عنوان: **Test Driven Development**
نویسنده: شهروز جعفری
تاریخ: ۱۳۹۲/۰۳/۰۸ ۲۲:۱۰
آدرس: www.dotnettips.info
گروه‌ها: Unit testing, TDD

نوشتن تست برای نرم افزار امری ضروریست، چه پس از تولید نرم افزار چه در حین تولید، در کل به وسیله تست می‌توان از به وجود آمدن باگ‌ها در هنگام گسترش دادن برنامه تا حد قابل توجهی جلوگیری کرد.
از معروف ترین روش‌های تست می‌توان عناوین زیر را نام برد:

Unit test

Integration test

Smoke test

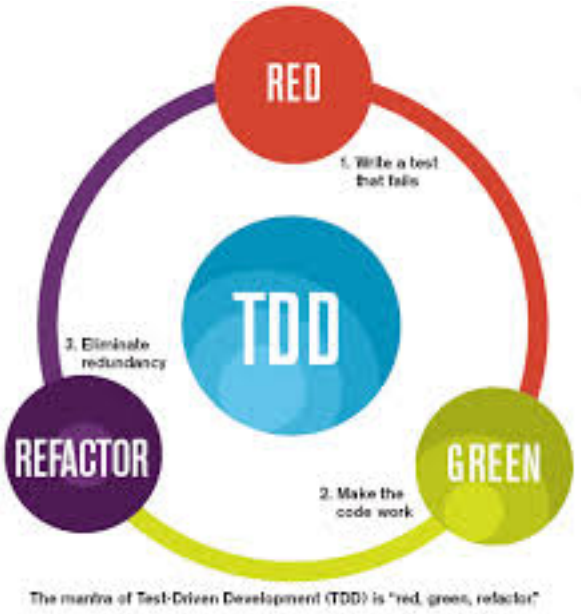
Regression test

Acceptance test

Test Driven Development
یک پروسه تولید نرم افزار است که برای اولین بار توسط Kent_Beck معرفی شد.
TDD شامل 4 مرحله کلی است:

نوشتن تست **قبل از نوشتن کد.**
کامپایل کردن کد و اطمینان از **Fail شدن** کامپایل.
پیاده سازی کد به طوری که تست ما پاس شود.
Refactoring

مراحل 4 گانه تست باید به صورت متناوب اجرا شوند.
البته بسیاری این 4 مورد را با عبارت red/green/ refactor نیز می‌شناسند.



همانطور که گفته شد در کل نوشتن تست باعث می‌شود که با اضافه شدن کدهای جدید در برنامه از به وجود آمدن باگ تا حدی جلوگیری شود.
اما مزایای TDD:

TDD باعث کاهش زمان تولید نرم افزار می‌شود.البته این حرف کمی عجیب است.(در ادامه بیشتر توضیح می‌دهم)

اعتماد شما نسبت به کد بالا می‌رود.
باگ کمتری تولید می‌شود بتابراین اعتماد مصرف کنندگان نیز نسبت به برنامه شما بالا می‌رود.
باعث نظم در کد می‌شود.
باعث انعطاف پذیری بیشتر در نرم افزار می‌شود.
ریسک تولید نرم افزار به علت باگ کمتر به حداقل می‌سد.
...

البته باید به این نکته نیز اشاره داشت که مایکروسافت تحقیقی انجام داده که بر طبق آن نوشتن کد به روش TDD می‌تواند 15 تا 30 در صد روند تولید نرم افزار را افزایش دهد ولی در عوض بین 40 تا 90 در صد می‌تواند از به وجود آمدن باگ جدید جلوگیری کند.
در بسیاری از محیطهای برنامه نویسی، نه تنها به این موضوع اهمیت داده نمی‌شود بلکه به طور کلی به اشتباه گرفته شده و حتی در پروژه هایی که تست نوشته می‌شود مفاهیم آن(که در بالا نام برده شده)جابجا شده و به اشتباه نام برده می‌شود.
هدف از نوشتن تست,تست کردن قطعات کوچک کد است,به عنوان مثال نباید تست به گونه ای باشد که یک متد با 300 خط کد را تحت پوشش قرار دهد.ابتدا باید کد به قطعات کوچک شکسته و بعد تست شود.
یک نمونه از متد تست:

```
[Test]
        public void TestFullName()
        {

            Person person = new Person ();
            person.lname = "Doe";
            person.mname = "Roe";
            person.fname = "John";

            string actual = person.GetFullName();
            string expected = "John Roe Doe";
            Assert.AreEqual(expected, actual, ”The GetFullName returned a different Value”);
        }
```

هدف از نوشتن این پست مقدمه ای بر شروع سری پست‌های TDD باستفاده از MVC.Net و فریم ورک قدرت مند تست Nunit است.

