عنوان: **آشنایی با آزمایش واحد (unit testing) در دات نت، قسمت 1**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۷/۱۰/۰۷ ۱۳:۰۸:۳۱
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

آزمایش واحد چیست؟

آزمایش واحد (unit testing) هنر و تمرین بررسی صحت عملکرد قطعه‌ای از کد (که در اینجا واحد نامیده شده است)، به وسیله کدهای دیگری است که توسط برنامه نویس نوشته خواهند شد. عموما این آزمایش‌ها جهت بررسی یک متد تهیه می‌شوند. در این مرحله باید درنظر داشت که هدف، بررسی کارآیی نرم افزار نیست. هدف این است که بررسی کنیم آیا قطعه کد جدیدی که به برنامه اضافه شده است درست کار می‌کند و آیا هدف اصلی از توسعه آن‌را برآورده می‌سازد؟
برای مثال متدی را توسعه داده‌اید که آدرس یک دومین را از آدرس اینترنتی دریافت شده، جدا می‌سازد. با استفاده از آزمایشات واحد متعدد می‌توان از صحت عملکرد آن اطمینان حاصل کرد.

اهمیت و مزایای آزمایش واحد کدامند؟

کامپایل شدن کد به معنای صحت عملکرد آن نیست. حتما نیاز به روش‌هایی برای آزمایش سیستم وجود دارد. صرفا به شما حقوق داده نمی‌شود که کد بنویسید. به شما حقوق داده می‌شود که کد قابل اجرایی را تهیه کنید.
نوشتن آزمایش‌های واحد به تولید کدهایی با کیفیت بالا در دراز مدت منجر خواهد شد. برای نمونه فرض کنید سیستمی را توسعه داده‌اید. امروز کارفرما از شما خواسته است که قابلیت جدیدی را به برنامه اضافه کنید. برای اعمال این تغییرات برای مثال نیاز است تا قسمتی از کدهای موجود تغییر کند، همچنین کلاس‌ها و متدهای جدیدی نیز به برنامه افزوده گردند. پس از انجام درخواست رسیده، چگونه می‌توانید اطمینان حاصل کنید که قسمت‌های پیشین سیستم که تا همین چند لحظه پیش کار می‌کردند، اکنون نیز همانند قبل کار می‌کنند؟ حجم کدهای نوشته شده بالا است. آزمایش دستی تک تک موارد شاید دیگر از لحاظ زمانی مقدور نباشد. آزمایش واحد روشی است برای اطمینان حاصل کردن از اینکه هنگام تحویل کار به کارفرما مرتبا سرخ و سفید نشویم! به این صورت عملیات refactoring کدهای موجود بدون ترس و لرز انجام خواهد شد، چون بلافاصله می‌توانیم آزمایشات قبلی را اجرا کرده و از صحت عملکرد سیستم اطمینان حاصل نمائیم. بدون اینکه در زمان تحویل برنامه در هنگام بروز خطا بگوئیم : "این غیرممکنه!"
روال‌های آزمایشات صورت گرفته در آینده تبدیل به مرجع مهمی جهت درک چگونگی عملکرد قسمت‌های مختلف سیستم خواهند شد. چگونه فراخوانی شده‌اند، چگونه باید به آن‌ها مقداری را ارجاع داد و امثال آن.
با استفاده از آزمایش‌های واحد، بدترین حالات ممکن را قبل از وقوع می‌توان در نظر گرفت و بررسی کرد.
نوشتن آزمایش‌های واحد در حین کار، برنامه نویس را وادار می‌کند که کار خود را به واحدهای کوچکتری که قابلیت بررسی مستقلی دارند، بشکند. برای مثال فرض کنید متدی را توسعه داده‌اید که پس از انجام سه عملیات مختلف بر روی یک رشته، خروجی خاصی را ارائه می‌دهد. هنگام آزمایش این متد چگونه می‌توان اطمینان حاصل کرد که کدام قسمت سبب شکست آزمایش شده است؟ به همین جهت برنامه نویس جهت ساده‌تر کردن آزمایشات، مجبور خواهد شد که کد خود را به قسمت‌های مستقل کوچکتری تقسیم کند.
با توجه به امکان اجرای خودکار این آزمایشات، به عنوان جزئی ایده‌آل از پروسه تولید نرم افزار محسوب می‌شوند.

حد و مرز یک آزمایش واحد کجاست؟

آزمایش شما، آزمایش واحد نامیده نخواهد شد اگر:
با دیتابیس سر و کار داشته باشد.
با شبکه در ارتباط باشد.
با فایل‌ها کار کند.
نیاز به تمهیدات ویژه‌ای برای اجرای آن وجود داشته باشد. مثلا وجود یک فایل config برای اجرای آن ضروری باشد.

همراه و همزمان با سایر کدهای آزمایش‌های واحد شما قابل اجرا نباشد.
برای مثال اگر یکی از متدهای شما بزرگترین عدد یک لیست را از دیتابیس دریافت می‌کند، در متدی که برای آزمایش واحد آن تهیه خواهید کرد نباید هیچگونه کدی جهت برقراری ارتباط با دیتابیس نوشته شود.
این امر سبب سریع‌تر اجرا شدن آزمایشات واحد خواهند شد و در آینده شما را از انجام آن به‌دلیل کند بودن روند انجام آزمایشات، منصرف نخواهد کرد. همچنین تغییرات انجام شده در لایه دسترسی به داده‌ها سبب غیرمعتبر شدن این نوع آزمایشات نخواهند شد. به بیان دیگر وظیفه متد آزمایش واحد، اتصال به دیتابیس یا شبکه و یا خواندن اطلاعات از یک فایل نیست.

ادامه دارد...

نظرات خوانندگان

نویسنده: بهروز راد
تاریخ: ۱۳۸۷/۱۰/۰۷ ۲۲:۱۶:۰۰

وحید جان اینو البته همه میدونیم که از Unit Testing و نرم افزارهای مرتبط با اون مثل MBUnit به ندرت استفاده میشه. این مورد مختص به ایران نیست. در بسیاری از وبلاگ ها و فروم های خارجی هم به این نکته اشاره شده. کسی حال و حوصله ی Unit Testing رو نداره. این یک واقعیته :)

نویسنده: وحید نصیری
تاریخ: ۱۳۸۷/۱۰/۰۸ ۰۱:۰۹:۰۰

نمی‌دونم! ولی در کل پروژه‌های خوب جدیدی رو که در codeplex می‌بینم اکثرشون unit testing رو دارند.
یا یک سری ویدیوی nhibernate معرفی کرده بودم، حتما وقت کردی ببین. تمام جلساتش پر از unit testing است.

نویسنده: Anonymous
تاریخ: ۱۳۸۷/۱۰/۰۸ ۰۹:۰۴:۰۰

ضمن تشکر از وحید خان بخاطر شروع کردن این بحث .

@ بهروز:
حداقل شرکتی که من کارمندش هستم اینطور نیست.
اصلا یونیت تستینگ رو اینجاانجام نمیدهند.یک بخشی از شرکت داخل هند هست که کل سافت ور تستینگ پروژه های شرکت (که همگی مربوط به نرم افزارهای پزشکی هستش) رو اون بخش در هند انجام میده.

عنوان: **آشنایی با آزمایش واحد (unit testing) در دات نت، قسمت 2**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۷/۱۰/۰۸ ۱۳:۴۶:۴۵
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

دلایل شانه خالی کردن از آزمایش واحد!

-1 نوشتن آزمایشات زمان زیادی را به خود اختصاص خواهند داد.

مهمترین دلیلی که برنامه‌نویس‌ها به سبب آن از نوشتن آزمایشات واحد امتناع می‌کنند، همین موضوع است. اکثر افراد به آزمایش به‌عنوان مرحله آخر توسعه فکر می‌کنند. اگر این چنین است، بله! نوشتن آزمایش‌های واحد واقعا سخت و زمانگیر خواهند بود. به همین جهت برای جلوگیری از این مساله روش pay-as-you-go مطرح شده است (ماخذ: کتاب Pragmatic Unit Testing در سی شارپ). یعنی با اضافه شدن هر واحد کوچکی به سیستم، آزمایش واحد آن‌را نیز تهیه کنید. به این صورت در طول توسعه سیستم با باگ‌های کمتری نیز برخورد خواهید داشت چون اجزای آن‌را در این حین به تفصیل مورد بررسی قرار داده‌اید. اثر این روش را در شکل زیر می‌توانید ملاحظه نمائید (تصویری از همان کتاب ذکر شده)

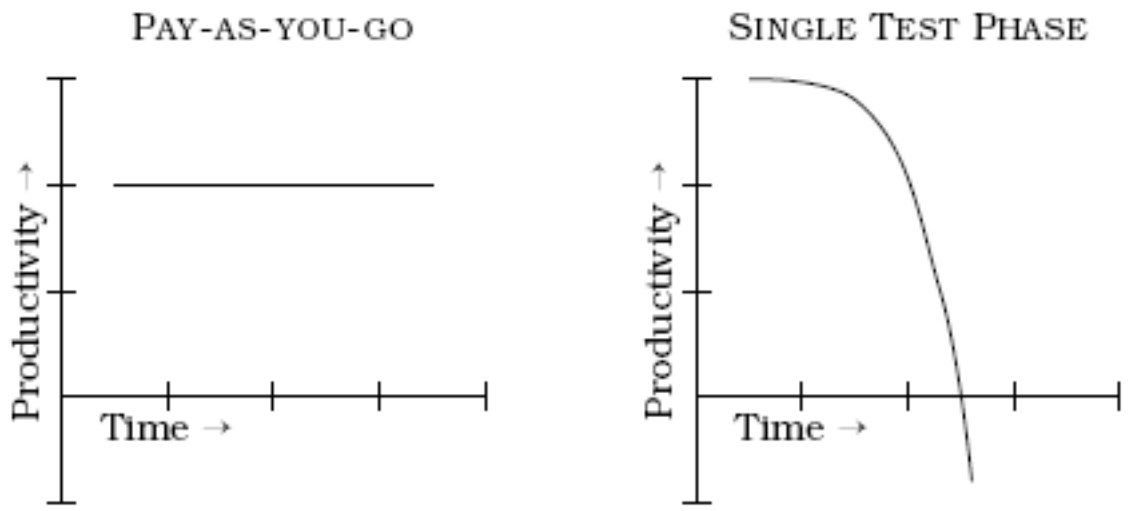


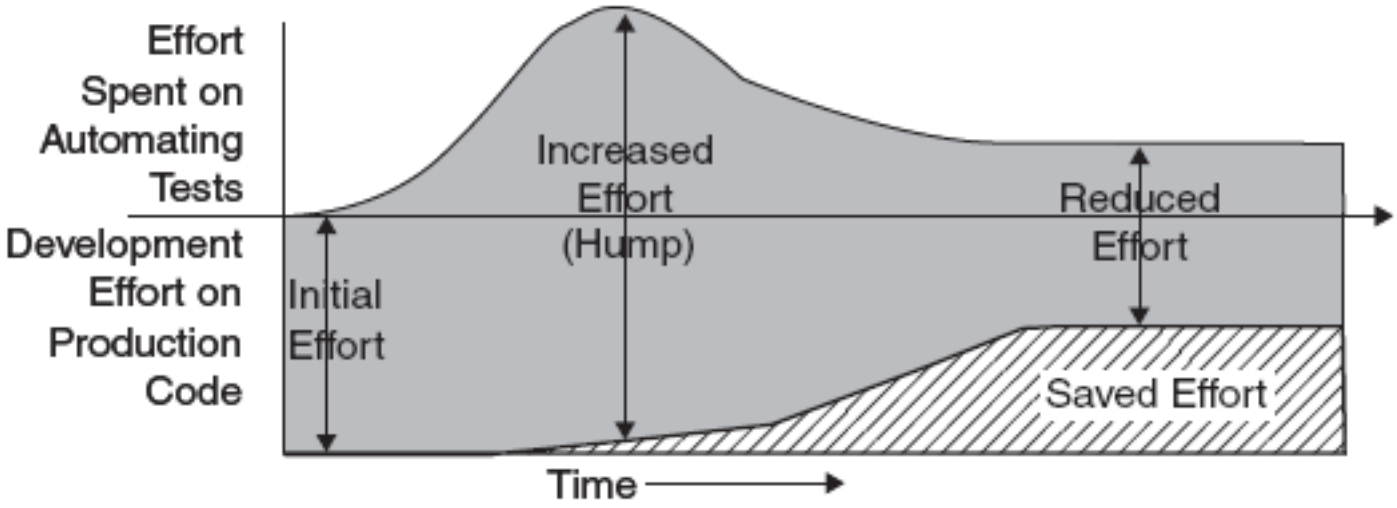
Figure 1.1: Comparison of Paying-as-you-go vs. Having a Single Testing Phase

نوشتن آزمایشات واحد زمانبر هستند اما توسعه پیوسته آن‌ها با به تاخیر انداختن آزمایشات به انتهای پروژه، همانند تصویر فوق تاثیر بسیار قابل توجهی در بهره وری شما خواهند داشت.

بنابراین اگر عنوان می‌کنید که وقت ندارید آزمایش واحد بنویسید، به چند سؤال زیر پاسخ دهید:
الف) چه مقدار زمان را صرف دیباگ کردن کدهای خود یا دیگران می‌کنید؟

ب) چه میزان زمان را صرف بازنویسی کدی کرده‌اید که تصور می‌رفت درست کار می‌کند اما اکنون بسیار مشکل زا ظاهر شده است؟
ج) چه مقدار زمان را صرف این کرده‌اید که منشاء باگ گزارش شده در برنامه را کشف کنید؟

برای افرادی که آزمایشات واحد را در حین پروسه توسعه در نظر نمی‌گیرند، این مقادیر بالا است و با ازدیاد تعداد خطوط سورس کدها، این ارقام سیر صعودی خواهند داشت.



تصویری از کتاب xUnit Test Patterns ، که بیانگر کاهش زمان و هزینه کد نویسی در طول زمان با رعایت اصول آزمایشات واحد است

-2 اجرای آزمایشات واحد زمان زیادی را تلف می‌کند.

نباید اینطور باشد. عموما اجرای هزاران آزمایش واحد، باید در کسری از ثانیه صورت گیرد. (برای اطلاعات بیشتر به قسمت حد و مرز یک آزمایش واحد در قسمت قبل مراجعه نمائید)

-3 امکان تهیه آزمایشات واحد برای کدهای قدیمی (legacy code) من وجود ندارد

برای بسیاری از برنامه نویس‌ها، تهیه آزمایش واحد برای کدهای قدیمی بسیار مشکل است زیرا شکستن آن‌ها به واحدهای کوچکتر قابل آزمایش بسیار خطرناک و پرهزینه است و ممکن است سبب از کار افتادن سیستم آن‌ها گردد. اینجا مشکل از آزمایش واحد نیست. مشکل از ضعف برنامه نویسی آن سیستم است. روش refactoring ، طراحی مجدد و نوشتن آزمایشات واحد، به تدریج سبب طراحی بهتر برنامه از دیدگاه‌های شیءگرایی شده و نگهداری سیستم را در طولانی مدت ساده‌تر می‌سازد. آزمایشات واحد این نوع سیستم‌ها را از حالت فلج بودن خارج می‌سازد.

-4 کار من نیست که کدهای نوشته شده را آزمایش کنم!

باید درنظر داشته باشید که این هم کار شما نیست که انبوهی از کدهای مشکل دار را به واحد بررسی کننده آن تحویل دهید! همچنین اگر تیم آزمایشات و کنترل کیفیت به این نتیجه برسد که عموما از کدهای شما کمتر می‌توان باگ گرفت، این امر سبب معروفیت و تضمین شغلی شما خواهد شد.

همچنین این کار شما است که تضمین کنید واحد تهیه شده مقصود مورد نظر را ارائه می‌دهد و این‌کار را با ارائه یک یا چندین آزمایش واحد می‌توان اثبات کرد.

-5 تنها قسمتی از سیستم به من واگذار شده است و من دقیقا نمی‌دانم که رفتار کلی آن چیست. بنابراین آن را نمی‌توانم آزمایش کنم!

اگر واقعا نمی‌دانید که این کد قرار است چه کاری را انجام دهید به طور قطع الان زمان مناسبی برای کد نویسی آن نیست!

-6 کد من کامپایل می‌شود!

باید دقت داشت که کامپایلر فقط syntax کدهای شما را بررسی کرده و خطاهای آن‌را گوشزد می‌کند و نه نحوه‌ی عملکرد آن‌را.

-7 من برای نوشتن آزمایشات حقوق نمی‌گیرم!

باید اذعان داشت که به شما جهت صرف تمام وقت یک روز خود برای دیباگ کردن یک خطا هم حقوق نمی‌دهند! شما برای تهیه یک کد قابل قبول و قابل اجرا حقوق می‌گیرید و آزمایش واحد نیز ابزاری است جهت نیل به این مقصود (همانند یک IDE و یا یک کامپایلر).

-8 احساس گناه خواهم کرد اگر تیم فنی کنترل کیفیت و آزمایشات را از کار بی کار کنم!!

نگران نباشید، این اتفاق نخواهد افتاد! بحث ما در اینجا آزمایش کوچکترین اجزا و واحدهای یک سیستم است. موارد دیگری مانند functional testing, acceptance testing, performance & environmental testing, validation & verification, formal analysis توسط تیم‌های کنترل کیفیت و آزمایشات هنوز باید بررسی شوند.

-9 شرکت من اجازه اجرای آزمایشات واحد را بر روی سیستم‌های در حال اجرا نمی‌دهد.

قرار هم نیست بدهد! چون دیگر نام آن آزمایش واحد نخواهد بود. این آزمایشات باید بر روی سیستم شما و توسط ابزار و امکانات شما صورت گیرد.

پ.ن.
در هشتمین دلیل ذکر شده، از acceptance testing نامبرده شده. تفاوت آن با unit testing به صورت زیر است:

آزمایش واحد:
توسط برنامه نویس‌ها تعریف می‌شود
سبب اطمینان خاطر برنامه نویس‌ها خواهد شد
واحدهای کوچک سیستم را مورد بررسی قرار می‌دهد
یک آزمایش سطح پائین (low level) به شمار می‌رود
بسیار سریع اجرا می‌شود
به صورت خودکار (100 درصد خودکار است) و با برنامه نویسی قابل کنترل است

اما در مقابل آزمایش پذیرش به صورت زیر است:
توسط مصرف کنندگان تعریف می‌شود
سبب اطمینان خاطر مصرف کنندگان می‌شود.
کل برنامه مورد آزمایش قرار می‌گیرد
یک آزمایش سطح بالا (high level) به شمار می‌رود
ممکن است طولانی باشد

عموما به صورت دستی یا توسط یک سری اسکریپت اجرا می‌شود
مثال : گزارش ماهیانه باید جمع صحیحی از تمام صفحات را در آخرین برگه گزارش به همراه داشته باشد

ادامه دارد...

نظرات خوانندگان

نویسنده: Anonymous
تاریخ: ۱۳۸۷/۱۰/۰۹ ۰۷:۴۸:۰۰

سلام از این که هر روز وبلاگت رو اپدیت می کنی متشکر از این بحث هم خیلی خوشم می اد ادامه بده

نویسنده: Alex's Blog
تاریخ: ۱۳۸۸/۰۱/۲۵ ۱۵:۴۹:۰۰

سلام آقای نصیری
یه خواهش ازاتون داشتم اونم این بود که من این روزا مجبورم یه روالی رو برای performance & environmental testing تهیه کنم. بخاطر همین میخواستم ازتون خواهش بکنم اگه منابعی در این مورد دارین یا نرم افزارهایی برای این تستها سراغ دارین بهم معرفی کنید. (من بیشتر روی performance سیستمها می خوام کار کنم)
ممنون.

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۱/۲۵ ۱۶:۲۳:۰۰

سلام
در مورد تست کارآیی مشخص نکردید که چه پلتفرمی مد نظر شما است. اگر دات نت مد نظر است، نرم افزار شرکت red gate در این زمینه حرف اول را می‌زند:
www.red-gate.com/Products/ants_profiler/index.htm
شرکت سازنده resharper هم یک محصول دیگر در این مورد دارد:
/www.jetbrains.com/profiler

در مورد سایر پلتفرم‌ها هم کمابیش هست. profiler و code profiling را جستجو کنید.

نویسنده: Alex's Blog
تاریخ: ۱۳۸۸/۰۱/۲۶ ۰۸:۴۷:۰۰

ممنون از لطفتون
منظورم پلت فرم دات نت بود.
بازم ممنون.

عنوان: **آشنایی با آزمایش واحد (unit testing) در دات نت، قسمت 3**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۷/۱۰/۰۹ ۱۵:۱۹:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

آشنایی با NUnit

NUnit یکی از فریم ورک‌های آزمایش واحد سورس باز مخصوص دات نت فریم ورک است. (کلا در دات نت هرجایی دیدید که N ، به ابتدای برنامه‌ای یا کتابخانه‌ای اضافه شده یعنی نمونه منتقل شده از محیط جاوا به دات نت است. برای مثال NHibernate از Hibernate جاوا گرفته شده است و الی آخر)
این برنامه با سی شارپ نوشته شده است اما تمامی زبان‌های دات نتی را پشتیبانی می‌کند (اساسا با زبان نوشته شده کاری ندارد و فایل اسمبلی برنامه را آنالیز می‌کند. بنابراین فرقی نمی‌کند که در اینجا چه زبانی بکار گرفته شده است).

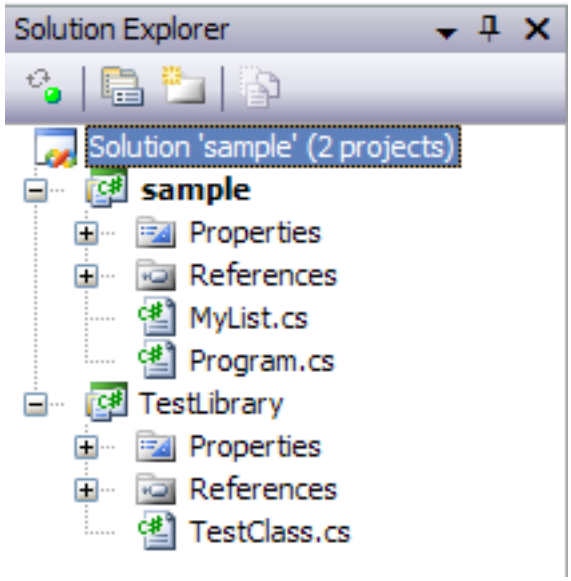
ابتدا NUnit را دریافت نمائید:
http://nunit.org/index.php?p=download

یک برنامه ساده از نوع console را در VS.net آغاز کنید.
کلاس MyList را با محتوای زیر به پروژه اضافه کنید:

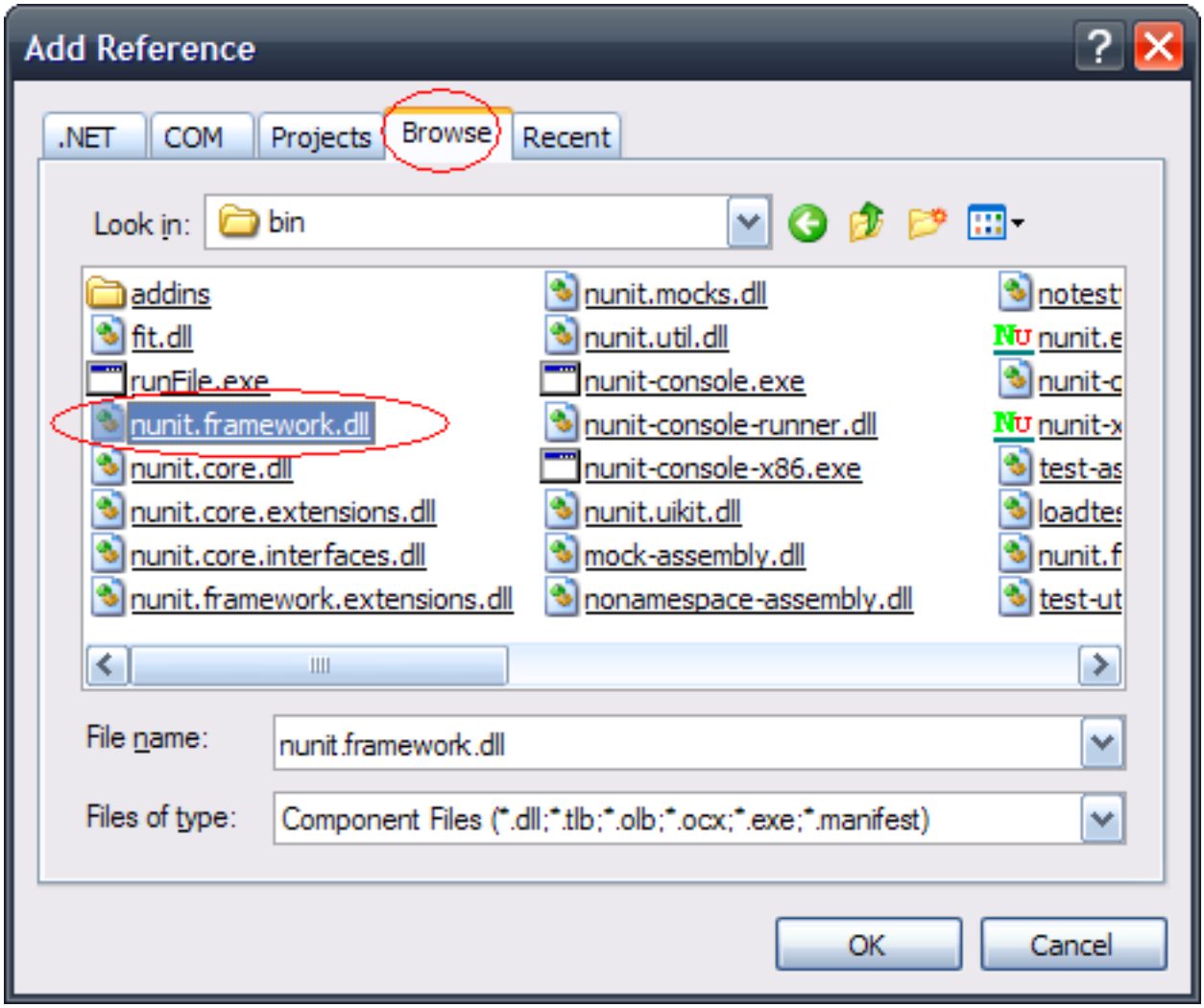
```
using System.Collections.Generic;

namespace sample
{
   public class MyList
   {
       public static List<int> GetListOfIntItems(int numberOfItems)
       {
           List<int> res = new List<int>();

           for (int i = 0; i < numberOfItems; i++)
               res.Add(i);

           return res;
       }
   }

}
```

یکبار پروژه را کامپایل کنید.

اکنون بر روی نام پروژه در قسمت solution explorer کلیک راست کرده و گزینه add->new project را انتخاب کنید. نوع این پروژه را که متدهای آزمایش واحد ما را تشکیل خواهد داد، class library انتخاب کنید. با نام مثلا TestLibrary (شکل زیر).



با توجه به اینکه NUnit ، اسمبلی برنامه (فایل exe یا dll آن‌را) آنالیز می‌کند، بنابراین می‌توان پروژه تست را جدای از پروژه اصلی ایجاد نمود و مورد استفاده قرار داد.
پس از ایجاد پروژه class library ، باید ارجاعی از NUnit framework را به آن اضافه کنیم. به محل نصب NUnit مراجعه کرده (پوشه bin آن) و ارجاعی به فایل nunit.framework.dll را به پروژه اضافه نمائید (شکل زیر).



سپس فضاهای نام مربوطه را به کلاس آزمایش واحد خود اضافه خواهیم کرد:

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

اولین نکته‌ای را که باید در نظر داشت این است که کلاس آزمایش واحد ما باید Public باشد تا در حین آنالیز اسمبلی پروژه توسط NUint، قابل دسترسی و بررسی باشد.
سپس باید ویژگی جدیدی به نام TestFixture را به این کلاس اضافه کرد.

```
[TestFixture]
   public class TestClass
```

این ویژگی به NUnit می‌گوید که در این کلاس به دنبال متدهای آزمایش واحد بگرد. (در NUnit از attribute ها برای توصیف عملکرد یک متد و همچنین دسترسی runtime به آن‌ها استفاده می‌شود)
سپس هر متدی که به عنوان متد آزمایش واحد نوشته می‌شود، باید دارای ویژگی Test باشد تا توسط NUnit بررسی گردد:

```
[Test]
       public void TestGetListOfIntItems()
```

نکته: متد Test ما باید public و از نوع void باشد و همچنین هیچ پارامتری هم نباید داشته باشد.

اکنون برای اینکه بتوانیم متد GetListOfIntItems برنامه خود را در پروژه دیگری تست کنیم، باید ارجاعی را به اسمبلی آن اضافه کنیم. همانند قبل، از منوی project گزینه add reference ، فایل exe برنامه کنسول خود را انتخاب کرده و ارجاعی از آن‌را به پروژه class library اضافه می‌کنیم. بدیهی است امکان اینکه کلاس تست در همان پروژه هم قرار می‌گرفت وجود داشت و صرفا جهت جداسازی آزمایش از برنامه اصلی این‌کار صورت گرفت.
پس از این مقدمات، اکنون متد آزمایش واحد ساده زیر را در نظر بگیرید:

```
[Test]
       public void TestGetListOfIntItems()
       {
           const int count = 5;
           List<int> items = sample.MyList.GetListOfIntItems(count);
           Assert.That(items.Count,Is.EqualTo(5));
       }
```

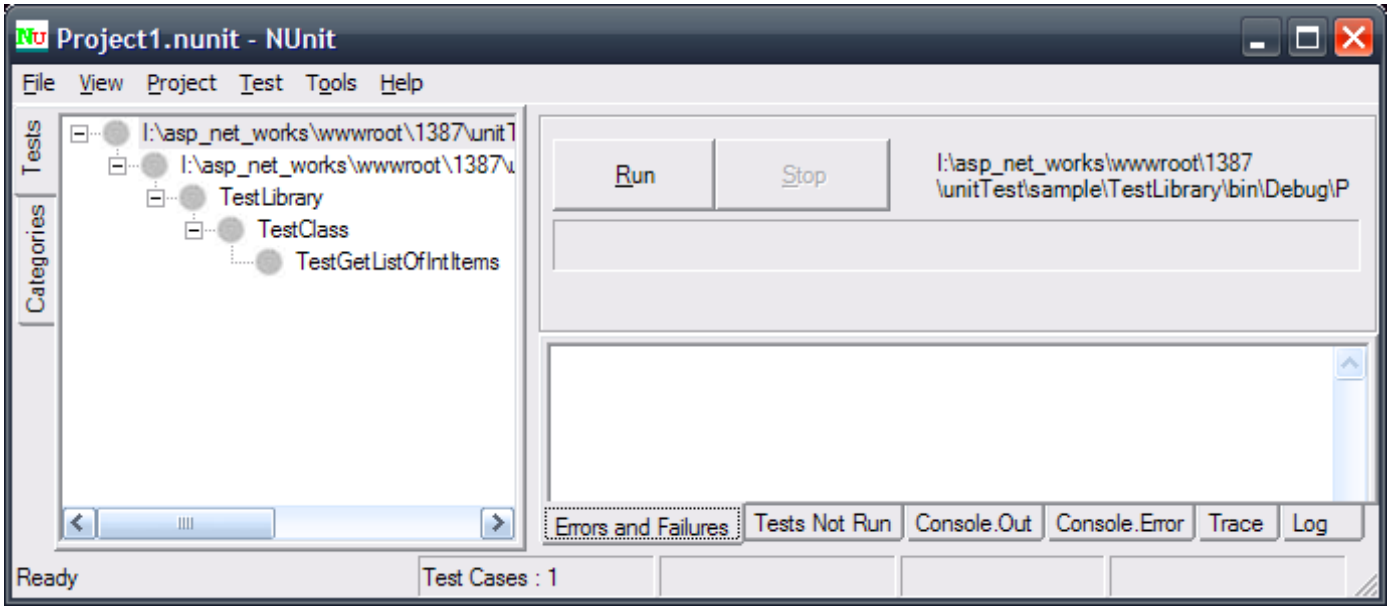
قصد داریم بررسی کنیم آیا متد GetListOfIntItems واقعا همان تعداد آیتمی را که باید برگرداند، بازگشت می‌دهد؟ عدد 5 به آن پاس شده است و در ادامه قصد داریم بررسی کنیم، count شیء حاصل (items در اینجا) آیا واقعا مساوی عدد 5 است؟
اگر آن را (سطر مربوط به Assert را) کلمه به کلمه بخواهیم به فارسی ترجمه کنیم به صورت زیر خواهد بود:
می‌خواهیم اثبات کنیم که count مربوط به شیء items مساوی 5 است.

پس از اضافه کردن متد فوق، پروژه را کامپایل نمائید.

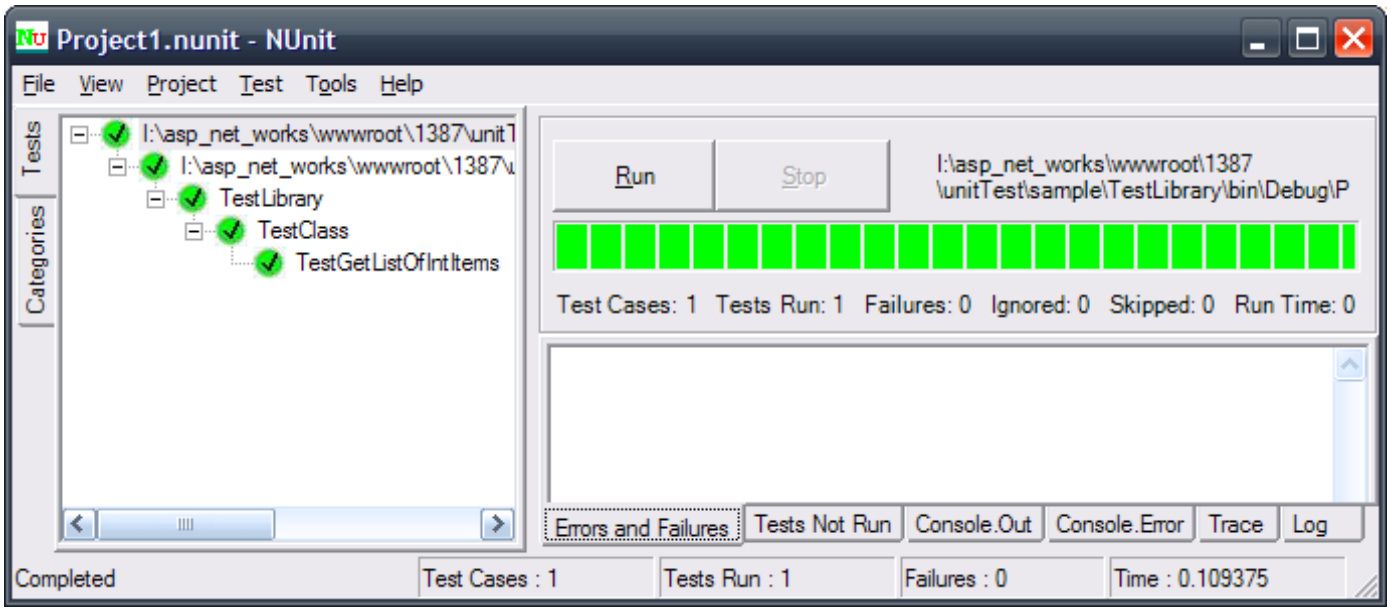
اکنون برنامه nunit.exe را اجرا کنید تا NUnit IDE ظاهر شود (در همان دایرکتوری bin مسیر نصب NUnit قرار دارد).
از منوی File آن یک پروژه جدید را آغاز نموده و آنرا ذخیره کنید.
سپس از منوی project آن، با استفاده از گزینه add assembly ، فایل dll کتابخانه تست خود را اضافه نمائید.
احتمالا پس از انجام این عملیات بلافاصله با خطای زیر مواجه خواهید شد:

```
---------------------------
Assembly Not Loaded
---------------------------
System.ApplicationException : Unable to load TestLibrary because it is not located under
the AppBase
----> System.IO.FileNotFoundException : Could not load file or assembly
'TestLibrary' or one of its dependencies. The system cannot find the file specified.
For further information, use the Exception Details menu item.
```

این خطا به این معنا است که پروژه جدید NUnit باید دقیقا در همان پوشه خروجی پروژه، جایی که فایل dll کتابخانه تست ما تولید شده است، ذخیره گردد. پس از افزودن اسمبلی، نمای برنامه NUnit باید به شکل زیر باشد:



همانطور که ملاحظه می‌کنید، NUnit با استفاده از قابلیت‌های reflection در دات نت، اسمبلی را بارگذاری می‌کند و تمامی کلاس‌هایی که دارای ویژگی TestFixture باشند در آن لیست خواهد شد.
اکنون بر روی دکمه run کلیک کنید تا اولین آزمایش ما انجام شود. (شکل زیر)



رنگ سبز در اینجا به معنای با موفقیت انجام شدن آزمایش است.

