Baby steps: Basic Unit Testing

Our running example

For this exercise, we will consider a frequent flyer scheme.

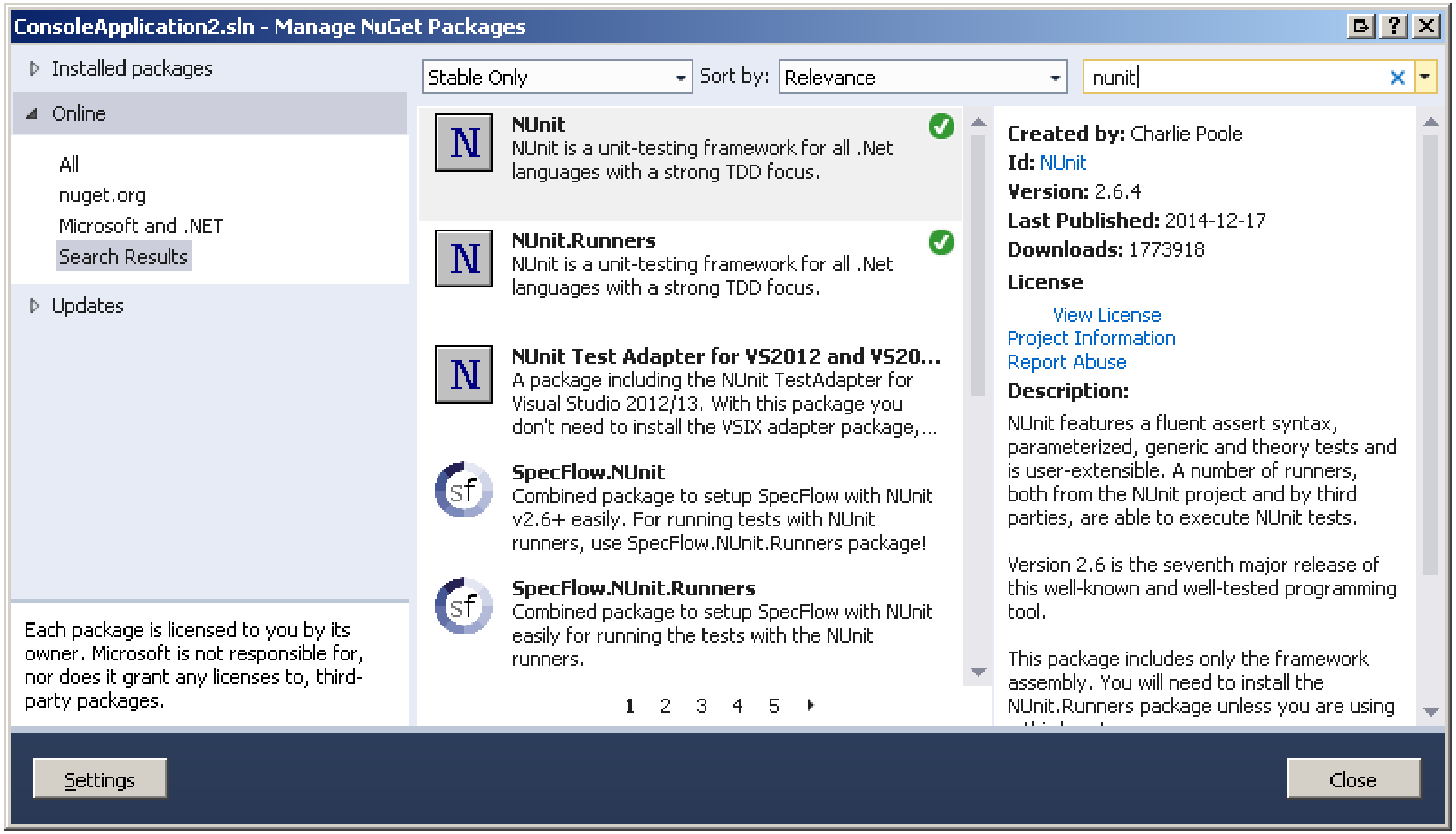
Our task is to implement the status calculator.

As points are accumulated, flyers move up towards a higher status. The status calculator keeps track of the points earned and can determine a flyer's current status.

We will use NUnit in as our testing framework.

Getting NUnit

The easiest way to get NUnit is with NuGet. Search for and install both **NUnit** (the testing framework itself) and **NUnit.Runners** (an external test runner).



ConsoleApplication2.sln - Manage NuGet Packages

Stable Only | Sort by: Relevance | nunit

NUnit
NUnit is a unit-testing framework for all .Net languages with a strong TDD focus.

NUnit.Runners
NUnit is a unit-testing framework for all .Net languages with a strong TDD focus.

NUnit Test Adapter for VS2012 and VS20...
A package including the NUnit TestAdapter for Visual Studio 2012/13. With this package you don't need to install the VSIX adapter package,...

SpecFlow.NUnit
Combined package to setup SpecFlow with NUnit v2.6+ easily. For running tests with NUnit runners, use SpecFlow.NUnit.Runners package!

SpecFlow.NUnit.Runners
Combined package to setup SpecFlow with NUnit easily for running the tests with the NUnit runners.

Created by: Charlie Poole
Id: NUnit
Version: 2.6.4
Last Published: 2014-12-17
Downloads: 1773918
License
View License
Project Information
Report Abuse
Description:
NUnit features a fluent assert syntax, parameterized, generic and theory tests and is user-extensible. A number of runners, both from the NUnit project and by third parties, are able to execute NUnit tests.