نظرات خوانندگان

نویسنده: رضا
تاریخ: ۱۳۹۲/۰۵/۲۰ ۰:۱۰

با سلام

با تشکر از مقاله خوبتون

خواستم ببینم پروژه وبی وجود داره که در اون قسمت‌های مختلف سایت رو با انواع تست‌های مختلف پیاده سازی کرده باشه (یا حداقل با روش unit test)؟

من از unit test استفاده می‌کنم ولی یه جورایی توش سر در گمم (تست‌ها رو می‌نویسم و عملکردش هم قابل قبوله ولی یه جورایی کدها خیلی بی نظم و بهم ریخته است)

نویسنده: حسینی
تاریخ: ۱۳۹۳/۰۲/۱۰ ۲۱:۴

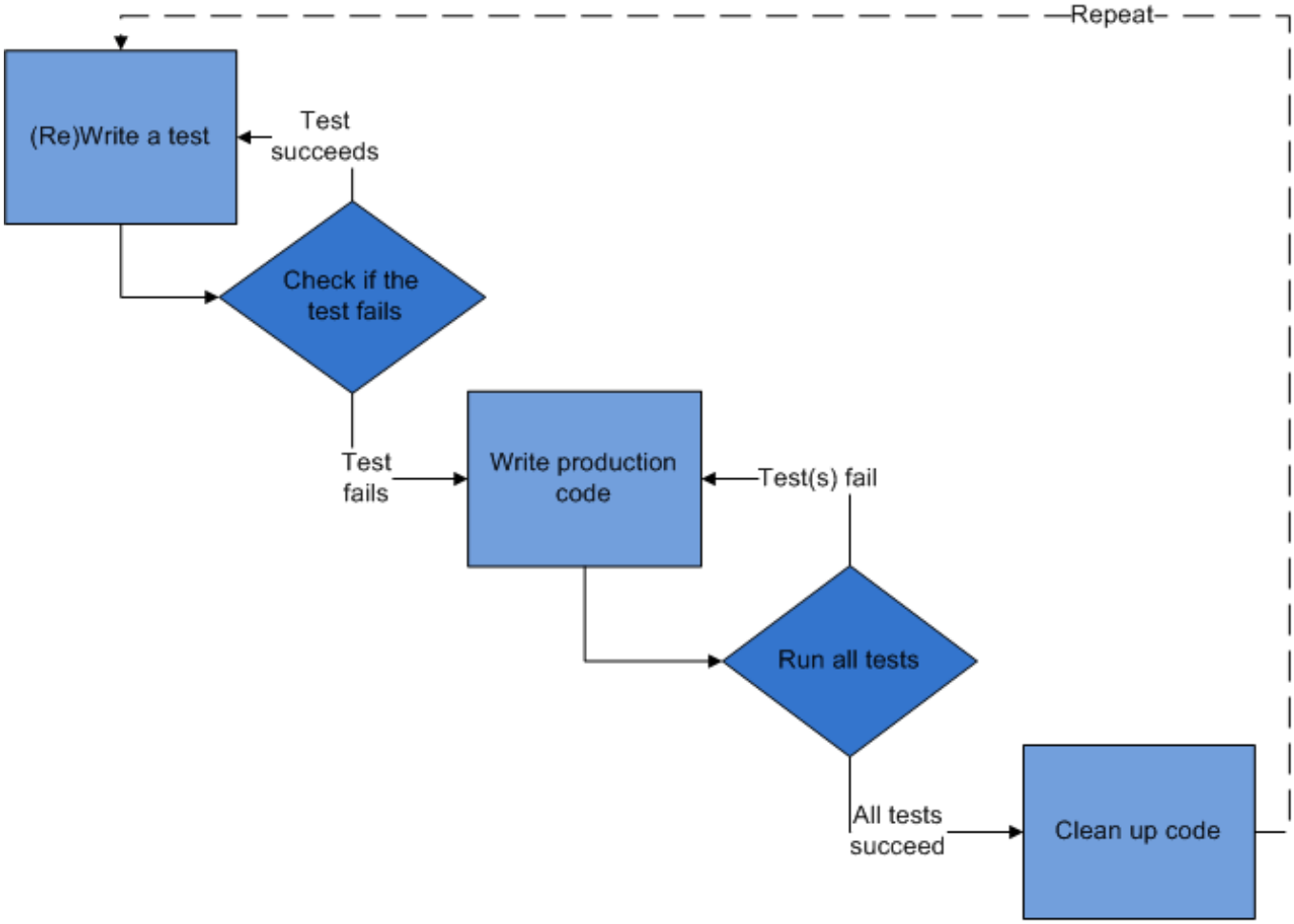
سلام

تفاوت TDD با unit testing چیه؟

همون مباحثی که برای tdd مطرح هست برای unit test هم مطرح میشه من تفاوت این 2 رو متوجه نمیشم.

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۳/۰۲/۱۱ ۱۱:۸

آزمون واحد بر می‌گردد به آنچه شما تست می‌کنید و TDD اشاره دارد به زمانی که تست می‌کنید،
در واقع فرض کنید برنامه‌ی ماشین حساب را توسعه داده اید، اکنون برای عملگر جمع تست می‌نویسید، این Unit Test هست. در TDD ، آزمون واحد شما توسعه و طراحی را پیش می‌برد، اگر مقالات مربوطه به TDD را مطالعه کنید، در TDD ابتدا بدون پیدا سازی هیچ ویژگی تست نوشته می‌شود.



به تصویر بالا توجه کنید، ابتدا تست نوشته شده، سپس کد محصول نوشته می‌شود..

عنوان: **Test Driven Development #2**
نویسنده: شهروز جعفری
تاریخ: ۱۳۹۲/۰۳/۰۹ ۲۰:۲۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing, TDD

در مطلب قبل شما با TDD آشنا شدید اکنون بهتر است با یک مثال نشان دهم منظور از Test Driven Development چیست.
برای شروع کافی است یک پروژه کنسول ساخته و Nunit را از طریق کنسول Nuget نصب کنید.

