عنوان: **استفاده از shim و stub برای mock کردن در آزمون واحد**
نویسنده: آرش خوشبخت
تاریخ: ۱۳۹۲/۰۱/۲۸ ۲۲:۵۰
آدرس: www.dotnettips.info
گروه‌ها: Unit testing, test, Moq, Mock, Testing, Shim, Stub

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 حتما 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 objectهای وابسته تغییر یابند.

Shims و Stubs:
نوع stub همانند فریم ورک mock می‌باشد که برای مقلد ساختن اینترفیس‌ها و کلاس‌های non-sealed virtual یا ویژگی ها، 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));
        }
    }
}
```

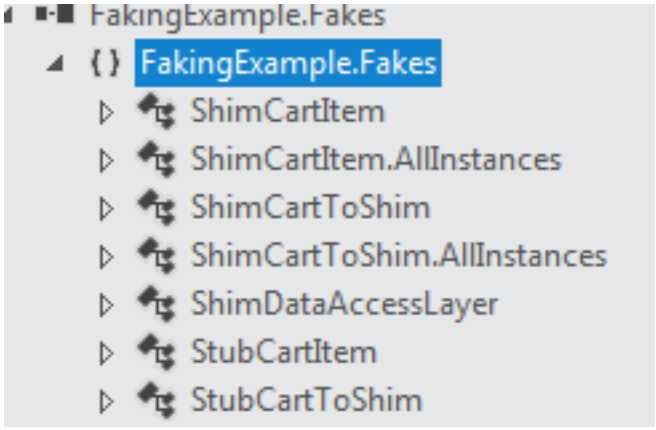
و همچنین کلاسی با نام CartItem به صورت زیر ایجاد کنید:

```
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، پوشه ای 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 اسکوپی 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 ویا stub آن را مقلد 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, ro 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 نظرم 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 گفتم شاید مقاله برای دیگران بیشتر از من قابل فهم است. ولی در کل مقاله خوبی بود و برای من کابردی 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 مثلا 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 CartToStub همان کد آخری است فقط خطی که نام کلاس را نوشته بود نگذاشتم. در مورد اینکه برای مثال مقدمه ای باید می‌گذاشتم راستش من دقیقا 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 می‌باشد)~~

تعریفی که از stup تو راهنماش اومده با مطلبی که شما ذکر کردید متفاوته

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 اینترفیس‌ها و abstractها و ... به کار میره همانطور که mock برای اینطور کلاس‌ها و متدها استفاده می‌شود

عنوان: **آشنایی با تست واحد و استفاده از کتابخانه Moq**
نویسنده: ارش کریمی
تاریخ: ۱۳۹۳/۰۹/۲۹ ۱۹:۵
آدرس: www.dotnettips.info
گروه‌ها: Unit testing, Dependency Injection, Moq, Mock, NUnit

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 چناچه بعدها تغییری در بیزنس متد ایجاد شود و ورودی و خروجی‌ها تغییر نکند، صحت این تغییر بیزنس، توسط تست بررسی مشود؛ 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 Annotationهای 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 عمدتا 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 ارایه فعالیت‌ها و خدمات متدهایتان با Interfaceهای 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 Interfaceها 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 Interfaceها 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 Objet 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.obectها 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 شده‌است 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 Interfaceها 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.

