

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 درست‌تر بنویسم اما اون 3 موردی را که گفتم کارهایی است که برای آزمون واحد انجام می‌شود یعنی باید اینترفیس داشته باشیم برای مقلد ساختن و کلاس‌ها 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 می‌باشد و هیچ فرقی با آن ندارد یعنی شما مجبور نیستید از stub استفاده کنید می‌توانید 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 می‌گذاشتم راستش من دقیقاً نمی‌دونم شاید هم حرف شما درست باشد ولی من فقط می‌خواستم 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 می‌کنم 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