PM> Install-Package NUnit

معمولا برای کلاس‌های تست یک پروژه جدا در نظر گرفته می‌شود، ولی برای شروع می‌توانید از همان پروژه اصلی استفاده کنید.
پس از نصب شدن Nunit می توانیم شروع به ساختن کلاس‌های تست کنیم:

```
[TestFixture]
    public class HelloWorldTest
    {
    }
```

همانطور که ملاحظه می‌کنید کلاس ما با Attribute به نام TestFixture مزین شده است که خاص فریمورک Nunit است، در صورتی که از فریمورک دیگری برای تست استفاده می‌کنید باید تنظیمات مربوط به آن را انجام دهید.متدهای تست ما نیز با Attribute به نام Test مزین می‌شوند.

```
[Test]
   public void ShouldSayHelloWorld()
  {
  }
```

همانطور که دقت کردید متد ما به صورتی نام گذاری شده است که مشخص کننده کاری باشد که قرار است انجام دهد.این یکی دیگر از مزایای تست نویسی است که یک داکیومنت تقریباً کامل در طول تولید نرم افزار ایجاد میشود.همچنین متد تست باید غیر استاتیک با خروجی void باشد .متدهای تست بهتر است فقط یک موضوع را تست کنند، به طور مثال نباید هم اضافه شدن یک رکورد و هم ریدایرکت شدن به صفحه ای خاص را تست کرد .
حالا وقت آن است که قبل از نوشتن کد اول تستش را بنویسیم.

```
[Test]
        public void ShouldSayHelloWorld()
        {
            const string result = "Hello World";
            Assert.AreEqual(result, HelloWorld.SayHello());
        }
```

کلاس Assert شامل توابعی بسیار قدرتمند است که مارا در اجرای تست بهتر کمک میکند.شامل متد هایی مانند .

AreEqual

AreNotEqual

AreNotSame

AssertDoublesAreEqual

Contains

DoesNotThrow

Equals

Fail

Greater

GreaterOrEqual

Ignore

IsEmpty

IsInstanceOf

IsNaN

IsNotNull

True

...

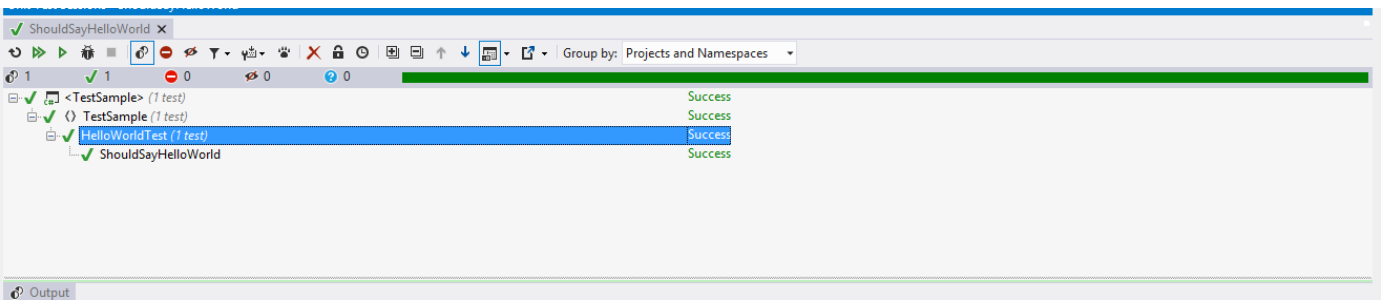
است.

هر کدام از متدهای بالا کاربرد خاصی را دارند که به طور جداگانه به آن می‌پردازیم.

به علت وجود نداشتن کلاس HelloWorld در زمان کامپایل با خطا مواجه می‌شویم.سپس کلاس مربوطه را ساخته و متد SayHello طوری پیاده سازی می‌کنیم که تست ما را پاس کند..(برنامه resharper برای اجرای متدهای تست بسیار کار آمد است)

```
public class HelloWorld
    {
        public static string SayHello()
        {
            return "Hello World";
        }
    }
```

حال دوباره تست را اجرا کرده و می‌بینید که تست ما پاس شد.



نیازی به مرحله ریفکتورینگ نیست زیرا کلاس ما به اندازه کافی ساده است.
برای مقایسه بین Nunit و ابزار توکار ویژوال استودیو می توانید به این سوال نگاهی بیاندازید.
در مطلب بعدی با استفاده از تست پذیری Mvc.net شروع به نوشتن تست هایی جدی‌تر خواهیم کرد.

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۳/۱۰ ۲۲:۵۶

با تشکر. روش دوم بدون استفاده از ری شارپر:

در VS 2012 بعد از نصب NUnit Test adaptor ، میشه از Visual Studio 2012 Test Runner مستقیما برای کار با NUnit استفاده کرد.