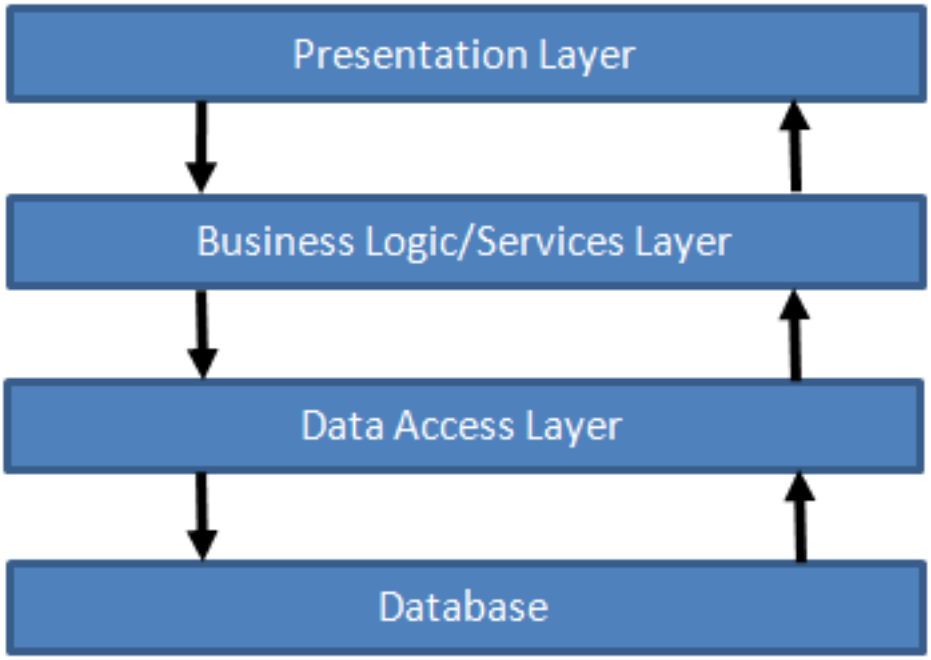
عنوان: **چگونه کد قابل تست بنویسیم - قسمت اول**
نویسنده: محمد بنازاده
تاریخ: ۱۳۹۳/۱۰/۰۱ ۱۳:۲۵
آدرس: www.dotnettips.info
گروه‌ها: Unit testing, Dependency Injection, Mock

مقدمه

نوشتن تست برای کدها بسیار عالی است، در صورتیکه بدانید چگونه این کار را بدرستی انجام دهید. متأسفانه بسیاری از منابع آموزشی موجود، این مطلب که چگونه کد قابل تست بنویسیم را رها می‌کنند؛ بدلیل اینکه آنها مراقبند در بین لایه هایی که در کدهای واقعی وجود دارند گیر نکنند، جایی که شما لایه‌های خدمات (Service Layer)، لایه‌های داده، و غیره را دارید. به ضرورت، وقتی میخواهید کدی را تست کنید که این وابستگی‌ها را دارد، تست‌ها بسیار کند و برای نوشتن دشوار هستند و اغلب بدلیل وابستگی‌ها شکست میخورند و نتیجه غیر قابل انتظاری خواهند داشت.

پیش زمینه

کدی که به خوبی نوشته شده باشد از لایه‌های جداگانه ای تشکیل شده است که هر لایه مسئول یک قسمت متفاوت از وظایف برنامه خواهد بود. لایه‌های واقعی بر اساس نیاز و نظر توسعه دهندگان متفاوت است، ولی یک ساختار رایج به شکل زیر خواهد بود.



لایه نمایش / رابط کاربری : این قسمت کد منطق نمایش و تعامل رابط کاربری می‌باشد. **منطق تجاری / لایه خدمات :** این قسمت منطق تجاری کد شما میباشد. برای نمونه در کد مربوط به یک کارت خرید، این کارت خرید میداند چگونه جمع کارت را محاسبه نماید و یا چگونه اقلام موجود در سفارش را شمارش کند. **لایه دستیابی به داده / لایه ماندگاری :** این کد میداند چگونه به منبع داده متصل شود و یک کارت خرید را بازگرداند و یا چگونه یک کارت را در منبع داده ذخیره نماید.
منبع داده : اینجا جایی است که محتویات کارت در آن ذخیره میشود.
مزیت مدیریت وابستگی‌ها

بدون مدیریت وابستگی‌ها، وقتی شما برای لایه نمایش تست می‌نویسید، کد شما در خدمات واقعی که به کد واقعی دستیابی به منبع داده وابسته است، گرفتار می‌شود و سپس به منبع داده اصلی متصل می‌شود. در واقع، وقتی شما در حال تست نویسی برای

گزینه "اضافه به کارت" و یا "دریافت تعداد اقلام" هستید، می‌خواهید کد را به صورت مجزا تست کنید و قادر باشید نتایج قابل پیش بینی را تضمین نمایید. بدون مدیریت وابستگی‌ها، تست‌های نمایش شما برای گزینه "اضافه به کارت" کند هستند و وابستگی‌ها نتایج غیر قابل پیش بینی بازمیگردانند که میتوانند باعث پاس نشدن تست شما شوند.