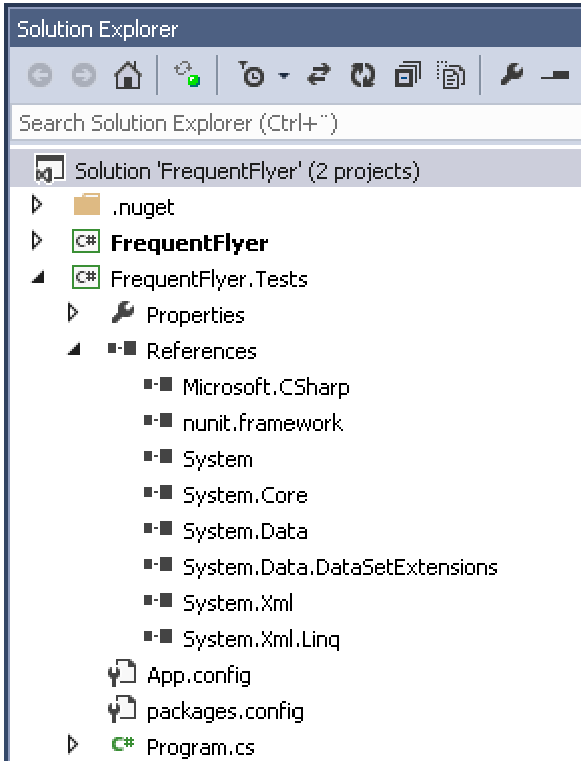
Version 2.6 is the seventh major release of this well-known and well-tested programming tool.

This package includes only the framework assembly. You will need to install the NUnit.Runners package unless you are using

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Our running example

We should always create a separate class-library project for our tests and **add a reference** to NUnit.



Solution Explorer

Search Solution Explorer (Ctrl+¨)

```
Solution 'FrequentFlyer' (2 projects)
  .nuget
  FrequentFlyer
  FrequentFlyer.Tests
    Properties
    References
      Microsoft.CSharp
      nunit.framework
      System
      System.Core
      System.Data
      System.Data.DataSetExtensions
      System.Xml
      System.Xml.Linq
    App.config
    packages.config
    Program.cs
```

Naming the project as **[ProjectName].Tests** is a good practice

Test fixtures and tests

Test cases are written as methods, marked with the **Test** attribute.

Such methods are written inside a class, which must be marked with the **TestFixture** attribute.

```
[TestFixture]
public class StatusCalculatorTests
{
    [Test]
    public void StartWithZeroPointsAndBasicStatus()
    {
        // Test code comes here
    }
}
```

A **TestFixture** class may contain one or multiple test methods.

An immediate design issue

TDD often helps bring design issues into focus. Right away, we should consider how the various status levels will be specified.

We'd like our tests not to break as our program evolves, and to control the data they are working with. Therefore, we will pass the status levels into the **StatusCalculator**`s constructor.

```
private List<Tuple<string, int>> statusLevels =
    new List<Tuple<string, int>> {
        new Tuple<string, int>("Basic", 0),
        new Tuple<string, int>("Silver", 50000),
        new Tuple<string, int>("Gold", 100000)
};

[Test]
public void StartWithZeroPointsAndBasicStatus()
{
    var sut = new StatusCalculator(statusLevels);
    // More to do here...
}
```

Class does not exist yet

Assertions

Asserts are used to check that some property or method of the object we are testing produces the expected value. Typically an assertion has:

- The expected value
- Something that obtains the value from the object
- A description

```
[Test]
public void StartWithZeroPointsAndBasicStatus()
{
    var sut = new StatusCalculator(statusLevels);

    Assert.AreEqual(0, sut.Points, "Zero points");
    Assert.AreEqual("Basic", sut.Status, "Basic status");
}
```

Method does not exist yet

Generate from usage

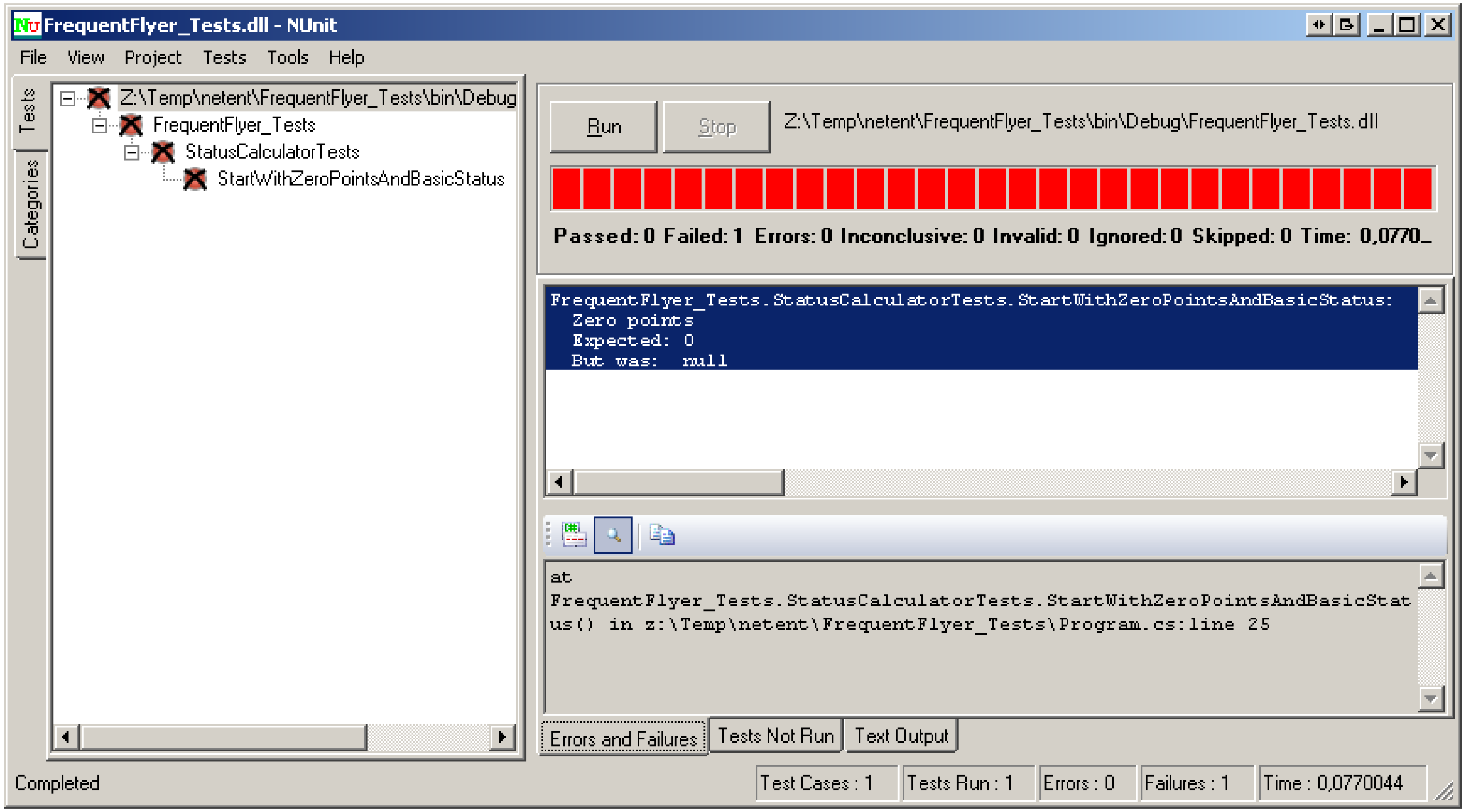
So far, our test won't even compile - we have no **StatusCalculator** class!