ادامه دارد...

نظرات خوانندگان

نویسنده: Anonymous
تاریخ: ۱۳۸۷/۱۱/۲۱ ۰۸:۱۲:۰۰

یک دید کلی نسبت به توابع تستینگ لازم داشتم فوری! که این مطلب به موقع به دادم رسید.
ایوالله.

نویسنده: محمد آزاد
تاریخ: ۱۳۸۸/۰۸/۱۲ ۱۰:۳۳:۴۸

سلام...
آقای نصیری من با
;using NUnit.Framework.SyntaxHelpers
مشکل دارم ...این فضای نام رو ندارم...اما اولیشو دارم..

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۸/۱۲ ۱۲:۵۳:۴۷

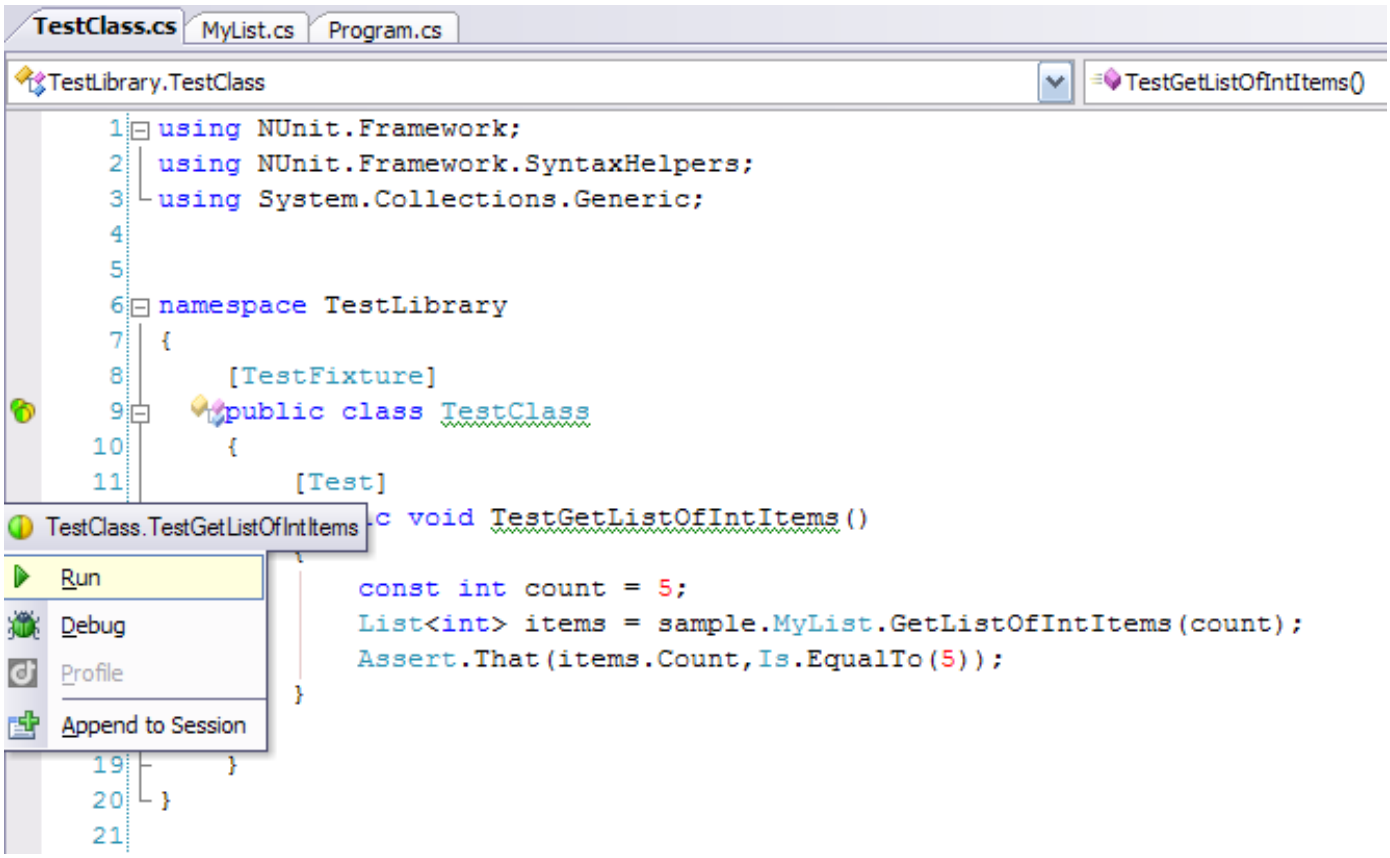
سلام
حق با شما است. مطابق مستندات نگارش آخر آن
.NUnit.Framework.SyntaxHelpers namespace no longer exists
.All classes that were in this namespace have been moved to the NUnit.Framework namespace

به این معنا که SyntaxHelpers الان با همان using NUnit.Framework به پروژه شما الحاق می‌شود (به این فضای نام منتقل شده).

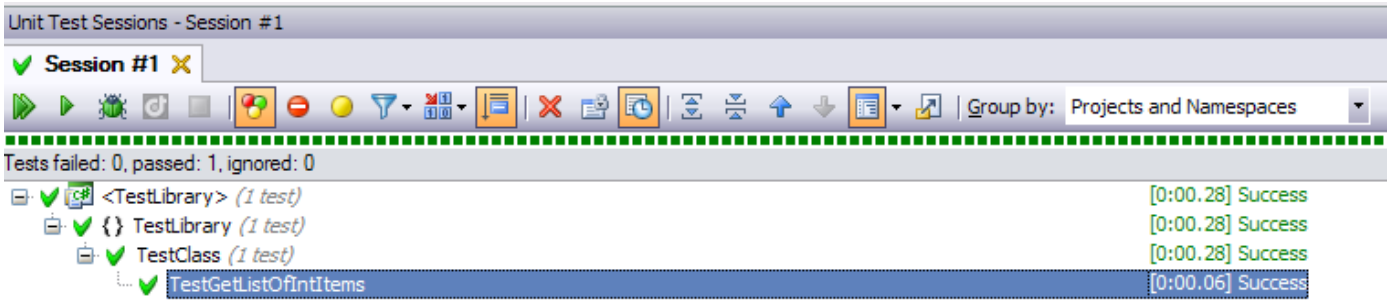
عنوان: **آشنایی با آزمایش واحد (unit testing) در دات نت، قسمت 4**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۷/۱۰/۱۳ ۰۰:۲۳:۲۸
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

ادامه آشنایی با NUnit

اگر قسمت سوم را دنبال کرده باشید احتمالا از تعداد مراحلی که باید در خارج از IDE صورت گیرد گلایه خواهید کرد (کامپایل کن، اجرا کن، اتچ کن، باز کن، ذخیره کن، اجرا کن و ...). خوشبختانه افزونه ReSharper این مراحل را بسیار ساده و مجتمع کرده است. این افزونه به صورت خودکار متدهای آزمایش واحد یک پروژه را تشخیص داده و آنها را با آیکون‌هایی (Gutter icons) متمایز مشخص می‌سازد (شکل زیر). پس از کلیک بر روی آن‌ها ، امکان اجرای آزمایش یا حتی دیباگ کردن سطر به سطر یک متد آزمایش واحد درون IDE ویژوال استودیو وجود خواهد داشت.



برای نمونه پس از اجرای آزمایش واحد قسمت قبل، نتیجه حاصل مانند شکل زیر خواهد بود:



راه دیگر، استفاده از افزونه TestDriven.NET است که نحوه استفاده از آن‌را اینجا می‌توانید ملاحظه نمائید. به منوی جهنده کلیک راست بر روی یک صفحه، گزینه run tests را اضافه می‌کند و نتیجه حاصل را در پنجره output ویژوال استودیو نمایش می‌دهد.

ساختار کلی یک کلاس آزمایش واحد مبتنی بر NUnit framework :

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

namespace TestLibrary
{
   [TestFixture]
   public class Test2
   {
       [SetUp]
       public void MyInit()
       {
           //کدی که در این قسمت قرار می‌گیرد پیش از اجرای هر متد تستی اجرا خواهد شد
       }

       [TearDown]
       public void MyClean()
       {
           //کدی که در این قسمت قرار می‌گیرد پس از اجرای هر متد تستی اجرا خواهد شد
       }

       [TestFixtureSetUp]
       public void MyTestFixtureSetUp()
       {
           // کدی که در اینجا قرار می‌گیرد در ابتدای بررسی آزمایش واحد و فقط یکبار اجرا می‌شود
       }

       [TestFixtureTearDown]
       public void MyTestFixtureTearDown()
       {
           // کدهای این قسمت در پایان کار یک کلاس آزمایش واحد اجرا خواهند شد
       }

       [Test]
       public void Test1()
       {
           //بدنه آزمایش واحد در اینجا قرار می‌گیرد
           Assert.That(2, Is.EqualTo(2));
       }
   }
}
```

شبیه به روال‌های رخداد گردان load و close یک فرم، یک کلاس آزمایش واحد NUnit نیز دارای ویژگی‌های TestFixtureSetUp و TestFixtureTearDown است که در ابتدا و انتهای آزمایش واحد اجرا خواهند شد (برای درک بهتر موضوع و دنبال کردن نحوه‌ی اجرای این روال‌ها، داخل این توابع break point قرار دهید و با استفاده از ReSharper ، آزمایش را در حالت دیباگ آغاز کنید)، یا SetUp و TearDown که در زمان آغاز و پایان بررسی هر متد آزمایش واحدی فراخوانی می‌شوند.
همانطور که در قسمت قبل نیز ذکر شد، به امضاهای متدها و کلاس فوق دقت نمائید (عمومی ، void و بدون آرگومان ورودی).

بهتر است از ویژگی‌های SetUp و TearDown با دقت استفاده نمود. عموما هدف از این روال‌ها ایجاد یک شیء و تخریب و پاک سازی آن است. حال اینکه این روال‌ها قبل و پس از اجرای هر متد آزمایش واحدی فراخوانی می‌شوند. بنابراین به این موضوع دقت داشته باشید.

همچنین توصیه می‌شود که کلاس‌های آزمایش واحد را در اسمبلی دیگری مجزا از پروژه اصلی پیاده سازی کنید (برای مثال یک پروژه جدید از نوع class library)، زیرا این موارد مرتبط با بررسی کیفیت کدهای شما هستند که موضوع جداگانه‌ای نسبت به پروژه اصلی محسوب می‌گردد (نحوه پیاده سازی آن‌را در قسمت قبل ملاحظه نمودید). همچنین در یک پروژه تیمی این جدا سازی، مدیریت آزمایشات را ساده‌تر می‌سازد و بعلاوه سبب حجیم شدن بی‌مورد اسمبلی‌های اصلی محصول شما نیز نمی‌گردند.

ادامه دارد...

نظرات خوانندگان

نویسنده: افشار محبی
تاریخ: ۱۳۸۸/۰۹/۲۳ ۱۰:۱۴:۲۹

ای کاش می‌شد در TestFixtureSetup به دیتابیس متصل شد. حتی اگر اجرای تست کند شود مهم نیست چون بررسی درستی بعضی عملیات مرتبط با دیتابیس خیلی مهم‌تر از زمان اجرای تست است.

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۹/۲۳ ۱۲:۱۴:۳۶

- در NHibernate برای این نوع تست‌ها تا جایی که دیدم از دیتابیس SQLite تشکیل شده در حافظه استفاده می‌کنند. به این صورت مزایای سرعت و همچنین حذف خودکار داده‌ها پس از پایان کار برقرار است.
- ضمناآزمایش واحدی که از مرزهای برنامه خارج شود دیگر آزمایش واحد نام ندارد به همین جهت mocking frameworks برای این نوع کارها ایجاد شده است. (برای کار با دیتابیس، کار با smtp server ، کار با فایل سیستم و مواردی از این دست)

نویسنده: افشار محبی
تاریخ: ۱۳۸۸/۰۹/۲۳ ۱۴:۴۶:۵۶

آره، mocking framework و ابزارهای تست دیتابیس کارهای جالب و قشنگی می‌کنند. من هم فهمیدم آن چیزی که بهش نیاز دارم همان integration test است نه unit test.
در NUnit همه کاری می‌شود انجام داد حتی اتصال به دیتابیس (البته اسمش می‌شود integration test) و راهش هم اضافه کردن app.config یا web.config به همان پروژه class library مخصوص تست است. راه این کار هم در خیلی جاها گفته شده ولی اگر مثل من در خواندن و استفاده از app.config در برنامه دچار مشکل شدید به لینک زیر مراجعه کنید:
http://david.givoni.com/blog/?p=4

عنوان: **آشنایی با آزمایش واحد (unit testing) در دات نت، قسمت 5**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۷/۱۰/۱۸ ۱۳:۱۰:۳۸
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

ادامه آشنایی با NUnit

حالت‌های مختلف Assert :

NUnit framework حالت‌های مختلفی از دستور Assert را پشتیبانی می‌کند که در ادامه با آنها آشنا خواهیم شد.

کلاس Assertion :
این کلاس دارای متدهای زیر است:

```
public static void Assert(bool condition)
public static void Assert(string message, bool condition)
```

تنها در حالتی این بررسی موفقیت آمیز گزارش خواهد شد که condition مساوی true باشد

```
public static void AssertEquals(string message, object expected, object actual)
public static void AssertEquals(string message, float expected, float actual, float delta)
public static void AssertEquals(string message, double expected, double actual, double delta)
public static void AssertEquals(string message, int expected, int actual)
public static void AssertEquals(int expected, int actual)
public static void AssertEquals(object expected, object actual)
public static void AssertEquals(float expected, float actual, float delta)
public static void AssertEquals(double expected, double actual, double delta)
```

تنها در صورتی این بررسی به اثبات خواهد رسید که اشیاء actual و expected یکسان باشند. (دلتا در اینجا به عنوان تلرانس آزمایش درنظر گرفته می‌شود)

```
public static void AssertNotNull(string message, object anObject)
public static void AssertNotNull(object anObject)
```

این بررسی تنها در صورتی موفقیت آمیز گزارش می‌شود که شیء مورد نظر نال نباشد.

```
public static void AssertNull(string message, object anObject)
public static void AssertNull(object anObject)
```

این بررسی تنها در صورتی موفقیت آمیز گزارش می‌شود که شیء مورد نظر نال باشد.

```
public static void AssertSame(string message, object expected, object actual)
public static void AssertSame(object expected, object actual)
```

تنها در صورتی این بررسی به اثبات خواهد رسید که اشیاء actual و expected یکسان باشند.

```
public static void Fail(string message)
public static void Fail()
```

همواره Fail خواهد شد. (در مورد کاربرد آن در قسمت بعد توضیح داده خواهد شد)

نکته:
در یک متد آزمایش واحد شما مجازید به هرتعدادی که لازم است از متدهای Assertion استفاده نمائید. در این حالت اگر تنها یکی از متدهای assertion با شکست روبرو شود، کل متد آزمایش واحد شما مردود گزارش شده و همچنین عبارات بعدی Assertion بررسی نخواهند شد. بنابراین توصیه می‌شود به ازای هر متد آزمایش واحد، تنها از یک Assertion استفاده نمائید.

مهم!
کلاس Assertion منسوخ شده است و توصیه می‌شود بجای آن از کلاس Assert استفاده گردد.

آشنایی با کلاس Assert :
این کلاس از متدهای زیر تشکیل شده است:

الف) بررسی حالت‌های تساوی

```
Assert.AreEqual( object expected, object actual );
```

جهت بررسی تساوی دو شیء مورد بررسی و شیء مورد انتظار بکار می‌رود.

```
Assert.AreNotEqual( object expected, object actual );
```

جهت بررسی عدم تساوی دو شیء مورد بررسی و شیء مورد انتظار بکار می‌رود.
برای مشاهده انواع و اقسام overload های آن‌ها می‌توانید به راهنمای NUnit که پس از نصب، در پوشه doc آن قرار می‌گیرد مراجعه نمائید.

همچنین دو متد زیر و انواع overload های آن‌ها جهت برسی اختصاصی حالت تساوی دو شیء بکار می‌روند:

```
Assert.AreSame( object expected, object actual );
Assert.AreNotSame( object expected, object actual );
```

بعلاوه اگر نیاز بود بررسی کنیم که آیا شیء مورد نظر حاوی یک آرایه یا لیست بخصوصی است می‌توان از متد زیر و oveload های آن استفاده نمود:

```
Assert.Contains( object anObject, IList collection );
```

ب) بررسی حالت‌های شرطی:

```
Assert.IsTrue( bool condition );
```

تنها در حالتی این بررسی موفقیت آمیز گزارش خواهد شد که condition مساوی true باشد

```
Assert.IsFalse( bool condition);
```

تنها در حالتی این بررسی موفقیت آمیز گزارش خواهد شد که condition مساوی false باشد

```
Assert.IsNull( object anObject );
```

این بررسی تنها در صورتی موفقیت آمیز گزارش می‌شود که شیء مورد نظر نال باشد.

```
Assert.IsNotNull( object anObject );
```

این بررسی تنها در صورتی موفقیت آمیز گزارش می‌شود که شیء مورد نظر نال نباشد.

```
Assert.IsNaN( double aDouble );
```

این بررسی تنها در صورتی موفقیت آمیز گزارش می‌شود که شیء مورد نظر عددی نباشد (اگر با جاوا اسکریپت کار کرده باشید حتما با isNan آشنا هستید، is not a numeric).

```
Assert.IsEmpty( string aString );
Assert.IsEmpty( ICollection collection );
```

جهت بررسی خالی بودن یک رشته یا لیست بکار می‌رود.

```
Assert.IsNotEmpty( string aString );
Assert.IsNotEmpty( ICollection collection );
```

جهت بررسی خالی نبودن یک رشته یا لیست بکار می‌رود.

ج) بررسی حالت‌های مقایسه‌ای

```
Assert.Greater( double arg1, double arg2 );
Assert.GreaterOrEqual( int arg1, int arg2 );
Assert.Less( int arg1, int arg2 );
Assert.LessOrEqual( int arg1, int arg2 );
```

نکته‌ای را که در اینجا باید در نظر داشت این است که همواره شیء اول با شیء دوم مقایسه می‌شود. مثلا در حالت اول، اگر شیء اول بزرگتر از شیء دوم بود، آزمایش مورد نظر با موفقیت گزارش خواهد شد.
از ذکر انواع و اقسام overload های این توابع جهت طولانی نشدن مطلب پرهیز شد.

د) بررسی نوع اشیاء

```
Assert.IsInstanceOfType( Type expected, object actual );
Assert.IsNotInstanceOfType( Type expected, object actual );
Assert.IsAssignableFrom( Type expected, object actual );
Assert.IsNotAssignableFrom( Type expected, object actual );
```

این توابع و Overload های آن‌ها امکان بررسی نوع شیء مورد نظر را میسر می‌سازند.

ه) متدهای کمکی

```
Assert.Fail();
Assert.Ignore();
```

در حالت استفاده از ignore ، آزمایش واحد شما در حین اجرا ندید گرفته خواهد شد. از متد fail برای طراحی یک متد assertion سفارشی می‌توان استفاده کرد. برای مثال:
طراحی متدی که بررسی کند آیا یک رشته مورد نظر حاوی عبارتی خاص می‌باشد یا خیر:

```
public void AssertStringContains( string expected, string actual,

string message )

{

if ( actual.IndexOf( expected )

Assert.Fail( message );

}
```

و) متدهای ویژه‌ی بررسی رشته‌ها

```
StringAssert.Contains( string expected, string actual );
StringAssert.StartsWith( string expected, string actual );
StringAssert.EndsWith( string expected, string actual );
StringAssert.AreEqualIgnoringCase( string expected, string actual );
StringAssert.IsMatch( string expected, string actual );
```

این متدها و انواع overload های آن‌ها جهت بررسی‌های ویژه رشته‌ها بکار می‌روند. برای مثال آیا رشته مورد نظر حاوی عبارتی خاص است؟ آیا با عبارتی خاص شروع می‌شود یا با عبارتی ویژه، پایان می‌پذیرد و امثال آن.

ز) بررسی فایل‌ها

```
FileAssert.AreEqual( Stream expected, Stream actual );
FileAssert.AreEqual( FileInfo expected, FileInfo actual );
FileAssert.AreEqual( string expected, string actual );

FileAssert.AreNotEqual( Stream expected, Stream actual );
```

```
FileAssert.AreNotEqual( FileInfo expected, FileInfo actual );
FileAssert.AreNotEqual( string expected, string actual );
```

این متدها جهت مقایسه دو فایل بکار می‌روند و ورودی‌های آن‌ها می‌تواند از نوع stream ، شیء FileInfo و یا مسیر فایل‌ها باشد.

ح) بررسی collections

```
CollectionAssert.AllItemsAreInstancesOfType( IEnumerable collection, Type expectedType );
CollectionAssert.AllItemsAreNotNull( IEnumerable collection );
CollectionAssert.AllItemsAreUnique( IEnumerable collection );
CollectionAssert.AreEqual( IEnumerable expected, IEnumerable actual );
CollectionAssert.AreEquivalent( IEnumerable expected, IEnumerable actual);
CollectionAssert.AreNotEqual( IEnumerable expected, IEnumerable actual );
CollectionAssert.AreNotEquivalent( IEnumerable expected,IEnumerable actual );
CollectionAssert.Contains( IEnumerable expected, object actual );
CollectionAssert.DoesNotContain( IEnumerable expected, object actual );
CollectionAssert.IsSubsetOf( IEnumerable subset, IEnumerable superset );
CollectionAssert.IsNotSubsetOf( IEnumerable subset, IEnumerable superset);
CollectionAssert.IsEmpty( IEnumerable collection );
CollectionAssert.IsNotEmpty( IEnumerable collection );
```

به صورت اختصاصی و ویژه نیز می‌توان بررسی مقایسه‌ای را بر روی اشیایی از نوع IEnumerable انجام داد. برای مثال آیا معادل هستند، آیا شیء مورد نظر نال نیست و امثال آن.

نکته: در تمامی overload های این توابع، آرگومان message نیز وجود دارد. از این آرگومان زمانیکه آزمایش با شکست مواجه شد، جهت ارائه اطلاعات بیشتری استفاده می‌گردد.

ادامه دارد...

عنوان: **آشنایی با آزمایش واحد (unit testing) در دات نت، قسمت 6**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۷/۱۰/۲۱ ۰۹:۱۵:۲۸
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

ادامه آشنایی با NUnit

فرض کنید یک RSS reader نوشته‌اید که فیدهای فارسی و انگلیسی را دریافت می‌کند. به صورت پیش فرض هم مشخص نیست که کدام فید اطلاعات فارسی را ارائه خواهد داد و کدامیک انگلیسی. تشخیص محتوای فارسی و از راست به چپ نشان دادن خودکار مطالب آن‌ها به عهده‌ی برنامه نویس است. بهترین روش برای تشخیص این نوع الگوها، استفاده از regular expressions است.
برای مثال الگوی تشخیص اینکه آیا متن ما حاوی حروف انگلیسی است یا خیر به صورت زیر است:

```
[a-zA-Z]
```

که بیان بصری آن به این شکل است.
در مورد تشخیص وجود حروف فارسی در یک عبارت، یکی از دو الگوی زیر بکار می‌رود:

```
[\u0600-\u06FF]
[ا-یءئ]
```

در مورد اینکه بازه یونیکد فارسی استاندارد از کجا شروع می‌شود می‌توان به مقاله‌ی آقای حاج‌لو مراجعه نمود (به صورت خلاصه، بازه مصوب عربی یونیکد، همان بازه یونیکد فارسی نیز می‌باشد . یا به بیان بهتر، بازه‌ی فارسی، جزئی از بازه‌ای است که عربی نام گرفته است). البته بازه‌ی مصوب دیگری هم در مورد ایران باستان وجود دارد به نام old Persian که مورد استفاده‌ی روزمره‌ای ندارد!

کلاس زیر را در مورد استفاده از این الگوها تهیه کرده‌ایم:

```
using System.Text.RegularExpressions;

namespace sample
{
   public static class CDetectFarsi
   {
       public static bool ContainsFarsiData(this string txt)
       {
           return !string.IsNullOrEmpty(txt) &&
                   Regex.IsMatch(txt, "[ا-یءئ]");
       }

       public static bool ContainsFarsi(this string txt)
       {
           return !string.IsNullOrEmpty(txt) &&
                   Regex.IsMatch(txt, @"[\u0600-\u06FF]");
       }
   }
}
```

همانطور که ملاحظه می‌کنید در اینجا از extension methods سی شارپ 3 جهت توسعه کلاس پایه string استفاده شد.
اکنون می‌خواهیم بررسی کنیم آیا این الگوها مقصود ما را برآورده می‌سازند یا خیر.

```
using NUnit.Framework;
using sample;

namespace TestLibrary
{
   [TestFixture]
   public class TestFarsiClass
   {
       /*****************************************************************************/
       [Test]
       public void TestContainsFarsi1()
       {
           Assert.IsTrue("وحید".ContainsFarsi());
       }

       [Test]
       public void TestContainsFarsi2()
       {
           Assert.IsTrue("گردان".ContainsFarsi());
       }

       [Test]
       public void TestContainsFarsi3()
       {
           Assert.IsTrue("سپیدTest".ContainsFarsi());
       }

       [Test]
       public void TestContainsFarsi4()
       {
           Assert.IsTrue("123456بررسی".ContainsFarsi());
       }

       [Test]
       public void TestContainsFarsi5()
       {
           Assert.IsFalse("Book".ContainsFarsi());
       }

       [Test]
       public void TestContainsFarsi6()
       {
           Assert.IsTrue("\۳۸۷".ContainsFarsi());
       }

       [Test]
       public void TestContainsFarsi7()
       {
           Assert.IsFalse("Здравствуйте!".ContainsFarsi()); //Russian hello!
       }


       /*****************************************************************************/
       [Test]
       public void TestContainsFarsiData1()
       {
           Assert.IsTrue("وحید".ContainsFarsiData());
       }

       [Test]
       public void TestContainsFarsiData2()
       {
           Assert.IsTrue("گردان".ContainsFarsiData());
       }

       [Test]
       public void TestContainsFarsiData3()
       {
           Assert.IsTrue("سپیدTest".ContainsFarsiData());
       }

       [Test]
       public void TestContainsFarsiData4()
       {
           Assert.IsTrue("123456بررسی".ContainsFarsiData());
       }

       [Test]
       public void TestContainsFarsiData5()
       {
           Assert.IsFalse("Book".ContainsFarsiData());
```

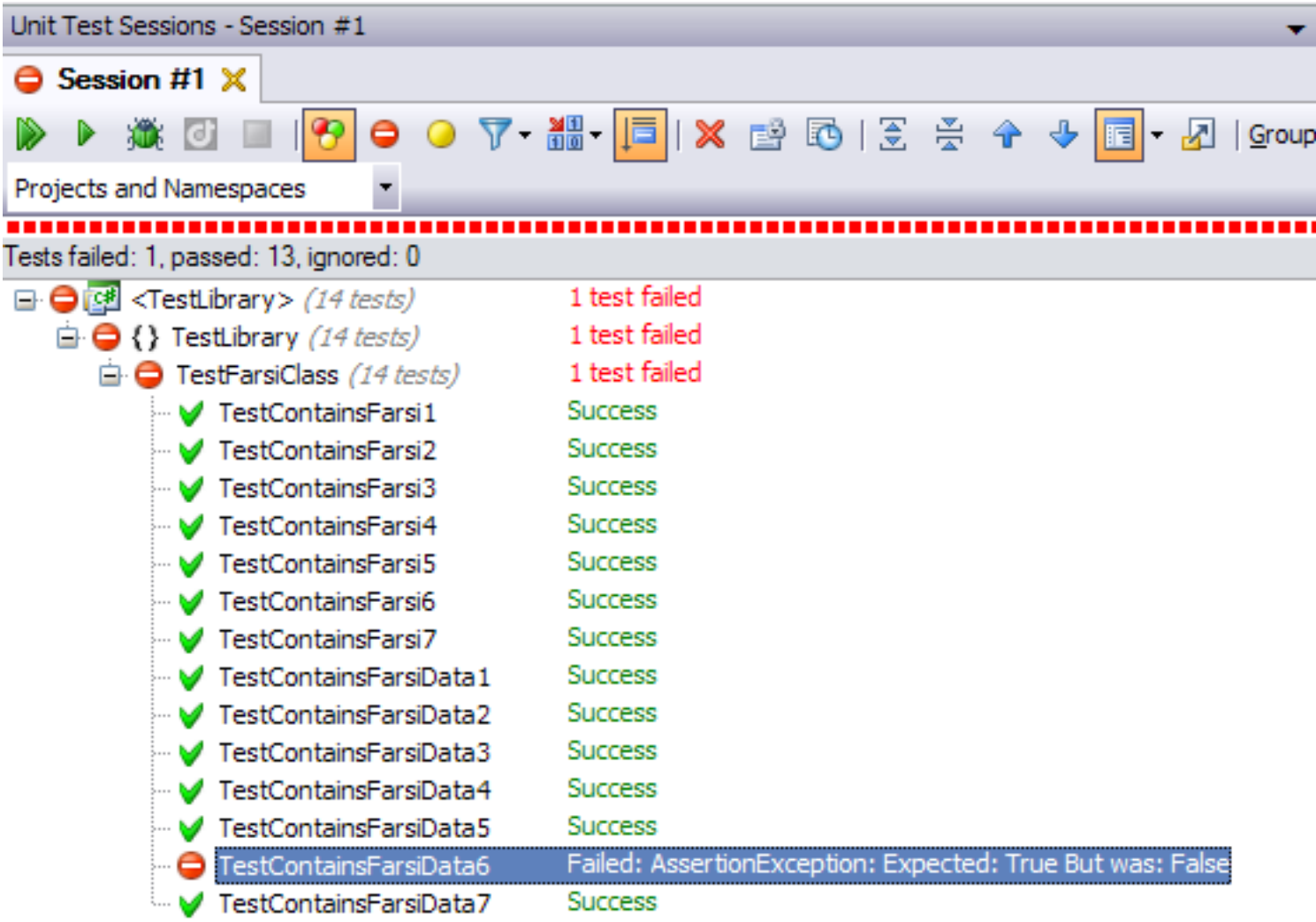
```
        }

        [Test]
        public void TestContainsFarsiData6()
        {
            Assert.IsTrue("\۳۸۷".ContainsFarsiData());
        }

        [Test]
        public void TestContainsFarsiData7()
        {
            Assert.IsFalse("Здравствуйте!".ContainsFarsiData()); //Russian hello!
        }
    }
}
```

در کلاس فوق هر دو متد را با آزمایش‌های واحد مختلفی بررسی کرده‌ایم، انواع و اقسام حروف فارسی، ترکیبی از فارسی و انگلیسی، ترکیبی از فارسی و اعداد انگلیسی، عبارت کاملا انگلیسی، عدد کاملا فارسی و یک عبارت روسی! (در یک کلاس عمومی با متدهای عمومی بدون آرگومان از نوع void)

کلاس CDetectFarsi در برنامه اصلی قرار دارد و کلاس TestFarsiClass در یک پروژه class library دیگر قرار گرفته است (در این مورد و جدا سازی آزمایش‌ها از پروژه اصلی در قسمت‌های قبل بحث شد)

همچنین به ازای هر عبارت Assert یک متد ایجاد گردید تا شکست یکی، سبب اختلال در بررسی سایر موارد نشود.

نتیجه اجرای این آزمایش واحد با استفاده از امکانات مجتمع افزونه ReSharper به صورت زیر است:



منهای یک مورد، سایر آزمایشات ما با موفقیت انجام شده‌اند. موردی که با شکست مواجه شده، بررسی اعداد کاملا فارسی است که البته در الگوی دوم لحاظ نشده است و انتظار هم نمی‌رود که آن‌را به این شکل پشتیبانی کند.
برای اینکه در حین اجرای آزمایشات بعدی این متد در نظر گرفته نشود، می‌توان ویژگی Test آن‌را به صورت زیر تغییر داد:

```
[Test,Ignore]
```

نکته: مرسوم شده است که نام متدهای آزمایش واحد به صورت زیر تعریف شوند (با Test شروع شوند، در ادامه نام متدی که بررسی می‌شود ذکر گردد و در آخر ویژگی مورد بررسی عنوان شود):