عنوان: **Test Driven Development #3**
نویسنده: شهروز جعفری
تاریخ: ۱۳۹۲/۰۳/۱۱ ۲۱:۳۵
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing, TDD

در پست قبلی با نوشتن یک تست ساده، با مفهوم TDD بیشتر آشنا شدیم .در این پست قصد بر این است که به وسیله Mvc.Net شروع به نوشتن تست‌های جدی‌تر کرده و از مزایای آن بهره ببریم .
برای شروع یک پروژه Mvc.Net ساخته و Nunit را در آن نصب می‌کنیم.
مدل زیر را در پوشه مدل‌ها می‌سازیم:

```
public class Idea
    {
        public static List<Idea> Ideas = new List<Idea>
            {
                new Idea{Content="سایتی که در ایده به اشتراک گذاشته شود",Title = "سایت ایده ها"},
                new Idea{Content="عینک گوگل را فارسی کنم",Title = "عینک گوگل"},
            };
        public string Content { get;  set;}
        public string Title { get; set; }
    }
```

```
[TestFixture]
    public class IdeaTest
    {
        [Test]
        public void ShouldDisplayListOfIdea()
        {
            var viewResult = new IdeaController().Index() as ViewResult;
            Assert.AreEqual(Idea.Ideas, viewResult.Model)
             Assert.IsNotNull(viewResult.Model);


        }
 }
```

کد بالا شامل مقایسه مقدار خروجی Action با لیستی از مدل Idea و همچنین اطمینان از خالی نبودن مدل ارسالی به view می‌باشد.در خط اول یک وهله از Controller می سازیم و Action مورد نظر را به شی از جنس ViewResult تبدیل(Cast) می‌کنیم پس از آن به وسیله viewResult.Model به مدلی که به سمت view پاس داده می‌شود دسترسی خواهیم داشت.اکنون اگر تست را اجرا کنیم با خطای کامپایل مواجه می‌شویم.حال Controller و Action مورد نظر را به صورتی که تست ما پاس شود پیاده سازی می‌کنیم.

```
public class IdeaController : Controller
    {

        public ActionResult Index()
        {
            return View(Idea.Ideas);
        }
      }
```

کد بالا مقدار Ideas را به view برمیگرداند.
در این دروره ما به تست کردن ویوها نخواهیم پرداخت.
تست بعدی تست ساده ای است که فقط میخواهیم از از وجود داشتن یک Action و نام view بازگشتی اطمینان حاصل کنیم.

```
[Test]
        public void ShouldLoadCreateIdeaView()
        {
            var viewResult = new IdeaController().Create() as ViewResult;
            Assert.AreEqual(string.Empty, viewResult.ViewName);
        }
```

در کد بالا مثل تست قبل، یک وهله از Controller می سازیم و سپس نام view بازگشتی را با string.Empty مقایسه میکنیم به این معنی که view خروجی Action ما نباید نامی داشته باشد و براساس قرار دادها باید هم نام اکشن باشد.
حال نوبت به پیاده سازی اکشن رسید.:

```
public ActionResult Create()
        {
            return View();
        }
```

در تست بعدی میخواهیم عملیات اضافه شدن یک Idea را به لیست بررسی کنیم:

```
[Test]
        public void ShouldAddIdeaItem()
        {
            var idea = new Idea { Title = "شبکه اجتماعی ", Content = " شبکه اجتماعی سینمایی" };
            var redirectToRouteResult = new IdeaController().Create(idea) as RedirectToRouteResult;
            Assert.Contains(idea, Idea.Ideas);
            Assert.AreEqual("Index",redirectToRouteResult.RouteValues["action"]);
        }
```

تست بالا نیز مانند دو تست قبل است با این تفاوت که مخواهیم ریدارکت شدن به یک Action خاص را نیز تست کنیم.برای همین مقدار خروجی را به RedirectToRouteResult تبدیل می‌کنیم.در ادامه یک Idea جدید ساخته و به لست اضافه میکنیم سپس از وجود داشتن آن در لیست Ideas اطمینان حاصل می‌کنیم.در خط آخر نیز نام Action که انتظار داریم بعد از اضافه شدن یک Idea ,کاربر به آن هدایت شود را ست می‌کنیم.
پیاده سازی Action به شکل زیر است:

```
public ActionResult Create(Idea idea)
        {
            Idea.Ideas.Add(idea);
            return RedirectToAction("Index");
        }
```

در این پست شما با مدل تست نویسی برایMvc.Net آشنا شدید.در مطلب بعدی شما با تست حذف و اصلاح Ideas آشنا خواهید شد.

نظرات خوانندگان

نویسنده: دنیس ریچی
تاریخ: ۱۳۹۲/۰۳/۱۸ ۱۵:۵۹

بسیار عالی بود. لطفن ادامه بدید. اگه میشه در قسمتهای بعدی راجع به TDD کار کردن برای جاوا اسکریپت در ویوها و QUnit هم توضیح بدید

نویسنده: s.t
تاریخ: ۱۳۹۲/۰۵/۰۹ ۱۷:۵۲

بسیار عالی،
منتظر ادامه‌ی این مبحث هستیم

عنوان: آماده سازی Jasmine برای پروژه های Asp.Net MVC
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۷/۰۸ ۹:۴۵
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing, test, TDD, JavaScriptFrameWork, Jasmine