Using the **Generate From Usage** feature of Visual Studio, we can produce stubs of the class, constructor and properties.

```
public class StatusCalculator
{
    private readonly List<Tuple<string, int>> _statusLevels;

    public StatusCalculator(List<Tuple<string, int>> statusLevels)
    {
        _statusLevels = statusLevels;
    }

    public int Points { get; set; }

    public string Status { get; set; }
}
```

See the test fail

We can now compile our test and run it using the NUnit runner. Naturally, it fails.


FrequentFlyer_Tests.dll - NUnit
File View Project Tests Tools Help
Tests
Categories
Z:\Temp\netent\FrequentFlyer_Tests\bin\Debug
FrequentFlyer_Tests
StatusCalculatorTests
StartWithZeroPointsAndBasicStatus
Run
Stop
Z:\Temp\netent\FrequentFlyer_Tests\bin\Debug\FrequentFlyer_Tests.dll
Errors and Failures
Tests Not Run
Text Output

```
Passed: 0 Failed: 1 Errors: 0 Inconclusive: 0 Invalid: 0 Ignored: 0 Skipped: 0 Time: 0,0770_
```

```
FrequentFlyer_Tests.StatusCalculatorTests.StartWithZeroPointsAndBasicStatus:
  Zero points
  Expected: 0
  But was:  null
```

```
at
FrequentFlyer_Tests.StatusCalculatorTests.StartWithZeroPointsAndBasicStat
us() in z:\Temp\netent\FrequentFlyer_Tests\Program.cs:line 25
```

```
Completed    Test Cases : 1   Tests Run : 1   Errors : 0   Failures : 1   Time : 0,0770044
```

Why might we want to see our test fail before we work on the feature it exercises?

Do as little as needed to make the test pass

The only thing we need to do to make the test pass is return sensible values from the **Points** and **Status** accessors.

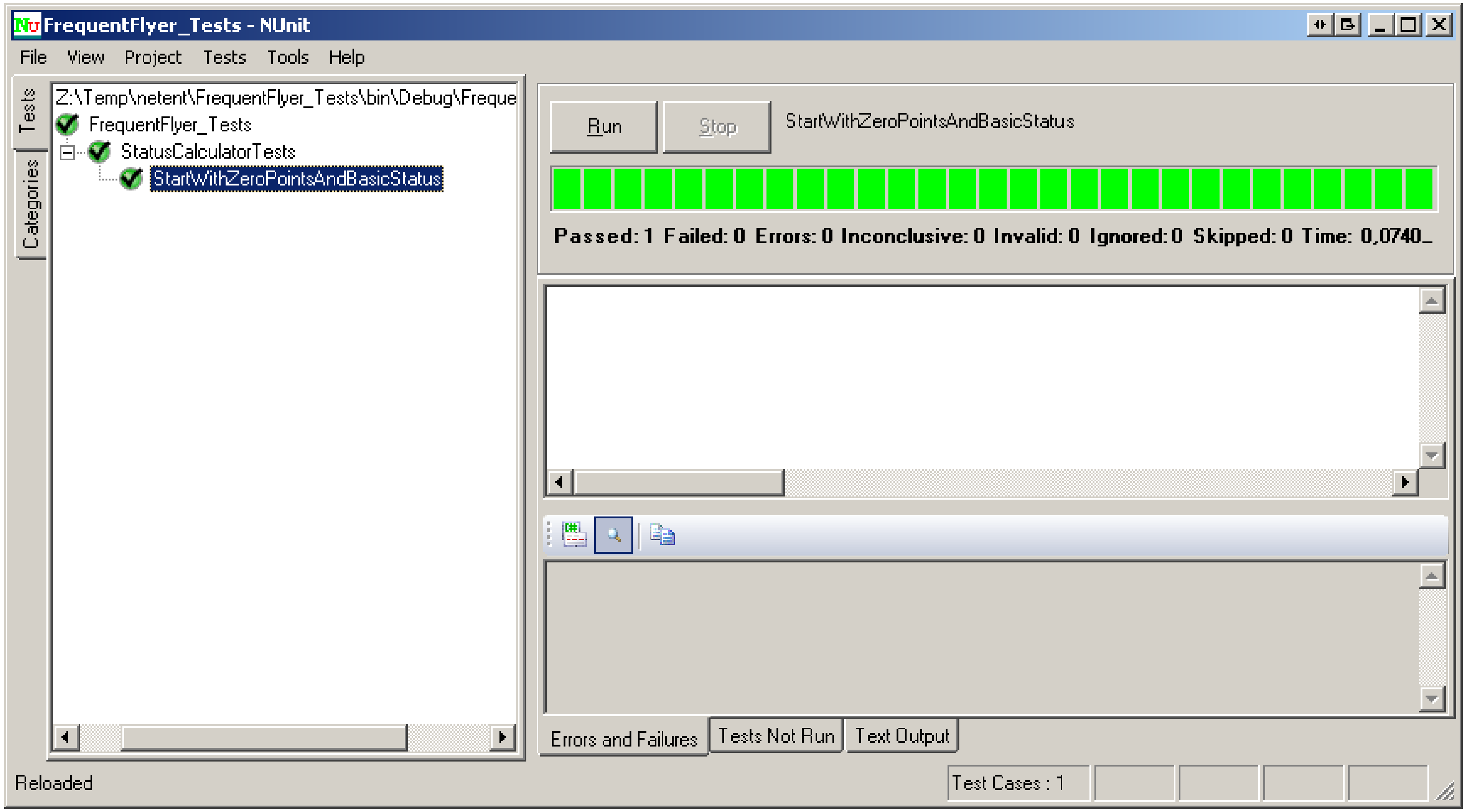
```
public int Points { get { return 0; } }

public string Status { get { return statusLevels[0].Item1; } }
```

When starting out with TDD, it's best to try and avoid thinking too far ahead. We could add many more things at this point - but try to resist the temptation. Sometimes, it can be surprising to learn what you don't need!

