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.