راه حل تزریق وابستگی است

راه حل این مسأله تزریق وابستگی است. تزریق وابستگی برای کسانی که تا بحال از آن استفاده نکرده اند، اغلب گیج کننده و پیچیده به نظر میرسد، اما در واقع، مفهومی بسیار ساده و فرآیندی با چند اصل ساده است. آنچه میخواهیم انجام دهیم مرکزیت دادن به وابستگی هاست. در این مورد، استفاده از شیء کارت خرید است و سپس رابطه بین کدها را کم می‌کنیم تا جاییکه وقتی شما برنامه را اجرا می‌کنید، از خدمات و منابع واقعی استفاده کند و وقتی آنرا تست می‌کنید، می‌توانید از خدمات جعلی (mocking) استفاده نمایید که سریع و قابل پیش بینی هستند. توجه داشته باشید که رویکردهای متفاوت بسیاری وجود دارند که میتوانید استفاده کنید، ولی من برای ساده نگهداشتن این مطلب، فقط رویکرد Constructor Injection را شرح میدهم.

گام 1 - وابستگی‌ها را شناسایی کنید

وابستگی‌ها وقتی اتفاق می‌افتند که کد شما از لایه‌های دیگر استفاده می‌نماید. برای نمونه، وقتی لایه نمایش از لایه خدمات استفاده می‌نماید. کد نمایش شما به لایه خدمات وابسته است، ولی ما میخواهیم کد لایه نمایش را به صورت مجزا تست کنیم.

```
public class ShoppingCartController : Controller
{
    public ActionResult GetCart()
    {
        //shopping cart service as a concrete dependency
        ShoppingCartService shoppingCartService = new ShoppingCartService();
        ShoppingCart cart = shoppingCartService.GetContents();
        return View("Cart", cart);
    }
    public ActionResult AddItemToCart(int itemId, int quantity)
    {
        //shopping cart service as a concrete dependency
        ShoppingCartService shoppingCartService = new ShoppingCartService();
        ShoppingCart cart = shoppingCartService.AddItemToCart(itemId, quantity);
        return View("Cart", cart);
    }
}
```

گام 2 – وابستگی‌ها را مرکزیت دهید

این کار با چندین روش قابل انجام است؛ در این مثال من میخواهم یک متغیر عضو از نوع ShoppingCartService ایجاد کنم و سپس آنرا به وهله ای که در Constructor ایجاد خواهم کرد، منتسب کنم. حال هرجا ShoppingCartService نیاز باشد بجای آنکه یک وهله جدید ایجاد کنم، از این وهله استفاده می‌نمایم.

```
public class ShoppingCartController : Controller
{
    private ShoppingCartService _shoppingCartService;
    public ShoppingCartController()
    {
        _shoppingCartService = new ShoppingCartService();
    }
    public ActionResult GetCart()
    {
        //now using the shared instance of the shoppingCartService dependency
        ShoppingCart cart = _shoppingCartService.GetContents();
        return View("Cart", cart);
    }
    public ActionResult AddItemToCart(int itemId, int quantity)
    {
        //now using the shared instance of the shoppingCartService dependency
        ShoppingCart cart = _shoppingCartService.AddItemToCart(itemId, quantity);
        return View("Cart", cart);
    }
}
```

در قسمت بعد درباره مواردی مانند از بین بردن ارتباط لایه ها، تزریق وابستگی و نوشتن تست بحث خواهم کرد.

نظرات خوانندگان

نویسنده: ح مرداد
تاریخ: ۱۳۹۳/۱۱/۱۸ ۰:۲۵

با سلام،

ابتدا از مقاله جذابتون تشکر می‌کنم.

سوالی ذهن بنده رو درگیر کرده :
طبق مقالات آموزش ام وی سی همین سایت بنده لایه سرویسی توی پروژه هام می‌سازم که کارش مشابه بیان شماست :
" **لایه دستیابی به داده / لایه ماندگاری :** این کد میداند چگونه به منبع داده متصل شود و یک کارت خرید را بازگرداند و یا چگونه یک کارت را در منبع داده ذخیره نماید. "

احساس می‌کنم که جای لایه بیزینس توی پروژهام خالیه ، لایه ای که کار محاسبات ریاضی و سایر محاسبات عددی رو به عهده داشته باشه.
از یکی از اساتیدم هم شنیدم که پروژها رو بصورت زیر می‌سازند
لایه دیتا - لایه بیزینس - لایه سرویس - لایه UI