"Always implement things when you **actually** need them,
never when you just **foresee** that you need them."
http://c2.com/xp/YouArentGonnaNeedIt.html

Now we have green

With the changes on the previous slide, we now have a passing test.



```
Z:\Temp\netent\FrequentFlyer_Tests\bin\Debug\Freque
✓ FrequentFlyer_Tests
  ✓ StatusCalculatorTests
    ✓ StartWithZeroPointsAndBasicStatus
```

StartWithZeroPointsAndBasicStatus

Passed: 1 Failed: 0 Errors: 0 Inconclusive: 0 Invalid: 0 Ignored: 0 Skipped: 0 Time: 0,0740

Reloaded

Test Cases : 1

At this point, our code is extremely simple, and there is no interesting refactor that we could do. Thus, we move on.

The "System Under Test" convention

A unit test should exercise a single unit (typically, an object). At the moment there is only one object involved in our tests, but in the future there may be others (for example, mock of stub objects).

Therefore, it is useful to adopt a naming convention for the object that is the **S**ystem **U**nder **T**est, so it can be easily identified. In this course, we will use the name **sut** consistently for this.

```
[Test]
public void StartWithZeroPointsAndBasicStatus()
{
    var sut = new StatusCalculator(statusLevels);

    Assert.AreEqual(0, sut.Points, "Zero points");
    Assert.AreEqual("Basic", sut.Status, "Basic status");
}
```

Testing point counting

We know from our requirements that we need a way to record when points are earned. We therefore write a test to exercise a (to be implemented) **AddPoints** method.

```
[Test]
public void RecordsExtraPointsAndKeepsCorrectStatus()
{
    var sut = new StatusCalculator(statusLevels);

    sut.AddPoints(2000);
    sut.AddPoints(5000);

    Assert.AreEqual(7000, sut.Points, "Correct number of points");
    Assert.AreEqual("Basic", sut.Status, "Still have basic status");
}
```

Method not implemented yet

The "Arrange, Act, Assert" practice

Most tests can be broken up into three distinct phases.

Phase	Description
Arrange	Get the System Under Test into a starting state for the test.
Act	Exercise the System Under Test, typically by calling a method.
Assert	Check that the consequences of the action were as expected.

Arranging your tests into these 3 distinct phases will help to make them more understandable, but also help you to get your tests to be a sensible size.

The "Arrange, Act, Assert" subdivision is considered good practice. Follow it unless there is a really good reason not to.

The "Arrange, Act, Assert" practice in our test

EDUMENT

Most tests can be broken up into three distinct phases.

Arr

```
var sut = new StatusCalculator(statusLevels);
```

Act

```
sut.AddPoints(2000);
sut.AddPoints(5000);
```

Ass

```
Assert.AreEqual(7000, sut.Points, "Correct number of points");
Assert.AreEqual("Basic", sut.Status, "Still have basic status");
```

Using this **AAA** convention will make your tests easy to read, maintain and share with other developers.

Passing the test

Add a private field to record the points:

```
private int points = 0;
```

Update the property to return it:

```
public int Points { get { return points; } }
```

And finally, implement the AddPoints method:

```
public void AddPoints(int p)
{
    points += p;
}
```

Passing the test

EDUMENT

Now our class under test should look something like:

```
public class StatusCalculator
   {
       private List<Tuple<string, int>> statusLevels;
       private int points = 0;

       public int Points { get { return points; } }
       public string Status { get { return statusLevels[0].Item1; } }

       public StatusCalculator(List<Tuple<string, int>> statusLevels)
       {
           this.statusLevels = statusLevels;
       }

       public void AddPoints(int p)
       {
           points += p;
       }
   }
```

A small refactor

We may not have much code yet, but that doesn't mean there's nothing wrong with it. In fact, this line...

```
points += p;
```

...is a pretty good hint that we should consider some variable renaming.

We rename the field to **currentPoints** and the parameter to **newPoints** to make the code clearer.

```
public void AddPoints(int newPoints)
{
    currentPoints += newPoints;
}
```

Testing that upgrades are given as expected

The next test adds enough points to reach silver status, then asserts that the status is correctly reported as silver.

```
[Test]
public void EnoughPointsEarnSilverStatus()
{
    var sut = new StatusCalculator(statusLevels);

    sut.AddPoints(20000);
    sut.AddPoints(35000);

    Assert.AreEqual(55000, sut.CurrentPoints, "Correct number of points");
    Assert.AreEqual("Silver", sut.Status, "Upgraded to silver status");
}
```

Now we need to implement this failing test.