```
Test[MethodToBeTested][SomeAttribute]
```

ادامه دارد...

نظرات خوانندگان

نویسنده: مسعود
تاریخ: ۱۳۸۷/۱۰/۲۱ ۲۰:۵۷:۰۰

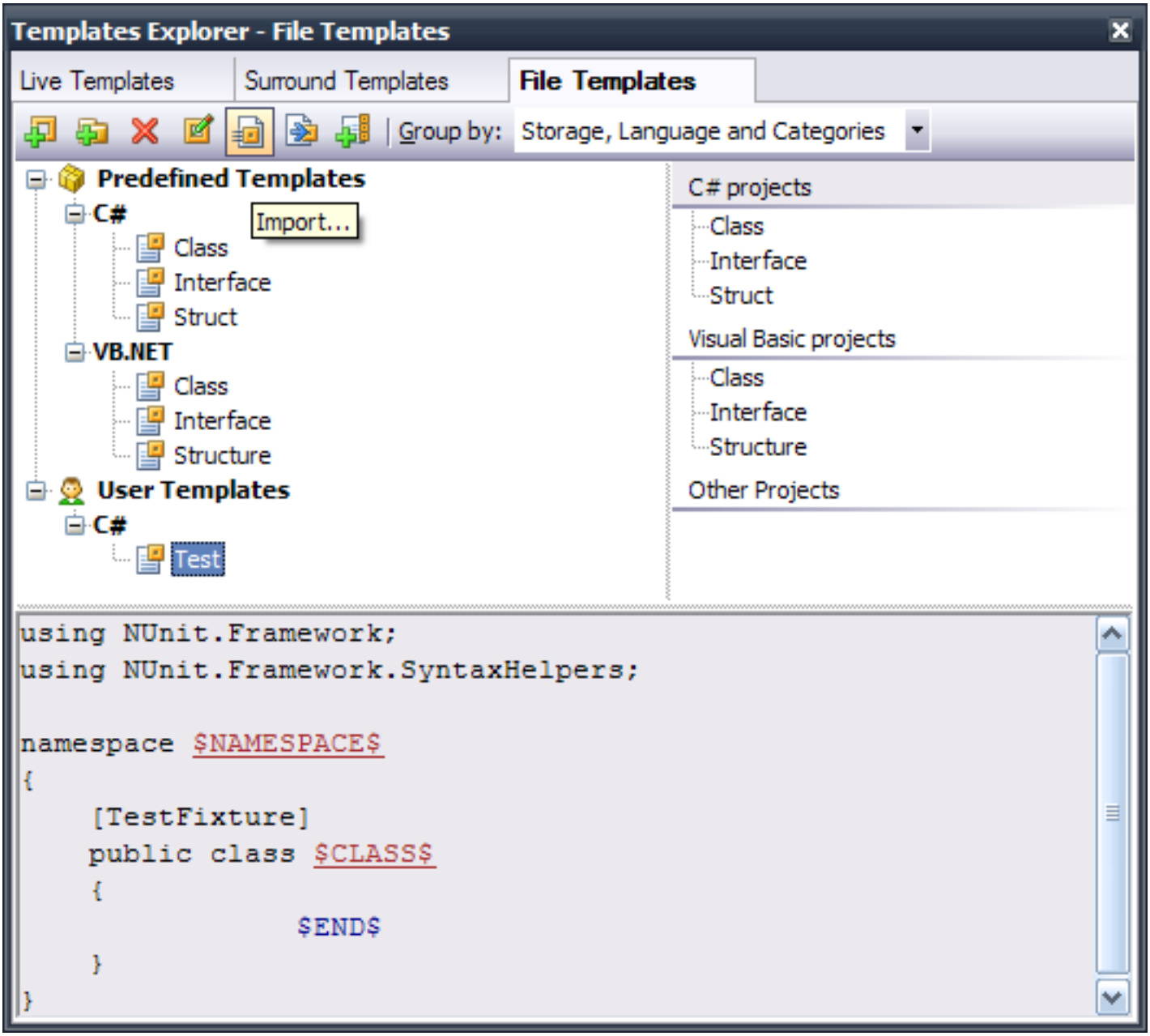
واقعا جالب !

کیف کردم ، تا حالا یه چند باری خواستم برم سراغ این Unit Testing ولی اصلا وقت نشد.

ممنون استاد

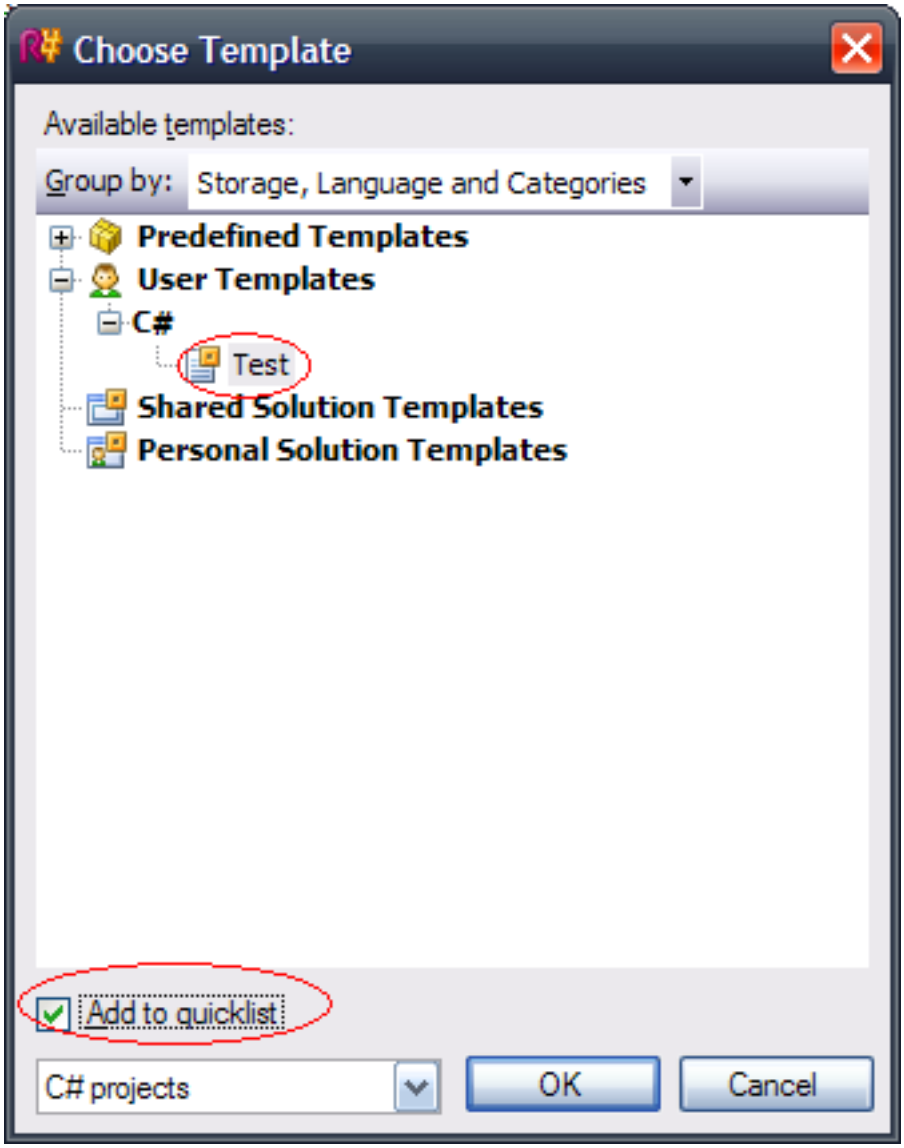
عنوان: **قالبی برای ایجاد آزمون‌های NUnit مخصوص ReSharper**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۱/۱۱ ۱۹:۳۱:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

افزونه‌ی ReSharper به‌دلیل یکپارچه کردن امکان استفاده از NUnit در ویژوال استودیو، یکی از انتخاب‌های اول جهت انجام آزمایشات واحد در این محیط به شمار می‌رود.
اخیرا آقای Genisio چند قالب ایجاد آزمون‌های NUnit را مخصوص ReSharper ایجاد کرده‌اند ، که در ادامه در مورد نحوه‌ی استفاده از آن‌ها توضیح داده خواهد شد.
پس از دریافت فایل‌ها ، برای استفاده، به منوی ReSharper گزینه‌ی live templates مراجعه نمائید. سپس بر روی نوار ابزار صفحه‌ی باز شده، روی دکمه‌ی import کلیک نموده و فایل‌ها را معرفی کنید.
NewTestFileTemplate.xml از نوع file template است.
TestTemplates.xml از نوع live template می‌باشد.



اکنون مجددا به منوی اصلی ReSharper مراجعه کنید و مسیر زیر را طی نمائید:

```
ReSharper -> new from template -> more …
```



گزینه‌ی Test اضافه شده را انتخاب کرده و سپس قسمت Add to quicklist را نیز انتخاب نمائید.
به این صورت گزینه‌ی Test به این منو افزوده خواهد شد و هر بار که بر روی آن کلیک شود، یک کلاس حاضر و آماده مطابق قالب اصلی یک کلاس استاندارد NUnit برای شما ایجاد خواهد شد.
همچنین در این مجموعه یک سری live template نیز موجود است که کار آن‌ها فعال سازی intellisense ویژوال استودیو جهت ایجاد یک سری متدها به صورت خودکار است. برای مثال اگر کلمه‌ی test را تایپ کنید و سپس دکمه‌ی tab و یا enter را فشار دهید، بلافاصله بدنه‌ی خالی یک متد تست برای شما ایجاد خواهد شد.
سایر میان‌برهای در نظر گرفته شده، به شرح زیر هستند:

```
test - Create a new [Test] method
setup - Create a [SetUp] method
teardown - Create a new [TearDown] method
ise - Assert that condition is equal to value
ist - Assert that condition is true
isf - Assert that condition is false
isn - Assert that condition is null
isnn - Assert that condition is not null
```

نظرات خوانندگان

نویسنده: fateme
تاریخ: ۱۳۸۹/۰۹/۲۷ ۱۵:۴۵:۰۸

سلام
ممکنه بهم بگید منو resharper در vs کجاست؟

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۹/۲۷ ۱۶:۴۷:۳۳

از اینجا باید دریافت شود: (+)

نویسنده: رضا
تاریخ: ۱۳۹۱/۰۶/۲۵ ۱۲:۲۹

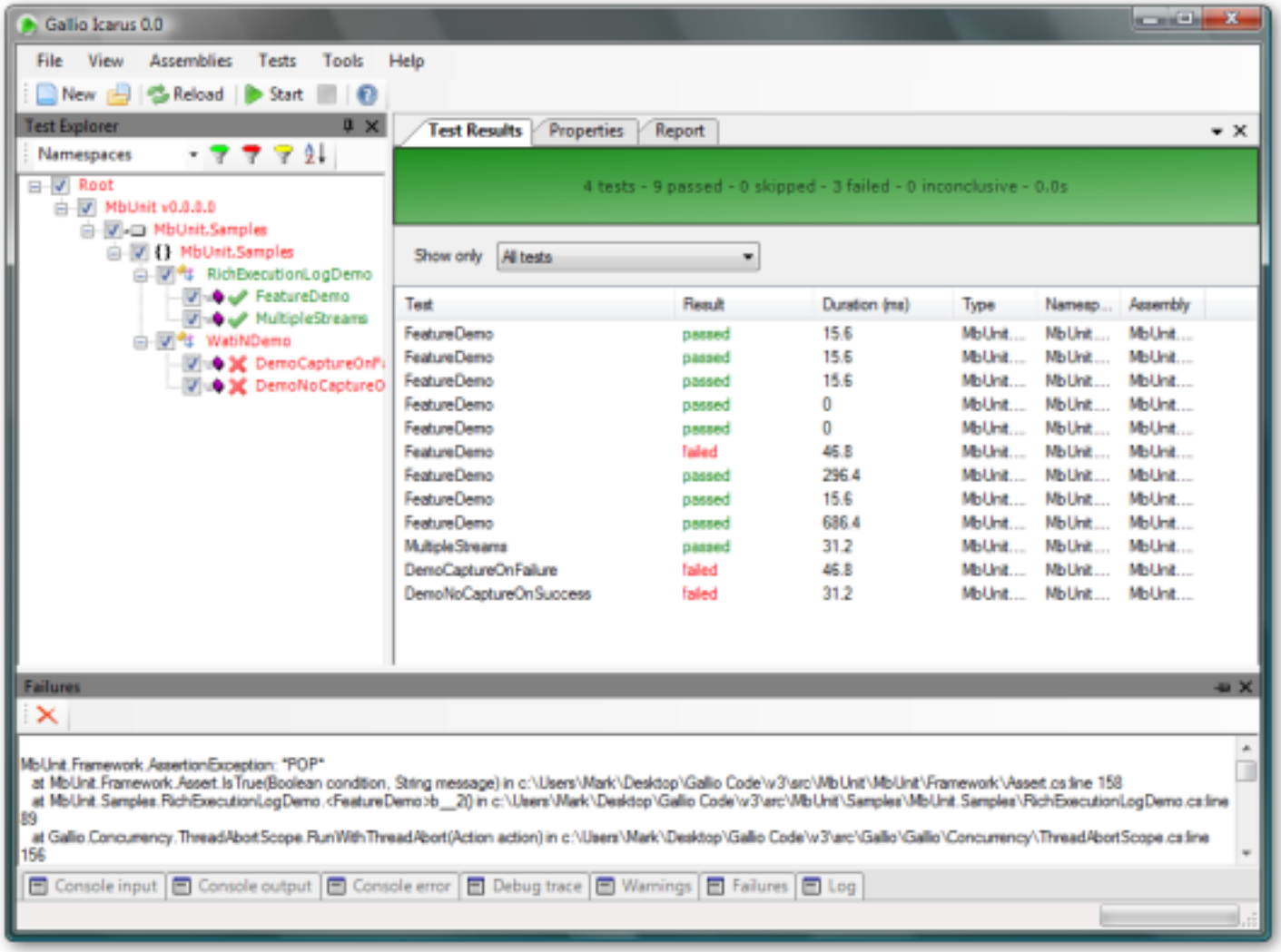
این Resharper نسخه رایگان نداره؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۲۵ ۱۲:۳۶

خیر. ولی یک سری nightly builds آزمایشی و مدت دار داره که رایگان است (حدود یک ماه) برای آزمایش. این برنامه زمان ارائه نگارش‌های جدید فعال می‌شود.

عنوان: **نگارش نهایی MBUnit 3 ارائه شد**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۱/۱۵ ۱۹:۴۴:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

MBUnit که یکی دیگر از فریم ورک‌های آزمایش واحد یا unit testing دات نت به شمار می‌رود، نگارش 3 بتا آن از سال 2007 شروع شده و اخیرا نگارش نهایی 3 آن ارائه گردیده است.



جهت مشاهده‌ی جزئیات آخرین تغییرات اعمال شده در نگارش جدید آن، MbUnit v3.0.6 Update 1 می‌توان به وبلاگ یکی از اعضای اصلی تیم مراجعه نمود .

```
Added support for TestDriven.Net category filters.
Added support for more powerful inclusion/exclusion test filter expressions.
Fixed ReSharper v3.1, v4.0 and v4.1 hangs.
Increased NCover v1.5.8 timeout.
Adopted new assembly version numbering scheme
```

MBUnit از جهات بسیاری از NUnit پیشرفته‌تر است برای مثال در هنگام انجام آزمایشات واحد بر روی یک دیتابیس، به صورت خودکار امکان حذف رکوردهای آزمایشی را داشته و کار به حالت اول بازگردان وضعیت دیتابیس را انجام می‌دهد .
همچنین نگارش 3 آن کار یکپارچه شدن با ReSharper را هم انجام می‌دهد که پیشتر به صورت پیش فرض مهیا نبود و این مورد یکی از دلایل مهم استفاده گسترده از NUnit به شمار می‌رفت.

پ.ن.
برای دریافت آن باید به گوگل‌کد مراجعه کرد که احتمالا با مشکلاتی همراه خواهد بود. نگارش نهایی آن تا این تاریخ را از اینجا دریافت کنید.

نظرات خوانندگان

نویسنده: Anonymous
تاریخ: ۱۳۸۸/۰۲/۰۱ ۲۰:۰۰:۰۰

Question
:
kodoomeshoon az in Testing tools .... Code Coverage ro ham dare for .net .... mesle chizi ke vase java hast

thanks
http://en.wikipedia.org/wiki/Code_coverage

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۲/۰۱ ۲۰:۲۷:۰۰

http://www.ncover.com

نویسنده: Anonymous
تاریخ: ۱۳۸۸/۰۲/۰۴ ۲۰:۵۹:۰۰

Question aqgain
Moshkele Mock Objects ro che konam? Tools hast vasash ? ya khodemoon bayadmock objects ro tooye test hamono dorost konim
?
thanks

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۲/۰۴ ۲۳:۲۸:۰۰

http://ayende.com/projects/rhino-mocks.aspx

نویسنده: مهدی پایروند
تاریخ: ۱۳۸۸/۰۳/۲۷ ۱۷:۲۵:۲۷

سلام باز خسته نباشی، مرجعی برای کسی که میخواد تازه شروع کنه حالا چه فارسی یا انگلیسی میشه معرفی کنید.

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۳/۲۷ ۱۷:۳۱:۱۶

برای شروع:
Tag/Unit%20testing/

عنوان: **آشنایی با mocking frameworks (چارچوب‌های تقلید) - قسمت اول**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۲/۱۵ ۲۱:۵۴:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

این مطلب در ادامه‌ی مطالب آزمون‌های واحد یا unit testing است.
نوشتن آزمون واحد برای کلاس‌هایی که با یک سری از الگوریتم‌ها ، مسایل ریاضی و امثال آن سر و کار دارند، ساده است. عموما این نوع کلاس‌ها وابستگی خارجی آنچنانی ندارند؛ اما در عمل کلاس‌های ما ممکن است وابستگی‌های خارجی بسیاری پیدا کنند؛ برای مثال کار با دیتابیس، اتصال به یک وب سرویس، دریافت فایل از اینترنت، خواندن اطلاعات از انواع فایل‌ها و غیره.
مطابق اصول آزمایشات واحد، یک آزمون واحد خوب باید ایزوله باشد. نباید به مرزهای سیستم‌های دیگر وارد شده و عملکرد سیستم‌های خارج از کلاس را بررسی کند.
این مثال ساده را در نظر بگیرید:
فرض کنید برنامه شما قرار است از یک وب سرویس لیستی از آدرس‌های IP یک کشور خاص را دریافت کند و در یک دیتابیس محلی آن‌ها را ذخیره نماید. به صورت متداول این کلاس باید اتصالی را به وب سرویس گشوده و اطلاعات را دریافت کند و همچنین آن‌ها را خارج از مرز کلاس در یک دیتابیس ثبت کند. نوشتن آزمون واحد برای این کلاس مطابق اصول مربوطه غیر ممکن است. اگر کلاس آزمون واحد آن‌را تهیه نمائید، این آزمون، integration test نام خواهد گرفت زیرا از مرزهای سیستم باید عبور نماید.

همچنین یک آزمون واحد باید تا حد ممکن سریع باشد تا برنامه نویس از انجام آن بر روی یک پروژه بزرگ منصرف نگردد و ایجاد این اتصالات در خارج از سیستم، بیشتر سبب کندی کار خواهند شد.

برای این ایزوله سازی روش‌های مختلفی وجود دارند که در ادامه به آن‌ها خواهیم پرداخت:

روش اول: استفاده از اینترفیس‌ها
با کمک یک اینترفیس می‌توان مشخص کرد که یک قطعه از کد "چه کاری" را قرار است انجام دهد؛ و نه اینکه "چگونه" باید آن‌را به انجام رساند.
یک مثال ساده از خود دات نت فریم ورک، اینترفیس IComparable است:

```
public static string GetComparisonText(IComparable a, IComparable b)
   {
       if (a.CompareTo(b) == 1)
           return "a is bigger";
       if (a.CompareTo(b) == -1)
           return "b is bigger";
       return "same";
   }
```

در این مثال چون از IComparable استفاده شده، متد ما از هر نوع داده‌ای جهت مقایسه می‌تواند استفاده کند. تنها موردی که برای آن مهم خواهد بود این است که a راهی را برای مقایسه با b ارائه دهد.

اکنون با توجه به این توضیحات، برای ایزوله کردن ارتباط با دیتابیس و وب سرویس در مثال فوق، می‌توان اینترفیس‌های زیر را تدارک دید:

```
public interface IEmailSource
   {
       IEnumerable<string> GetEmailAddresses();
   }

   public interface IEmailDataStore
   {
       void SaveEmailAddresses(IEnumerable<string> emailAddresses);

   }
```

در اینجا استفاده و تعریف اینترفیس‌ها چندین خاصیت را به همراه خواهد داشت :
الف) به این صورت تنها مشخص می‌شود که چه کاری را قصد داریم انجام دهیم و کاری به پیاده سازی آن نداریم.
ب) ساخت کلاس بدون وجود یا دسترسی به یک دیتابیس میسر می‌شود. این مورد خصوصا در یک کار تیمی که قسمت‌های مختلف کار به صورت همزمان در حالت پیشرفت و تهیه است حائز اهمیت می‌شود.
ج) با توجه به اینکه در اینجا به پیاده سازی توجهی نداریم، می‌توان از این اینترفیس‌ها جهت تقلید دنیای واقعی استفاده کنیم. (که در اینجا mocking نام گرفته است)

جهت تقلید رفتار و عملکرد این دو اینترفیس، به کلاس‌های تقلید زیر خواهیم رسید:

```
public class MockEmailSource : IEmailSource
{
   public IEnumerable<string> EmailAddressesToReturn { get; set; }
   public IEnumerable<string> GetEmailAddresses()
   {
       return EmailAddressesToReturn;
   }
}

public class MockEmailDataStore : IEmailDataStore
{
   public IEnumerable<string> SavedEmailAddresses { get; set; }
   public void SaveEmailAddresses(IEnumerable<string> emailAddresses)
   {
       SavedEmailAddresses = emailAddresses;
   }

}
```

تا اینجا اولین قدم در مورد ایزوله سازی کلاس‌هایی که به مرز سیستم‌های دیگر وارد می‌شوند، برداشته شد. اما به مرور زمان مدیریت این اینترفیس‌ها و افزودن رفتارهای جدید به کلاس‌های مشتق شده از آن‌ها مشکل می‌شود. به همین جهت تا حد ممکن از پیاده سازی دستی آن‌ها خودداری شده و روش پیشنهادی استفاده از mocking frameworks است.

ادامه دارد ....

عنوان: **آشنایی با mocking frameworks - قسمت دوم**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۲/۱۷ ۱۶:۱۰:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

استفاده از mocking frameworks :

تعدادی از چارچوب‌های تقلید نوشته شده برای دات نت فریم ورک مطابق لیست زیر بوده و هدف از آن‌ها ایجاد ساده‌تر اشیاء تقلید برای ما می‌باشد:

Nmock : http://www.nmock.org
Moq : http://code.google.com/p/moq Rhino Mocks : http://ayende.com/projects/rhino-mocks.aspx
TypeMock : http://www.typemock.com
EasyMock.Net : http://sourceforge.net/projects/easymocknet

در این بین Rhino Mocks که توسط یکی از اعضای اصلی تیم NHibernate به وجود آمده است، در مجامع مرتبط بیشتر مورد توجه است. برای آشنایی بیشتر با آن می‌توان به این ویدیوی رایگان آموزشی در مورد آن مراجعه نمود (حدود یک ساعت است).

خلاصه‌ای در مورد نحوه‌ی استفاده از Rhino Mocks :
پس از دریافت کتابخانه سورس باز Rhino Mocks ، ارجاعی را به اسمبلی Rhino.Mocks.dll آن، در پروژه آزمون واحد خود اضافه نمائید.
یک Rhino mock test با ایجاد شیءایی از MockRepository شروع می‌شود و کلا از سه قسمت تشکیل می‌گردد:
الف) ایجاد شیء Mock یا Arrange . هدف از ایجاد شیء mock ، جایگزین کردن و یا تقلید یک شیء واقعی جهت مباحثی مانند ایزوله سازی آزمایشات، بالابردن سرعت آن‌ها و متکی به خود کردن این آزمایشات می‌باشد. همچنین در این حالت نتایج false positive نیز کاهش می‌یابند. منظور از نتایج false positive این است که آزمایش باید با موفقیت به پایان برسد اما اینگونه نشده و علت آن بررسی سیستمی دیگر در خارج از مرزهای سیستم فعلی است و مشکل از جای دیگری نشات گرفته که اساسا هدف از تست ما بررسی عملکرد آن سیستم نبوده است. کلا در این موارد از mocking objects استفاده می‌شود:
- دسترسی به شیء مورد نظر کند است مانند دسترسی به دیتابیس یا محاسبات بسیار طولانی
- شیء مورد نظر از call back استفاده می‌کند

- شیء مورد آزمایش باید به منابع خارجی دسترسی پیدا کند که اکنون مهیا نیستند. برای مثال دسترسی به شبکه.
- شیءایی که می‌خواهیم آن‌را تست کنیم یا برای آن آزمایشات واحد تهیه نمائیم، هنوز کاملا توسعه نیافته و نیمه کاره است.

ب) تعریف رفتارهای مورد نظر یا Act

ج) بررسی رفتارهای تعریف شده یا Assert

مثال:

متد ساده زیر را در نظر بگیرید:

```
public class ImageManagement
  {
      public string GetImageForTimeOfDay()
      {
          int currentHour = DateTime.Now.Hour;
          return currentHour > 6 && currentHour < 21 ? "sun.jpg" : "moon.jpg";
      }

  }
```

آزمایش این متد، وابسته است به زمان جاری سیستم.

```
using System;
using NUnit.Framework;

  [TestFixture]
  public class CMyTest
  {
      [Test]
      public void DaytimeTest()
      {
          int currentHour = DateTime.Now.Hour;

          if (currentHour > 6 && currentHour < 21)
          {
              const string expectedImagePath = "sun.jpg";
              ImageManagement image = new ImageManagement();
              string path = image.GetImageForTimeOfDay();
              Assert.AreEqual(expectedImagePath, path);
          }
          else
          {
              Assert.Ignore("تنها در طول روز قابل بررسی است");
          }
      }

      [Test]
      public void NighttimeTest()
      {
          int currentHour = DateTime.Now.Hour;

          if (currentHour < 6 || currentHour > 21)
          {
              const string expectedImagePath = "moon.jpg";
              ImageManagement image = new ImageManagement();
              string path = image.GetImageForTimeOfDay();
              Assert.AreEqual(expectedImagePath, path);
          }
          else
          {
              Assert.Ignore("تنها در طول شب قابل بررسی است");
          }
      }

  }
```

برای مثال اگر بخواهیم تصویر ماه را دریافت کنیم باید تا ساعت 21 صبر کرد. همچنین بررسی اینکه چرا یکی از متدهای آزمون واحد ما نیز با شکست مواجه شده است نیز نیازمند بررسی زمان جاری است و گاهی ممکن است با شکست مواجه شود و گاهی خیر. در این‌جا با استفاده از یک mock object ، این وضعیت غیرقابل پیش بینی را با منطقی از پیش طراحی شده جایگزین کرده و آزمون خود را بر اساس آن انجام خواهیم داد.

برای این‌کار باید DateTime.Now.Hour را تقلید نموده و اینترفیسی را بر اساس آن طراحی نمائیم. سپس Rhino Mocks کار پیاده سازی این اینترفیس را انجام خواهد داد:

```
using NUnit.Framework;
using Rhino.Mocks;

namespace testWinForms87
{
  public interface IDateTime
  {
      int GetHour();
  }

  public class ImageManagement
  {
      public string GetImageForTimeOfDay(IDateTime time)
      {
          int currentHour = time.GetHour();

          return currentHour > 6 && currentHour < 21 ? "sun.jpg" : "moon.jpg";
      }
  }

  [TestFixture]
  public class CMocking
  {
      [Test]
      public void DaytimeTest()
      {
          MockRepository mocks = new MockRepository();
          IDateTime timeController = mocks.CreateMock<IDateTime>();

          using (mocks.Record())
          {
              Expect.Call(timeController.GetHour()).Return(15);
          }

          using (mocks.Playback())
          {
              const string expectedImagePath = "sun.jpg";
              ImageManagement image = new ImageManagement();
              string path = image.GetImageForTimeOfDay(timeController);
              Assert.AreEqual(expectedImagePath, path);
          }
      }

      [Test]
      public void NighttimeTest()
      {
          MockRepository mocks = new MockRepository();
          IDateTime timeController = mocks.CreateMock<IDateTime>();
          using (mocks.Record())
          {
              Expect.Call(timeController.GetHour()).Return(1);
          }

          using (mocks.Playback())
          {
              const string expectedImagePath = "moon.jpg";
              ImageManagement image = new ImageManagement();
              string path = image.GetImageForTimeOfDay(timeController);
              Assert.AreEqual(expectedImagePath, path);
          }
      }
  }

}
```

همانطور که در ابتدای مطلب هم عنوان شد، mocking از سه قسمت تشکیل می‌شود:

```
MockRepository mocks = new MockRepository();
```

ابتدا شیء mocks را از MockRepository کتابخانه Rhino Mocks ایجاد می‌کنیم تا بتوان از خواص و متدهای آن استفاده کرد.
سپس اینترفیسی باید به آن پاس شود تا انتظارات سیستم را بتوان در آن بر پا نمود:

```
IDateTime timeController = mocks.CreateMock<IDateTime>();
using (mocks.Record())
{
 Expect.Call(timeController.GetHour()).Return(15);

}
```

به عبارت دیگر در اینجا به سیستم مقلد خود خواهیم گفت: زمانیکه شیء ساعت را تقلید کردی، لطفا عدد 15 را برگردان.
به این صورت آزمایش ما بر اساس وضعیت مشخصی از سیستم صورت می‌گیرد و وابسته به ساعت جاری سیستم نخواهد بود.

همانطور که ملاحظه می‌کنید، روش Test Driven Development بر روی نحوه‌ی برنامه نویسی ما و ایجاد کلاس‌ها و اینترفیس‌های اولیه نیز تاثیر زیادی خواهد گذاشت. استفاده از اینترفیس‌ها یکی از اصول پایه‌ای برنامه نویسی شیءگرا است و در اینجا مقید به ایجاد آن‌ها خواهیم شد.

پس از آن‌که در قسمت mocks.Record ، انتظارات خود را ثبت کردیم، اکنون نوبت به وضعیت Playback می‌رسد:

```
using (mocks.Playback())
{
 string expectedImagePath = "sun.jpg";
 ImageManagement image = new ImageManagement();
 string path = image.GetImageForTimeOfDay(timeController);
 Assert.AreEqual(expectedImagePath, path);

}
```

در اینجا روش کار همانند ایجاد متدهای آزمون واحد متداولی است که تاکنون با آن‌ها آشنا شده‌ایم و تفاوتی ندارد.
با توجه به اینکه پس از تغییر طراحی متد GetImageForTimeOfDay ، این متد اکنون از شیء IDateTime به عنوان ورودی استفاده می‌کند، می‌توان پیاده سازی آن اینترفیس را در آزمایشات واحد تقلید نمود و یا جایی که قرار است در برنامه استفاده شود، می‌تواند پیاده سازی واقعی خود را داشته باشد و دیگر آزمایشات ما وابسته به آن نخواهد بود:

```
public class DateTimeController : IDateTime
{
  public int GetHour()
  {
     return DateTime.Now.Hour;
  }
}
```

نظرات خوانندگان

نویسنده: SirAsad
تاریخ: ۱۳۸۸/۰۲/۱۹ ۱۵:۰۰:۰۰

آقا بسیار مطلب جالبی بود... در این مورد در بلاگ یه مطلب گذاشتم ... امیدوارم مفید واقع بشه.

http://sir.blogsky.com/1388/02/19/post-115

موفق باشید

عنوان: **تولید خودکار آزمون‌های واحد NUnit**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۱۱/۱۲ ۰۹:۴۳:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

تعدادی ابزار برای تولید خودکار متدهای آزمون‌های واحد NUnit از روی کلاس‌های موجود در یک اسمبلی وجود دارند که به دو دسته تقسیم می‌شود:

الف) آن‌هایی که فقط نام کلاس‌های آزمون واحد و نام متدهای آن‌را به صورت خودکار تولید می‌کنند

NStub
Top Coder .Net Test Generator
CodeSmith NUnit Test Generator

این ابزارها و کتابخانه‌ها، تنها کاری که انجام می‌دهند یافتن کلاس‌ها و متدهای عمومی موجود در یک اسمبلی توسط Reflection و سپس تولید یک سری فایل آماده از روی این اطلاعات است. برای مثال اگر نام کلاس شما Class1 است فایلی به نام TestClass1 را تولید می‌کنند و اگر یکی از متدهای عمومی این کلاس به نام Method1 باشد، یک متد خالی را به نام Method1Test ایجاد خواهند کرد و همین.
تبدیل CodeSmith NUnit Test Generator فوق به یک T4 template کار ساده‌ای است.

ب) ابزارهایی که علاوه بر مورد الف، سعی می‌کنند بدنه‌ای را نیز برای متدهای واحد تولید شده تهیه کنند

Novell NUnitGen AddIn
Edwinyeah TestGen.Net
NUnit Test Case Code Generator

این افزونه‌ها و برنامه‌ها سعی می‌کنند به کمک Reflection و همچنین امکانات تولید کد موجود در VS.NET نسبت به تولید کلاس‌ها، متدها و بدنه‌های نمونه آن‌ها اقدام کنند. برای مثال اگر نام متد کلاسی، Method1 به همراه یک پارامتر از نوع int باشد، بدنه تولید شده به همراه وهله سازی از کلاس آن و فراخوانی این متد به همراه پارامتر آن خواهد بود.
مشکل مهم این پروژه‌های سورس باز کوچک هم عدم تعهد به نگهداری آن‌ها است. برای مثال آخرین به روز رسانی موجود افزونه‌ی NUnitGen شرکت ناول، مخصوص VS2008 است یا آخرین به روز رسانی TestGen.Net مربوط به دات نت یک است (سورسی هم که در سایت سورس فورج قرار داده ناقص است) یا مقاله‌ی سایت CodeProject که ذکر گردید، با نگارش‌های جدید NUnit درست کار نمی‌کند و کامپایل نمی‌شود.

در بین این‌ها به نظر من Edwinyeah TestGen.Net کار جالبی را انجام داده است و چندین زبان را هم پشتیبانی می‌کند. البته همانطور که عنوان شد توانایی بارگذاری اسمبلی‌های نگارش‌های جدید دات نت را ندارد که موضوع مهمی نیست. سورس آن‌را می‌توان دریافت و سپس جهت دات نت 4 کامپایل کرد. البته یک سری از کلاس‌های آن هم که در سورس موجود نیستند را می‌شود از اسمبلی کامپایل شده‌ی آن با Reflector درآورد، به پروژه اصلی اضافه و سپس کامپایل کرد!
کامپایل شده‌ی آن‌را جهت دات نت 4 از اینجا دریافت کنید.

عنوان: **بررسی میزان پوشش آزمون‌های واحد به کمک برنامه PartCover**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۱/۲۴ ۲۳:۴۵:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

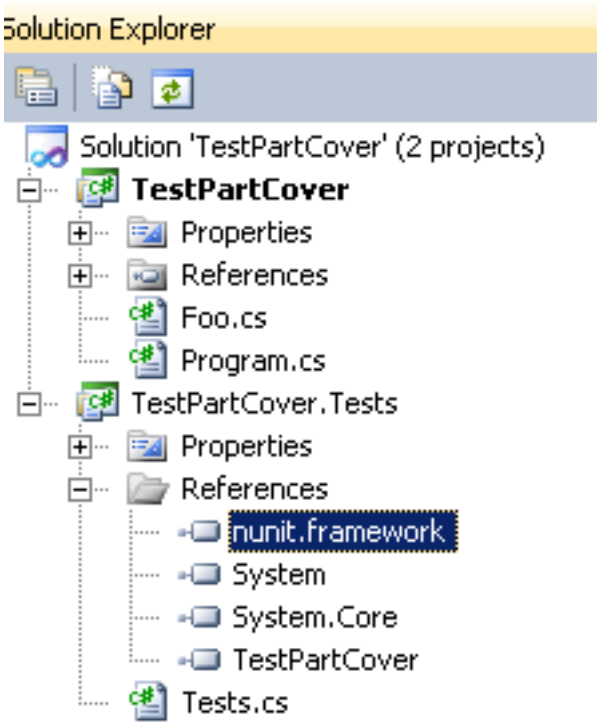
همیشه در حین توسعه‌ی یک برنامه این سؤالات وجود دارند:
- چند درصد از برنامه تست شده است؟
- برای چه تعدادی از متدهای موجود آزمون واحد نوشته‌ایم؟
- آیا همین آزمون‌های واحد نوشته شده و موجود، کامل هستند و تمام عملکردهای متدهای مرتبط را پوشش می‌دهند؟

این سؤالات به صورت خلاصه مفهوم Code coverage را در بحث Unit testing ارائه می‌دهند: برای چه قسمت‌هایی از برنامه آزمون واحد ننوشته‌ایم و میزان پوشش برنامه توسط آزمون‌های واحد موجود تا چه حدی است؟
بررسی این سؤالات در یک پروژه‌ی کم حجم، ساده بوده و به صورت بازبینی بصری ممکن است. اما در یک پروژه‌ی بزرگ نیاز به ابزار دارد. به همین منظور تعدادی برنامه جهت بررسی code coverage مختص پروژه‌های دات نتی تابحال تولید شده‌اند که در ادامه لیست آن‌ها را مشاهده می‌کنید:

NCover
Pex & Mole
DotCover
و ...

تمام این‌ها تجاری هستند. اما در این بین برنامه‌ی PartCover سورس باز و رایگان بوده و همچنین مختص به NUnit نیز تهیه شده است. این برنامه را از اینجا می‌توانید دریافت و نصب کنید. در ادامه نحوه‌ی تنظیم آن‌را بررسی خواهیم کرد:

الف) ایجاد یک پروژه آزمون واحد جدید
جهت توضیح بهتر سه سؤال مطرح شده در ابتدای این مطلب، بهتر است یک مثال ساده را در این زمینه مرور نمائیم: (پیشنیاز: (+))
یک Solution جدید در VS.NET آغاز شده و سپس دو پروژه جدید از نوع‌های کنسول و Class library به آن اضافه شده‌اند:



پروژه کنسول، برنامه اصلی است و در پروژه Class library ، آزمون‌های واحد برنامه را خواهیم نوشت.
کلاس اصلی برنامه کنسول به شرح زیر است:

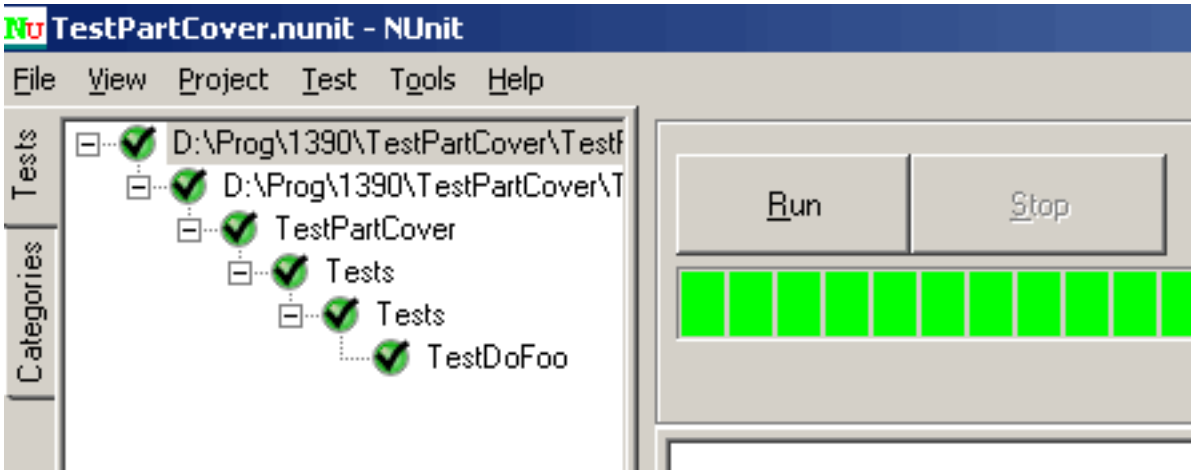
```
namespace TestPartCover
{
public class Foo
{
    public int DoFoo(int x, int y)
    {
        int z = 0;
        if ((x > 0) && (y > 0))
        {
            z = x;
        }
        return z;
    }

    public int DoSum(int x)
    {
        return ++x;
    }
}
}
```

و کلاس آزمون واحد آن در پروژه class library مثلا به صورت زیر خواهد بود:

```
using NUnit.Framework;

namespace TestPartCover.Tests
{
[TestFixture]
public class Tests
{
    [Test]
    public void TestDoFoo()
    {
        var result = new Foo().DoFoo(-1, 2);
        Assert.That(result == 0);
    }
```

```
}
}
```

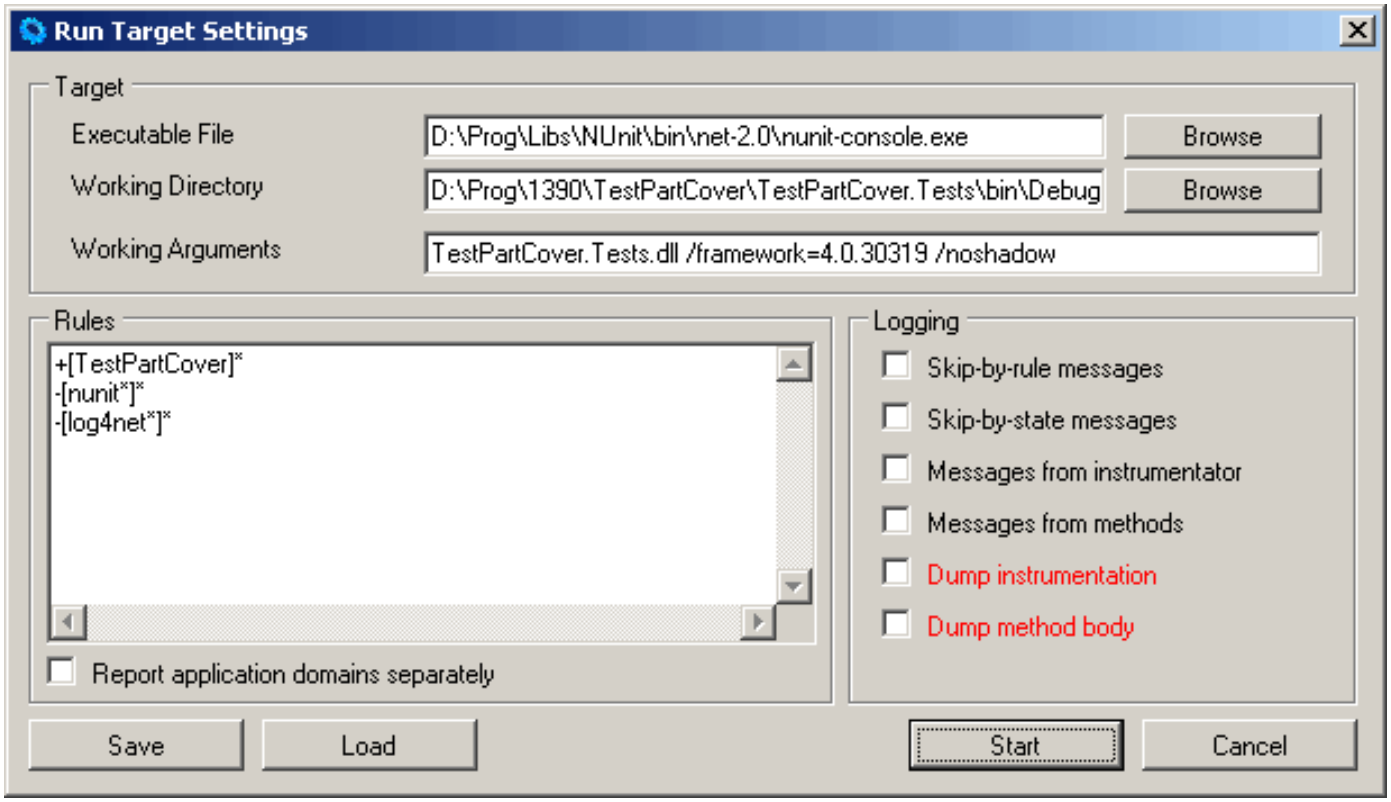
که نتیجه‌ی بررسی آن توسط NUnit test runner به شکل زیر خواهد بود:



به نظر همه چیز خوب است! اما آیا واقعا این آزمون کافی است؟!

ب) در ادامه به کمک برنامه‌ی PartCover می‌خواهیم بررسی کنیم میزان پوشش آزمون‌های واحد نوشته شده تا چه حدی است؟

پس از نصب برنامه، فایل PartCover.Browser.exe را اجرا کرده و سپس از منوی فایل، گزینه‌ی Run Target را انتخاب کنید تا صفحه‌ی زیر ظاهر شود:



توضیحات:

در قسمت executable file آدرس فایل nunit-console.exe را وارد کنید. این برنامه چون در حال حاضر برای دات نت 2 کامپایل شده امکان بارگذاری dll های دات نت 4 را ندارد. به همین منظور فایل nunit-console.exe.config را باز کرده و تنظیمات زیر را به آن اعمال کنید (مهم!):

```
<configuration>
<startup>
<supportedRuntime version="v4.0.30319" />
</startup>
```

و همچنین

```
<runtime>
<loadFromRemoteSources enabled="true" />
```

در ادامه مقابل working directory ، آدرس پوشه bin پروژه unit test را تنظیم کنید.

در این حالت working arguments به صورت زیر خواهند بود (در غیراینصورت باید مسیر کامل را وارد نمائید):

```
TestPartCover.Tests.dll /framework=4.0.30319 /noshadow
```

نام dll وارد شده همان فایل class library تولیدی است. آرگومان بعدی مشخص می‌کند که قصد داریم یک پروژه‌ی دات نت 4 را توسط NUnit بررسی کنیم (اگر ذکر نشود پیش فرض آن دات نت 2 خواهد بود و نمی‌تواند اسمبلی‌های دات نت 4 را بارگذاری کند). منظور از noshadow این است که NUnit مجاز به تولید shadow copies از اسمبلی‌های مورد آزمایش نیست. به این صورت

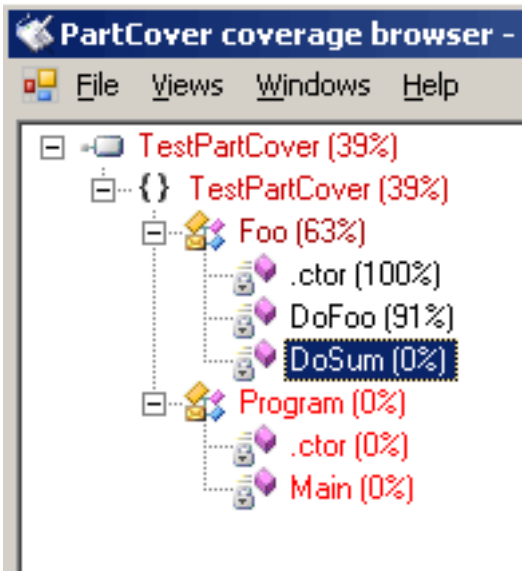
برنامه‌ی PartCover می‌تواند بر اساس StackTrace نهایی، سورس متناظر با قسمت‌های مختلف را نمایش دهد.
اکنون نوبت به تنظیم Rules آن است که یک سری RegEx هستند؛ به عبارتی چه اسمبلی‌هایی آزمایش شوند و کدام‌ها خیر:

```
+[TestPartCover]*
-[nunit*]*
-[log4net*]*
```

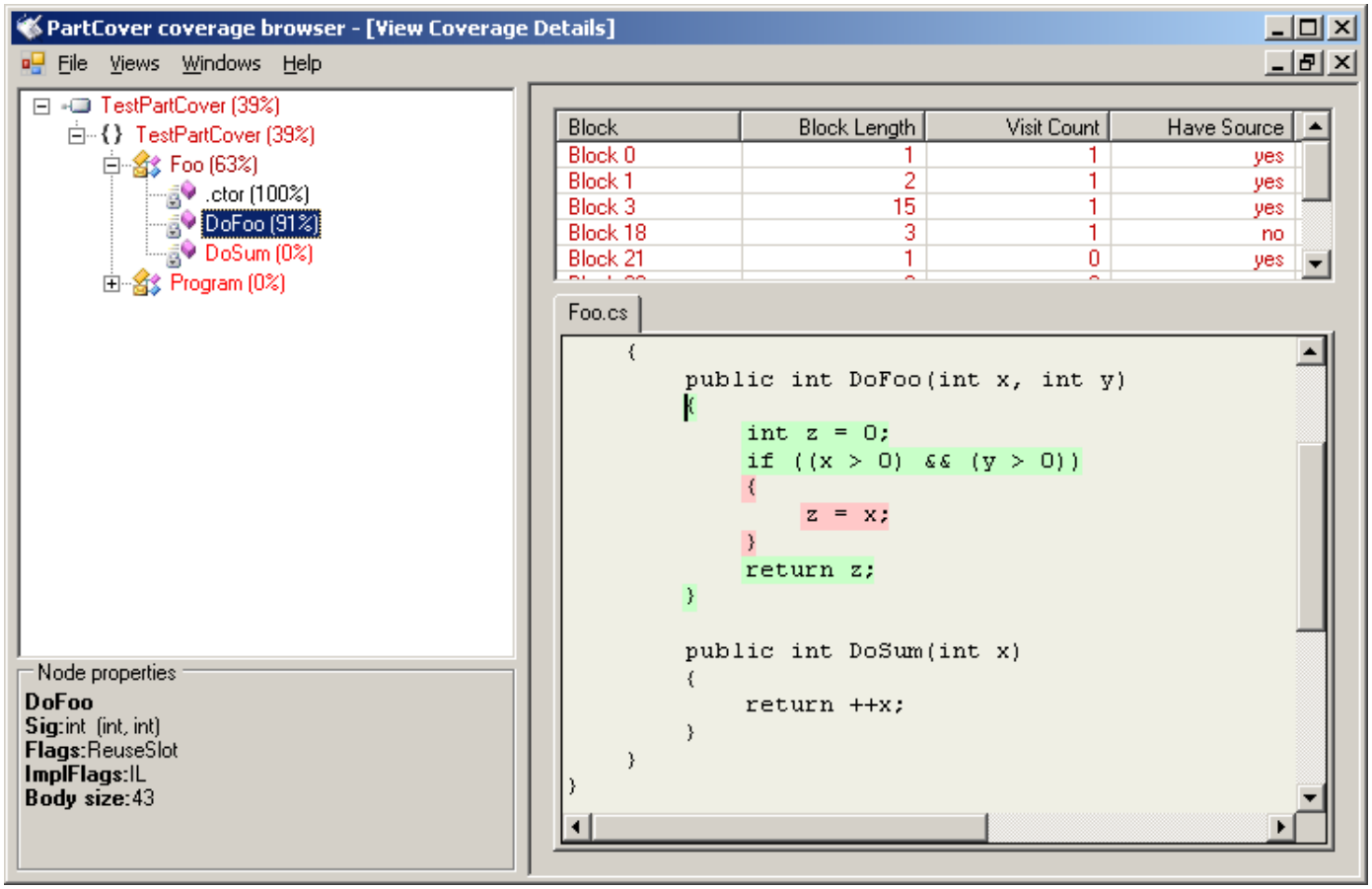
همانطور که ملاحظه می‌کنید در اینجا از اسمبلی‌های NUnit و log4net صرفنظر شده است و تنها اسمبلی TestPartCover (همان برنامه کنسول، نه اسمبلی برنامه آزمون واحد) بررسی خواهد گردید.
اکنون بر روی دکمه Save در این صفحه کلیک کرده و فایل نهایی را ذخیره کنید (بعدا توسط دکمه Load در همین صفحه قابل بارگذاری خواهد بود). حاصل باید به صورت زیر باشد:

```
<PartCoverSettings>
<Target>D:\Prog\Libs\NUnit\bin\net-2.0\nunit-console.exe</Target>
<TargetWorkDir>D:\Prog\1390\TestPartCover\TestPartCover.Tests\bin\Debug</TargetWorkDir>
<TargetArgs>TestPartCover.Tests.dll /framework=4.0.30319 /noshadow</TargetArgs>
<Rule>+[TestPartCover]*</Rule>
<Rule>-[nunit*]*</Rule>
<Rule>-[log4net*]*</Rule>
</PartCoverSettings>
```

برای شروع به بررسی، بر روی دکمه Start کلیک نمائید. پس از مدتی، نتیجه به صورت زیر خواهد بود:



بله! آزمون واحد تهیه شده تنها 39 درصد اسمبلی TestPartCover را پوشش داده است. مواردی که با صفر درصد مشخص شده‌اند، یعنی فاقد آزمون واحد هستند و نکته مهم‌تر پوشش 91 درصدی متد DoFoo است. برای اینکه علت را مشاهده کنید از منوی View ، گزینه‌ی Coverage detail را انتخاب کنید تا تصویر زیر نمایان شود:



قسمت نارنجی در اینجا به معنای عدم پوشش آن در متد TestDoFoo تهیه شده است. تنها قسمت‌های سبز را توانسته‌ایم پوشش دهیم و برای بررسی تمام شرط‌های این متد نیاز به آزمون‌های واحد بیشتری می‌باشد.

روش نهایی کار نیز به همین صورت است. ابتدا آزمون واحد تهیه می‌شود. سپس میزان پوشش آن بررسی شده و در ادامه سعی خواهیم کرد تا این درصد را افزایش دهیم.

عنوان: **تهیه آزمون واحد جهت کار با محتوای فایل‌ها**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۲/۰۵ ۱۳:۲۲:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

یکی از شروط تهیه آزمون‌های واحد، خارج نشدن از مرزهای سیستم در حین بررسی آزمون‌های مورد نظر است؛ تا بتوان تمام آزمون‌ها را با سرعت بسیار بالایی، بدون نگرانی از در دسترس نبودن منابع خارجی، درست در لحظه انجام آزمون‌ها، به پایان رساند. اگر این خروج صورت گیرد، بجای unit tests با integration tests سر و کار خواهیم داشت. در این میان، کار با فایل‌ها نیز مصداق بارز خروج از مرزهای سیستم است.
برای حل این مشکل راه حل‌های زیادی توصیه شده‌اند؛ منجمله تهیه یک اینترفیس محصور کننده فضای نام System.IO و سپس استفاده از فریم ورک‌های mocking و امثال آن. یک نمونه از پیاده سازی آن‌را اینجا می‌توانید پیدا کنید : (+)
اما راه حل ساده‌تری نیز برای این مساله وجود دارد و آن هم افزودن فایل‌های مورد نظر به پروژه آزمون واحد جاری و سپس مراجعه به خواص فایل‌ها و تغییر Build Action آن‌ها به Embedded Resource می‌باشد. به این صورت پس از کامپایل پروژه، فایلهای ما در قسمت منابع اسمبلی جاری قرار گرفته و به کمک متد زیر قابل دسترسی خواهند بود:

```
using System.IO;
using System.Reflection;

public class UtHelper
{
 public static string GetInputFile(string filename)
 {
   var thisAssembly = Assembly.GetExecutingAssembly();
   var stream = thisAssembly.GetManifestResourceStream(filename);
   return new StreamReader(stream).ReadToEnd();
 }
}
```

نکته‌ای را که اینجا باید به آن دقت داشت، filename متد GetInputFile است. چون این فایل دیگر به صورت متداول از فایل سیستم خوانده نخواهد شد، نام واقعی آن به صورت namespace.filename می‌باشد (همان نام منبع اسمبلی جاری).
اگر جهت یافتن این نام با مشکل مواجه شدید، تنها کافی است اسمبلی آزمون واحد را با برنامه Reflector یا ابزارهای مشابه گشوده و نام منابع آن‌را بررسی کنید.

عنوان: **استفاده یکپارچه از NUnit در VS.NET بدون نیاز به افزونه‌ها**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۶/۱۷ ۱۱:۰۲:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing

برای استفاده ساده‌تر از ابزارهای unit testing در ویژوال استودیو افزونه‌های زیادی وجود دارند، از ری شارپر تا CodeRush تا حتی امکانات نسخه‌ی کامل VS.NET که با MSTest یکپارچه است. اما اگر نخواهیم از MSTest استفاده کنیم و همچنین افزونه‌ها را هم بخواهیم حذف کنیم (مثلا از نسخه‌ی رایگان express استفاده کنیم)، چطور؟
برای حل این مشکل چندین روش وجود دارد. یا می‌شود از test runner این‌ها استفاده کرد که اصلا نیازی به IDE ندارند و مستقل است؛ یا می‌توان به صورت زیر هم عمل کرد:
به خواص پروژه در VS.NET مراجعه کنید. برگه‌ی Build events را باز کنید. در اینجا می‌خواهیم post-build event را مقدار دهی کنیم. به این معنا که پس از هر build موفق، لطفا این دستورات خط فرمان را اجرا کن.
NUnit به همراه test runner خط فرمان هم ارائه می‌شود و نام آن nunit-console.exe است. اگر به محل نصب آن مراجعه کنید، عموما در آدرس C:\Program Files\NUnit xyz\bin\nunit-console.exe قرار دارد. برای استفاده از آن تنها کافی است تنظیم زیر صورت گیرد:

```
c:\path\nunit-console.exe /nologo $(TargetPath)
```

TargetPath به صورت خودکار با نام اسمبلی جاری پروژه در زمان اجرا جایگزین می‌شود.
اکنون پس از هر Build، به صورت خودکار nunit-console.exe اجرا شده، اسمبلی برنامه که حاوی آزمون‌های واحد است به آن ارسال گردیده و سپس خروجی کار در output window نمایش داده می‌شود. اگر خطایی هم رخ داده باشد در قسمت errors قابل مشاهده خواهد بود.
در اینجا حتی بجای برنامه کنسول یاده شده می‌توان از برنامه nunit.exe هم استفاده کرد. در این حالت GUI اصلی پس از هر Build نمایش داده می‌شود:

```
c:\path\nunit.exe $(TargetPath)
```

چند نکته:
1- برنامه nunit-console.exe چون در حال حاضر برای دات نت 2 کامپایل شده امکان بارگذاری dll های دات نت 4 را ندارد. به همین منظور فایل nunit-console.exe.config را باز کرده و تنظیمات زیر را به آن اعمال کنید:

```
<configuration>
<startup>
      <supportedRuntime version="v4.0.30319" />
</startup>
```

و همچنین:

```
<runtime>
```

```
    <loadFromRemoteSources enabled="true" />
```

-2 خروجی نتایج اجرای آزمون‌ها را به صورت XML هم می‌توان ذخیره کرد. مثلا:

```
c:\path\nunit-console.exe /xml:$(ProjectName)-tests.xml /nologo $(TargetPath)
```

-3 از فایل xml ذکر شده می‌توان گزارشات زیبایی تهیه کرد. برای مثال:

Generating Report for NUnit

NUnit2Report Task

جهت مطالعه بیشتر:

Setting up Visual C#2010 Express with NUnit

Use Visual Studio's Post-Build Events to Automate Unit Testing Running

Ways to Run NUnit From Visual Studio 3

نظرات خوانندگان

نویسنده: علی
تاریخ: ۱۳۹۲/۰۴/۱۰ ۱۰:۵۵

با عرض سلام و تشکر از وب سایت پر محتواتون.
اونچه رو که در بالا فرموده بودین رو بنده آزمایش کردم ولی متاسفانه به هنگام build کردن پروژه با خطای زیر روبرو می‌شوم :

```
Error 1 The command "c:\program files\nunit 2.6.2\bin\nunit.exe C:\Users\Ali\documents\visual studio 2010\Projects\NUnitExample\NUnitExample.UnitTests\bin\Debug\NUnitExample.UnitTests.dll /run" exited with code 9009. NUnitExample.UnitTests
```

ممنون می‌شوم راهنماییم کنید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۴/۱۰ ۱۱:۳۷

- علت اینجا است که در مسیر فایل dll شما space وجود دارد؛ در قسمت visual studio 2010. به همین منظور نیاز است برای اجرای یک دستور خط فرمان، این نوع مسیرها داخل "" قرار گیرند (یک اصل کلی است در مورد تمام فرامین خط فرمان).
- ضمنا اگر از VS 2012 استفاده می‌کنید، بهتر است از NUnit Test Adapter کمک بگیرید، تا با یک سیستم یکپارچه بتوانید کار کنید.

عنوان: **قبل از رفع باگ، برای آن تست بنویسید**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۰ ۲۱:۵۸:۰۰
آدرس: www.dotnettips.info
گروه‌ها: Unit testing

از دقت کردن در نحوه اداره پروژه‌های خوب و بزرگ در سطح دنیا، می‌توان به نکات آموزنده‌ای رسید. برای مثال NHibernate را درنظر بگیرید. این پروژه شاید روز اول کپی مطابق اصل نمونه جاوای آن بوده، اما الان از خیلی از جهات یک سر و گردن از آن بالاتر است. پشتیبانی LINQ را اضافه کرده، خودش Syntax جدیدی را به نام QueryOver ارائه داده و همچنین معادلی را جهت حذف فایل‌های XML به کمک امکانات جدید زبان‌های دات نتی مانند lambda expressions ارائه کرده. خلاصه این تیم، فقط یک کپی کار نیست. پایه رو از یک جایی گرفته اما سبب تحول در آن شده. از اهداف پروژه‌های سورس باز هم همین است: برای هر کاری چرخ را از صفر ابداع نکنید.

اگر به نحوه اداره کلی پروژه NHibernate دقت کنید یک مورد مشهود است:
تمام گزارش‌های باگ بدون Unit test ندید گرفته می‌شوند.
از کلیه بهبودهای ارائه شده (وصله‌ها یا patch ها) بدون Unit test صرفنظر می‌شود.
از کلیه موارد جدید ارائه شده بدون Unit test هم صرفنظر خواهد شد.

بنابراین اگر در issue tracker این تیم رفتید و گفتید: «سلام، اینجا این مشکل هست»، خیالتان راحت باشد که ندید گرفته خواهید شد.

سؤال : چرا این‌ها اینطور رفتار می‌کنند؟!
- وجود Unit test دقیقا مشخص می‌کند که چه قسمت یا قسمت‌هایی به گزارش باگ شما مرتبط هستند. نیازی نیست حتما بتوانید یک خطا را با جملات ساده شرح دهید. این مساله خصوصا در پروژه‌های بین المللی حائز اهمیت است. ضعف زبان انگلیسی همه جا هست. همینقدر که توانسته‌اید برای آن یک Unit test بنویسید که مثلا در این عملیات با این ورودی، نتیجه قرار بوده بشود 10 و مثلا شده 5 یا حتی این Exception صادر شده که باید کنترل شود، یعنی مشکل را کاملا مشخص کرده‌اید.
- وجود Unit tests ، انجام Code review و همچنین Refactoring را تسهیل می‌بخشند. در هر دو حالت یاد شده، هدف تغییر کارکرد سیستم نیست؛ هدف بهبود کیفیت کدهای موجود است. بنابراین دست به یک سری تغییرات زده خواهد شد. اما سؤال اینجا است که از کجا باید مطمئن شد که این تغییرات، سیستم را به هم نریخته‌اند. پروژه‌ی جاری چند سال است که در حال توسعه است. قسمت‌های زیادی به آن اضافه شده. با نبود Unit tests ممکن است بعضی از قسمت‌ها زاید یا احمقانه به نظر برسند.
- بهترین مستندات کدهای تهیه شده، Unit tests آن هستند. برای مثال علاقمند هستید که NHibernate را یاد بگیرید؟ هرچه می‌گردید مثال‌های کمی را در اینترنت در این زمینه پیدا می‌کنید؟ وقت خودتان را تلف نکنید! این پروژه بالای 2000 آزمون واحد دارد. هر کدام از این آزمون‌ها نحوه‌ی بکارگیری قسمت‌های مختلف را به نحوی کاربردی بیان می‌کنند.
- وجود Unit tests از پیدایش مجدد باگ‌ها جلوگیری می‌کنند. اگر آزمون واحدی وجود نداشته باشد، امروز کدی اضافه می‌شود. فردا همین کد توسط عده‌ای دیگر زاید تشخیص داده شده و حذف می‌شود! بنابراین احتمال بروز مجدد این خطا در آینده وجود خواهد داشت. با وجود Unit tests، فلسفه وجودی هر قسمتی از کدهای موجود پروژه دقیقا مشخص می‌شود و در صورت حذف آن، با اجرای آزمون‌های خودکار سریعا می‌توان به کمبودهای حاصل پی‌برد.

نظرات خوانندگان

نویسنده: بهزاد
تاریخ: ۱۳۹۰/۱۰/۲۰ ۲۳:۲۷:۰۱

وحید عزیز، من وبلاگ ت رو دنبال می کنم و کتاب فوق العاده ات درباره Exchange Server را خوانده ام
مدتهاست که به دنبال یک جزوه یا کتاب مختصر برای یاد گرفتن تست هستم، کلیات رو می دونم، ولی به نظرم یک پروژه نمونه که تا حد زیادی واقعی باشه خیلی بیشتر از دهها کتاب قطور می تونه کمک کنه
می تونی در این زمینه منبعی معرفی کنی؟ پلتفرمی که من کار می کنم ASP.NET WebForm چهار لایه است که از NH هم به عنوان لایه دیتابیس استفاده می کنم

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۱ ۰۰:۳۸:۴۲

- برای NH اگر از الگوی Repository استفاده می‌کنید می‌تونید از SQLite به عنوان ابزار نوشتن آزمون‌های واحد استفاده کنید. SQLite یک مزیت جالبی که دارد این است که امکان تشکیل دیتابیس در حافظه را دارد. این یعنی همان پیشنیاز اصلی نوشتن آزمون‌های واحد: سرعت بالای انجام کار، خارج نشدن از مرزهای سیستم. ضمن اینکه این بانک اطلاعاتی تشکیل شده، یک بانک اطلاعاتی واقعی است اما پس از پایان کار به صورت خودکار نابود می‌شود که برای آزمون‌های واحد بسیار مفید است.
برای ORM های دیگر چون پشتیبانی از سایر بانک‌های اطلاعاتی آن‌ها ضعیف است، روش‌های mocking و غیره مطرح می‌شود (که اینبار دیگر با یک دیتابیس واقعی کار نمی‌شود و سطح کار کمی پایین‌تر است) اما با NH راحت می‌شود از SQLite تشکیل شده در حافظه استفاده کرد. فقط باید تنظیمات اتصال ابتدای برنامه را عوض کرد.
- خوب؛ تا اینجا واژه کلیدی مورد نیاز برای جستجو مشخص شد، مابقی را در اینجا (^) جستجو کنید.

نویسنده: Shima2012
تاریخ: ۱۳۹۰/۱۰/۲۲ ۲۰:۴۸:۳۳

با سلام.
جناب نصیری در حال حاضر برای Unit testing استفاده از MBUnit 3 رو پیشنهاد میدید یا NUnit

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۲ ۲۱:۳۲:۰۳

فریم ورک زیاد هست. حتی خود مایکروسافت هم مثلا MSTest رو داره که با VS.NET یکپارچه است. نکته مهم این ابزارها نیستند. مهم نوشتن تست است. مهم این نیست که از SVN استفاده کنید یا از GIT . مهم این است که از یک سورس کنترل استفاده شود.

نویسنده: پریسا زاهدی
تاریخ: ۱۳۹۳/۱۰/۱۲ ۲۱:۲۱

سلام؛ من تا حالا از Unit Testing تو پروژه‌ها استفاده نکردم و شرکت هایی که با اونا همکاری می‌کردم اصلا آزمایش واحد رو تو پروژه‌ها استفاده نمی‌کنند و یک کار اضافی و حتی بیهوده قلمداد میکنن اینو .
لطفا مسیر راهی برای فراگیری Unit Testing مشخص کنین تا من و بقیه دوستان مثل من بدونیم ازکجا شروع کنیم ای مهم رو و در پروژه‌ها انجامش بدیم ؟

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۰/۱۲ ۲۱:۴۰

گروه Unit Testing را در سایت پیگیری کنید. مطالب آن نظم خوبی دارند و مرحله به مرحله هست.

عنوان: **ساخت ActionResult سفارشی**
نویسنده: محمد صاحب
تاریخ: ۱۳۹۱/۱۱/۲۱ ۱۷:۵۵
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing, ASP.Net MVC

پیشتر با انواع ActionResult آشنا شدید. حال فرض کنید می‌خواهید نوعی رو برگردونید که براش ActionResult موجود نباشه مثلا RSS و یا فایل از نوع Excel و...
خوب، فرض کنید می‌خواهید اکشن متدی رو بنویسید که قراره نام یک فایل متنی رو بگیره و انو تو مروگر به کاربر نمایش بده.
برای اینکار از کلاس ActionResult، کلاس دیگه‌ی رو بنام TextResult به ارث می‌بریم و از این ActionResult سفارشی شده، در اکشن متد مربوطه استفاده می‌کنیم:

```
public class TextResult : ActionResult
{
    public string FileName { get; set; }
    public override void ExecuteResult(ControllerContext context)
    {
        var filePath = Path.Combine(context.HttpContext.Server.MapPath(@"~/Files/"), FileName);
        var data = File.ReadAllText(filePath);
        context.HttpContext.Response.Write(data);
    }
}
```

نحوه استفاده

```
public ActionResult DownloadTextFile(string fileName)
    {
        return new TextResult { FileName = fileName };
    }
```

در واقع متد اصلی اینجا ExecuteResult هست که نتیجه‌ی کار یک اکشن رو می‌تونیم پردازش کنیم.
خوب، سوالی که اینجا پیش میاد اینه که چرا این همه کار اضافی، چرا از Return File استفاده نمی‌کنی؟

```
public ActionResult DownloadTextFile(string fileName)
    {
        var filePath = Path.Combine(HttpContext.Server.MapPath(@"~/Files/"), fileName);
        return File(filePath, "text");
    }
```

یا کلا دلیل استفاده از ActionResult سفارشی چیه؟

جلوگیری از پیچیدگی و تکرار کد
همیشه کار مثل مورد بالا راحت و کم کد! نیست.
به مثال زیر توجه کنید که قراره خروجی CSV بهمون بده.

```
public class CsvActionResult : ActionResult
{
    public IEnumerable ModelListing { get; set; }

    public CsvActionResult(IEnumerable modelListing)
    {
        ModelListing = modelListing;
    }

    public override void ExecuteResult(ControllerContext context)
    {
        byte[] data = new CsvFileCreator().AsBytes(ModelListing);
        var fileResult = new FileContentResult(data, "text/csv")
        {
            FileDownloadName = "CsvFile.csv"
        };
        fileResult.ExecuteResult(context);
    }
```

```
}
```

و نحوه استفاده:

```
public ActionResult ExportUsers()
    {
        IEnumerable<User> model = UserRepository.GetUsers();
        return new CsvActionResult(model);
    }
```

حال فرض کنید بخواهیم همه این کدها رو داخل اکشن متد داشته باشیم، یکم پیچیده میشه و یا فرض کنید کنترلر دیگه‌ای نیاز به خروجی CSV داشته باشه، تکرار کد زیاد میشه.

راحت کردن گرفتن تست واحد از اکشن‌ها متدها

کاربرد ActionResult سفارشی تو تست واحد اینه که وابستگی‌های یک اکشن رو که Mock کردنش سخته می‌بریم داخل ActionResult و هنگام نوشتن تست واحد درگیر کار با اون وابستگی نمی‌شیم.
به مثال زیر توجه کنید که قراره برای اکشن Logout تست واحد بنویسیم
ابتدا بردن وابستگی‌ها به خارج از اکشن به کمک ActionResult سفارشی

```
public class LogoutActionResult : ActionResult
{
    public RedirectToRouteResult ActionAfterLogout { get; set; }
    public LogoutActionResult(RedirectToRouteResult actionAfterLogout)
    {
        ActionAfterLogout = actionAfterLogout;
    }
    public override void ExecuteResult(ControllerContext context)
    {
        FormsAuthentication.SignOut();
        ActionAfterLogout.ExecuteResult(context);
    }
}
```

نحوه استفاده از ActionResult سفارشی

```
public ActionResult Logout()
    {
        var redirect = RedirectToAction("Index", "Home");
        return new LogoutActionResult(redirect);
    }
```

و سپس نحوه تست واحد نوشتن

```
[TestMethod]
    public void The_Logout_Action_Returns_LogoutActionResult()
    {
        //arrange
        var account = new AccountController();

        //act
        var result = account.Logout() as LogoutActionResult;

        //assert
        Assert.AreEqual(result.ActionAfterLogout.RouteValues["Controller"], "Home");
    }
```

خوب به راحتی ما میایم فراخوانی متد SignOut رو از داخل اکشن می‌کشیم بیرون و این کار از اجرای متد SignOut از داخل اکشن متد جلوگیری می‌کنه و همچنین با این کار هنگام تست واحد نوشتن نیاز نیست با Mock کردن کلاس FormsAuthentication سروکار داشته باشیم و فقط کافیه چک کنیم خروجی از نوع LogoutActionResult هست یا خیر و یا می‌تونیم ActionAfterLogout

رو چک کنیم.

منابع و مراجع: + و +

عنوان: UnitTest برای کلاس های Abstract با استفاده از Rhino Mocks
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۱/۱۱/۲۳ ۱۲:۴۵
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing, Rhino Mocks, abstract

در این پست قصد دارم کلاس زیر رو براتون آزمایش کنم:

```
public abstract class myabstractclass
{
    public abstract string dosomething( string input );

    public double round( double number , int decimals )
    {
        return math.round( number , decimals );
    }
}
```

در کلاس بالا که abstract هستش، متدی دارم که abstract است و بدنه‌ای نداره و از متد بعدی به اسم round برای گرد کردن اعداد استفاده می‌شه. برای تست کلاس بالا و اطمینان از درست بودن متدها باید به روش زیر عمل نمود:

روش اول:

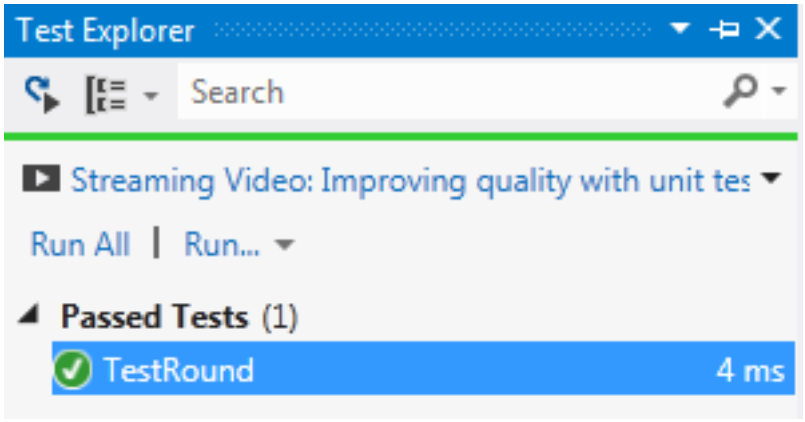
در این روش ابتدا باید کلاسی نوشت تا کلاس abstract بالا رو پیاده سازی کنه:

```
public class mynewclass : myabstractclass
   {
       public override string dosomething( string input )
       {
           return input;
       }
   }
```

بعد می‌شه خیلی راحت برای کلاس دوم متدهای تست رو نوشت. به روش زیر:

```
[testclass]
    public class mytest
    {
        [testmethod]
        public void testround()
        {
            mynewclass mynewclass = new mynewclass();

            var result = mynewclass.round( 5.55 , 1 );

            assert.areequal( 5.6 , result );
        }
    }
```

که بعد از اجرا نتیجه زیر رو خواهید دید



البته روش بالا خیلی مورد پسند من نیست.

در روش دوم که من خیلی بیشتر بهش علاقه دارم دیگه نیازی به استفاده از یک کلاس دوم برای پیاده سازی کلاس abstract نیست. بلکه در این روش از ابزار rhinomocks برای این کار استفاده می‌کنیم . استفاده از rhino mocks به چندین روش امکان پذیره که امروز 2 روش اونو براتون توضیح میدم:
در روش اول از mockrepository استفاده می‌کنیم و در روش دوم از روش aaa یا arrange-act-assert

استفاده از mockrepository :
ابتدا کدهای مربوطه رو می‌نویسم:

```
[testmethod]
       public void testwithmockrepository()
       {
           var mockrepository = new rhino.mocks.mockrepository();
           var mock = mockrepository.partialmock<myabstractclass>();

           using ( mockrepository.record() )
           {
               expect.call( mock.dosomething( arg<string>.is.anything ) ).return( "hi..."
).repeat.once();
           }
           using ( mockrepository.playback() )
           {
               assert.areequal( "hi..." , mock.dosomething( "salam" ) );
           }
       }
```

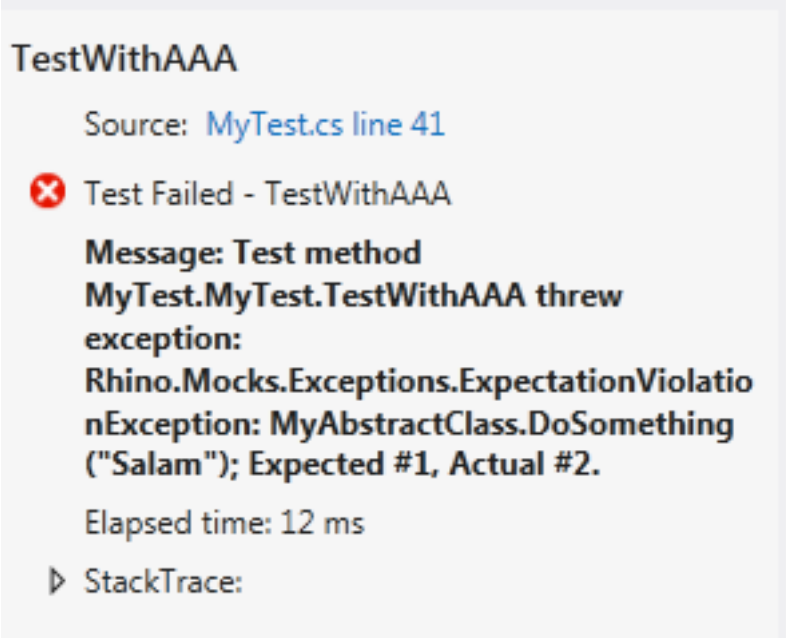
همانطور که در کدهای نوشته شده بالا می‌بینید ابتدا یک mockrepository ساخته شده، سپس از نمونه اون کلاس برای ساخت partialmock کلاس myabstractclass استفاده کردم. در این روش متدهای expect حتما باید بین بلاک record نوشته شوند تا بتونیم از اون‌ها در بلاک playback استفاده کنیم. نتیجه اجرای این متد تست هم مثل متد تست قبلی درست است.
در مورد expect.call باید بگم که از این کلاس برای شبیه سازی رفتار یک متد استفاده میشه (مثلا در مواقعی که یک متد برای انجام عملیات باید به دیتا بیس وصل شده و یک query را اجرا کنه) برای اینکه در تست، از این عملیات صرف نظر بشه از mockها استفاده کرده و رفتار متد رو به روش بالا شبیه سازی می‌کنیم. البته کار کردن با rhino mocks به صورت بالا دیگه از مد افتاده و جدیدا از روش aaa استفاده میشه که اونو در پایین توضیح می‌دم:

```
[testmethod]
      public void testwithaaa()
      {
          var mock = mockrepository.generatepartialmock<myabstractclass>();

          mock.expect( x => x.dosomething( arg<string>.is.anything ) ).return( "hi..."
).repeat.once();//arange

          var result = mock.dosomething( "salam" );//act
```

```
            assert.areequal( "hi..." , result );//assert
        }
```

توی این روش دیگه خبری از record و playback نیست و همانطور که مشخصه از سه مرحله arrange-act-assert تشکیل شده که هر مرحله رو براتون مشخص کردم. مزیت استفاده از این روش اینه که اولا تعداد خطوط کمتری برای کد نویسی نیاز داره و دوما سرعت اجراش از روش قبلی خیلی بیشتره. در مورد repeat.once هم بگم که این دستور نشون می‌ده فقط یک بار اجازه انجام عملیات act رو دارید. یعنی اگر کد هارو به صورت زیر تغییر بدیم با خطا روبرو می‌شیم:

```
[testmethod]
        public void testwithaaa()
        {
            var mock = mockrepository.generatepartialmock<myabstractclass>();

            mock.expect( x => x.dosomething( arg<string>.is.anything ) ).return( "hi..."
).repeat.once();//arange

            var result = mock.dosomething( "salam" );//act
            var result2 = mock.dosomething( "bye" );//act
            assert.areequal( "hi..." , result );//assert
        }
```



خطای آن هم واضح داره میگه که expected#1 هستش در حالی که actual#2 (تعداد دفعات حقیقی از دفعات مورد انتظار بیشتره)

توی پست‌های بعدی (اگه وقت بشه) حتما در مورد rhino mocks بیشتر توضیح میدم

عنوان: **آزمون واحد در MVVM به کمک تزریق وابستگی**
نویسنده: شاهین کیاست
تاریخ: ۱۳۹۲/۰۱/۰۴ ۱۷:۰
آدرس: www.dotnettips.info
برچسب‌ها: MVVM, Unit testing, Dependency Injection

یکی از خوبی‌های استفاده از Presentation Patternها بالا بردن تست پذیری برنامه و در نتیجه نگهداری کد می‌باشد.
MVVM الگوی محبوب برنامه نویسان WPF و Silverlight می‌باشد. به صرف استفاده از الگوی MVVM نمی‌توان اطمینان داشت که ViewModel کاملا تست پذیری داریم. به عنوان مثلا اگر در ViewModel خود مستقیما DialogBox کنیم یا ارجاعی از View دیگری داشته باشیم نوشتن آزمون‌های واحد تقریبا غیر ممکن می‌شود. قبلا درباره‌ی این مشکلات و راه حل آن مطلب در سایت منتشر شده است :

- MVVM و نمایش دیالوگ‌ها

در این مطلب قصد داریم سناریویی را بررسی کنیم که ViewModel از Background Worker جهت انجام عملیات مانند دریافت داده‌ها استفاده می‌کند.

Background Worker کمک می‌کند تا اعمال طولانی در یک Thread دیگر اجرا شود در نتیجه رابط کاربری Freeze نمی‌شود.

به این مثال ساده توجه کنید :

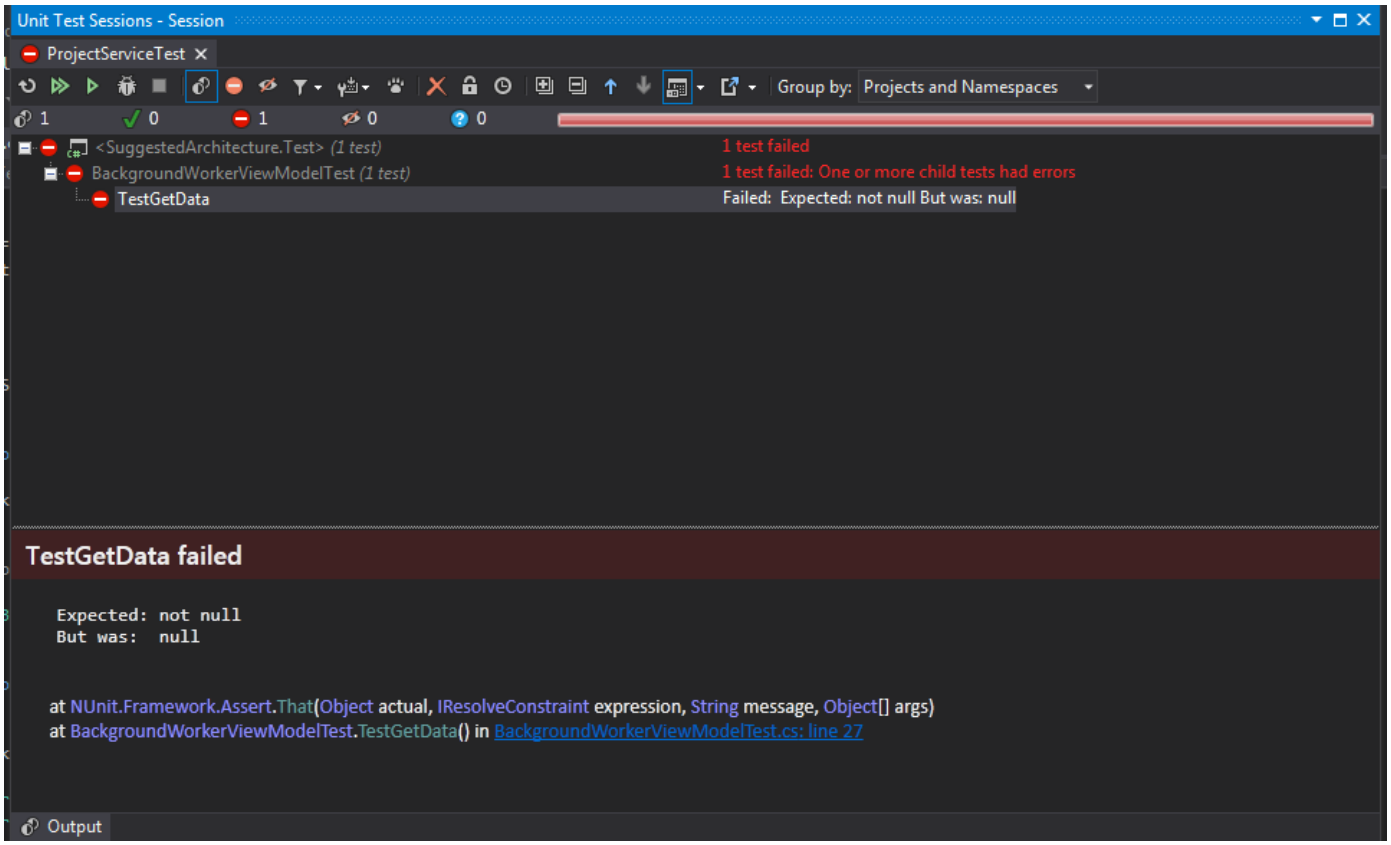
```
public class BackgroundWorkerViewModel : BaseViewModel
    {
        private List<string> _myData;

        public BackgroundWorkerViewModel()
        {
            LoadDataCommand = new RelayCommand(OnLoadData);
        }

        public RelayCommand LoadDataCommand { get; set; }

        public List<string> MyData
        {
            get { return _myData; }
            set
            {
                _myData = value;
                RaisePropertyChanged(() => MyData);
            }
        }

        public bool IsBusy { get; set; }

        private void OnLoadData()
        {
            var backgroundWorker = new BackgroundWorker();
            backgroundWorker.DoWork += (sender, e) =>
                                {
                                    MyData = new List<string> {"Test"};
                                    Thread.Sleep(1000);
                                };
            backgroundWorker.RunWorkerCompleted += (sender, e) => { IsBusy = false; };
            backgroundWorker.RunWorkerAsync();
        }
    }
```

در این ViewModel با اجرای دستور LoadDataCommand داده‌ها از یک منبع داده دریافت می‌شود. این عمل می‌تواند چند ثانیه طول بکشد ، در نتیجه برای قفل نشدن رابط کاربر این عمل را به کمک Background Worker به صورت Async در پشت صحنه انجام شده است.
آزمون واحد این ViewModel اینگونه خواهد بود :

```
[TestFixture]
```

```
public class BackgroundWorkerViewModelTest
{
    #region Setup/Teardown

    [SetUp]
    public void SetUp()
    {
        _backgroundWorkerViewModel = new BackgroundWorkerViewModel();
    }

    #endregion

    private BackgroundWorkerViewModel _backgroundWorkerViewModel;

    [Test]
    public void TestGetData()
    {

        _backgroundWorkerViewModel.LoadDataCommand.Execute(_backgroundWorkerViewModel);

        Assert.NotNull(_backgroundWorkerViewModel.MyData);
        Assert.IsNotEmpty(_backgroundWorkerViewModel.MyData);
    }
}
```

با اجرای این آزمون واحد نتیجه با آن چیزی که در زمان اجرا رخ می‌دهد متفاوت است و با وجود صحیح بودن کدها آزمون واحد شکست می‌خورد.
چون Unit Test به صورت همزمان اجرا می‌شود و برای عملیات‌های پشت صحنه صبر نمی‌کند در نتیجه این آزمون واحد شکست می‌خورد.



یک راه حل تزریق BackgroundWorker به صورت وابستگی به ViewModel می‌باشد. همانطور که قبلا اشاره شده یکی از مزایای استفاده از تکنیک‌های تزریق وابستگی سهولت Unit testing می‌باشد.

در نتیجه یک Interface عمومی و 2 پیاده سازی همزمان و غیر همزمان جهت استفاده در برنامه‌ی واقعی و آزمون واحد تهیه می‌کنیم :

```
public interface IWorker
    {
        void Run(DoWorkEventHandler doWork);
        void Run(DoWorkEventHandler doWork, RunWorkerCompletedEventHandler onComplete);
    }
```

جهت استفاده در برنامه‌ی واقعی :

```
public class AsyncWorker : IWorker
    {
        public void Run(DoWorkEventHandler doWork)
        {
            Run(doWork, null);
        }

        public void Run(DoWorkEventHandler doWork, RunWorkerCompletedEventHandler onComplete)
        {
            var backgroundWorker = new BackgroundWorker();
            backgroundWorker.DoWork += doWork;
            if (onComplete != null)
                backgroundWorker.RunWorkerCompleted += onComplete;
            backgroundWorker.RunWorkerAsync();


        }
    }
```

جهت اجرا در آزمون واحد :

```
public class SyncWorker : IWorker
    {
        #region IWorker Members

        public void Run(DoWorkEventHandler doWork)
        {
            Run(doWork, null);
        }

        public void Run(DoWorkEventHandler doWork, RunWorkerCompletedEventHandler onComplete)
        {
            Exception error = null;
            var doWorkEventArgs = new DoWorkEventArgs(null);
            try
            {
                doWork(this, doWorkEventArgs);
            }
            catch (Exception ex)
            {
                error = ex;
                throw;
            }
            finally
            {
                onComplete(this, new RunWorkerCompletedEventArgs(doWorkEventArgs.Result, error,
doWorkEventArgs.Cancel));
            }
        }

        #endregion
    }
```

در نتیجه ViewModel اینگونه تغییر خواهد کرد :

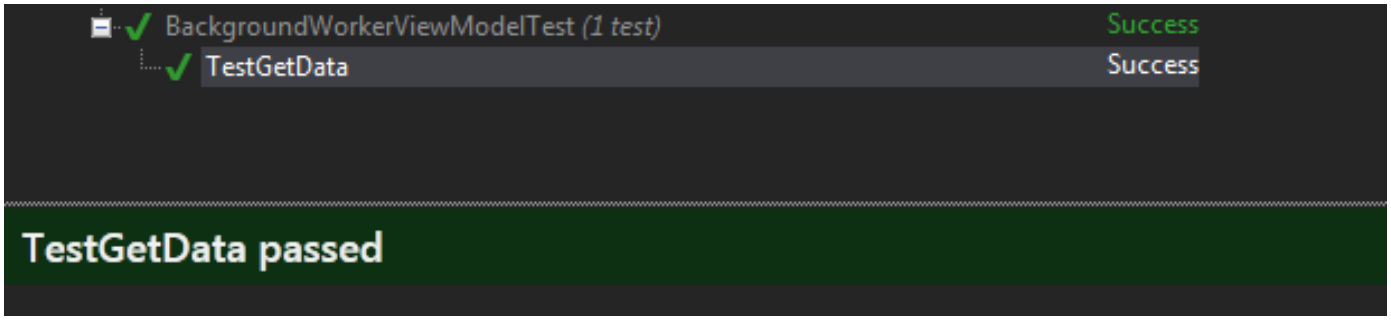
```
public class BackgroundWorkerViewModel : BaseViewModel
    {
        private readonly IWorker _worker;
        private List<string> _myData;
```

```
        public BackgroundWorkerViewModel(IWorker worker)
        {
            _worker = worker;
            LoadDataCommand = new RelayCommand(OnLoadData);
        }

        public RelayCommand LoadDataCommand { get; set; }

        public List<string> MyData
        {
            get { return _myData; }
            set
            {
                _myData = value;
                RaisePropertyChanged(() => MyData);
            }
        }

        public bool IsBusy { get; set; }

        private void OnLoadData()
        {
            IsBusy = true; // view is bound to IsBusy to show 'loading' message.

            _worker.Run(
                (sender, e) =>
                    {
                        MyData = new List<string> {"Test"};
                        Thread.Sleep(1000);
                    },
                (sender, e) => { IsBusy = false; });
        }
    }
```

کلاس مربوطه به آزمون واحد را مطابق با تغییرات ViewModel :

```
[TestFixture]
    public class BackgroundWorkerViewModelTest
    {
        #region Setup/Teardown

        [SetUp]
        public void SetUp()
        {
            _backgroundWorkerViewModel = new BackgroundWorkerViewModel(new SyncWorker());
        }

        #endregion

        private BackgroundWorkerViewModel _backgroundWorkerViewModel;

        [Test]
        public void TestGetData()
        {

            _backgroundWorkerViewModel.LoadDataCommand.Execute(_backgroundWorkerViewModel);

            Assert.NotNull(_backgroundWorkerViewModel.MyData);
            Assert.IsNotEmpty(_backgroundWorkerViewModel.MyData);
        }
    }
```

اکنون اگر Unit Test را اجرا کنیم نتیجه اینگونه خواهد بود :


BackgroundWorkerViewModelTest (1 test)
Success
TestGetData
Success
TestGetData passed

عنوان: **WatIn - Web Application Testing in .Net**
نویسنده: آرش خوشبخت
تاریخ: ۱۳۹۲/۰۱/۲۴ ۲۱:۵
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing, Integration tests, test

معرفی:

امروزه تست کردن کدها به دلیل وجود ابزارهای مختلف زیادی، کار آسانی شده است. اما بعضی‌ها در web application ها، یکی از تست‌هایی را که خیلی هم مهم است را فراموش می‌کنند که آن هم تست UI است. شما را در این مقاله با یکی از روش‌های خوب تست UI آشنا خواهم کرد. ابزارهای زیادی برای تست UI وجود دارد که کار کردن با آنها نه تنها زمان بر بلکه بسیار خسته کننده می‌باشند و به خاطر همین خیلی‌ها از انجام تست UI صرف نظر می‌کنند.

WatIn چیست؟

WatIn مخفف Web Application Testing in .Net می‌باشد؛ که یک فریم ورک تست web applicationها است. WatIn این اجازه را به شما می‌دهد که با استفاده از IE ویا FireFox عناصر داخل صفحات را مقدار دهی کنید و یا حتی رویدادی را برای عناصر فراخوانی کنید.

شروع کار با WatIn:

در زیر یک نمونه از کار با WatIn را می‌توانید مشاهده کنید:

```
[TestMethod]
public void SearchForWatiNOnGoogle()
{
  using (var browser = new IE("http://www.google.com"))
  {
    browser.TextField(Find.ByName("q")).TypeText("WatiN");
    browser.Button(Find.ByName("btnG")).Click();
    Assert.IsTrue(browser.ContainsText("WatiN"));
  }
}
```

WatIn یک فریم ورک کاربر پسند است و در ادامه متوجه می‌شوید که استفاده از این فریم ورک چه مزایایی دارد. برای نصب، WatIn را می‌توانید از اینجا دانلود کنید ویا اگر خواستید می‌توانید با NuGet هم این فریم ورک را دانلود کرده و نصب نمایید. برای شروع کار با Watin باید reference هایی را به پروژه تان اضافه کنید که یکی از این referenceها WatiN.Core.dll می‌باشد و برای استفاده از IE ویا FireFox باید فضای نام Watin.Core را اضافه کنیم. Watin چند فضای نام دیگری را هم به همراه دارد که در زیر به توضیح مختصری از آن‌ها می‌پردازیم:

1- **Watin.Core.DialogHandlers:** این فضای نام این امکان را به شما می‌دهد تا دیالوگ هایی را که مرورگر می‌تواند به کاربر نمایش دهد، مدیریت کنید. از handlerهای این فضای نام، AlertDialogHandler، ConfirmDialogHandler، FileUploadDialogHandler، PrintDialogHandler و LoginDialogHandler می‌باشد.

2- **Watin.Core.Exceptions:** این فضای نام دارای یک سری exception می‌باشد و این امکان را به ما می‌دهد تا یک سری رفتارهای ناخواسته را کنترل کنیم. بعضی از این exceptionها، IENotFoundException، ElementNotFoundException و TimeoutException و WatinException می‌باشد.

3- **Watin.Core.Logging:** این فضای نام کلاس هایی را در اختیار ما می‌گذارد تا بتوانیم عملیاتی را که در کدمان انجام می‌دهیم log کنیم.

مثالی از watin که در بالا نشان دادیم به این صورت عمل می‌کند که مرورگر IE را باز کرده و به سایت google خواهد رفت. در این صفحه جعبه متنی یا TextBox با نام "q" را پیدا کرده و عبارت "Watin" را در آن تایپ می‌کند و همچنین Buttonی با نام "btnG" پیدا کرده و آن را کلیک می‌نماید و در آخر بررسی می‌کند که در مرورگر متنی شامل WatIn وجود دارد یا خیر.

مشاهده کردید که به همین سادگی یک تست UI نوشتیم. به نظر شما جالب نبود؟ فرض کنید که اگر می‌خواستید با مثلا Microsoft Test Manager این کار را انجام دهید چه دردسرهایی را باید تحمل می‌کردید. حالا تست UI برای همه برنامه نویس‌ها جذاب خواهد شد.

به جای مثال بالا می‌توانیم به صورت زیر هم عمل کنیم:

```
[TestMethod]
public void SearchForWatiNOnGoogle()
{
  using (var browser = new IE("http://www.google.com"))
  {
    browser.TextField(Find.ByName("q")).Value="WatiN";
    browser.Button(Find.ByName("btnG")).ClickNoWait();
    Thread.Sleep(3000);
    Assert.IsTrue(browser.ContainsText("WatiN"));
  }
}
```

تفاوت کد دوم با کد اول این است چون در کد اول از متد TypeText استفاده کردیم یک مقدار سرعت تست را پایین می‌آورد ولی اگر از Value ویا از SetAttribute استفاده کنیم دیگر عمل تایپ را انجام نداده و مقدار را مستقیما در مقدار TextField قرار می‌دهد. شاید بپرسید چرا بعد از متد ClickNoWait چند ثانیه صبر می‌کنم؟ چون صفحه برای اینکه بارگذاری شود و نتیجه جستجو را نشان دهد کمی طول کشیده و Assert.IsTrue شما Failed می‌شود. البته به جای Thread.Sleep می‌توانیم از متدهای مربوط به Watin هم استفاده کنیم مانند WaitUntilComplete ویا از WaitUntilContainsText.

نظرات خوانندگان

نویسنده: آرش خوشبخت
تاریخ: ۱۳۹۲/۰۱/۲۴ ۲۱:۳۰

اگر سوالی یا مشکلی در کارکردن با این فریم ورک داشتین می‌توانید اینجا بپرسین

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:

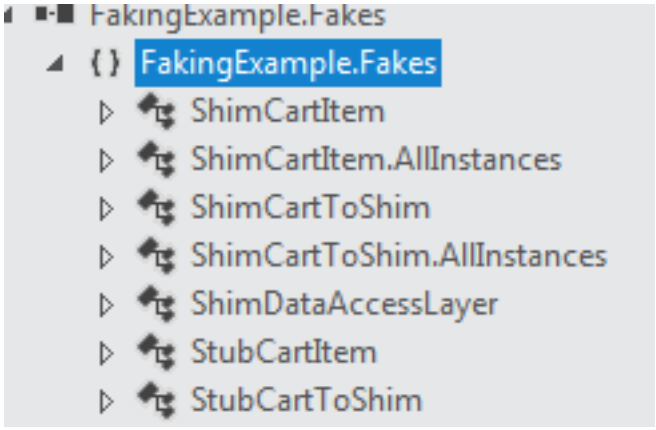
```
namespace FakingExample
{
    public class CartToShim
    {
        public int CartId { get; private set; }
        public int UserId { get; private set; }
        private List<CartItem> _cartItems = new List<CartItem>();
        public ReadOnlyCollection<CartItem> CartItems { get; private set; }
        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: **Microsoft Test Manager - قسمت اول**
نویسنده: آرش خوشبخت
تاریخ: ۱۳۹۲/۰۱/۳۱ ۲۳:۵۵
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing, Integration tests, test, Testing, Test Manager 2012, sprint, Test Case, Test Plan

مقدمه:

مدیریت آزمون مایکروسافت یا Microsoft Test Manager یک ابزار تست نویسی است که به تسترها این اجازه را می‌دهد تا بتوانند برای UI برنامه‌های خود یا sprintهای پروژه خود تست بنویسند. این ابزار برای نوشتن آزمون‌های پیشرفته و مجتمع سازی مدیریت طرح‌های تست یا test plans همراه با موردهای تست یا test case در طول توسعه برنامه است. یکی از مزایایی که این ابزار دارد این است که در طول انجام تست می‌توانید اشکالات تست را ثبت کنید و هم چنین می‌توانید شرحی در مورد انجام تست یا اشکالی که در آن تست وجود دارد، ثبت کنید. همچنین می‌توانید گزارشی از تست هایی که انجام داده اید و پاس شدن یا پاس نشدن تست‌ها و تاریخ انجام آن‌ها را نیز مشاهده کنید. قبل از کار با نرم افزار MTM باید یک سری مطالب مهم را در مورد انجام تست و مفهوم Agile بدانیم.

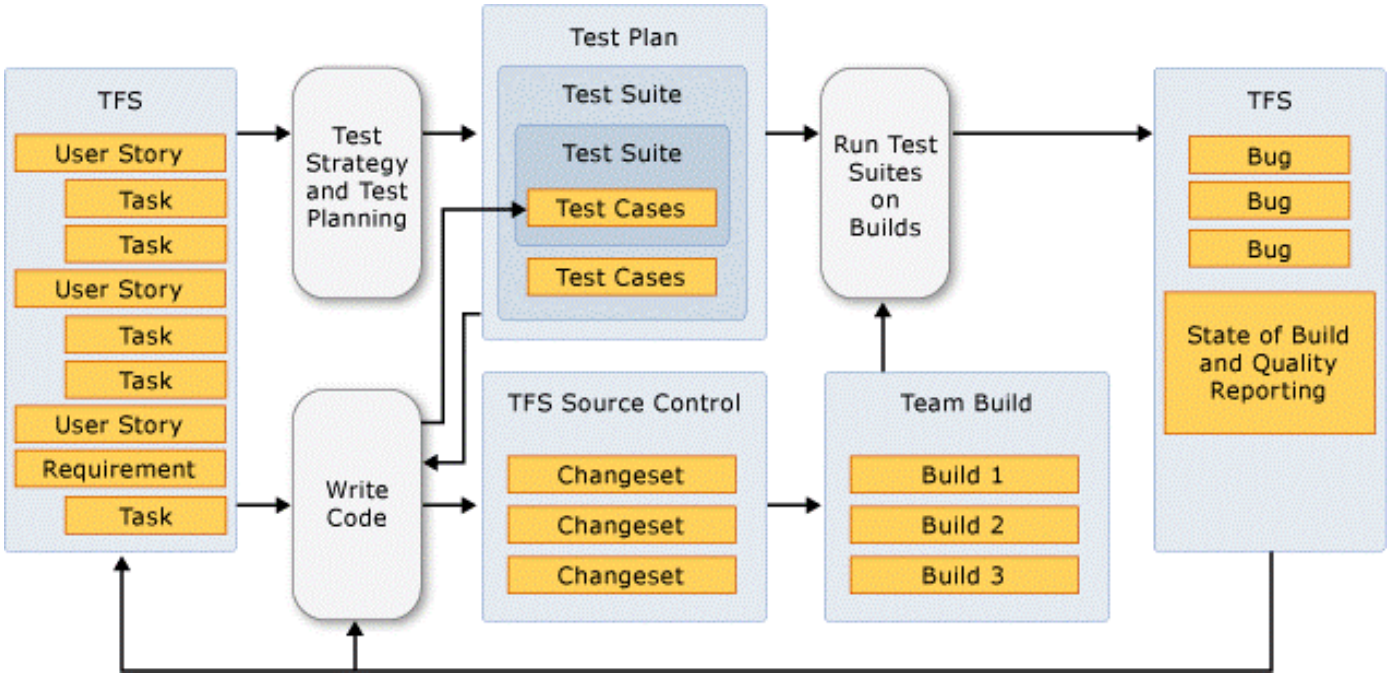
استراتژی تست:

زمانی که شما تست Agile را معرفی می‌کنید تیم برنامه نویسی شما می‌تواند بر روی تست‌های شما هم در سطح sprint و هم در سطح پروژه تمرکز کنند. تست در سطح sprint شامل تست هایی می‌شود که همه user story ها در بر بگیرد یعنی در واقع همان تست‌های واحد شما می‌شود. در سطح پروژه هم شامل تست هایی می‌شود که چندین sprint را در بر می‌گیرد که در واقع می‌توان تست‌های integrated گفت. بهتر است زمانی که تیم برنامه نویسی کدنویسی می‌کنند شما طرح تست‌های خود را بسازید و برای انجام تست کاملا آماده باشید. این تست‌ها شامل تست واحد، تست performance ، تست امنیتی و تست usability و غیره می‌باشد.

برای آماده کردن تست Agile در ابتدا شما باید یک تاریخچه یا history از برنامه یا سیستم خود داشته باشید. شما می‌توانید با استفاده از Microsoft Test Manager طرح تست خود را برای هر یک از sprint های پروژتان بسازید و موردهای تست را مشخص کنید.

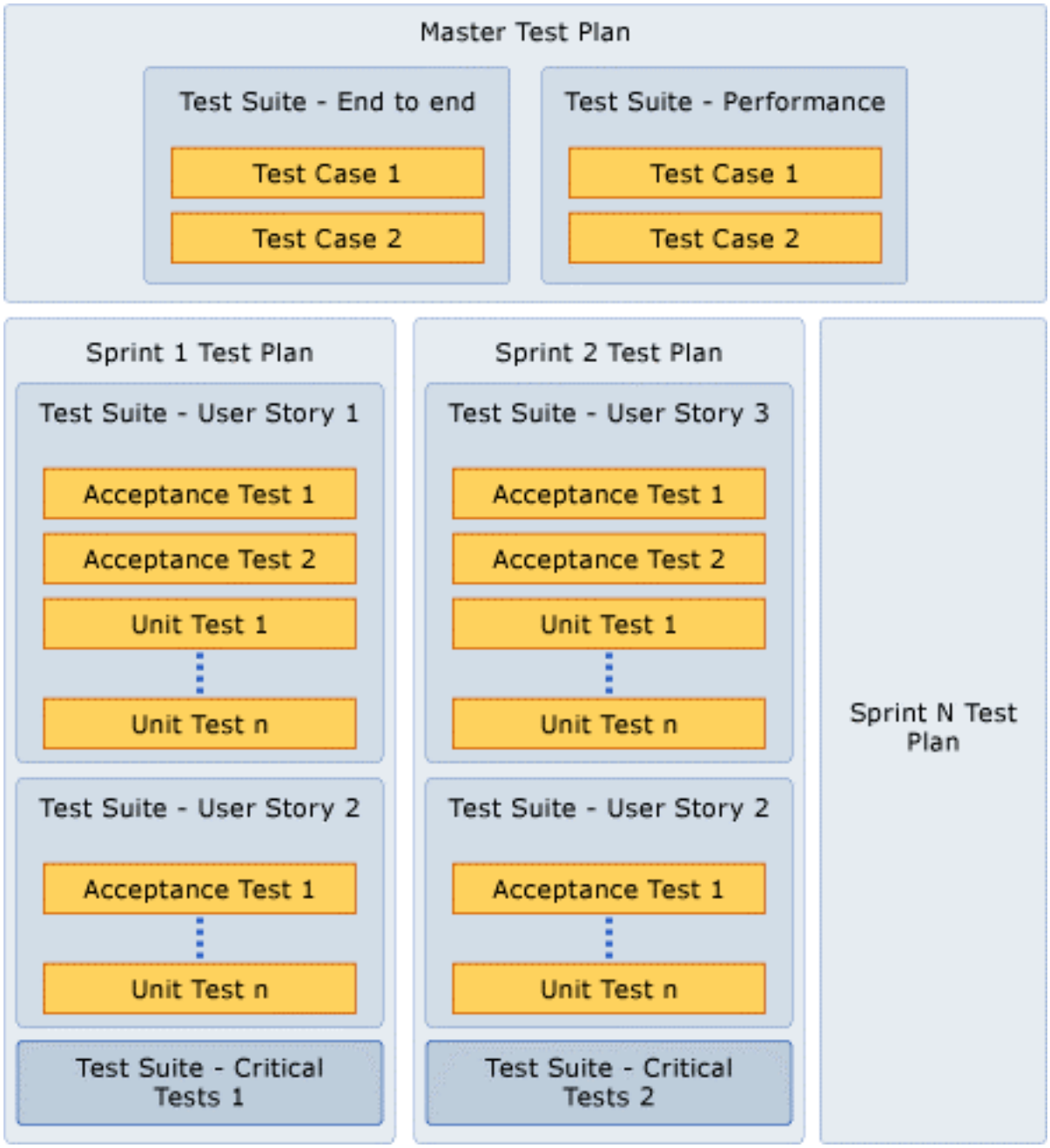
سپس باید کدهایی که برنامه نویسان می‌نویسند قابلیت تست را داشته باشند و شما به عنوان یک تستر باید آشنایی کاملی از ساختار و الگوهای برنامه تان داشته باشید.

تست یک فرآیند تکراری می‌باشد که همزمان با اجرای پروژه تان صورت می‌گیرد در زیر می‌توانید فرآیند کار تست و انجام کدنویسی را مشاهده نمایید:



: Test Planning

Test Planning فرآیندی است که به تیم شما کمک می‌کند تا درک درستی از پروژه داشته باشند و همچنین تیم را برای انجام هر گونه تستی آماده کند. تست Agile در سطح Sprint انجام می‌شود که در هر Sprint تیم شما تست هایی را ایجاد می‌کنند تا user story هایی که در هر Sprint وجود دارد، مورد بررسی قرار گیرند. در شکل زیر قالبی از test plan های شما در یک پروژه را نمایش می‌دهد:



البته این قالب‌ها بر اساس سلیقه شخصی است اما در کل می‌توانیم قالب تست را به صورت بالا در نظر بگیریم.

همیشه باید این را در نظر داشته باشیم که در طول هر sprint حتما باید تست‌ها را اجرا کرده و در صورت وجود خطا، آن خطا را رفع کنیم تا در مراحل بالاتر با مشکلی مواجه نشویم. در قسمت بعد با Microsoft Test Manager و روش‌های نوشتن sprint و تست‌ها آشنا خواهیم شد.

نظرات خوانندگان

نویسنده: سیروان عفیفی
تاریخ: ۱۳۹۲/۰۲/۰۱ ۰:۵

با تشکر از شما، مطلب خوبی بود.

نویسنده: آرش خوشبخت
تاریخ: ۱۳۹۲/۰۲/۰۱ ۸:۶

خواهش می‌کنم امیدوارم مطالبم خوب نوشته شده باشه چون در نوشتن کمی ضعیف هستم

نویسنده: مهدی
تاریخ: ۱۳۹۲/۰۲/۰۳ ۱۹:۱۱

با سلام خدمت دوست عزیز و تشکر از این مقاله مفید.
لطفا اگر می‌شود در مورد اصطلاحاتی که بیان می‌کنید در اول مقاله یه تعریفی از ان‌ها بیان کنید.
با تشکر.

نویسنده: آرش خوشبخت
تاریخ: ۱۳۹۲/۰۲/۰۳ ۲۱:۱۲

خواهش می‌کنم ولی منظور شما کدام اصطلاحات است؟ چون در قسمت دوم خیلی‌های این اصطلاحات رو گفتم اگر اصطلاحی رو متوجه نشدین بگین تا واستون توضیح بدم

عنوان: **Microsoft Test Manager - قسمت دوم**
نویسنده: آرش خوشبخت
تاریخ: ۱۳۹۲/۰۲/۰۱ ۲۳:۱۵
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing, Integration tests, test, Testing, Test Manager 2012, Test Case, Test Plan, Test Suite

تا اینجا متوجه شدیم که test plan چیست و چگونه ساخته می‌شود و برای نوشتن تست‌ها چه مراحلی را باید طی کنیم. در این مطلب قصد بر این است که آموزش نوشتن تست‌ها با استفاده از MTM را آموزش دهیم. در این آموزش فرض بر این است که شما آشنایی کمی با محیط این ابزار، نیازمندی‌ها و Story ها، اشکالات یا Bug ها و Task ها دارید.