که به نظرم لایه دیتا عملیات CRUD رو به عهده داشته باشه و لایه بیزینس هم محسابات و کارای پیچیده رو انجام بده و لایه سرویس هم لایه ای است که متدهای لازم جهت دسترسی UI به متدهای مورد نیاز در لایه‌های دیتا و بیزینس رو فراهم می‌کنه.
مثلا ثبت یک خرید جدید که موجب اجرای متد Add در کلاس ProductService میشه که در این متد ، متد CalcCommission جهت محاسبه پورسانت‌ها اجرا میشه و سپس نتیجه دریافتیبه کمک متدهای مربوطه در لایه دیتا در دیتابیس ثبت میشه.

به نظر میاد این لایه بندی قشنگ‌تر باشه.
(کل لایه‌های DomainClassess و DatabaseContext و Services پروژه‌های من در لایه دیتا قرار می‌گیرن)

می‌خواستم نظر شما رو درباره لایه بندی بدونم ؟
(به دنبال بهترین روش لایه بندی می‌گردم ، یک استاندارد مطمئن)

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۱/۱۸ ۰:۴۶

شما به نظر پرسش و پاسخ‌های EF 12 رو نخوندید یکبار. خلاصه‌اش اینه: زمانیکه از یک ORM استفاده می‌کنید، اون ORM هست که لایه Data شما رو تشکیل می‌ده و لازم نیست که بازنویسی‌اش کنید. لایه بیزنس هم همون لایه سرویس هست (با اون ادغام شده). اگر در مثالی که زده شده، لایه سرویس داخلش فقط Add یا Get هست (که نتیجه‌ی کار با لایه‌ی دیتا رو در اختیار لایه UI قرار می‌ده)، مابقی رو به خلاقیت خواننده واگذار کرده تا خودش جاهای خالی رو پر کنه. مثلا محاسبات هم انجام بده یا کارهای دیگر. هدفش بیشتر این بوده نمایش بده چطور می‌تونید از لایه دیتا در لایه بیزنس استفاده کنید و به اون دسترسی پیدا کنید.
ضمنا سعی کنید دچار over engineering (مدام لایه جدید اختراع کردن) و طراحی باقلوایی نشید.

عنوان: **چگونه کد قابل تست بنویسیم - قسمت دوم و پایانی**
نویسنده: محمد بنازاده
تاریخ: ۱۳۹۳/۱۰/۰۱ ۱۳:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Unit testing, Dependency Injection, Mock

گام 3 - از بین بردن ارتباط لایه‌ها (Loose Coupling) بجای استفاده از اشیاء واقعی ، براساس interfaceها برنامه نویسی کنید.
اگر شما کد خود را با استفاده از IShoppingCartService به عنوان یک interface بجای استفاده از شیء واقعی ShoppingCartService نوشته باشید، زمانیکه تست را مینویسید، میتوانید یک سرویس کارت خرید جعلی (mocking) که IShoppingCartService را پیاده سازی کرده جایگزین شیء اصلی نمایید. در کد زیر، توجه کنید تنها تغییر این است که متغیر عضو اکنون از نوع IShoppingCartService است بجای ShoppingCartService .

```
public interface IShoppingCartService
{
    ShoppingCart GetContents();
    ShoppingCart AddItemToCart(int itemId, int quantity);
}
public class ShoppingCartService : IShoppingCartService
{
    public ShoppingCart GetContents()
    {
        throw new NotImplementedException("Get cart from Persistence Layer");
    }
    public ShoppingCart AddItemToCart(int itemId, int quantity)
    {
        throw new NotImplementedException("Add Item to cart then return updated cart");
    }
}
public class ShoppingCart
{
    public List<product> Items { get; set; }
}
public class Product
{
    public int ItemId { get; set; }
    public string ItemName { get; set; }
}
public class ShoppingCartController : Controller
{
    //Concrete object below points to actual service
    //private ShoppingCartService _shoppingCartService;
    //loosely coupled code below uses the interface rather than the
    //concrete object
    private IShoppingCartService _shoppingCartService;
    public ShoppingCartController()
    {
        _shoppingCartService = new ShoppingCartService();
    }
    public ActionResult GetCart()
    {
        //now using the shared instance of the shoppingCartService dependency
        ShoppingCart cart = _shoppingCartService.GetContents();
        return View("Cart", cart);
    }
    public ActionResult AddItemToCart(int itemId, int quantity)
    {
        //now using the shared instance of the shoppingCartService dependency
        ShoppingCart cart = _shoppingCartService.AddItemToCart(itemId, quantity);
        return View("Cart", cart);
    }
}
```