Isolated changes

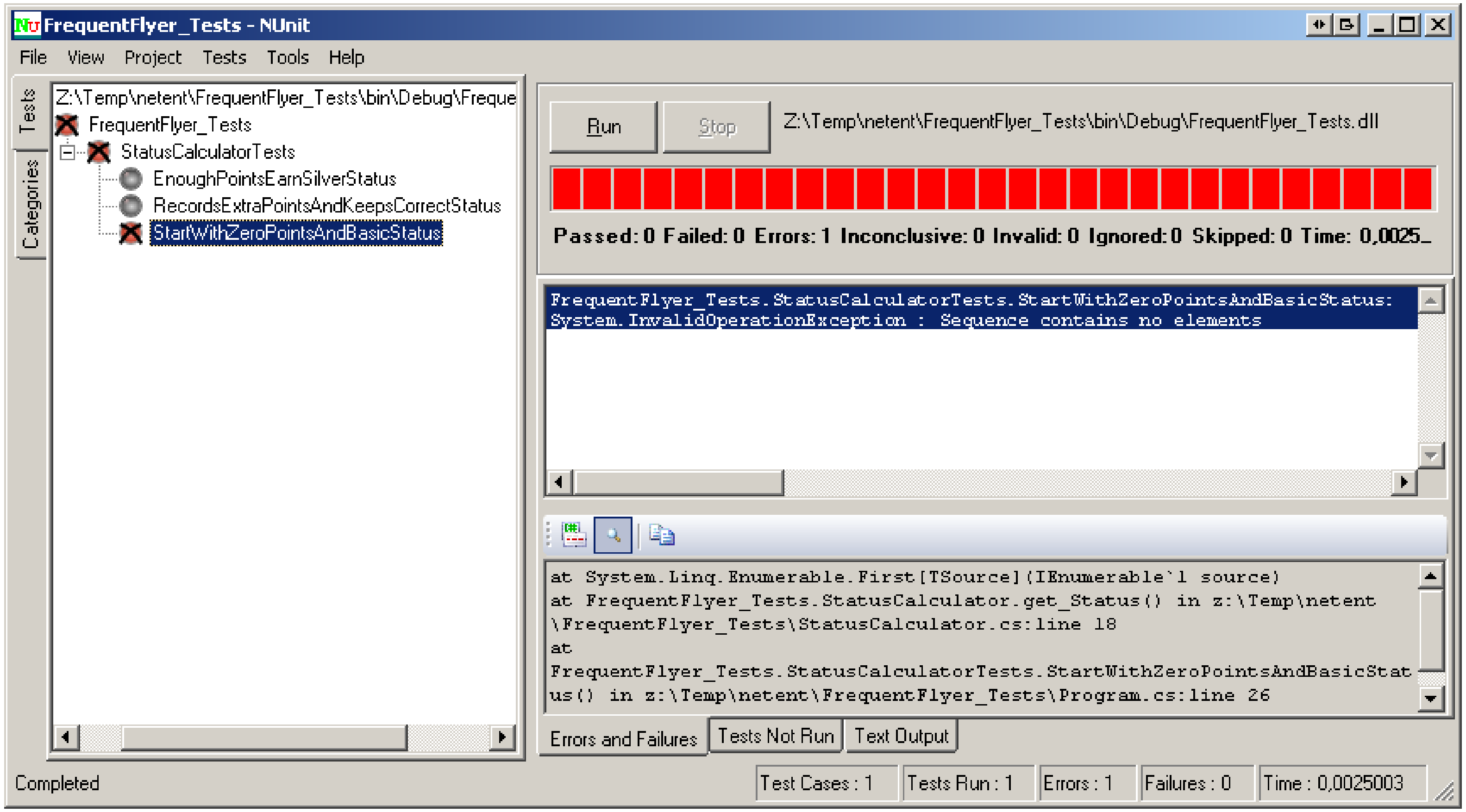
We may be tempted to add a field for the current status and update it as points are added. However, there's a simpler way that needs an update to just one property: **calculate it on demand.**

```
public string Status
{
    get
    {
        return (from sl in statusLevels
                where sl.Item2 < currentPoints
                orderby sl.Item2 descending
                select sl.Item1).First();
    }
}
```

In short, this takes the status levels, chooses those whose point requirement is below what we currently have, sorts them and then chooses the name of the first status.

Oops, regression!

The code seemed reasonable enough. And it does, in fact, pass the test that we just wrote.



```
Z:\Temp\netent\FrequentFlyer_Tests\bin\Debug\Freque
FrequentFlyer_Tests
  StatusCalculatorTests
    EnoughPointsEarnSilverStatus
    RecordsExtraPointsAndKeepsCorrectStatus
    StartWithZeroPointsAndBasicStatus
```

```
Z:\Temp\netent\FrequentFlyer_Tests\bin\Debug\FrequentFlyer_Tests.dll
Passed: 0 Failed: 0 Errors: 1 Inconclusive: 0 Invalid: 0 Ignored: 0 Skipped: 0 Time: 0,0025
```

```
FrequentFlyer_Tests.StatusCalculatorTests.StartWithZeroPointsAndBasicStatus:
System.InvalidOperationException : Sequence contains no elements
```

```
at System.Linq.Enumerable.First[TSource](IEnumerable`1 source)
at FrequentFlyer_Tests.StatusCalculator.get_Status() in z:\Temp\netent
\FrequentFlyer_Tests\StatusCalculator.cs:line 18
at
FrequentFlyer_Tests.StatusCalculatorTests.StartWithZeroPointsAndBasicStat
us() in z:\Temp\netent\FrequentFlyer_Tests\Program.cs:line 26
```

```
Completed        Test Cases : 1   Tests Run : 1   Errors : 1   Failures : 0   Time : 0,0025003
```

However, we broke a previously passing test! Our tests have caught a regression: having zero points no longer indicates basic status.

Fixing the regression

EDUMENT

The fix comes from realizing that you are eligible for a level as soon as you have reached the required number of points. Thus, the **<** should have been **<=**.

```
public string Status
{
    get
    {
        return (from sl in statusLevels
                where sl.Item2 <= currentPoints
                orderby sl.Item2 descending
                select sl.Item1).First();
    }
}
```

Now, our tests pass.

(Stop and ponder all the ways this regression could be discovered later if the tests hadn't caught it. Which way do you prefer?)

Summing up what we've done

EDUMENT

We've already started to see how tests influence the design process, and once they are passing they serve as regression tests.

Some key things to remember:

- A unit test should be given a **descriptive, but concise name**.
- It is wise to check that your tests **fail first**, before working on making them pass.
- The **Arrange Act Assert** practice helps divide a test into easily recognizable parts.
- The **S**ystem **U**nder **T**est naming convention helps to pinpoint the unit we're currently testing.

