    public OrganizationalService(IUnitOfWork unitOfWork, IUnitTypeRepository unitTypeRepository,
    IOrganizationUnitRepository organizationUnitRepository, IOrganizationUnitDomainService
    organizationUnitDomainService)
    {
        _unitOfWork = unitOfWork;
        _unitTypeRepository = unitTypeRepository;
        _organizationUnitRepository = organizationUnitRepository;
        _organizationUnitDomainService = organizationUnitDomainService;
    }

```

مشاهده میکنید که 4 Interface استفاده شده و در متد سازنده نیز مقدار دهی شده اند. پس 4 Mock نیاز داریم. در پروژه تست به صورت زیر و در ClassInitialize عمل میکنیم.

```

[TestClass]
public class OrganizationServiceTest
{
    private static OrganizationalService _organizationalService;
    private static Mock<IUnitTypeRepository> _mockUnitTypeRepository;
    private static Mock<IUnitOfWork> _mockUnitOfWork;
    private static Mock<IOrganizationUnitRepository> _mockOrganizationUnitRepository;
    private static Mock<IOrganizationUnitDomainService> _mockOrganizationUnitDomainService;

    [ClassInitialize]
    public static void ClassInit(TestContext context)
    {
        TestBootstrapper.ConfigureDependencies();
        _mockUnitOfWork = new Mock<IUnitOfWork>();
        _mockUnitTypeRepository = new Mock<IUnitTypeRepository>();
        _mockOrganizationUnitRepository = new Mock<IOrganizationUnitRepository>();
        _mockOrganizationUnitDomainService = new Mock<IOrganizationUnitDomainService>();
        _organizationalService = new OrganizationalService(_mockUnitOfWork.Object,
        _mockUnitTypeRepository.Object,
        _mockOrganizationUnitRepository.Object, _mockOrganizationUnitDomainService.Object);
    }

```

از خود لایه سرویس با نام OrganizationService یک آبجکت میگیریم و 4 واسط دیگر به صورت Mock شده تعریف میشوند. همچنین در کلاس بارگذار از همان نوع مقدار دهی میگردند تا در اجرای تمامی متدهای تست، در دست کامپایلر باشند. همچنین برای new کردن خود سرویس از mock.obect که حاوی مقدار اصلی است استفاده میکنیم. خود متد اصلی به صورت زیر است:

```

/// <summary>
/// یک نوع واحد سازمانی را حذف مینماید
/// </summary>
/// <param name="command"></param>
public void Handle>DeleteUnitTypeCommand command)
{
    var unitType = _unitTypeRepository.FindBy(command.UnitTypeId);
    if (unitType == null)
        throw new DeleteEntityNotFoundException();

    ICanDeleteUnitTypeSpecification canDeleteUnitType = new
    CanDeleteUnitTypeSpecification(_organizationUnitRepository);
    if (canDeleteUnitType.IsSatisfiedBy(unitType))
        throw new UnitTypeIsUnderUsingException(unitType.Title);
    _unitTypeRepository.Remove(unitType);
}

```

متدهای تست این متد نیز به صورت زیر هستند:

```

/// <summary>
/// کامند حذف نوع واحد سازمانی باید به درستی حذف کند.
/// </summary>
[TestMethod]
public void DeleteUnitTypeCommand_Should_Delete_UnitType()
{
    //Arrange
    var unitTypeId = new Guid();
    var deleteUnitTypeCommand = new DeleteUnitTypeCommand { UnitTypeId = unitTypeId };
    var unitType = new UnitType("خوشه");
    var org = new List<OrganizationUnit>();

```