گام 4 - وابستگی‌ها را تزریق کنید

اکنون ما تمام وابستگی‌ها را در یک جا مرکزیت داده‌ایم و کد ما رابطه کمی با آن وابستگی‌ها دارد. همانند گذشته، چندین راه برای پیاده سازی این گام وجود دارد. بدون استفاده از ابزارهای کمکی برای این مفهوم، ساده‌ترین راه دوباره نویسی (Overload) متد ایجاد کننده است:

```
//loosely coupled code below uses the interface rather
//than the concrete object
private IShoppingCartService _shoppingCartService;
```

```
//MVC uses this constructor
public ShoppingCartController()
{
    _shoppingCartService = new ShoppingCartService();
}
//You can use this constructor when testing to inject the
//ShoppingCartService dependency
public ShoppingCartController(IShoppingCartService shoppingCartService)
{
    _shoppingCartService = shoppingCartService;
}
```

گام 5 - تست را با استفاده از یک شیء جعلی (Mocking) انجام دهید

مثالی از یک سناریوی تست ممکن در زیر آمده است. توجه کنید که یک شیء جعلی از نوع کلاس ShoppingCartService ساخته‌ایم. این شیء جعلی فرستاده می‌شود به شیء Controller و متد GetContents پیاده سازی میشود تا بجای آنکه کد اصلی را که به منبع داده مراجعه می‌کند اجرا نماید، داده‌های جعلی و شبیه سازی شده را برگرداند. بدلیل آنکه تمام کد را نوشته ایم، بسیار سریعتر از اجرای کوئری بر روی دیتابیس اجرا خواهد شد و دیگر نگرانی بابت تهیه داده تستی و یا تصحیح داده بعد از اتمام تست (بازگرداندن داده به حالت قبل از تست) نخواهم داشت. توجه داشته باشید که بدلیل مرکزیت دادن به وابستگی‌ها در گام 2 ، تنها باید یکبار آنرا تزریق نماییم و بخاطر کاری که در گام 3 انجام شد، وابستگی ما به حدی پایین آمده که میتوانیم هر شیء ایی را (جعلی و یا حقیقی) ارسال کنیم و فقط کافیست شیء مورد نظر IShoppingCartService را پیاده سازی کرده باشد.

```
[TestClass]
public class ShoppingCartControllerTests
{
    [TestMethod]
    public void GetCartSmokeTest()
    {
        //arrange
        ShoppingCartController controller =
           new ShoppingCartController(new ShoppingCartServiceStub());
        // Act
        ActionResult result = controller.GetCart();
        // Assert
        Assert.IsInstanceOfType(result, typeof(ViewResult));
    }
}
/// <summary>
/// This is is a stub of the ShoppingCartService
/// </summary>
public class ShoppingCartServiceStub : IShoppingCartService
{
    public ShoppingCart GetContents()
    {
        return new ShoppingCart
        {
            Items = new List<product> {
                new Product {ItemId = 1, ItemName = "Widget"}
            }
        };
    }
    public ShoppingCart AddItemToCart(int itemId, int quantity)
    {
        throw new NotImplementedException();
    }
}
```

مطالب تکمیلی از یک ابزار کنترل وابستگی (IoC/DI) استفاده کنید:

از رایج‌ترین و عمومی‌ترین ابزارهای کنترل وابستگی برای .Net می‌توان به StructureMap و CastleWindsor اشاره کرد. در کد نویسی واقعی، شما وابستگی‌های بسیاری خواهید داشت، که این وابستگی‌ها هم وابستگی هایی دارند که به سرعت از مدیریت شما خارج خواهند شد. راه حل این مشکل استفاده از یک ابزار کنترل وابستگی خواهد بود. **از یک چارچوب تجزیه (Isolation Framework) استفاده نمایید:**

برای ایجاد اشیاء جعلی ممکن است کار زیادی لازم باشد و استفاده از یک Isolation Framework میتواند زمان و میزان کد نویسی شمارا کم کند. از رایج‌ترین این ابزارها میتوان Rhino Mocks و Moq را نام برد.