EDUMENT

Test Explorer

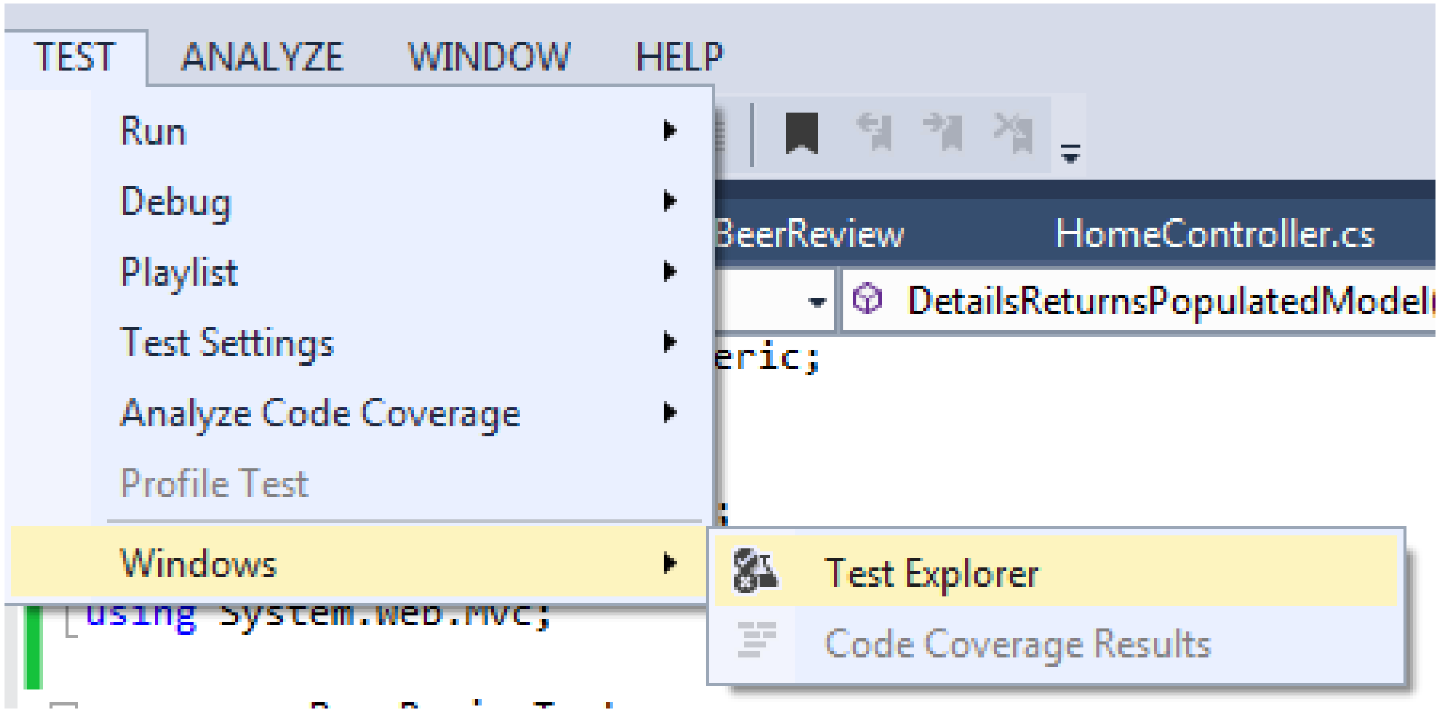
Test Explorer

Running tests using the **NUnit test runner** works, but it might not so practical in the long run.

In Visual Studio we have a **Test Explorer** that allows us to run our unit tests within Visual Studio.

Test Explorer

We open the test explorer via
Test* ⇨ *Windows* ⇨ *Test Explorer menu



Test Explorer

But by default Test Explorer **does not support** NUnit tests.

To add this support we need to first add **a test adapter** for NUnit.

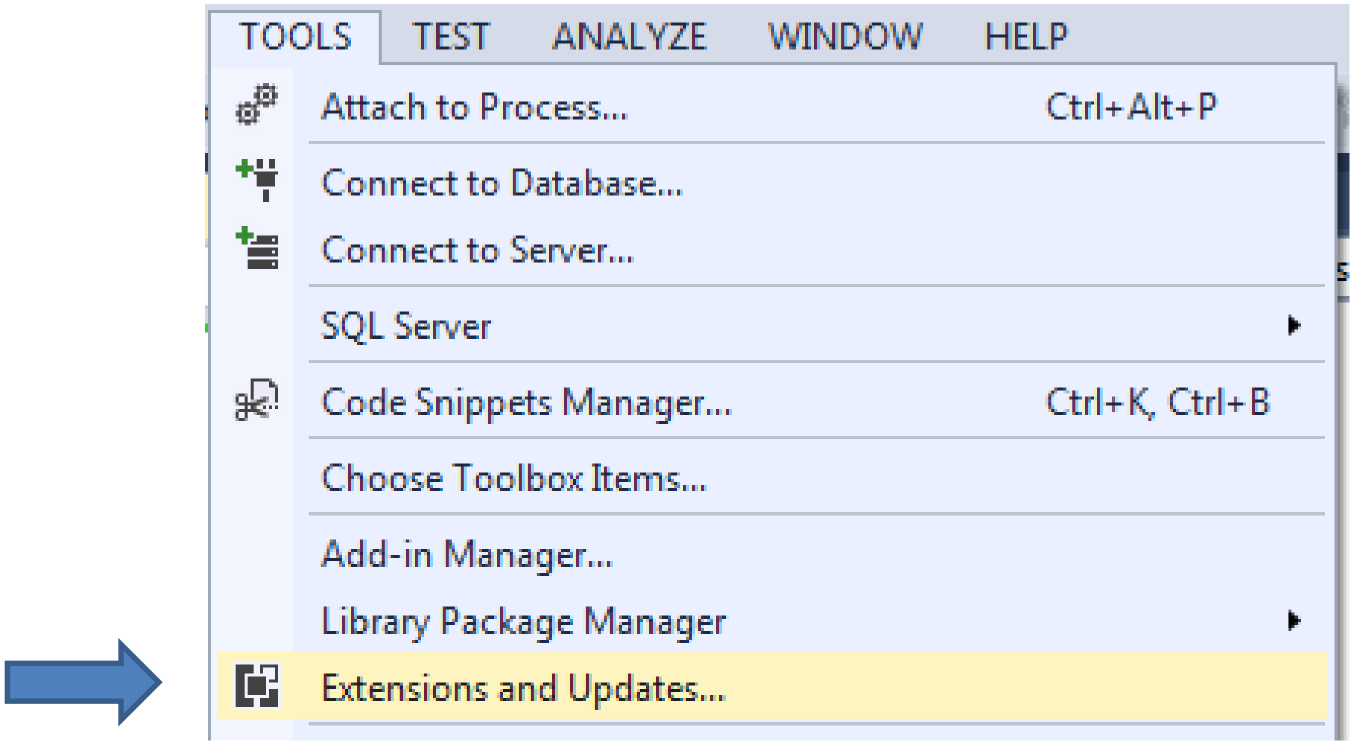
We do this simplest using the built in "**Extensions and Update**" feature.