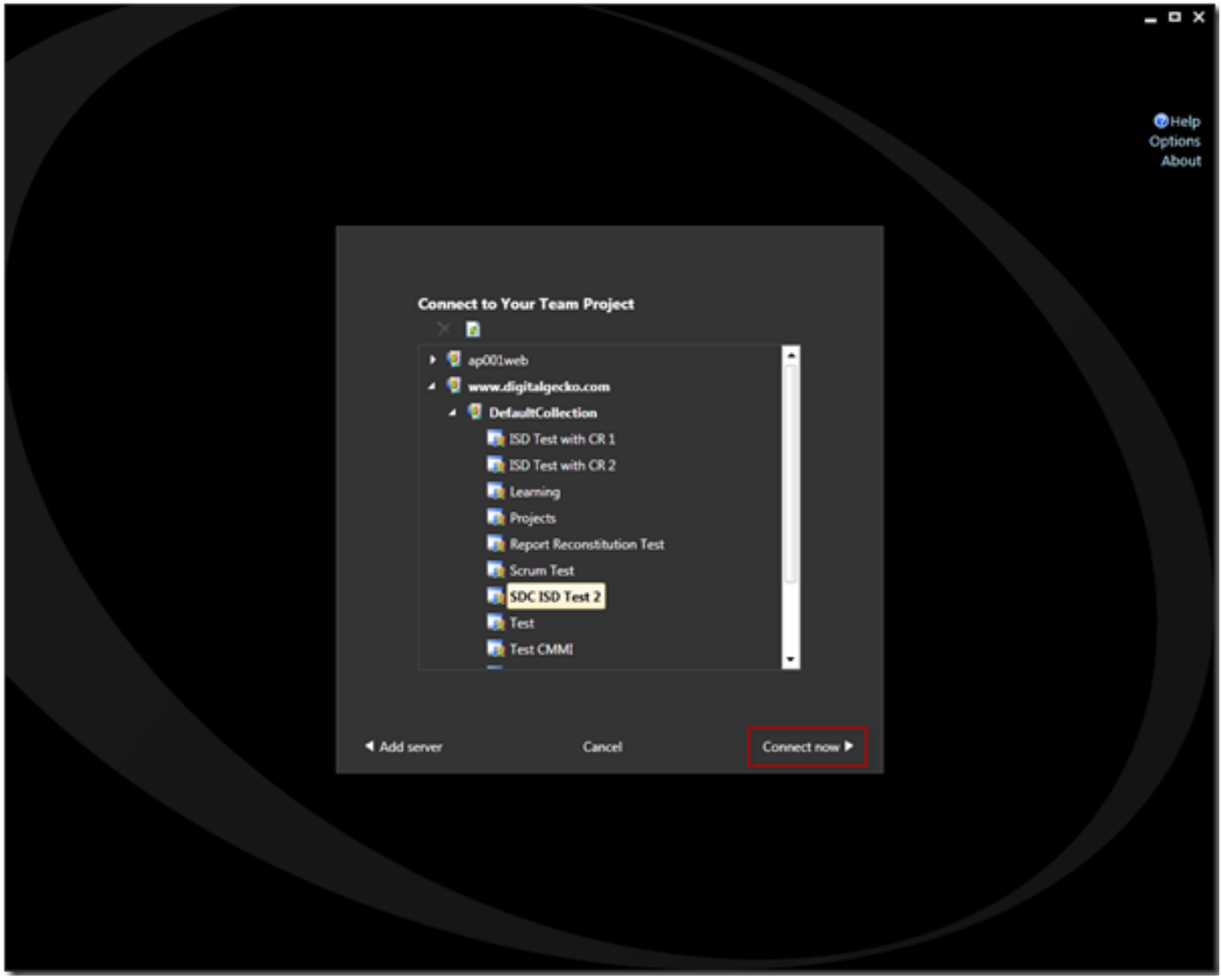
در MTM سه لایه وجود دارد:

1- **Test Plan :** شما در آغاز کار با MTM ابتدا باید Test Plan خود را ایجاد کنید.

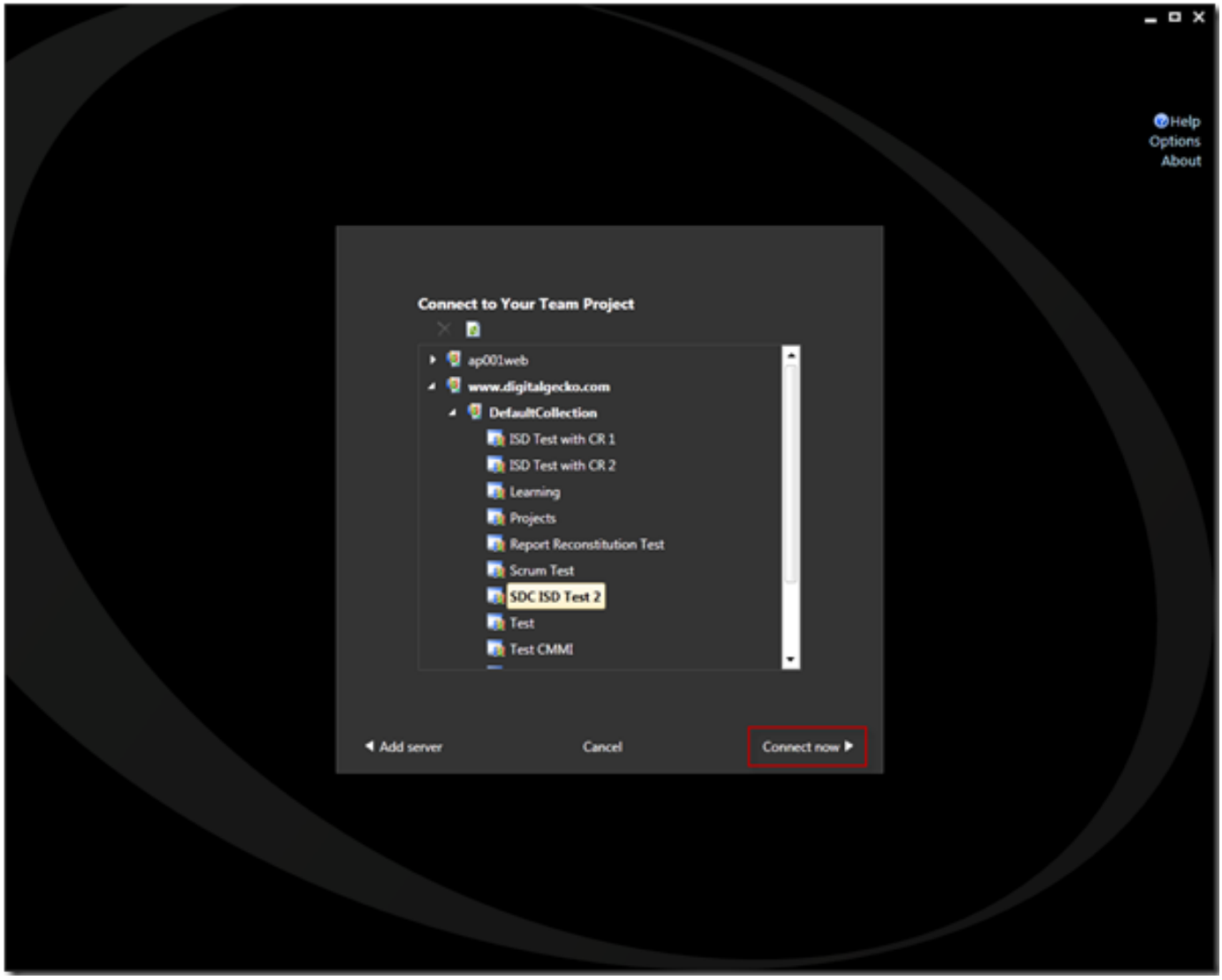
2- **Test Suite :** در هر Test Plan شما می‌توانید چندین Test Suite ایجاد کنید.

3- **Test Case :** هر Test Suite از چندین Test Case ترکیب شده است.

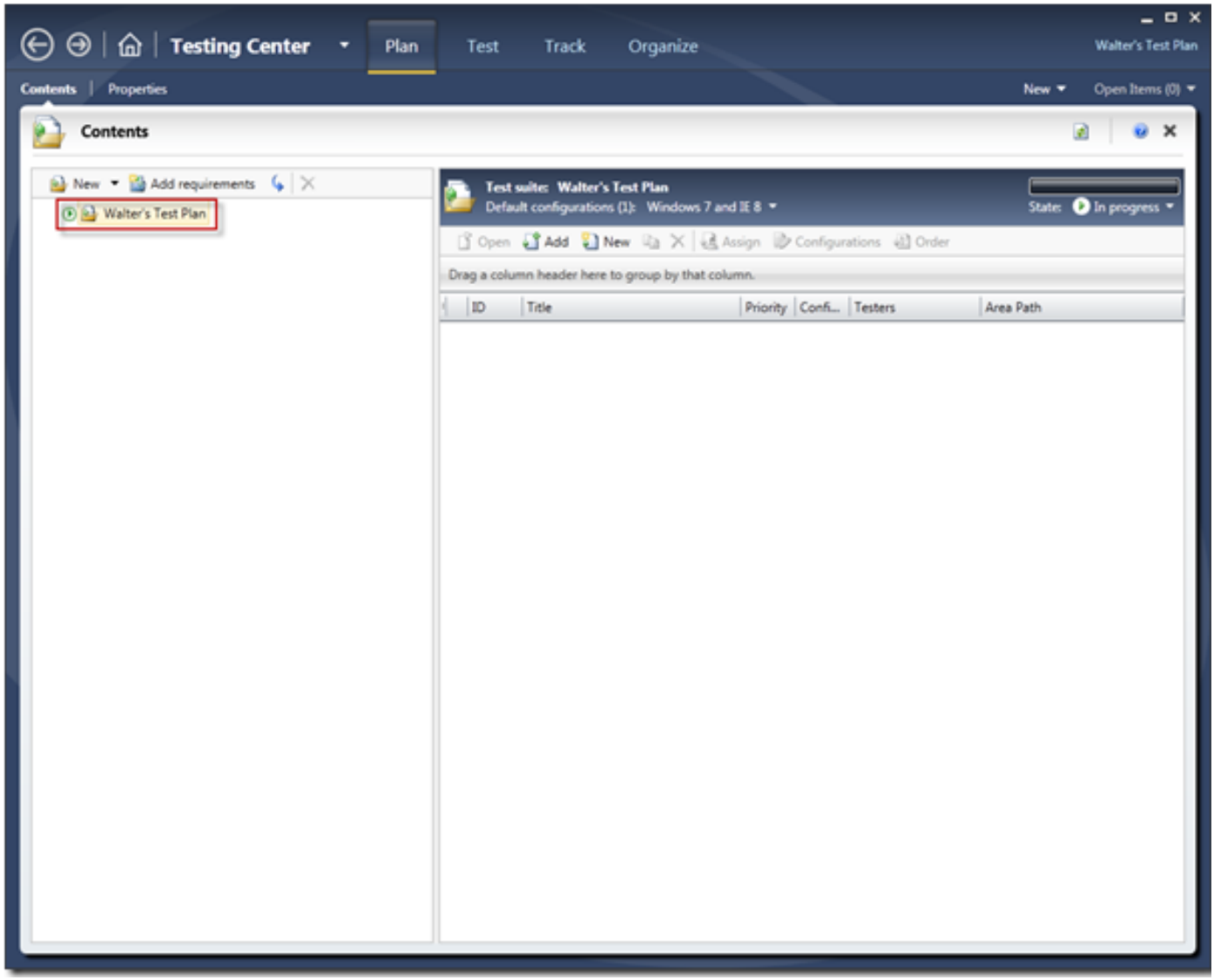
برای اولین بار که شما MTM را اجرا می‌کنید باید team project ی را که قرار است برای آن تست بنویسید را انتخاب کنید.
می‌توانید در زیر نمایی از MTM و اتصال به team project را مشاهده کنید:



بعد از اینکه پروژه خود را انتخاب کردید، می‌توانید لیستی از طرح‌های تست تان که برای این پروژه ایجاد کرده اید را مشاهده کنید که می‌توانید از این لیست یک طرحی را انتخاب نمایید و یا یک طرح جدید را ایجاد کنید همانطور که در شکل زیر مشاهده می‌کنید.



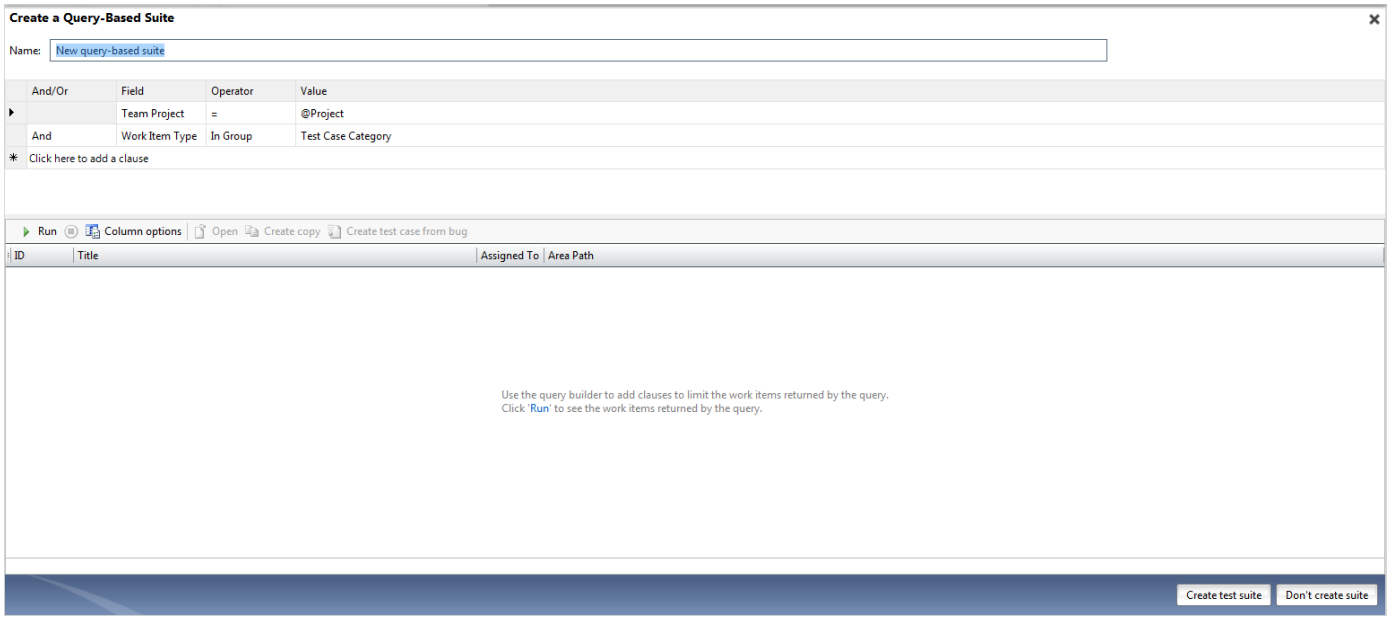
وقتی plan یا طرحی را انتخاب می‌کنید به صفحه testing center وارد می‌شوید که به صورت پیش فرض در کاربرگ plan و بخش contents قرار دارید.



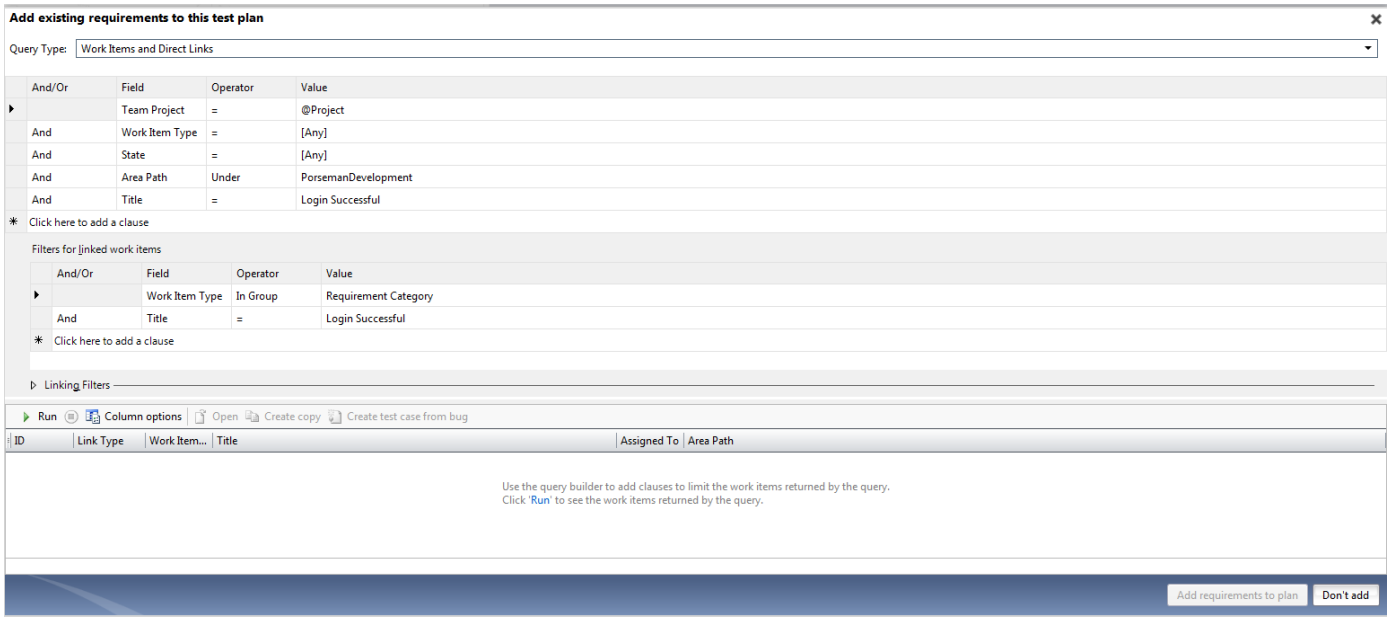
همانطور که در تصویر بالا مشاهده می‌کنید و در سمت چپ پنجره، plan شما در ریشه قرار دارد و test suite هایی را که ایجاد می‌کنید به عنوان فرزندان plan تان قرار می‌گیرند. در سمت راست test caseهای شما قرار می‌گیرند که با توجه به test suite ی که شما در سمت چپ انتخاب کرده اید test caseهای مربوط به آن در سمت راست قابل مشاهده است. برای ایجاد test suite به plan تان، باید روی plan راست کلیک کرده و گزینه new suite را انتخاب کنید و برای آن عنوانی را وارد می‌کنید. وقتی روی plan راست کلیک می‌کنید پند گزینه وجود دارد که می‌توانید با توجه به کارتان این گزینه‌ها را انتخاب کنید:

-1 وقتی new suite را انتخاب می‌کنید یک suite خالی برای شما ایجاد می‌کند.

-2 وقتی گزینه new query-based suite را انتخاب می‌کنید این اجازه را به شما می‌دهد که از test caseهای موجود در پروژه خود یک یا چندین مورد تست را انتخاب نمایید که پنجره ای مانند زیر باز می‌شود که می‌توانید با اعمال فیلتر، test caseهای موجود در پروژه را پیدا و یک یا چندین مورد را به suite خود اضافه نمایید.



3- گزینه add requirement to plan این اجازه را به شما می‌دهد تا بتوانید از planهای موجود در TFS تان استفاده نمایید. بعد از انتخاب این گزینه پنجره ای مشابه تصویر بالا باز می‌شود که می‌توانید با اعمال فیلتر موردهای تست را پیدا کرده و به آن بیافزاید.



4- با انتخاب گزینه copy suite from another plan همانطور که از اسمش پیداست می‌توانید از suiteهای مربوط به planهای دیگر کپی برداری کنید.

نظرات خوانندگان

نویسنده: علیرضا پونه
تاریخ: ۱۳۹۲/۰۲/۰۲ ۸:۴۹

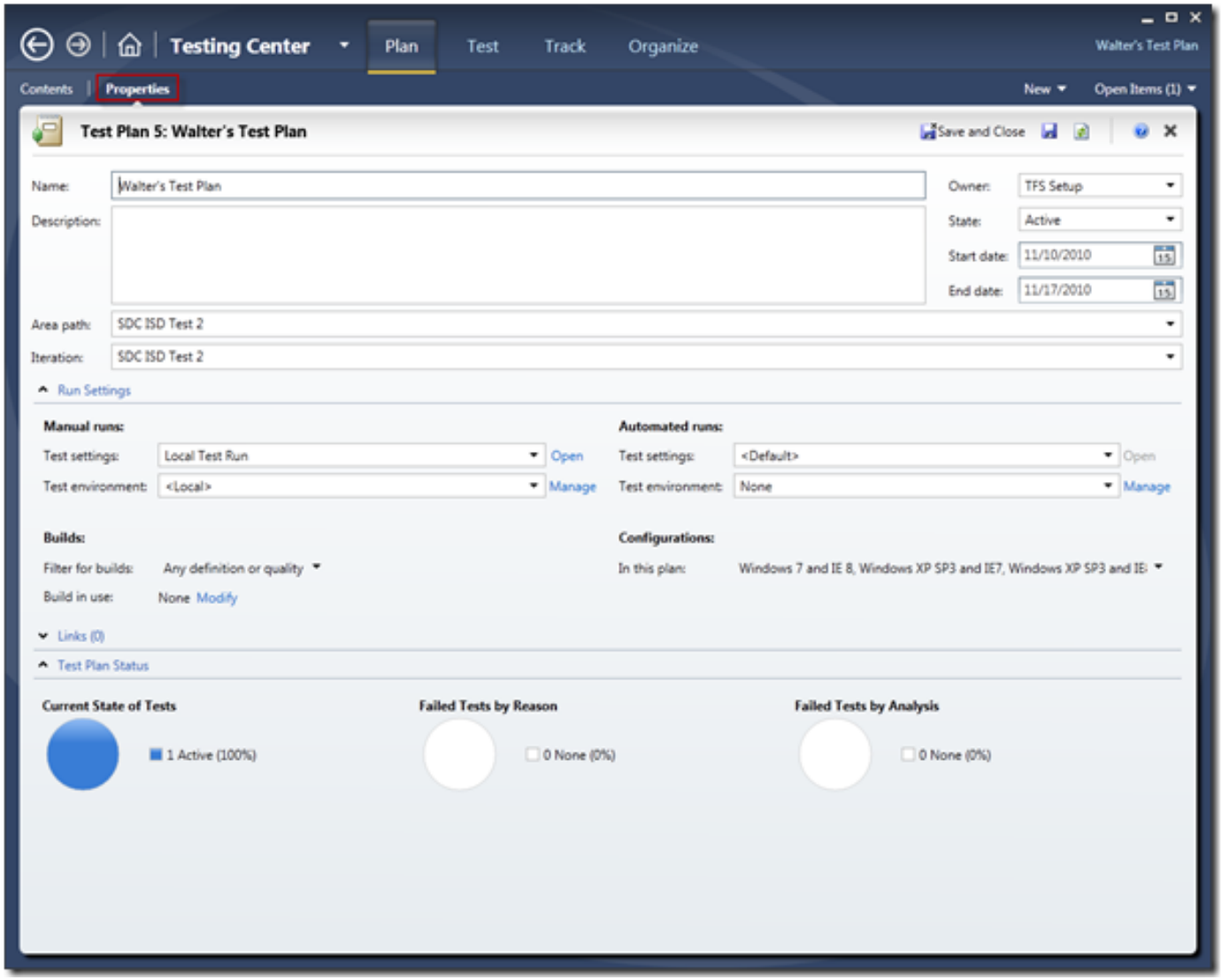
ممنونم. فقط اینکه تو هر پست مطلب رو کاملتر و قسمت بیشتری رو بگین تا در تعداد پست کمتری بشه همه چیز رو گفت و هم اینکه خواننده تا پست بعدی، خیلی از مطلب دور نشه. بازم بابت مطلب بسیار مهمی که دارین آموزش میدین خیلی خیلی ممنون.

نویسنده: آرش خوشبخت
تاریخ: ۱۳۹۲/۰۲/۰۲ ۱۲:۱

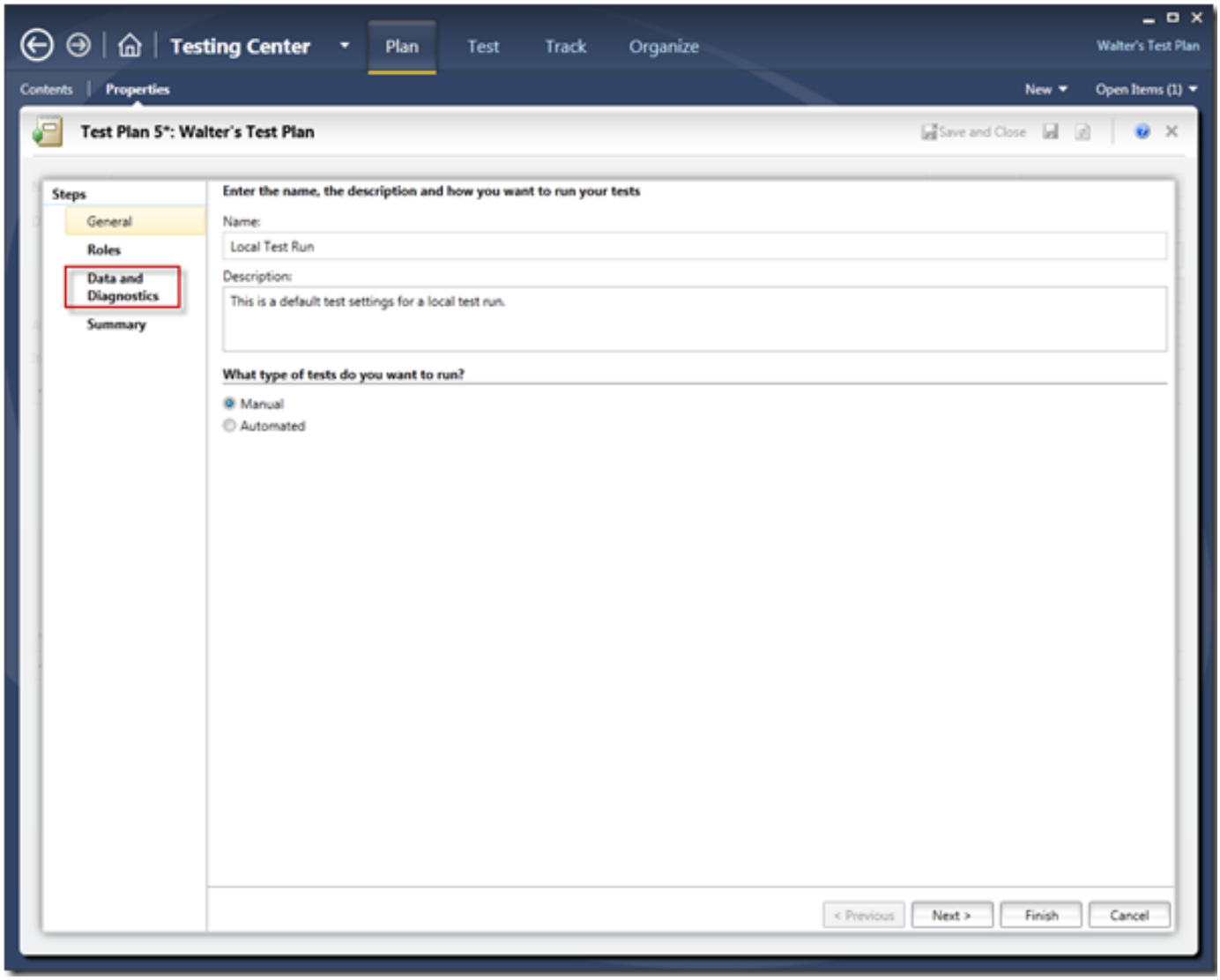
دلیل اینکه مطلب زیاد نمیزارم چون می‌گم شاید کاربران خسته شن یا حوصله‌ی خوندن مطلب زیاد رو نداشته باشن و بعد اینکه مبحث جدیدی که بخواد شروع بشه مجمبورم قسمت قبل رو قطع کنم قسمت بعدی در مورد یک سری تنظیمات در MTM است و ربطی به این بخش نداره

عنوان: **Microsoft Test Manager - قسمت سوم**
نویسنده: آرش خوشبخت
تاریخ: ۱۳۹۲/۰۲/۰۳ ۲۳:۵۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing, Integration tests, test, Testing, Test Manager 2012, Test Case, Test Plan, Test Suite

در کنار کاربرگ contents کاربرگی با نام Properties وجود دارد که می‌توانید یک سری تنظیمات را برای plan خود انجام دهید. این تنظیمات از قبیل تغییر عنوان plan ، تعیین مسیر پروژه، تاریخ شروع و پایان، کاربری که مالک این plan است، وضعیت جاری تست‌های plan و تعیین مرورگر و ویندوز نیز می‌باشد که می‌توانید در تصویر زیر آن را مشاهده کنید.

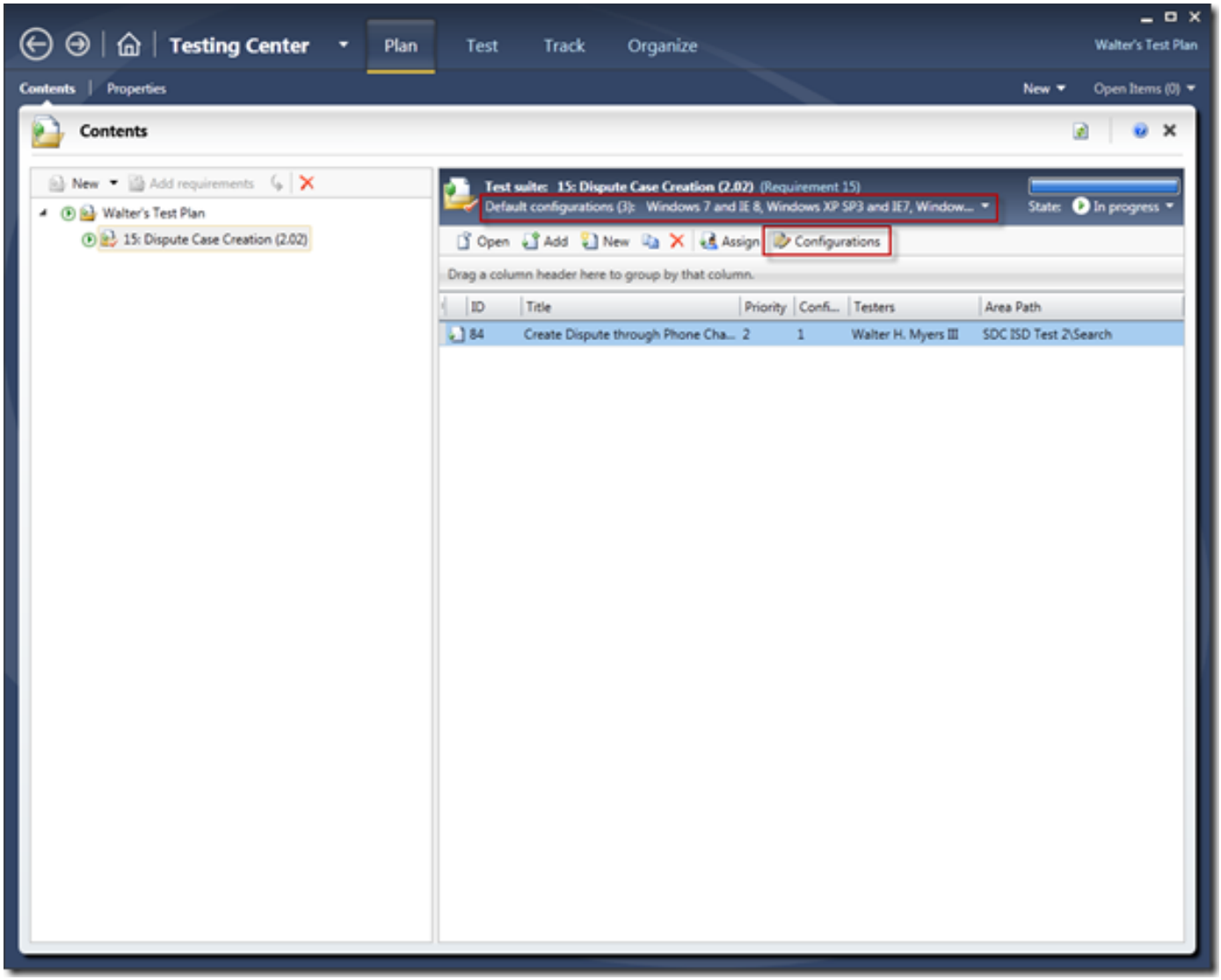


اگر در لیست کشویی مربوط به test settings مقدار <default> قرار داشت می‌توانید با انتخاب آیتم new از لیست settings جدیدی را ایجاد نمایید و یا می‌توانید لیست test settings هایی را که قبلا ایجاد کرده اید انتخاب نمایید و برای ویرایش آن با کلیک بر روی لینک open که کنار لیست قرار دارد، می‌توانید تنظیمات را ویرایش نمایید.

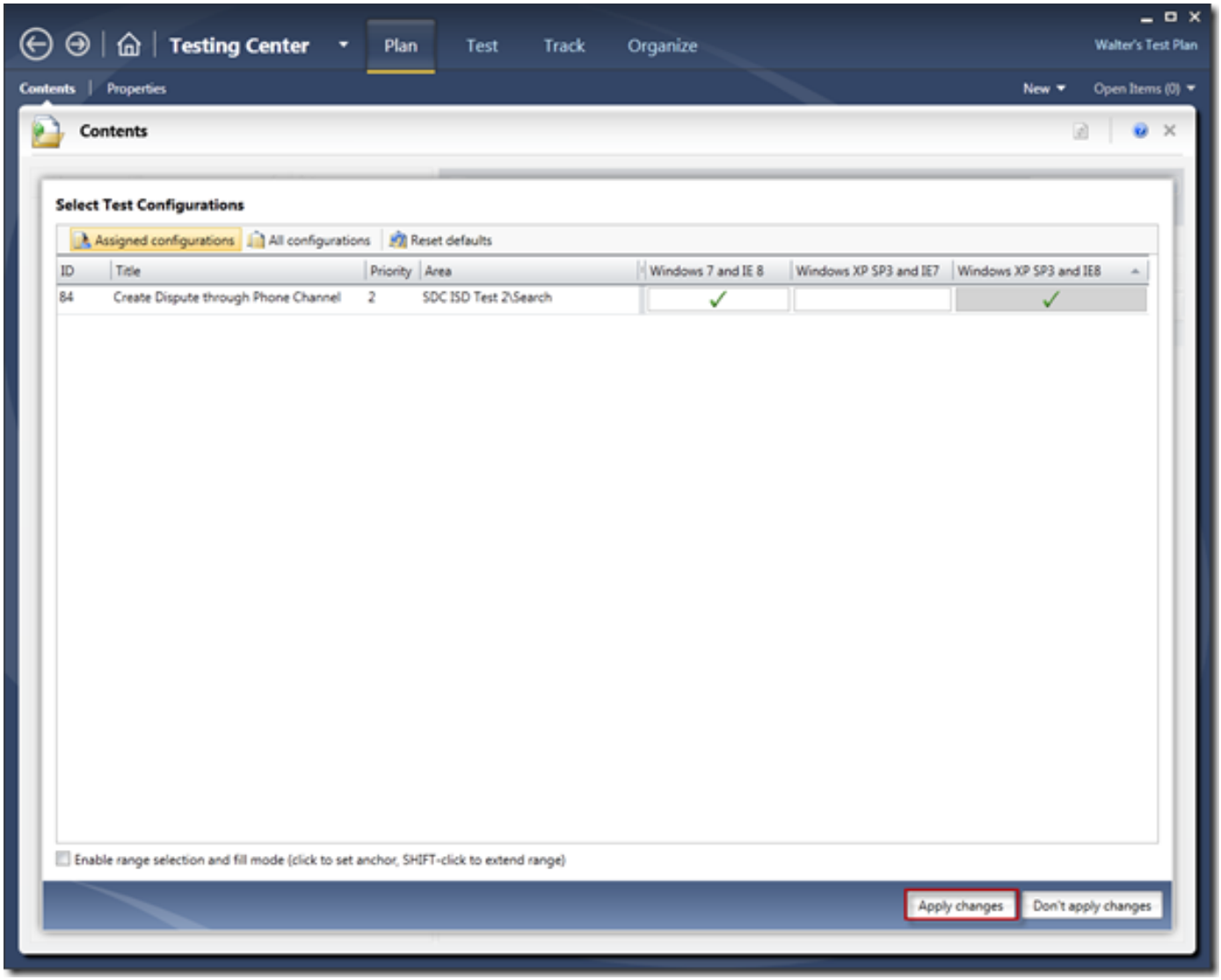


همانطور که در تصویر بالا مشاهده می‌کنید، در سمت چپ، بخش هایی برای انجام تنظیمات مربوط به تست وجود دارد. در قسمت general تنظیماتی از قبیل عنوان test settings، شرح و نوع اجرای دستی یا اتومات بودن تستتان وجود دارد. در بخش roles می‌توانید نقش هایی را برای این تست انتخاب نمایید و در قسمت data and diagnostics می‌توانید یک سری اطلاعاتی را که می‌خواهید در زمان تست دریافت کنید، انتخاب کنید. برای اطلاعات بیشتر در مورد این بخش می‌توانید در سایت مایکروسافت مطالعه کنید.