با گسترش روز افزون برنامه‌های تحت وب، نیاز به یک سری ابزار برای تست و اطمینان از نحوه عملکرد صحیح کدهای نوشته شده احساس می‌شود. Jasmine یکی از این ابزارهای قدرتمند برای تست کدهای JavaScript است.
چندی پیش در سایت جاری چند مقاله خوب توسط یکی از دوستان درباره Qunit منتشر شد. Qunit یک ابزار قدرتمند و مناسب برای تست کدهای جاوااسکریپت است و در اثبات صحت این گفته همین کافیست که بدانیم برای تست کدهای نوشته شده در پروژه‌های متن بازی هم چون Backbone.Js و JQuery از این فریم ورک استفاده شده است. اما به احتمال قوی در ذهن شما این سوال مطرح شده است که خب! در صورت آشنایی با Qunit چه نیاز به یادگیری Jasmine یا خدای نکرده Mocha و FuncUnit است؟ هدف صرفا معرفی یک ابزار غیر برای تست کد است نه مقایسه و نتیجه گیری برای تعیین میزان برتری این ابزارها. اصولا مهم‌ترین دلیل برای انتخاب، علاوه بر امکانات و انعطاف پذیری، فاکتور راحتی و آسان بودن در هنگام استفاده است که به صورت مستقیم به شما و تیم توسعه نرم افزار بستگی دارد.

اما به عنوان توسعه دهنده نرم افزار که قرار است از این ابزار استفاده کنیم بهتر است با تفاوت‌ها و شباهت‌های مهم این دو فریم ورک آشنا باشیم:

»Jasmine یک فریم ورک تست کدهای جاوا اسکریپ بر مبنای Behavior-Driven Development است در حالی که Qunit بر مبنای Test-Driven Development است و همین مسئله مهم‌ترین تفاوت بین این دو فریم ورک می‌باشد.
»اگر قصد دارید که از Qunit نیز به روش BDD استفاده نمایید باید از ترکیب Pavlov به همراه Qunit استفاده کنید.
»Jasmine از مباحث مربوط به Spies و Mocking به خوبی پشتیبانی می‌کند ولی این امکان به صورت توکار در Qunit فراهم نیست. برای اینکه بتوانیم این مفاهیم را در Qunit پیاده سازی کنیم باید از فریم ورک‌های دیگر نظیر SinonJS به همراه Qunit استفاده کنیم.
»هر دو فریم ورک بالا به سادگی و راحتی کار معروف هستند
»تمام موارد مربوط به الگوهای Matching در هر دو فریم ورک به خوبی تعبیه شده است
» هر دو فریم ورک بالا از مباحث مربوط به Asynchronous Testing برای تست کدهای Ajax ای به خوبی پشتیبانی می‌کنند.

بررسی چند مفهوم
قبل از شروع، بهتر است که با چند مفهوم کلی و در عین حال مهم این فریم ورک آشنا شویم

```
describe('JavaScript addition operator', function () {
 it('adds two numbers together', function () {
  expect(1 + 2).toEqual(3);
 });
});
```

در کد بالا یک نمونه از تست نوشته شده با استفاده از Jasmine را مشاهده می‌کنید. دستور describe برای تعریف یک تابع تست مورد استفاده قرار می‌گیرد که دارای دو پارامتر ورودی است. ابتدا یک نام را به این تست اختصاص دهید(بهتر است که این عنوان به صورت یک جمله قابل فهم باشد). سپس یک تابع به عنوان بدنه تست نوشته می‌شود. به این تابع Spec گفته می‌شود.
در تابع it کد بالا شما می‌توانید کدهای مربوط بدنه توابع تست خود را بنویسید. برای پیاده سازی Assert در توابع تست مفهوم expectationها وجود دارد. در واقع expect برای بررسی مقادیر حقیقی با مقادیر مورد انتظار مورد استفاده قرار می‌گیرد و شامل مقادیر true یا false خواهد بود.
برای Setup و Teardown توابع تست خود باید از توابع beforeEach و afterEach که بدین منظور تعبیه شده اند استفاده کنید.

```
describe("A spec (with setup and tear-down)", function() {
```

```
  var foo;

  beforeEach(function() {
    foo = 0;
    foo += 1;
  });

  afterEach(function() {
    foo = 0;
  });

  it("is just a function, so it can contain any code", function() {
    expect(foo).toEqual(1);
  });

  it("can have more than one expectation", function() {
    expect(foo).toEqual(1);
    expect(true).toEqual(true);
  });
});
```

کاملا واضح است که در تابع beforeEach مجموعه دستورالعمل‌های مربوط به setup تست وجود دارد. سپس دو تابع it برای پیاده سازی عملیات Assertion نوشته شده است. در پایان هم دستورات تابع afterEach ایجاد می‌شوند.

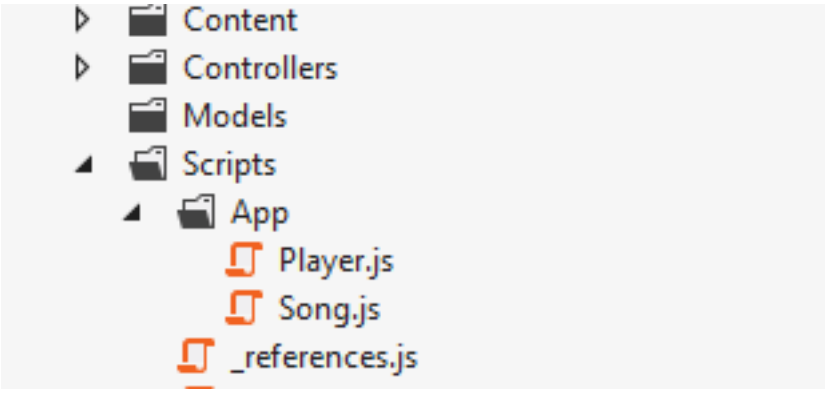
اگر در کد تست خود قصد دارید که یک تابع describe یا it را غیر فعال کنید کافیست یک x به ابتدای آن‌ها اضافه کنید و دیگر نیاز به هیچ کار اضافه دیگری برای comment کردن کد نیست.

```
xdescribe("A spec", function() {
  var foo;

  beforeEach(function() {
    foo = 0;
    foo += 1;
  });

  xit("is just a function, so it can contain any code", function() {
    expect(foo).toEqual(1);
  });
});
```