```

        _mockUnitTypeRepository.Setup(d =>
d.FindBy(deleteUnitTypeCommand.UnitTypeId)).Returns(unitType);
        _mockUnitTypeRepository.Setup(x => x.Remove(unitType));
        _mockOrganizationUnitRepository.Setup(z => z.FindBy(unitType)).Returns(org);
        try
        {
            //Act
            _organizationalService.Handle(deleteUnitTypeCommand);
        }
        catch (Exception ex)
        {
            //Assert
            Assert.Fail(ex.Message);
        }
    }
}

```

همانطور که مشاهده میشود ابتدا یک Guid به عنوان آی دی نوع واحد سازمانی گرفته میشود و همان آی دی برای تعریف کامند حذف به آن ارسال میشود. سپس یک نوع واحد سازمانی دلخواه تستی ساخته میشود و همچنین یک لیست خالی از واحدهای سازمانی که برای چک شدن توسط خود متد Handle استفاده شده است ساخته میشود. در اینجا این متد خالی است تا شرط غلط شود و عمل حذف به درستی صورت پذیرد.

برای اعمالی که در Handle انجام میشود و متدهایی که از Interface ها صدا زده میشوند Setup میکنیم و آنهایی را که Return دارند به object هایی که مورد انتظار خودمان هست نسبت میدهیم. در Setup اول میگوییم که آن Guid مربوط به "خوشه" است. در Setup بعدی برای عمل Remove کدی مینویسیم و چون عمل حذف Return ندارد میتواند، این خط به کل حذف شود! به طور کلی Setup هایی که Return ندارند میتوانند حذف شوند. در Setup بعدی از Interface دیگر متد FindBy که قرار است چک کند این نوع واحد سازمانی برای تعریف واحد سازمانی استفاده شده است، در Return به آن یک لیست خالی اختصاص میدهیم تا نشان دهیم لیست خالی برگشته است. عملیات Act را وارد Try میکنیم تا اگر به هر دلیل انجام نشد، Assert ما باشد. دو حالت رخداد استثناء که در متد اصلی تست شده است در دو متد تست به طور جداگانه تست گردیده است:

```

/// <summary>
/// کامند حذف یک نوع واحد سازمانی باید پیش از حذف بررسی کند که این شناسه داده شده برای حذف
/// موجود باشد.
/// </summary>
[TestMethod]
[ExpectedException(typeof(DeleteEntityNotFoundException))]
public void DeleteUnitTypeCommand_ShouldNot_Delete_When_UnitTypeId_NotExist()
{
    //Arrange
    var unitTypeId = new Guid();
    var deleteUnitTypeCommand = new DeleteUnitTypeCommand();
    var unitType = new UnitType("خوشه");
    var org = new List<OrganizationUnit>();
    _mockUnitTypeRepository.Setup(d => d.FindBy(unitTypeId)).Returns(unitType);
    _mockUnitTypeRepository.Setup(x => x.Remove(unitType));
    _mockOrganizationUnitRepository.Setup(z => z.FindBy(unitType)).Returns(org);

    //Act
    _organizationalService.Handle(deleteUnitTypeCommand);
}

/// <summary>
/// کامند حذف یک نوع واحد سازمانی نباید اجرا شود وقتی که نوع واحد برای تعریف واحدهای سازمان
/// استفاده شده است.
/// </summary>
[TestMethod]
[ExpectedException(typeof(UnitTypeIsUnderUsingException))]
public void
DeleteUnitTypeCommand_ShouldNot_Delete_When_UnitType_Exist_but_UsedForDefineOrganizationUnit()
{
    //Arrange
    var unitTypeId = new Guid();
    var deleteUnitTypeCommand = new DeleteUnitTypeCommand { UnitTypeId = unitTypeId };
    var unitType = new UnitType("خوشه");
    var org = new List<OrganizationUnit>()
    {
        new OrganizationUnit("مدیریت یک", unitType, null),
        new OrganizationUnit("مدیریت دو", unitType, null)
    };
    _mockUnitTypeRepository.Setup(d =>
d.FindBy(deleteUnitTypeCommand.UnitTypeId)).Returns(unitType);

```

```
_mockUnitTypeRepository.Setup(x => x.Remove(unitType));  
_mockOrganizationUnitRepository.Setup(z => z.FindBy(unitType)).Returns(org);  
  
//Act  
_organizationalService.Handle(deleteUnitTypeCommand);  
}
```

متد `DeleteUnitTypeCommand_ShouldNot_Delete_When_UnitTypeId_NotExist` همانطور که از نامش معلوم است بررسی میکند که نوع واحد سازمانی که ID آن برای حذف ارسال میشود در Database وجود دارد و اگر نباشد Exception مطلوب ما باید داده شود.

در متد `DeleteUnitTypeCommand_ShouldNot_Delete_When_UnitType_Exist_but_UsedForDefineOrganizationUnit` بررسی میشود که از این نوع واحد سازمانی برای تعریف واحد سازمانی استفاده شده است یا نه و صحت این مورد با الگوی Specification صورت گرفته است. استثنای مطلوب ما Assert و شرط درستی این متد تست، میباشد.