حالا بر می‌گردیم به بخش contents و موارد تست خود را می‌سازیم. همانطور که در تصویر پایین مشاهده می‌کنید در بخش contents و در سمت راست پنجره یک گزینه ای به نام configuration وجود دارد.

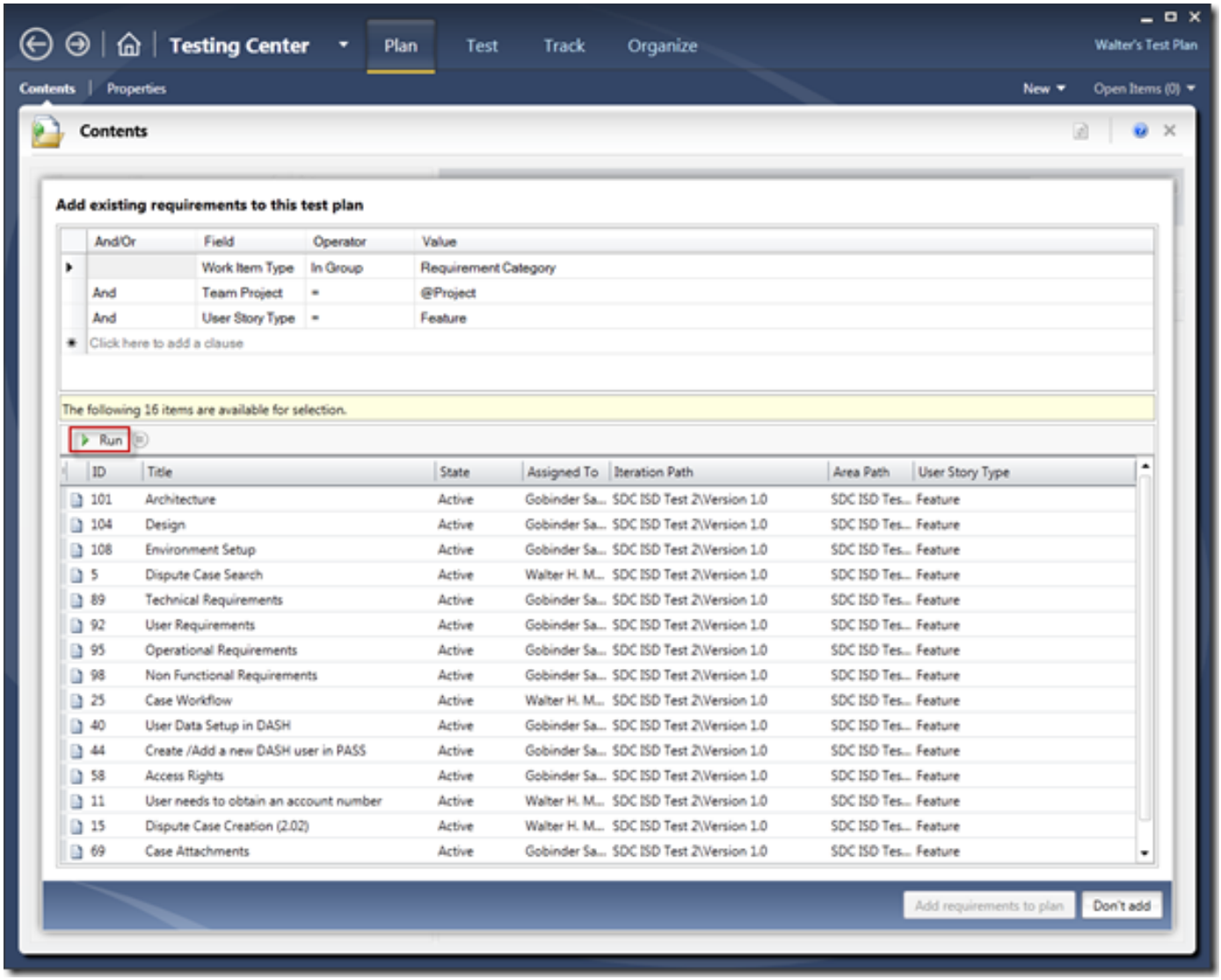


در configuration شما می‌توانید یک سری تنظیمات مربوط به test شما است انجام دهید مثلا نوع مرورگری که می‌خواهید تست خود را اجرا کنید و یا اولویت تست را مشخص نمایید یا حتی نوع سیستم عامل را مشخص کنید. هم چنین می‌توانید چندین configuration تعریف کنید و از هر کدام برای یک test suite استفاده کنید. به صورت پیش فرض test suite از تنظیمات config والد خودش یعنی test plan استفاده می‌کند.



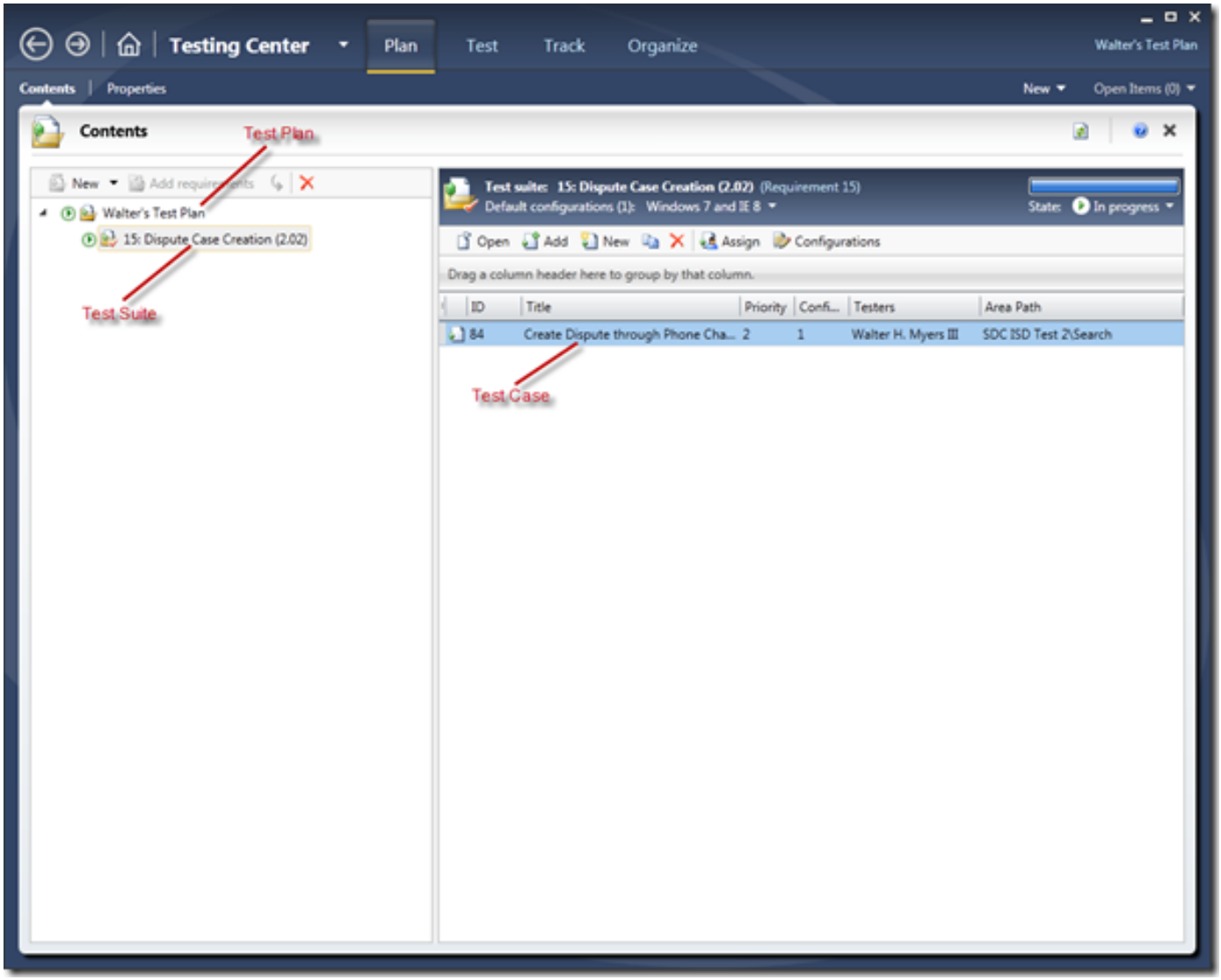
دوباره برمی گردیم به بخش contents و می‌خواهیم یک test suite با استفاده از add requirements بسازیم. همانطور که در بخش‌های قبل توضیح دادم می‌توانیم به چند روش test suite بسازیم که یکی از آن‌ها همین add requirements بود که می‌توانستید از test suite هایی که قبلا ساخته اید به این پروژه تستتان اضافه کنید.

با انتخاب گزینه add requirements پنجره ای باز می‌شود که می‌توانید همه test suiteها را مشاهده کنید و حتی می‌توانید براساس عنوان و یا وضعیت تست و ... فیلتر کنید.



بعد از اینکه در قسمت بالا کوئری خود را تنظیم کردید با انتخاب گزینه run می‌توانید کوئری خود را اجرا کرده و لیست testsuiteها را براساس آن کوئری فیلتر کنید. می‌توانید یک یا چند سطر را انتخاب کرده و با زدن دکمه add requirements to plan آن‌ها را به plan خود اضافه نمایید. حالا ما یک test suite با استفاده از test suite هایی که قبلا ساخته ایم ایجاد کردیم. حالا باید مورد تست‌های مان را به این test suite اضافه کنیم. در سمت راست با کلیک بر روی گزینه add پنجره ای مشابه پنجره بالا باز می‌شود که شما می‌توانید test caseها را فیلتر کنید و یک یا چند مورد را انتخاب کرده و با زدن دکمه add test cases آن‌ها را به test suite تان اضافه کنید. برای اضافه کردن مورد تست جدید هم می‌توانید با کلیک بر روی new که در کنار گزینه Add قرار دارد مورد تست جدیدی را بسازید.

در تصویر زیر می‌توانید بخش‌های مختلف تست را که در بخش‌های قبل هم توضیح دادم ببینید.


Testing Center
Plan
Test
Track
Organize
Walter's Test Plan
Contents
Properties
New
Open Items (0)
Contents
Test Plan
New
Walter's Test Plan
15: Dispute Case Creation (2.02)
Test Suite
Test suite: 15: Dispute Case Creation (2.02) (Requirement 15)
Default configurations (1): Windows 7 and IE 8
State: In progress
Open
Add
New
Assign
Configurations
Drag a column header here to group by that column.
ID
Title
Priority
Confi...
Testers
Area Path
84
Create Dispute through Phone Cha...
2
1
Walter H. Myers III
SDC ISD Test 2\Search
Test Case

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

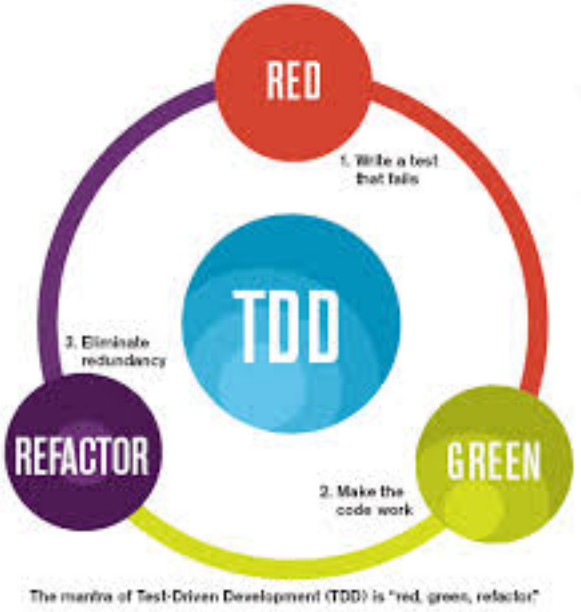
Regression test

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

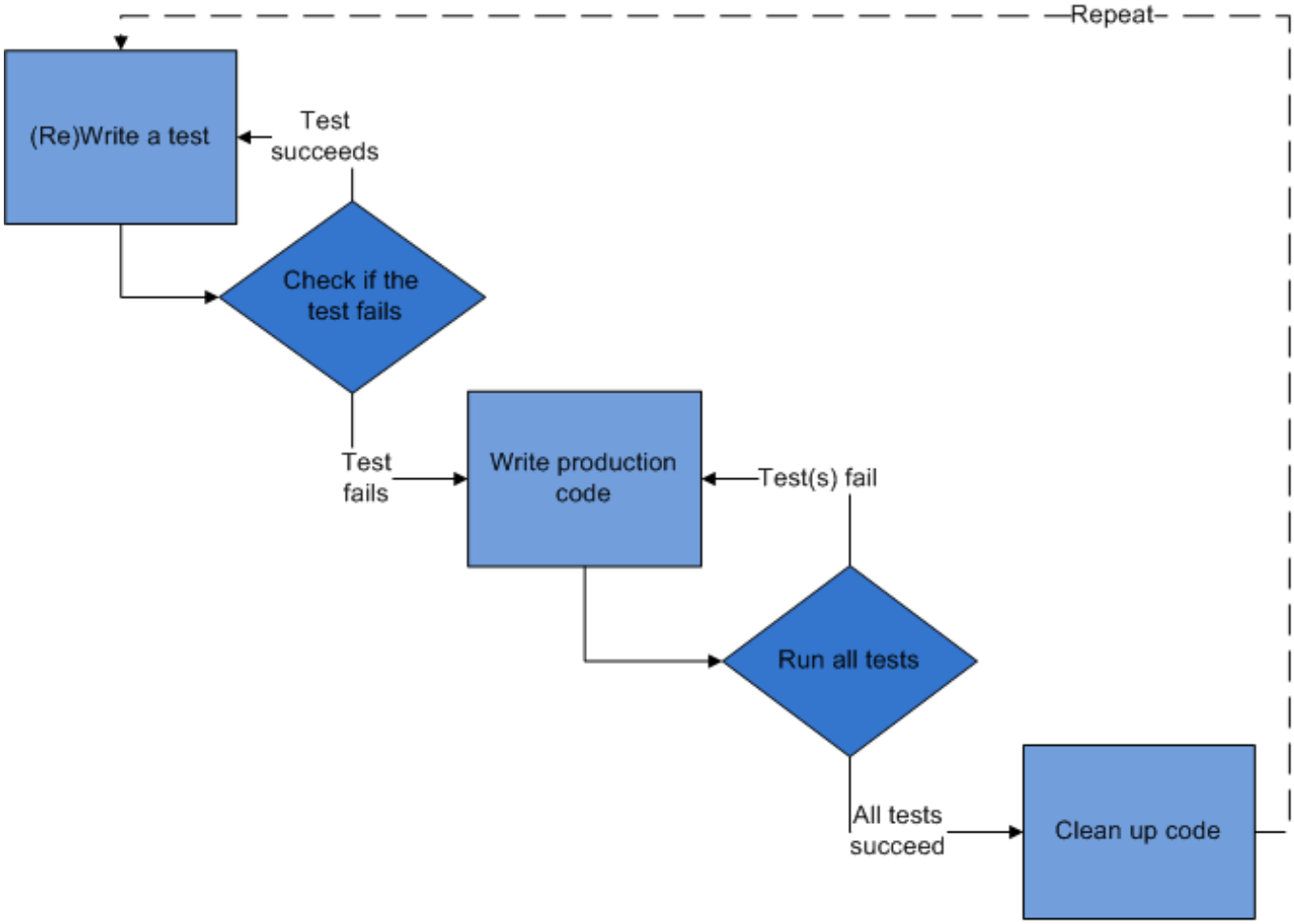
سلام

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

...

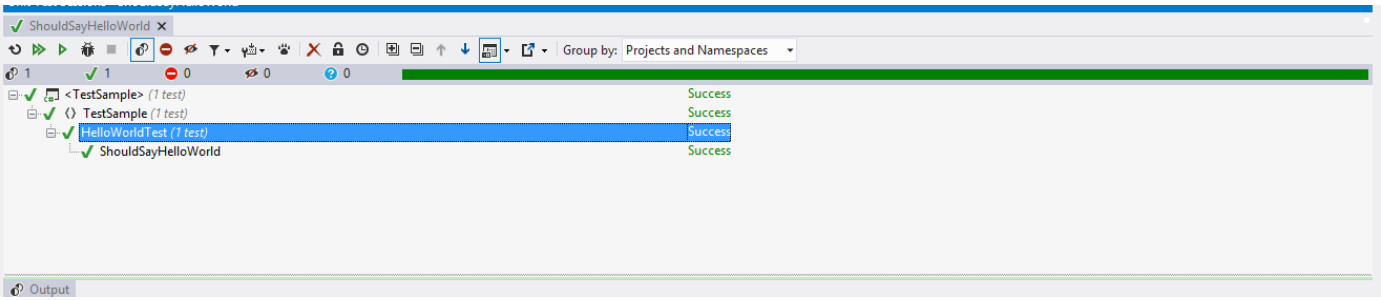
است.

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

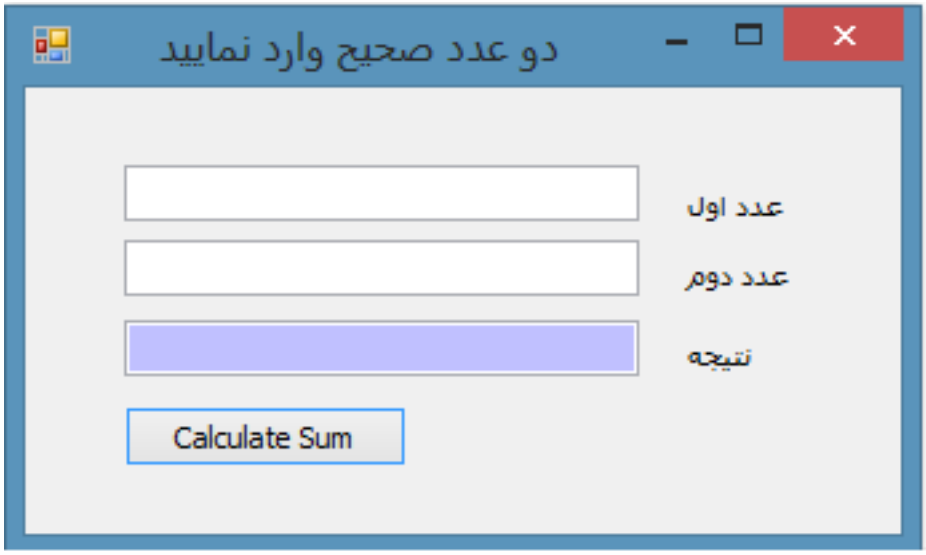
بسیار عالی،
منتظر ادامه‌ی این مبحث هستیم

عنوان: **آموزش Coded UI Test #1**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۴/۰۷ ۸:۳۰
آدرس: www.dotnettips.info
برچسب‌ها: Unit testing, Testing, Coded UI Test

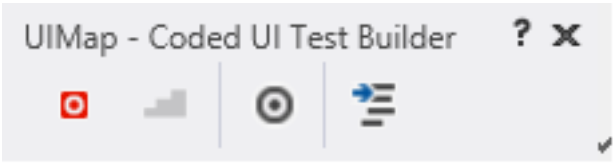
اکثر برنامه نویسان با مباحث Unit Testing آشنایی دارند و بعضی برنامه نویسان هم، از این مباحث در پروژه‌های خود استفاده می‌کنند. ساختار الگوهای MVC و MVVM به گونه ای است که به راحتی می‌توان برای این گونه پروژه‌ها Unit Test بنویسیم. در پروژه‌های MVC به دلیل عدم وابستگی بین View و Controller به طور مستقیم، امکان نوشتن Unit Test برای Controller امکان پذیر است و از طرفی در الگوی MVVM به دلیل منطق وجود ViewModel می‌توان برای اینگونه پروژه‌ها نیز Unit Test نوشت. اما ساختار سایر پروژه‌ها به گونه ای است که نوشتن Unit Test برای آن‌ها مشکل و در بعضی مواقع غیر ممکن می‌شود. برای مثال در پروژهای Desktop نظیر Windows Application و حتی وب به صورت Asp.Net Web Forms به دلیل وابستگی مستقیم کنترل‌های UI به منطق اجرای برنامه، طراحی و نوشتن Unit Test بسیار مشکل و در برخی موارد بیهوده است. در VS.Net ابزاری وجود دارد به نام Coded UI Test که برای تست این گونه پروژه‌ها طراحی شده است و همان طور که از نامش پیداست صرفا برای تست کنترل‌های UI و رویدادهای کنترل‌ها و تست درستی برنامه با توجه به داده‌های ورودی به کار می‌رود. یکی از مزیت‌های اصلی آن تسریع عملیات تست در حجم بالا است و زمان ایجاد unit test را به حداقل می‌رساند. مزیت دوم آن امکان ایجاد unit test برای پروژه‌های که در مراحل پایانی تولید هستند ولی هنوز اطمینانی به عملکرد صحیح برنامه در حالات مختلف نیست.
در این پست قصد دارم روش استفاده از این گونه پروژه‌های تست را با ذکر یک مثال بررسی کنیم و در پست‌های بعدی به بررسی امکانات دیگر خواهیم پرداخت.

نکته : فقط در Vs.Net با نسخه‌های Ultimate و Premium می‌تونید از Code UI Test استفاده کنید که البته به دلیل اینکه در ایران پیدا کردن نسخه‌های دیگر Vs.Net به غیر از Ultimate سخت‌تر است به طور قطع این محدودیت برای برنامه نویسان ما وجود نخواهد داشت. برای اینکه از نسخه Vs.Net خود اطمینان حاصل کنید از منوی Help گزینه About Microsoft Visual Studio رو انتخاب کنید. پنجره ای به شکل زیر مشاهده خواهید کرد که در آن مشخصات کامل Vs.Net ذکر شده است.

در این مرحله قصد داریم برای فرم زیر Unit Test طراحی کنیم. پروژه به صورت زیر است:



کاملا واضح است که در این فرم دو عدد به عنوان ورودی دریافت می‌شود و بعد از کلیک بر روی CalculateSum نتیجه در textbox سوم نمایش داده می‌شود. برای تست عملکرد صحیح فرم بالا ایتدا به Solution مورد نظر از منوی test Project یک Coded UI Test Project اضافه می‌کنیم. به دلیل اینکه این قبلا در این Solution پروژه تست از نوع Coded UI Test نبود بلافاصله یک پنجره نمایش داده می‌شود. مطمئن شوید گزینه اول انتخاب شده و بعد بر روی Ok کلیک کنید.(گزینه اول به معنی است که قصد داریم عملیات مورد نظر بر روی UI را رکورد کنیم و گزینه دوم به معنی است که قصد داریم از عملیات رکورد شده قبلی استفاده کنیم). یک کلاس به نام CodeUITest1 به همراه یک متد تست به نام CodedUITestMethod1 ساخته می‌شود. اولین چیزی که جلب توجه می‌کند این است که این کلاس به جای TestClassAttribute دارای نشان CodeUITestAttrbiute است. در گوشه سمت راست Vs.Net خود یک پنجره کوچک به نام UI Map Test Builder مانند شکل زیر خواهید دید.



دکمه قرمز رنگ به نام Record Button است و عملیات تست را رکورد خواهد کرد. دکمه دایره ای به رنگ مشکی برای تعیین Assertion به کار می‌رود. و در نهایت گزینه آخر کدهای مورد نظر مراحل قبل را به صورت خودکار تولید خواهد کرد.

#روش کار

روش کار به این صورت است که ابتدا شما مراحل تست خود را شبیه سازی خواهید کرد و بعد از آن Test Builder مراحل تست شما را به صورت کامل به صورت کدهای قابل فهم تولید خواهد کرد. (دقیقا شبیه به ایجاد UnitTest به روش Arrange/Act/Assert است با این تفاوت که این مراحل توسط UI Map رکورد شده و نیازی به کد نویسی ندارد). در پایان باید یک Data Driven Coded UI Test طراحی کنید تا بتوانید از این مراحل رکورد استفاده نمایید.

#چگونگی شبیه سازی :

پروژه را اجرا نمایید. زمانی که فرم مورد نظر ظاهر شد بر روی گزینه Record در TestBuilder کلیک کنید. عملیات ذخیره سازی شروع شده است. در نتیجه به فرم مربوطه رفته و در Textbox اول مقدار 10 و در textbox دوم مقدار 5 را وارد نمایید. با کلیک بر روی دکمه CalculateSum مقدار 15 نمایش داده خواهد شد. از برنامه خارج شوید و بعد بر روی گزینه Generate Code در TestBuilder کلیک کنید با از کلیدهای ترکیبی Alt + G استفاده نمایید.(اگر در این مرحله، از برنامه خارج نشده باشید با خطا مواجه خواهید شد.) در پنجره نمایش داده شده یک نام به متد اختصاص دهید. عملیات تولید کد شروع خواهد شد. بعد کدی مشابه زیر را در متد مربوطه مشاهده خواهید کرد.

```
[TestMethod]
        public void CodedUITestMethod1()
        {
            this.UIMap.CalculateSum();
            this.UIMap.txtSecondValueMustBe10();
        }
```

بخشی از سورس کد تولید شده برای متد CalculateSum به شکل زیر است:

```
public void CodedUITestMethod1
```

```
()
        {
            #region Variable Declarations
            WinEdit uITxtFirstNumberEdit =
this.UIدوعددصحیحواردنماییدWindow.UITxtFirstNumberWindow.UITxtFirstNumberEdit;
            WinEdit uITxtSecondNumberEdit =
this.UIدوعددصحیحواردنماییدWindow.UITxtSecondNumberWindow.UITxtSecondNumberEdit;
            WinButton uICalculateSumButton =
this.UIدوعددصحیحواردنماییدWindow.UICalculateSumWindow.UICalculateSumButton;
            #endregion

            // Type '10' in 'txtFirstNumber' text box
            uITxtFirstNumberEdit.Text = this.CalculateSumParams.UITxtFirstNumberEditText;

            // Type '{Tab}' in 'txtFirstNumber' text box
            Keyboard.SendKeys(uITxtFirstNumberEdit,
this.CalculateSumParams.UITxtFirstNumberEditSendKeys, ModifierKeys.None);

            // Type '10' in 'txtSecondNumber' text box
            uITxtSecondNumberEdit.Text = this.CalculateSumParams.UITxtSecondNumberEditText;

            // Click 'Calculate Sum' button
            Mouse.Click(uICalculateSumButton, new Point(83, 12));

            // Type '10' in 'txtFirstNumber' text box
            uITxtFirstNumberEdit.Text = this.CalculateSumParams.UITxtFirstNumberEditText1;

            // Type '{Tab}' in 'txtFirstNumber' text box
            Keyboard.SendKeys(uITxtFirstNumberEdit,
this.CalculateSumParams.UITxtFirstNumberEditSendKeys1, ModifierKeys.None);

            // Type '10' in 'txtSecondNumber' text box
            uITxtSecondNumberEdit.Text = this.CalculateSumParams.UITxtSecondNumberEditText1;

            // Type '{Tab}' in 'txtSecondNumber' text box
            Keyboard.SendKeys(uITxtSecondNumberEdit,
this.CalculateSumParams.UITxtSecondNumberEditSendKeys, ModifierKeys.None);

            // Click 'Calculate Sum' button
            Mouse.Click(uICalculateSumButton, new Point(49, 11));

            // Type '10' in 'txtFirstNumber' text box
            uITxtFirstNumberEdit.Text = this.CalculateSumParams.UITxtFirstNumberEditText2;

            // Type '{Tab}' in 'txtFirstNumber' text box
            Keyboard.SendKeys(uITxtFirstNumberEdit,
this.CalculateSumParams.UITxtFirstNumberEditSendKeys2, ModifierKeys.None);

            // Type '5' in 'txtSecondNumber' text box
            uITxtSecondNumberEdit.Text = this.CalculateSumParams.UITxtSecondNumberEditText2;

            // Type '{Tab}' in 'txtSecondNumber' text box
            Keyboard.SendKeys(uITxtSecondNumberEdit,
```

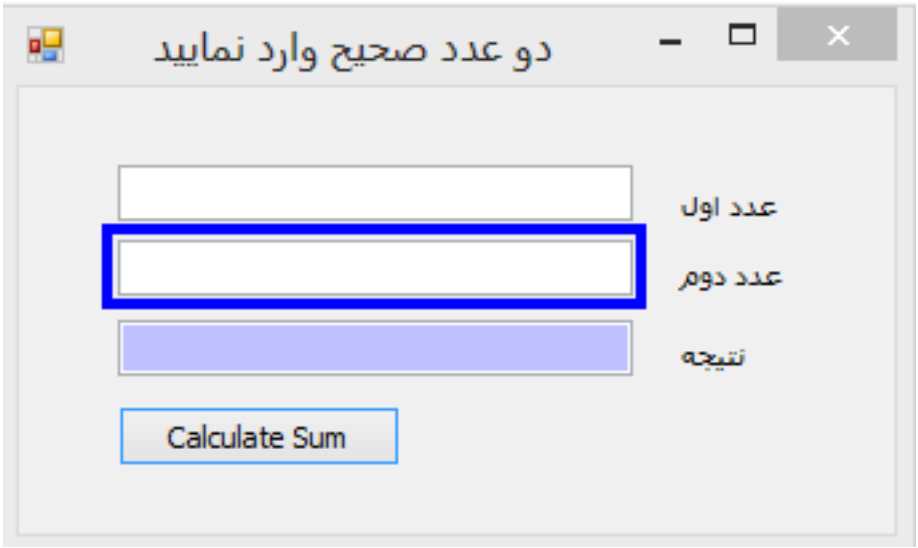
```
this.CalculateSumParams.UITxtSecondNumberEditSendKeys1, ModifierKeys.None);

            // Click 'Calculate Sum' button
            Mouse.Click(uICalculateSumButton, new Point(74, 16));
        }
```

همان طور که می‌بینید تمام مراحل تست شما رکورد شده است و به صورت کد قابل فهم بالا ایجاد شده است.

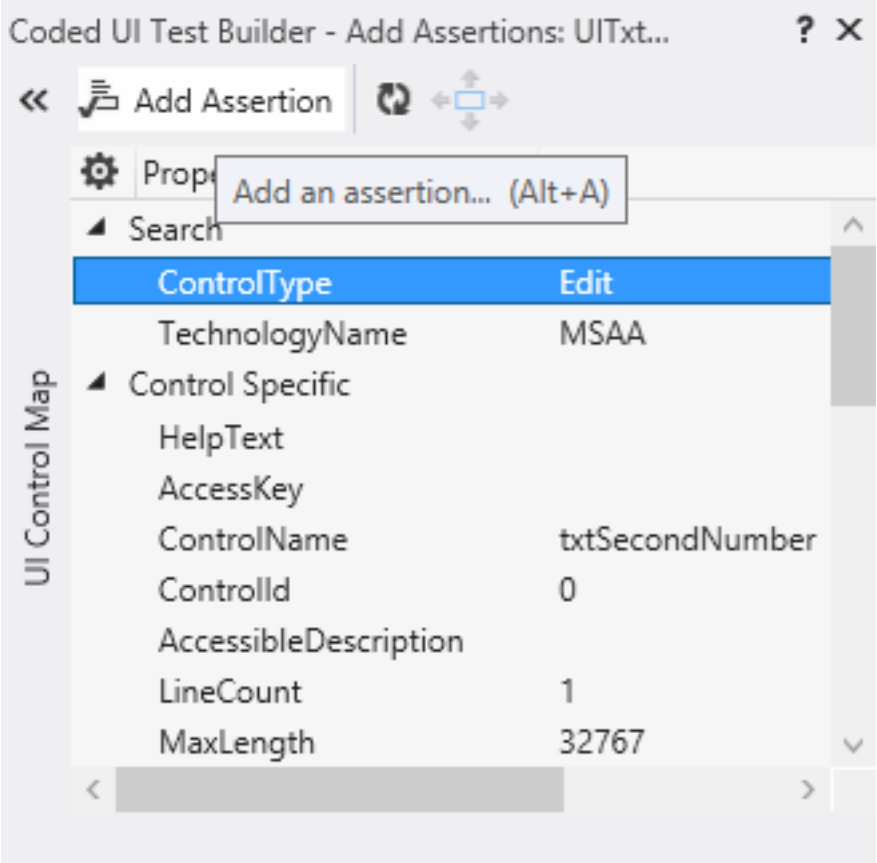
چگونگی ایجاد Assertion

اگر به کد متد تست CodedUITestMethod1 در بالا دقت کنید یک متد به صورت this.UIMap.txtSecondValueMustBe10 فراخوانی شده است. این در واقع یک Assertion است که در هنگام عملیات رکورد ایجاد کردم و به این معنی است که مقدار TextBox دوم حتما باید 10 باشد. حال روش تولید Assertionها را بررسی خواهیم کرد.

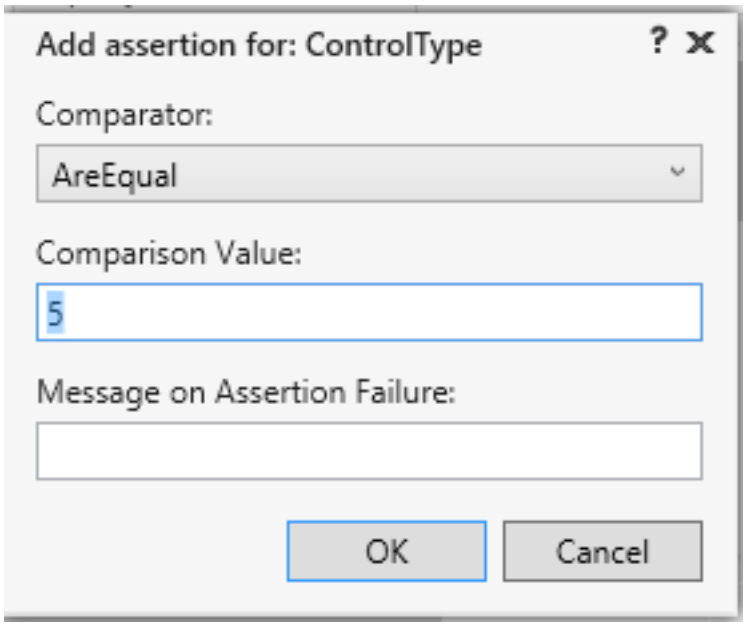
بعد از شروع شدن مرحله رکورد اگر قصد دارید برای یک کنترل خاص Assert بنویسید، دکمه assertion (به رنگ مشکی و به صورت دایره است) را بر روی کنترل مورد نظر drag&drop کنید. یک border آبی برای کنترل مورد نظر ایجاد خواهد شد:



به محض اتمام عملیات drag&drop منوی زیر ظاهر خواهد شد:



از گزینه Add Assertion استفاده کنید و برای کنترل مورد نظر یک assert بنویسید. در شکل زیر یک assert برای textbox دوم نوشتم به صورتی که مقدار آن باید با 5 برابر باشد.