توابع describe و it بالا در هنگام تست نادیده گرفته می‌شوند و خروجی آن‌ها مشاهده نخواهد شد.

درادامه قصد پیاده سازی یک مثال را با استفاده از Jasmine و RequireJs در پروژه Asp.Net MVC دارم.
برای شروع آخرین نسخه Jasmine را از اینجا دریافت نمایید. یک پروژه Asp.Net MVC به همراه پروژه تست به صورت Empty ایجاد کنید(در هنگام ایجاد پروژه، گزینه create unit test را انتخاب نمایید). فایل دانلود شده را unzip نمایید و دو پوشه lib و spec ،به همراه فایل specRunner.html را در پروژه تست خود کپی نمایید.
فولدر lib شامل فایل‌ها کدهای Jasmine برای setup و tear down و spice و تست کدهای شما می‌باشد.
فایل specRunner.html به واقع یک فایل برای نمایش فایل‌های تست و همچنین نمایش نتیجه تست است.
فولدر spec نیز شامل کدهای Jasmine برای کمک به نوشتن تست می‌باشد.

در این مثال قصد داریم فایل‌های player.js و song.js که به عنوان نمونه به همراه این فریم ورک قرار دارد را در قالب یک پروژه MVC به همراه RequireJs، تست نماییم. در نتیجه این فایل‌ها را از فولدر src انتخاب نمایید و آن‌ها را در قسمت Scripts پروژه اصلی خود کپی کنید(ابتدا یک پوشه به نام App بسازید و فایل‌ها را در آن قرار دهید)



برای استفاده از requireJs باید دستور define را در ابتدا این فایل‌ها اضافه نماییم. در نتیجه فایل‌های Player.js و Song.js را باز کنید و تغییرات زیر را در ابتدای این فایل‌ها اعمال نمایید.

Song.js

```
define(function () {
    function Song() {
    }

    Song.prototype.persistFavoriteStatus = function (value) {
        // something complicated
        throw new Error("not yet implemented");
    };
});
```

Player.js

```
define(function () {
    function Player() {
    }
    Player.prototype.play = function (song) {
        this.currentlyPlayingSong = song;
        this.isPlaying = true;
    };

    Player.prototype.pause = function () {
        this.isPlaying = false;
    };

    Player.prototype.resume = function () {
        if (this.isPlaying) {
            throw new Error("song is already playing");
        }

        this.isPlaying = true;
    };

    Player.prototype.makeFavorite = function () {
        this.currentlyPlayingSong.persistFavoriteStatus(true);
    };
});
```

حال فایل SpecRunner.html را بازکنید و کدهای مربوط به تگ script که به مسیر اصلی فایل‌های تست اشاره می‌کند را Comment نمایید و به جای آن تگ Script مربوط به RequireJs را اضافه نمایید. برای پیکر بندی RequireJs باید از baseUrl و paths استفاده کرد.

```
<link rel="shortcut icon" type="image/png" href="lib/jasmine-1.2.0/jasmine_favicon.png">
<link rel="stylesheet" type="text/css" href="lib/jasmine-1.2.0/jasmine.css">
<script type="text/javascript" src="lib/jasmine-1.2.0/jasmine.js"></script>
<script type="text/javascript" src="lib/jasmine-1.2.0/jasmine-html.js"></script>

<script type="text/javascript" src="../RequireJsMvcStarter/Scripts/require.js"></script>

<!-- include source files here... -->
<!--<script type="text/javascript" src="spec/SpecHelper.js"></script>-->
<!--<script type="text/javascript" src="spec/PlayerSpec.js"></script>-->

<!-- include spec files here... -->
<!--<script type="text/javascript" src="src/Player.js"></script>
<script type="text/javascript" src="src/Song.js"></script>-->

<script type="text/javascript">

    require.config({
        baseUrl: '../RequireJsMvcStarter/Scripts/App',
        paths: {
            spec: '../../../RequireJsMvcStarter.Scripts.Test/spec'
        }
    });
</script>
```