Read more about the adapter here:
http://nunit.org/index.php?p=vsTestAdapter&r=2.6.4

Test Explorer

EDUMENT

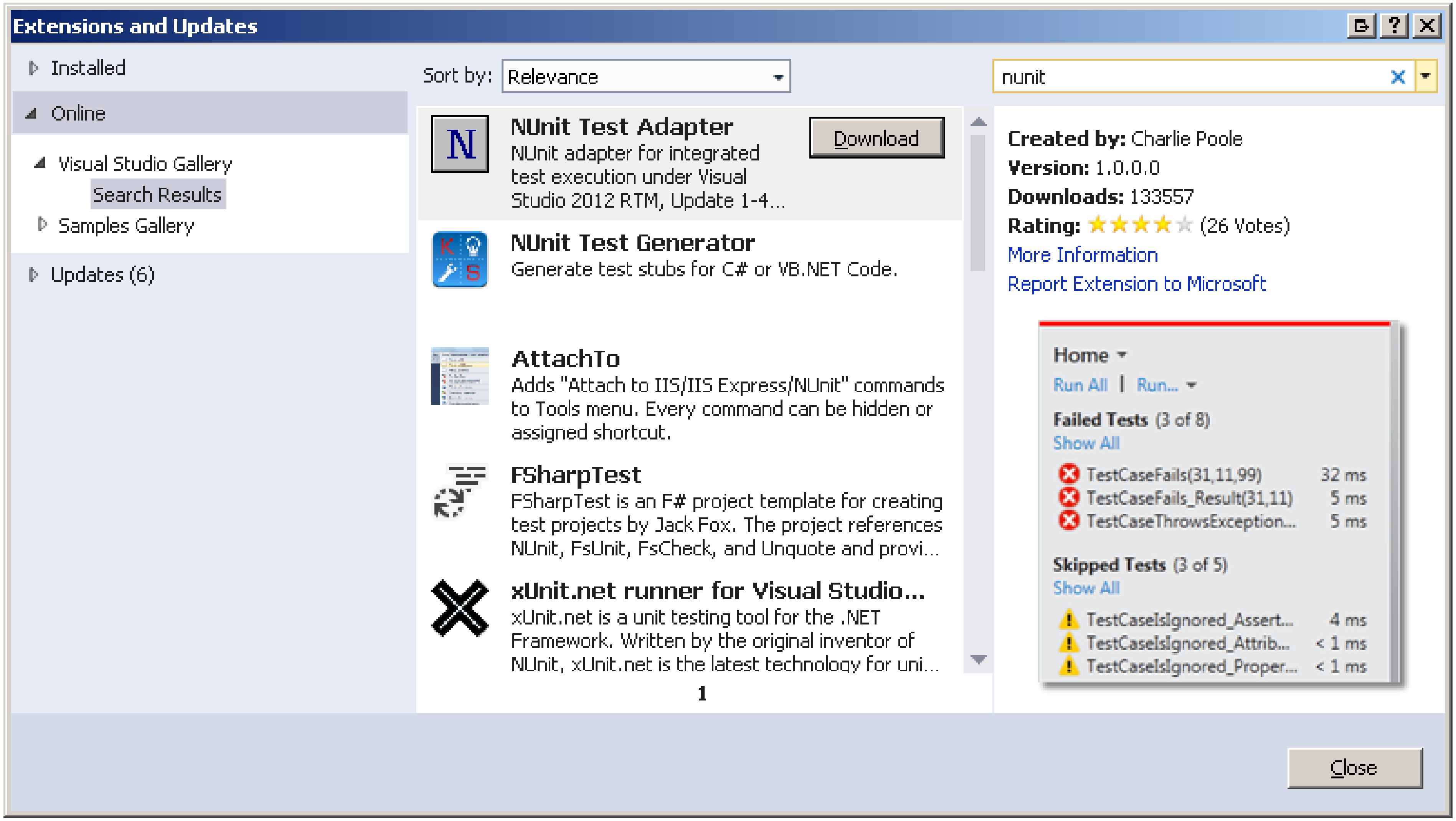
Select the "Extensions and Updates…" option on the Tools menu.



Test Explorer

EDUMENT

Search for NUnit test adapter and install it:



Extensions and Updates

- Installed
- Online
  - Visual Studio Gallery
    - Search Results
  - Samples Gallery
- Updates (6)

Sort by: Relevance

nunit

NUnit Test Adapter — Download
NUnit adapter for integrated test execution under Visual Studio 2012 RTM, Update 1-4...

NUnit Test Generator
Generate test stubs for C# or VB.NET Code.

AttachTo
Adds "Attach to IIS/IIS Express/NUnit" commands to Tools menu. Every command can be hidden or assigned shortcut.

FSharpTest
FSharpTest is an F# project template for creating test projects by Jack Fox. The project references NUnit, FsUnit, FsCheck, and Unquote and provi...

xUnit.net runner for Visual Studio...
xUnit.net is a unit testing tool for the .NET Framework. Written by the original inventor of NUnit, xUnit.net is the latest technology for uni...

1

Created by: Charlie Poole
Version: 1.0.0.0
Downloads: 133557
Rating: ★★★★☆ (26 Votes)
More Information
Report Extension to Microsoft

Home
Run All | Run...