از گزینه آخر برای نمایش پیغام مورد نظر خودتون در هنگامی که aseert با شکست مواجه می‌شود استفاده کنید.
کد تولید شده زیر برای عملیات assert بالا است:

```
public void txtSecondValueMustBe10()
        {
            #region Variable Declarations
            WinEdit uITxtSecondNumberEdit =
this.UIدوعددصحیحواردنماییدWindow.UITxtSecondNumberWindow.UITxtSecondNumberEdit;
            #endregion

            // Verify that the 'ControlType' property of 'txtSecondNumber' text box equals '10'
            Assert.AreEqual(this.txtSecondValueMustBe10ExpectedValues.UITxtSecondNumberEditControlType,
uITxtSecondNumberEdit.ControlType.ToString());
        }
```

مرحله اول انجام شد. برای تست این مراحل باید یک Data DrivenTest بسازید که در پست بعدی به صورت کامل شرح داده خواهد شد.

عنوان: **آموزش QUnit #1**
نویسنده: مجتبی کاویانی
تاریخ: ۱۳۹۲/۰۴/۱۷ ۱۹:۵۵
آدرس: www.dotnettips.info
برچسب‌ها: JavaScript, jQuery, QUnit, Unit testing

مقدمه: تست و آزمایش کد برنامه‌ها و وب سایت‌هایمان، بهترین راه کاهش خطا و مشکلات آنها بعد از انتشار است. از جمله روش‌های موجود، تست واحد است که ویژوال استادیو نیز از آن برای پروژه‌های دات نت پشتیبانی می‌کند. با افزایش روز افزون کتابخانه‌های جاوا اسکریپتی و جی کوئری، نیاز به تست کدهای جاواسکریپتی نیز بیشتر به نظر می‌رسد و بهتر است تست واحد و آزمایش شوند. اما برخلاف کدهای C# و ASP.NET تست کدهای جاوا اسکریپت، مخصوصا زمانی که به دستکاری عناصر DOM می‌پردازیم و یا رویدادهای درون صفحه وب را با استفاده از جی کوئری می‌نویسیم، حتی اگر در فایل جداگانه‌ای نوشته شود، این بدان معنی نیست که آماده تست واحد است و ممکن است امکان نوشتن تست وجود نداشته باشد.
بنابراین چه چیزی یک تست واحد است؟ در بهترین حالت توابعی که مقداری را برمی گردادنند، بهترین حالت برای تست واحد است. اما در بیشتر موارد شما نیاز دارید تا تاثیر کد را بر روی عناصر صفحه نیز مشاهد نمایید.

ساخت تست واحد

برای تست پذیری بهتر، توابع جاوا اسکریپت و هر کد دیگری، آن را می‌بایست طوری بنویسید که مقادیر تاثیر گذار در اجرای تابع به عنوان ورودی تابع در نظر گرفته شده باشند و همیشه نتیجه به عنوان خروجی تابع برگردانده شود؛ قطعه کد زیر را در نظر بگیرید:

```
function prettyDate(time){
    var date = new Date(time || ""),
      diff = (((new Date()).getTime() - date.getTime()) / 1000),
      day_diff = Math.floor(diff / 86400);

    if ( isNaN(day_diff) || day_diff < 0 || day_diff >= 31 )
      return;

    return day_diff == 0 && (
        diff < 60 && "just now" ||
        diff < 120 && "1 minute ago" ||
        diff < 3600 && Math.floor( diff / 60 ) +
          " minutes ago" ||
        diff < 7200 && "1 hour ago" ||
        diff < 86400 && Math.floor( diff / 3600 ) +
          " hours ago") ||
      day_diff == 1 && "Yesterday" ||
      day_diff < 7 && day_diff + " days ago" ||
      day_diff < 31 && Math.ceil( day_diff / 7 ) +
        " weeks ago";
  }
```

تابع pertyDate اختلاف زمان حال را نسبت به زمان ورودی، بصورت یک رشته برمی گرداند. اما در اینجا مقدار زمان حال، در خط سوم، در خود تابع ایجاد شده است و در صورتی که بخواهیم برای چندین مقدار آن را تست کنیم زمان حال متفاوتی در نظر گرفته می‌شود و حداکثر، زمان 31 روز قبل را نمایش داده و در بقیه تاریخ ها undefined را بر می‌گرداند. برای تست واحد، چند تغییر می‌دهیم.

بهینه سازی، مرحله اول:

پارامتری به عنوان مقدار زمان جاری برای تابع در نظر می‌گیریم و تابع را جدا کرده و در یک فایل جداگانه قرار می‌دهیم. فایل prettydate.js بصورت زیر خواهد شد.

```
function prettyDate(now, time){
  var date = new Date(time || ""),
    diff = (((new Date(now)).getTime() - date.getTime()) / 1000),
    day_diff = Math.floor(diff / 86400);

  if ( isNaN(day_diff) || day_diff < 0 || day_diff >= 31 )
    return;

  return day_diff == 0 && (
      diff < 60 && "just now" ||
      diff < 120 && "1 minute ago" ||
      diff < 3600 && Math.floor( diff / 60 ) +
```

```
        " minutes ago" ||
      diff < 7200 && "1 hour ago" ||
      diff < 86400 && Math.floor( diff / 3600 ) +
        " hours ago") ||
    day_diff == 1 && "Yesterday" ||
    day_diff < 7 && day_diff + " days ago" ||
    day_diff < 31 && Math.ceil( day_diff / 7 ) +
      " weeks ago";
}
```

حال یک تابع برای تست داریم، چند تست واحد واقعی می‌نویسیم

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Refactored date examples</title>
  <script src="prettydate.js"></script>
  <script>
  function test(then, expected) {
    results.total++;
    var result = prettyDate("2013/01/28 22:25:00", then);
    if (result !== expected) {
      results.bad++;
      console.log("Expected " + expected +
        ", but was " + result);
    }
  }
  var results = {
    total: 0,
    bad: 0
  };
  test("2013/01/28 22:24:30", "just now");
  test("2013/01/28 22:23:30", "1 minute ago");
  test("2013/01/28 21:23:30", "1 hour ago");
  test("2013/01/27 22:23:30", "Yesterday");
  test("2013/01/26 22:23:30", "2 days ago");
  test("2012/01/26 22:23:30", undefined);
  console.log("Of " + results.total + " tests, " +
    results.bad + " failed, " +
    (results.total - results.bad) + " passed.");
  </script>
</head>
<body>

</body>
</html>
```

در کد بالا یک تابع بدون استفاده از Qunit برای تست واحد نوشته ایم که با آن تابع prettyDate را تست می‌کند. تابع test مقدار زمان حال و رشته خروجی را گرفته و آن را با تابع اصلی تست می‌کند در آخر تعداد تست ها، تست‌های شکست خورده و تست های پاس شده گزارش داده می‌شود.
خروجی می‌تواند مانند زیر باشد:

.Of 6 tests, 0 failed, 6 passed

.Expected 2 day ago, but was 2 days ago

.f 6 tests, 1 failed, 5 passed

عنوان: **آموزش QUnit #2**
نویسنده: مجتبی کاویانی
تاریخ: ۱۳۹۲/۰۴/۲۱ ۱۲:۵۵
آدرس: www.dotnettips.info
برچسب‌ها: JavaScript, Unit testing, jQuery, QUnit

فریم ورک تست جاوا اسکریپت QUnit: انتخاب و استفاده از یک فریم ورک برای تست کدهای جاوا اسکریپت، قطعا نتیجه بهتری را به همراه خواهد داشت. من در این جا از QUnit که یکی از بهترین‌های تست واحد است، استفاده می‌کنم. برای این کار فایل‌های qunit.js و qunit.css را دانلود و مانند زیر برای تست واحد آماده کنید:

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Refactored date examples</title>

  <link rel="stylesheet" href="../qunit.css">
  <script src="../qunit.js"></script>
  <script src="prettydate.js"></script>

  <script>
  test("prettydate basics", function() {
    var now = "2013/01/28 22:25:00";
    equal(prettyDate(now, "2013/01/28 22:24:30"), "just now");
    equal(prettyDate(now, "201308/01/28 22:23:30"), "1 minute ago");
    equal(prettyDate(now, "2013/01/28 21:23:30"), "1 hour ago");
    equal(prettyDate(now, "2013/01/27 22:23:30"), "Yesterday");
    equal(prettyDate(now, "2013/01/26 22:23:30"), "2 days ago");
    equal(prettyDate(now, "2012/01/26 22:23:30"), undefined);
  });
  </script>
</head>
<body>

<div id="qunit"></div>

</body>
</html>
```

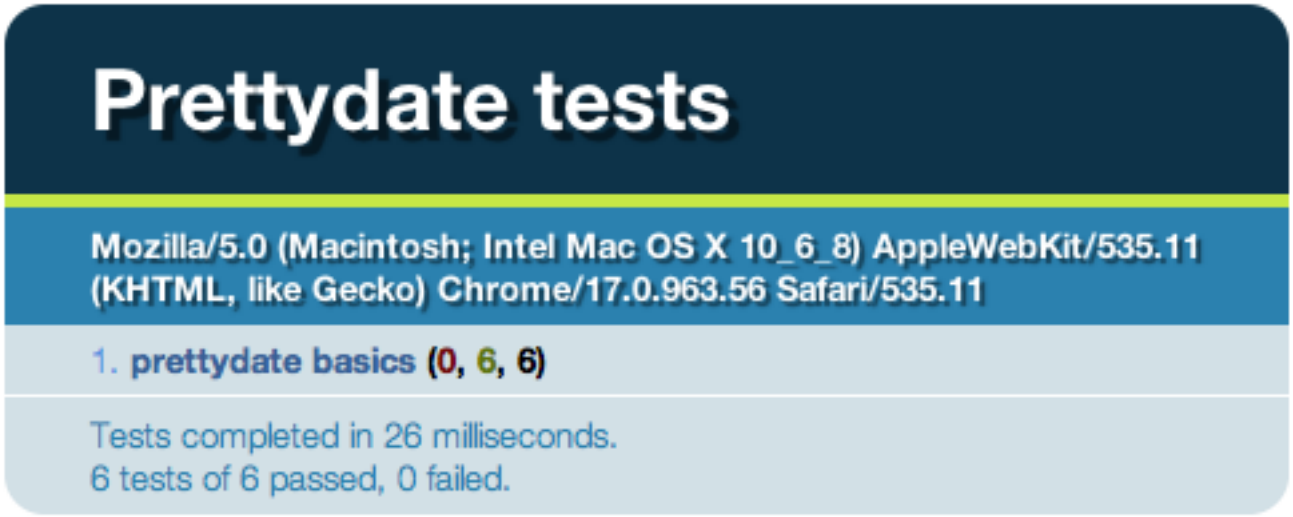
در کد بالا ابتدا فایل‌های فریم ورک و فایل prettydate.js را اضافه کردیم. برای نمایش نتیجه تست، یک تگ div با نام qunit در بین تگ body اضافه می‌کنیم.

تابع test:

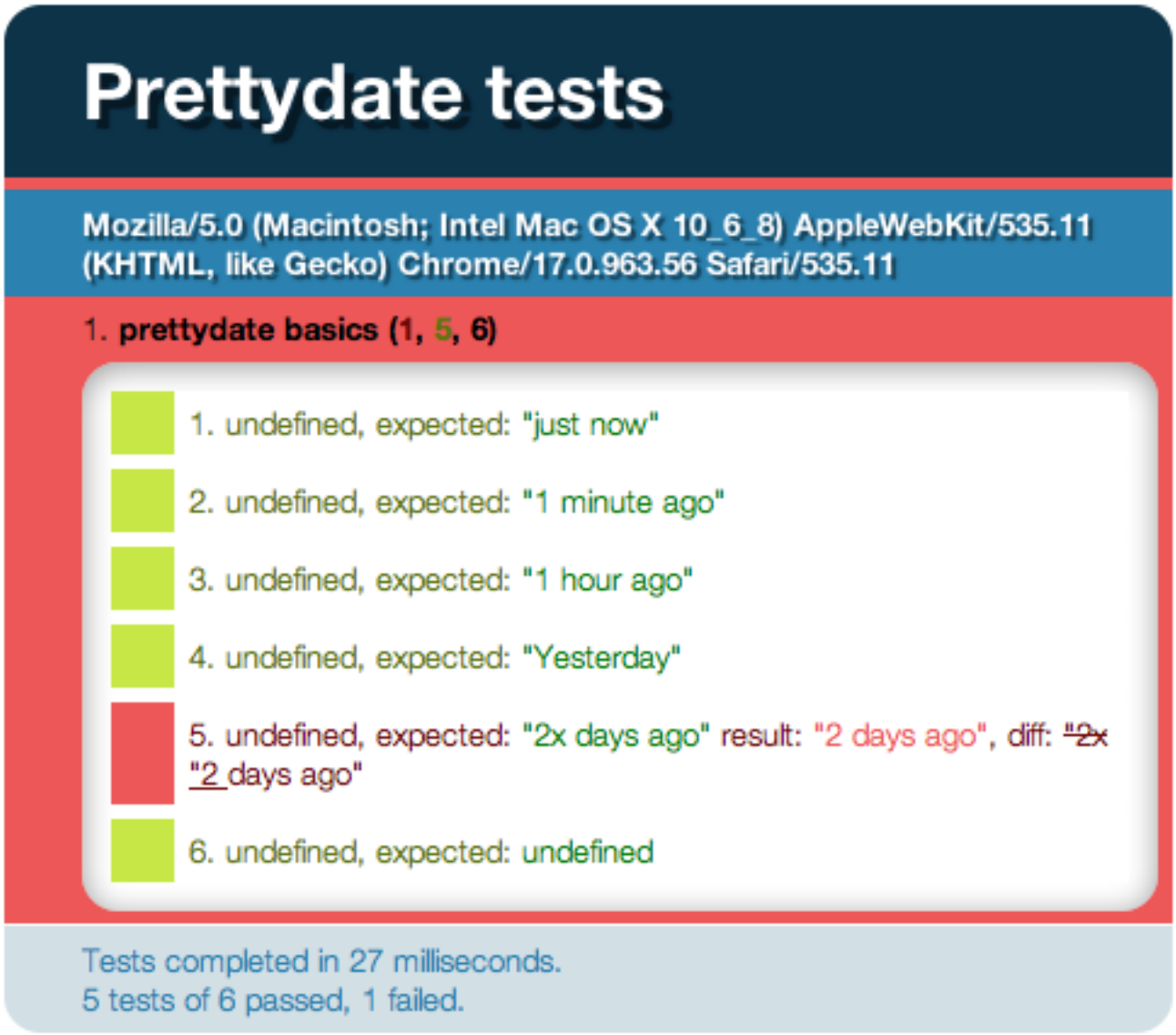
این تابع برای تست توابع نوشته شده، استفاده می‌شود. ورودی‌های این تابع، یکی عنوان تست و دومی یک متود دیگر، به عنوان ورودی دریافت می‌کند که در آن بدنه تست نوشته می‌شود.

تابع equal:

اولین تابع برای سنجش تست واحد equal است و در آن، تابعی که می‌خواهیم تست کنیم با مقدار خروجی آن مقایسه می‌شود.
فایل را با نام test.htm ذخیره و آن را در مرورگر خود باز نمایید. خروجی در شکل آورده شده است:



همین طور که در تصویر بزرگ می‌بینید اطلاعات مرورگر، زمان تکمیل تست و تعداد تست، تعداد تست پاس شده و تعداد تست شکست خورده، نشان داده شده است.



اگر یکی از تست‌ها با شکست روبرو شود رنگ پس زمینه قرمز و جزئیات شکست نمایش داده می‌شوند.

بهینه سازی، مرحله اول:

در حال حاضر تست ما کامل نیست زیرا امکان تست n weeks ago یا تعداد هفته پیش میسر نیست. قبل از آنکه این را به آزمون اضافه کنیم، تغییراتی در تست می‌دهیم

```
test("prettydate basics", function() {
  function date(then, expected) {
    equal(prettyDate("2013/01/28 22:25:00", then), expected);
  }
  date("2013/01/28 22:24:30", "just now");
  date("2013/01/28 22:23:30", "1 minute ago");
  date("2013/01/28 21:23:30", "1 hour ago");
  date("2013/01/27 22:23:30", "Yesterday");
  date("2013/01/26 22:23:30", "2 days ago");
  date("2012/01/26 22:23:30", undefined);
});
```

تابع prettyDate را در تابع دیگری به نام date قرار می‌دهیم. این تغییر سبب می‌شود تا امکان مقایسه زمان ورودی تست جاری با تست قبلی فراهم شود.

تست دستکاری عناصر DOM:

تا اینجا با تست توابع آشنا شدید، حالا می‌خواهیم تغییراتی در prettyDate دهیم تا امکان انتخاب عناصر DOM و به روزرسانی آن نیز وجود داشته باشد. فایل prettyDate2.js در زیر آورده شده است:

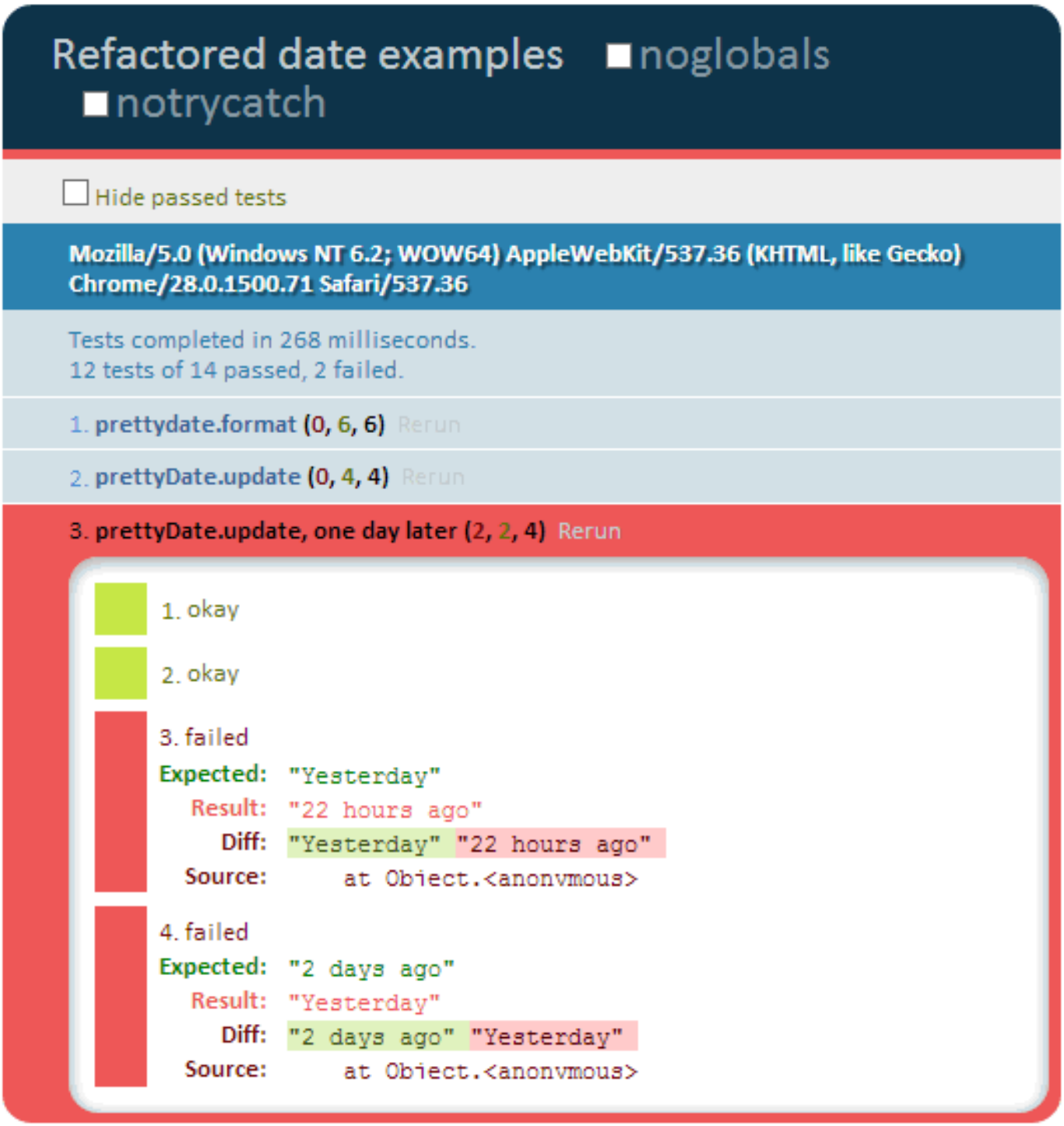
```
var prettyDate = {
  format: function(now, time){
    var date = new Date(time || ""),
      diff = (((new Date(now)).getTime() - date.getTime()) / 1000),
      day_diff = Math.floor(diff / 86400);

    if ( isNaN(day_diff) || day_diff < 0 || day_diff >= 31 )
      return;

    return day_diff === 0 && (
        diff < 60 && "just now" ||
        diff < 120 && "1 minute ago" ||
        diff < 3600 && Math.floor( diff / 60 ) +
          " minutes ago" ||
        diff < 7200 && "1 hour ago" ||
        diff < 86400 && Math.floor( diff / 3600 ) +
          " hours ago") ||
      day_diff === 1 && "Yesterday" ||
      day_diff < 7 && day_diff + " days ago" ||
      day_diff < 31 && Math.ceil( day_diff / 7 ) +
        " weeks ago";
  },

  update: function(now) {
    var links = document.getElementsByTagName("a");
    for ( var i = 0; i < links.length; i++ ) {
      if ( links[i].title ) {
        var date = prettyDate.format(now, links[i].title);
        if ( date ) {
          links[i].innerHTML = date;
        }
      }
    }
  }
};
```

prettyDate شامل دو تابع، یکی format که weeks ago به آن اضافه گردیده و تابع update که با انتخاب تگ‌ها، مقدار title را به تابع فرمت و خروجی آن را در Html هر عنصر قرار می‌دهد. حال یک تست واحد می‌نویسیم:

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Refactored date examples</title>
  <link rel="stylesheet" href="../qunit.css">
  <script src="../qunit.js"></script>
  <script src="prettydate2.js"></script>
  <script>
  test("prettydate.format", function() {
    function date(then, expected) {
      equal(prettyDate.format("2013/01/28 22:25:00", then),
        expected);
    }
    date("2013/01/28 22:24:30", "just now");
    date("2013/01/28 22:23:30", "1 minute ago");
    date("2013/01/28 21:23:30", "1 hour ago");
    date("2013/01/27 22:23:30", "Yesterday");
    date("2013/01/26 22:23:30", "2 days ago");
    date("2012/01/26 22:23:30", undefined);
  });

  function domtest(name, now, first, second) {
    test(name, function() {
      var links = document.getElementById("qunit-fixture")
        .getElementsByTagName("a");
      equal(links[0].innerHTML, "January 28th, 2013");
      equal(links[2].innerHTML, "January 27th, 2013");
      prettyDate.update(now);
      equal(links[0].innerHTML, first);
      equal(links[2].innerHTML, second);
    });
```

```
  }
  domtest("prettyDate.update", "2013-01-28T22:25:00Z",
    "2 hours ago", "Yesterday");
  domtest("prettyDate.update, one day later", "2013/01/29 22:25:00",
    "Yesterday", "2 days ago");
  </script>
</head>
<body>

<div id="qunit"></div>
<div id="qunit-fixture">

<ul>
  <li id="post57">
    <p>blah blah blah...</p>
    <small>
      Posted <span>
        <a href="/2013/01/blah/57/" title="2013-01-28T20:24:17Z"
          >January 28th, 2013</a>
      </span>
      by <span><a href=""></a></span>
    </small>
  </li>
  <li id="post57">
    <p>blah blah blah...</p>
    <small>
      Posted <span>
        <a href="/2013/01/blah/57/" title="2013-01-27T22:24:17Z"
          >January 27th, 2013</a>
      </span>
      by <span><a href=""></a></span>
    </small>
  </li>
</ul>

</div>

</body>
</html>
```

همین طور که مشاهد می‌کنید در تست واحد اول خود تابع prettyDate.format را تست نموده ایم. در تست بعدی عناصر DOM نیز دستکاری و تست شده است. تابع domtest با جستجوی تگ qunit-fixture و تگ‌های a درون آن، مقدار نهایی html آن با مقدار داده شده، مقایسه شده است.



در شکل بالا نتیجه تست واحد نشان داده شده است.

عنوان: آموزش QUnit #3
نویسنده: مجتبی کاویانی
تاریخ: ۱۳۹۲/۰۴/۲۴ ۱۷:۳۰
آدرس: www.dotnettips.info
برچسب‌ها: JavaScript, Unit testing, jQuery, QUnit

در قسمت‌های قبلی با مفهوم تست واحد و کتابخانه quint آشنا شدید و مثالی را نیز با هم بررسی کردیم. در ادامه به قابلیت‌های بیشتر این کتابخانه می‌پردازیم.

توابع اعلان نتایج:

qunit سه تابع را جهت اعلان نتایج تست واحد فراهم نموده است

تابع ok:

تابع پایه‌ای تست واحد، دو پارامتر را به عنوان ورودی دریافت می‌کند و در صورتیکه بررسی نتیجه پارامتر اول برابر true باشد، تست با موفقیت روبرو شده است. پارامتر دوم برای نمایش یک پیام است. در مثال زیر حالت‌های مختلف آن بررسی شده است. مقادیر non-empty string، true به معنی موفقیت و مقادیر null،"",NaN،0،false و undefined به معنی شکست تست می‌باشد. در واقع خروجی تابع ارسالی به اعلان ok یکی از نتایج بالا می‌تواند باشد.

```
//ok( truthy [, message ] )

test( "ok test", function() {
  ok( true, "true succeeds" );
  ok( "non-empty", "non-empty string succeeds" );

  ok( false, "false fails" );
  ok( 0, "0 fails" );
  ok( NaN, "NaN fails" );
  ok( "", "empty string fails" );
  ok( null, "null fails" );
  ok( undefined, "undefined fails" );
});
```

تابع equal:

این اعلان یک مقایسه ساده بین پارامتر اول و دوم تابع می‌باشد که شرط برابری(==) را بررسی می‌نماید. وقتی مقدار اول و دوم برابر باشند، اعلان موفقیت و در غیر این صورت، تست با شکست رویرو شده و هر دو پارامتر نمایش داده می‌شوند.

```
//equal( actual, expected [, message ] )

test( "equal test", function() {
  equal( 0, 0, "Zero; equal succeeds" );
  equal( "", 0, "Empty, Zero; equal succeeds" );
  equal( "", "", "Empty, Empty; equal succeeds" );
  equal( 0, 0, "Zero, Zero; equal succeeds" );

  equal( "three", 3, "Three, 3; equal fails" );
  equal( null, false, "null, false; equal fails" );
});
```

زمانی که می‌خواهید مؤکداً شرط === را بررسی نمایید از ()strictEqual استفاده کنید.

تابع deepEqual:

تکمیل شده دو تابع قبل می‌باشد و حتی امکان مقایسه دو شی را نیز با هم دارا است. علاوه بر این، امکان مقایسه NaN، تاریخ، عبارات باقاعده، آرایه‌ها و توابع نیز وجود دارند.

```
//deepEqual( actual, expected [, message ] )

test( "deepEqual test", function() {
  var obj = { foo: "bar" };
```

```
  deepEqual( obj, { foo: "bar" }, "Two objects can be the same in value" );
});
```

در صورتیکه نمی‌خواهید محتوای دو مقدار را با هم مقایسه کنید، از equal استفاده نمایید اما عموما deepEqual انتخاب بهتری می‌باشد.

تست عملیات کاربر:

گاهی لازم است رویدادهایی که از عملیات کاربران صدا زده می‌شوند تست شوند. در این موارد با صدا زدن تابع trigger جی‌کوئری، تابع مورد نظر را تست نمایید. به مثال زیر توجه نمایید:

```
function KeyLogger( target ) {
  if ( !(this instanceof KeyLogger) ) {
    return new KeyLogger( target );
  }
  this.target = target;
  this.log = [];

  var self = this;

  this.target.off( "keydown" ).on( "keydown", function( event ) {
    self.log.push( event.keyCode );
  });
}
```

این مثال یک گزارش دهنده است و در صورتیکه کاربر، کلیدی را فشار دهد، کد آن را گزارش می‌دهد و در آرایه log ذخیره می‌نماید. حال لازم است بصورت دستی این رویداد را صدا زده و تایع را تست کنیم. تست را بصورت زیر می‌نویسیم:

```
test( "keylogger api behavior", function() {

  var event,
      $doc = $( document ),
      keys = KeyLogger( $doc );

  // trigger event
  event = $.Event( "keydown" );
  event.keyCode = 9;
  $doc.trigger( event );

  // verify expected behavior
  equal( keys.log.length, 1, "a key was logged" );
  equal( keys.log[ 0 ], 9, "correct key was logged" );

});
```

برای این کار تابع KeyLogger را با شی document جی کوئری صدا زدیم و نتیجه را در متغییر keys قرار داده‌ایم. بعد رویداد keydown را با کد 9 پرکرده تایع trigger متغییر $doc را با مقدار event صدا زده‌ایم که در واقع بصورت دستی، یک رویداد اتفاق افتاده است. در آخر هم با اعلان equal تست واحد را انجام داده‌ایم.

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.

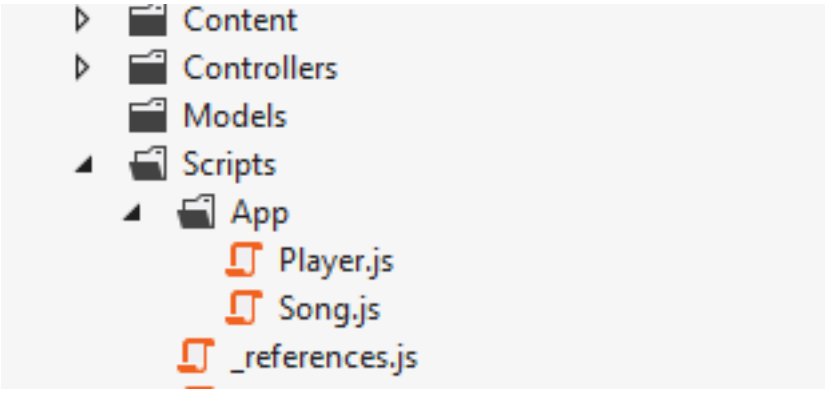
اگر در کد تست خود قصد دارید که یک تابع describe یا it را غیر فعال کنید کافیست یک x به ابتدای آن‌ها اضافه کنید و دیگر نیاز به هیچ کار اضافه دیگری برای comment کردن کد نیست.

```
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
```

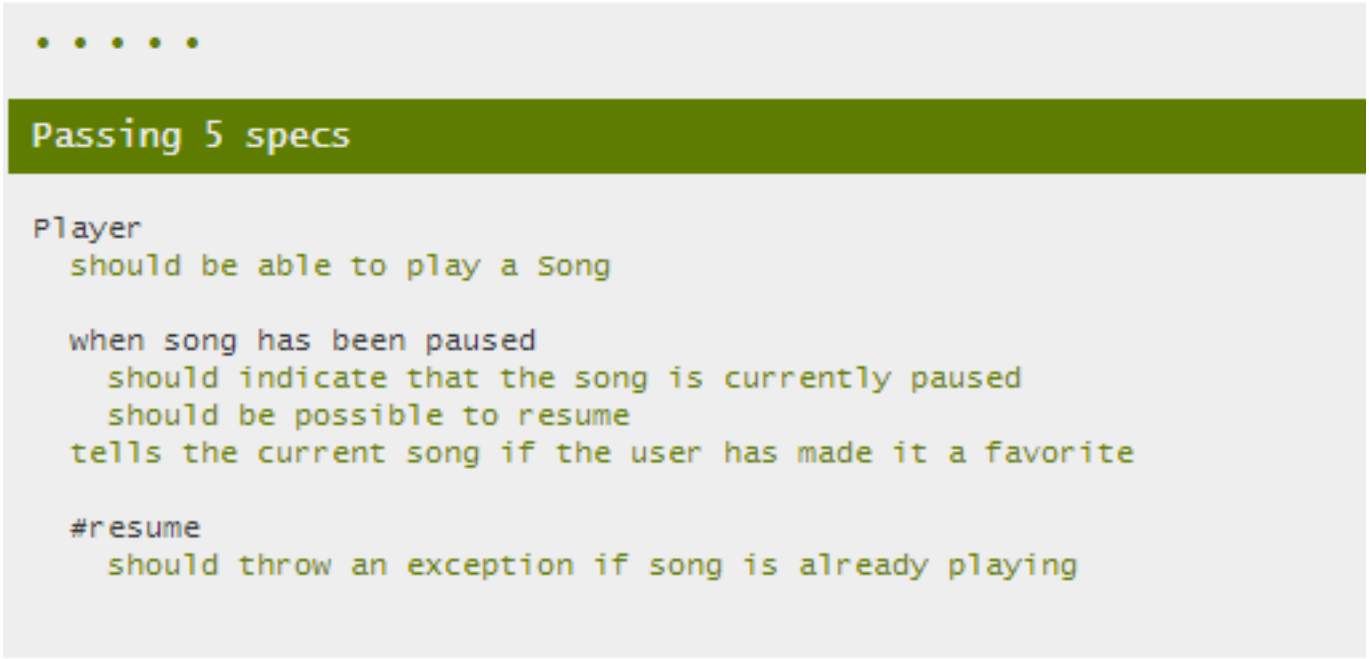
یادآوری :

»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>();
}
```

عنوان: **آزمون واحد Entity Framework به کمک چارچوب تقلید**
نویسنده: شاهین کیاست
تاریخ: ۱۳۹۳/۰۸/۱۷ ۱۲:۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Unit testing, Dependency Injection, Moq

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 میاید 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 که در آن مستقیما 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 مستقیما وهله 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 مستقیما 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، مستقیما 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، لطفا 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 می توان از بسته‌ی نیوگت آن بهره جست. پیشتر مطلبی در رابطه با چارچوب‌های تقلید در سایت نوشته شده است.
با توجه به اینکه PoductService 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 DbSet شده 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 شده اند. 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<Entites>();
            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 کالاها 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 تزریق وابستگی و آزمون‌های واحد نوشته شده است.

نظرات خوانندگان

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

-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 مرداف
تاریخ: ۱۳۹۳/۱۱/۱۸ ۰:۳۶

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)

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.

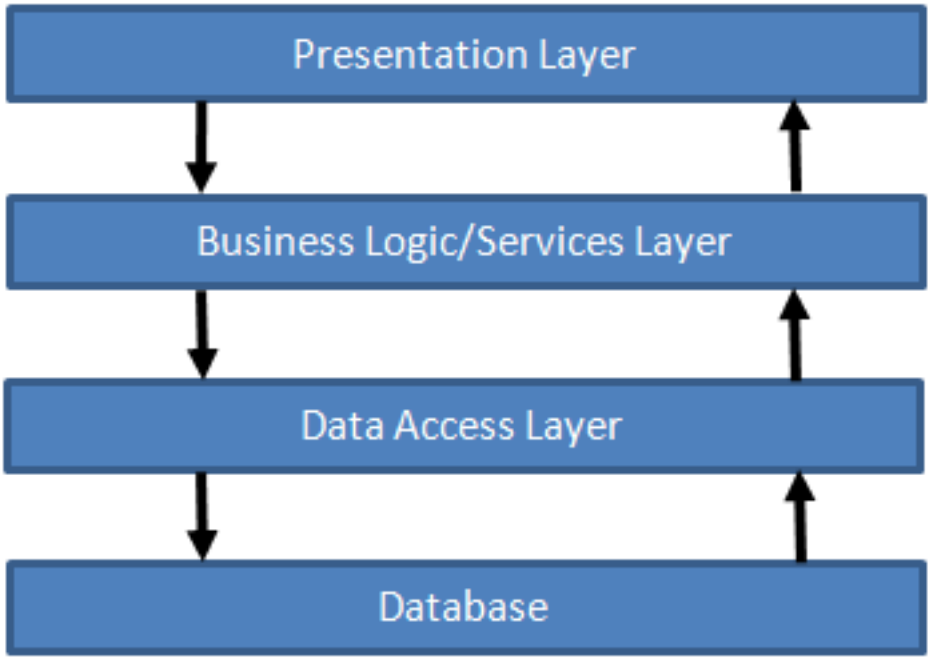
عنوان: **چگونه کد قابل تست بنویسیم - قسمت اول**
نویسنده: محمد بنازاده
تاریخ: ۱۳۹۳/۱۰/۰۱ ۱۳:۲۵
آدرس: www.dotnettips.info
گروه‌ها: Unit testing, Dependency Injection, Mock

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.