baseUrl در پیکر بندی requireJs به مسیر فایل‌های پروژه که در پروژه اصلی MVC قرار دارد اشاره می‌کند. paths برای تعیین مسیر فایل‌های تست که در پوشه spec در پروژه تست قرار دارد اشاره می‌کند. اگر دقت کرده باشید به دلیل اینگه تگ‌های script مربوط به لود فایل‌های SpecHelper.js و PlayerSpec.js به صورت comment در آمده اند در نتیجه این فایل‌ها لود نخواهند شد و خروجی مورد نظر مشاهده نمی‌شود. در این جا باید از مکانیزم AMD موجود در RequireJs استفاده نماییم و فایل‌های مربوطه را لود کنیم. برای این کار نیاز به اضافه کردن دستور require در ابتدای تگ script به صورت زیر در این فایل است. در نتیجه فایل‌های PlayerSpec و SpecHelper نیز توسط RequireJs لود خواهند شد.

```
<script type="text/javascript">

    require(['spec/PlayerSpec', 'spec/SpecHelper'], function() {
        var jasmineEnv = jasmine.getEnv();
        jasmineEnv.updateInterval = 1000;

        var htmlReporter = new jasmine.HtmlReporter();

        jasmineEnv.addReporter(htmlReporter);

        jasmineEnv.specFilter = function(spec) {
            return htmlReporter.specFilter(spec);
        };

        var currentWindowOnload = window.onload;

        window.onload = function() {
            if (currentWindowOnload) {
                currentWindowOnload();
            }
            execJasmine();
        };

        function execJasmine() {
            jasmineEnv.execute();
        }
    });
</script>
```

نیاز به یک تغییر کوچک دیگر نیز وجود دارد. فایل PlayerSpec را باز نمایید و وابستگی فایل‌های آن را تعیین نمایید. از آن جا که این فایل برای تست فایل‌های Player , Song ایجاد شده است در نتیجه باید از define برای تعیین این وابستگی‌ها استفاده نماییم.

PlayerSpec.js

```
1  define(['Player', 'Song'], function(Player, Song) {
2      describe("Player", function() {
3          var player;
4          var song;
5
6          beforeEach(function() {
7              player = new Player();
8              song = new Song();
9          });
10
```

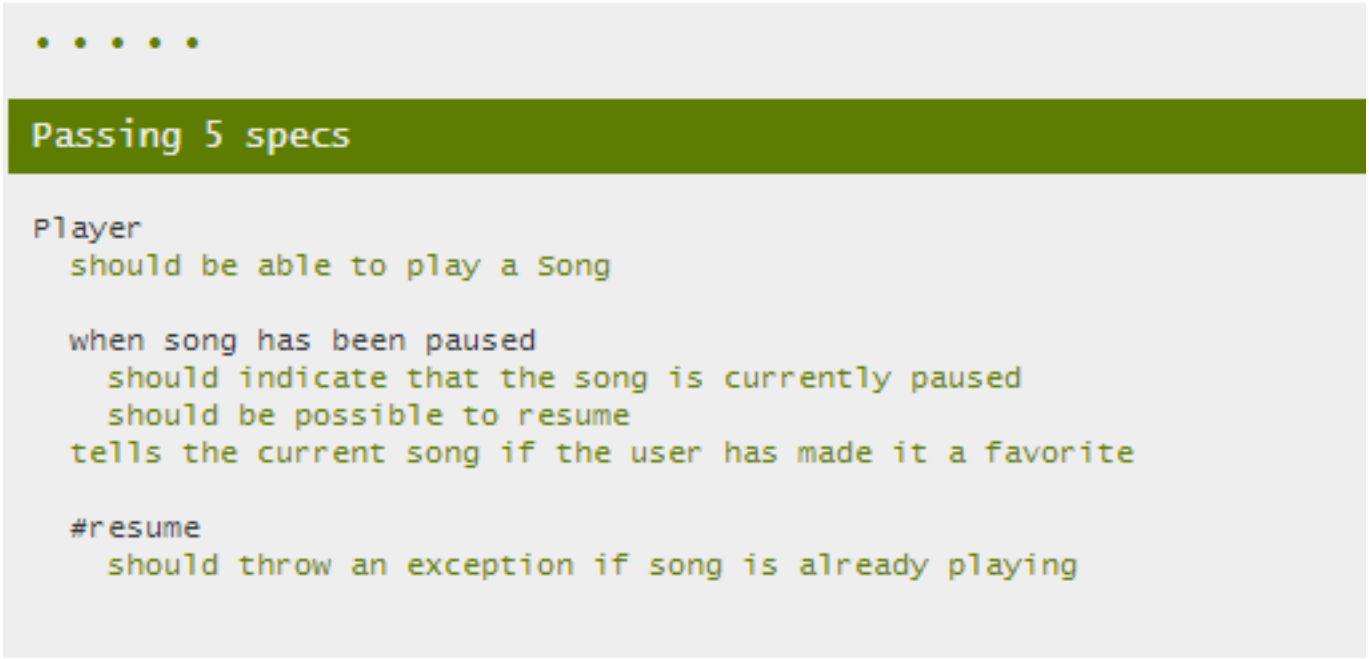
یادآوری :

»دستور describe در فایل بالا برای تعریف تابع تست است. همان طور که می‌بینید بک نام به آن داده می‌شود به همراه بدنه تابع تست.

»دستور beforeEach برای آماده سازی مواردی است که قصد داریم در تست مورد استفاده قرار گیرند. همانند متدهای Setup در UnitTest.

» دستور expect نیز معادل Assert در UnitTest است و برای بررسی صحت عملکرد تست نوشته می‌شود.

اگر فایل SpecRunner.html را دوباره در مرورگر خود باز نمایید تصویر زیر را مشاهده خواهید کرد که به عنوان موفقیت آمیز بودن پیکر بندی پروژه و تست‌های آن می‌باشد.



عنوان: **آشنایی با Should Library**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۸/۱۵ ۱۲:۱۵
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing, TDD, Should Library

نوشتن Assert در کدهای تست، وابستگی مستقیم به انتخاب کتابخانه تست دارد. برای مثال:

:NUnit

```
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;

namespace TestLibrary
{
   [TestFixture]
   public class MyTest
   {
       [Test]
       public void Test1()
       {
           var expectedValue = 2;
           Assert.That(expectedValue , Is.EqualTo(2));
       }
   }
}
```

: Microsoft UnitTesting

```
using Microsoft.VisualStudio.TestTools.UnitTesting ;

namespace TestLibrary
{
   [TesClass]
   public class MyTest
   {
       [TestMethod]
       public void Test1()
       {
           var expectedValue = 2;
           Assert.AreEqual (expectedValue , 2);
       }
   }
}
```

کدهای Assert نوشته شده در مثال بالا با توجه به فریم ورک مورد استفاده متفاوت است. در حالی که کتابخانه Should، مجموعه ای از Extension Method هاست برای قسمت Assert در UnitTestهای نوشته شده. با استفاده از این کتابخانه دیگر نیازی به نوشتن Assert به سبک و سیاق فعلی نیست. کدهای Assert بسیار **خواناتر و قابل درک** خواهند بود و از طرفی وابستگی به سایر کتابخانه‌های تست از بین خواهد رفت.
نکته: مورد استفاده این کتابخانه فقط در قسمت Assert کدهای تست است و استفاده از سایر کتابخانه‌های جانبی الزامی است.
این کتابخانه به دو صورت مورد استفاده قرار می‌گیرد:
»Standard که باید از Should.dll استفاده نمایید؛
»Fluent که باید از Should.Fluent.dll استفاده نمایید؛(پیاده سازی همان فریم ورک Should به صورت Static Reflection)

نصب کتابخانه Should با استفاده از nuget (آخرین نسخه آن در حال حاضر 1.1.20 است) :

```
Install-Package Should
```

نصب کتابخانه Should.Fluent با استفاده از nuget(آخرین نسخه آن در حال حاضر 1.1.19 است):

```
Install-Package ShouldFluent
```

در ابتدا همان مثال قبلی را با این کتابخانه بررسی خواهیم کرد:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TestLibrary
{
   [TesClass]
   public class MyTest
   {
       [TestMethod]
       public void Test1()
       {
           var expectedValue = 2;
           expectedValue.Should().Equal( 2 );
       }
   }
}
```

در نگاه اول چیز خاصی به چشم نمی‌خورد، اما اگر از این پس قصد داشته باشیم کدهای تست خود را تحت فریم ورک NUnit پیاده سازی کنیم در قسمت Assert کدهای خود هیچ گونه خطایی را مشاهده نخواهیم کرد.

مثال:

```
[TestMethod]
public void AccountConstructorTest()
{
     const int expectedBalance = 1000;
     Account bankAccount = new Account();

     // Assert.IsNotNull(bankAccount, "Account was null.");
    // Assert.AreEqual(expectedBalance, bankAccount.AccountBalance, "Account balance not mathcing");



    bankAccount.ShouldNotBeNull("Account was null");
    bankAccount.AccountBalance.ShouldEqual(expectedBalance, "Account balance not matching");
}
```

در مثال بالا ابتدا با استفاده از Ms UnitTesting دو Assert نوشته شده است سپس در خطوط بعدی همان دو شرط با استفاده از کتابخانه Should نوشتم. در ذیل چند مثال از استفاده این کتابخانه (البته نوع Fluent آن) در هنگام کار با رشته ها، آبجکت ها، boolean و Collectionها را بررسی خواهیم کرد:

Should.Fluent#

```
public void Should_fluent_assertions()
{
    object obj = null;
    obj.Should().Be.Null();

    obj = new object();
    obj.Should().Be.OfType(typeof(object));
    obj.Should().Equal(obj);
    obj.Should().Not.Be.Null();
    obj.Should().Not.Be.SameAs(new object());
    obj.Should().Not.Be.OfType<string>();
    obj.Should().Not.Equal("foo");

    obj = "x";
    obj.Should().Not.Be.InRange("y", "z");
    obj.Should().Be.InRange("a", "z");
    obj.Should().Be.SameAs("x");

    "This String".Should().Contain("This");
    "This String".Should().Not.Be.Empty();
    "This String".Should().Not.Contain("foobar");

    false.Should().Be.False();
    true.Should().Be.True();
```

```
    var list = new List<object>();
    list.Should().Count.Zero();
    list.Should().Not.Contain.Item(new object());

    var item = new object();
    list.Add(item);
    list.Should().Not.Be.Empty();
    list.Should().Contain.Item(item);
};
```

#مثال‌های استفاده از متغیرهای DateTime و Guid

```
public void Should_fluent_assertions()
{
    var id = new Guid();
    id.Should().Be.Empty();

    id = Guid.NewGuid();
    id.Should().Not.Be.Empty();

    var date = DateTime.Now;
    date1.Should().Be.Today();

    var str = "";
    str.Should().Be.NullOrEmpty();

    var one = "1";
    one.Should().Be.ConvertableTo<int>();

    var idString = Guid.NewGuid().ToString();
    idString.Should().Be.ConvertableTo<Guid>();
}
```