Failed Tests (3 of 8)
Show All

```
TestCaseFails(31,11,99)        32 ms
TestCaseFails_Result(31,11)     5 ms
TestCaseThrowsException...      5 ms
```

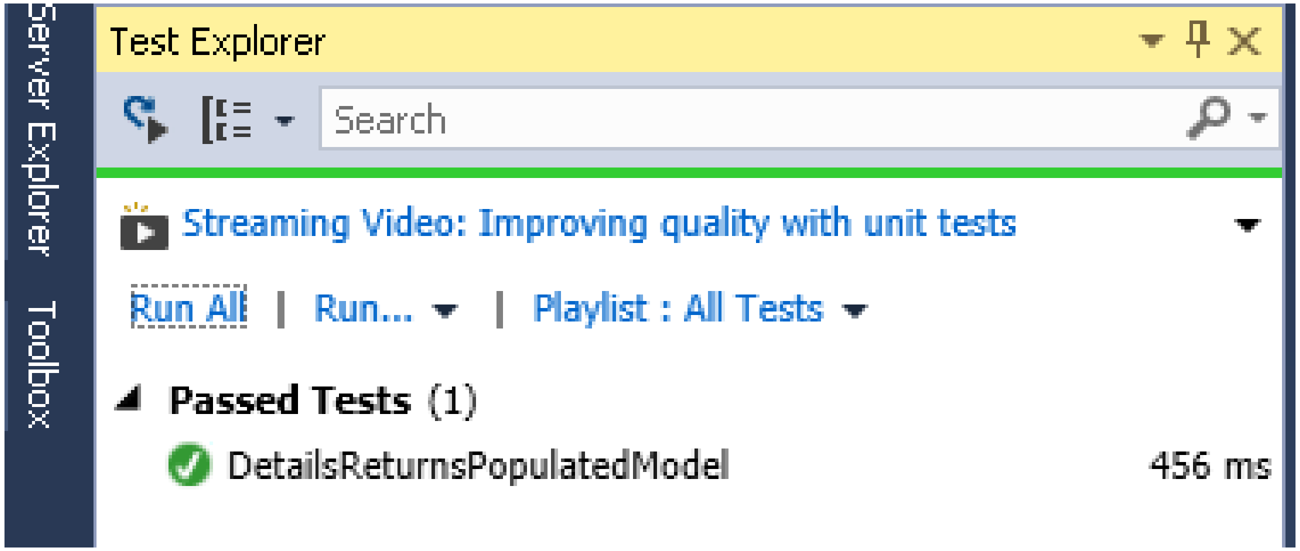
Skipped Tests (3 of 5)
Show All

```
TestCaseIsIgnored_Assert...     4 ms
TestCaseIsIgnored_Attrib...   < 1 ms
TestCaseIsIgnored_Proper...   < 1 ms
```

Close

Test Explorer

After a **restart** of Visual Studio we can now run our unit tests from the Test Explorer.



Test Explorer

Search

Streaming Video: Improving quality with unit tests

Run All | Run... | Playlist : All Tests

Passed Tests (1)

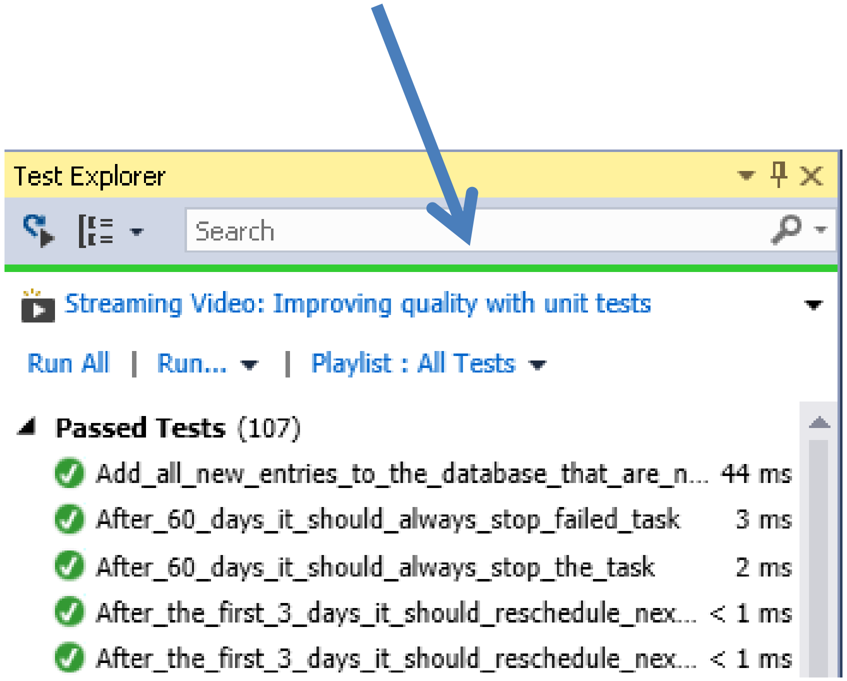
DetailsReturnsPopulatedModel 456 ms

Click on "Run All" to run all the tests discovered by the test explorer.

Test Explorer

EDUMENT

The line under the search box indicates the current status of the tests.



Test Explorer

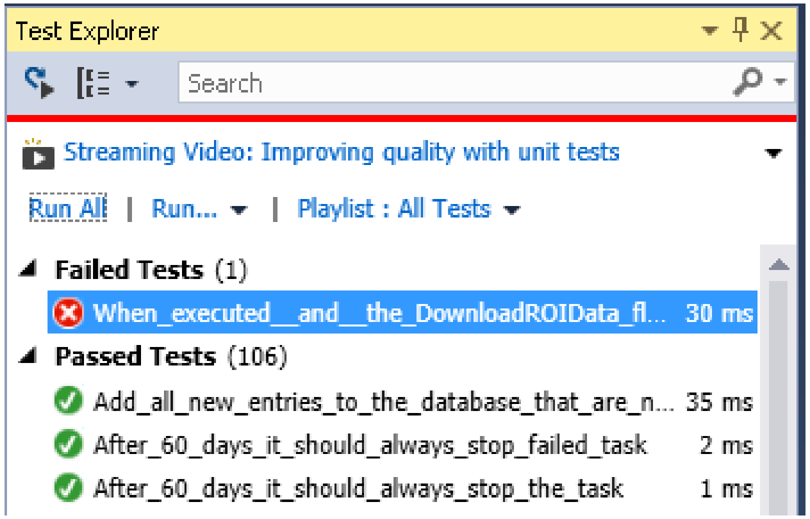
Search

Streaming Video: Improving quality with unit tests

Run All | Run... | Playlist : All Tests

Passed Tests (107)

Add_all_new_entries_to_the_database_that_are_n...	44 ms
After_60_days_it_should_always_stop_failed_task	3 ms
After_60_days_it_should_always_stop_the_task	2 ms
After_the_first_3_days_it_should_reschedule_nex...	< 1 ms
After_the_first_3_days_it_should_reschedule_nex...	< 1 ms



Test Explorer

Search

Streaming Video: Improving quality with unit tests

Run All | Run... | Playlist : All Tests

Failed Tests (1)

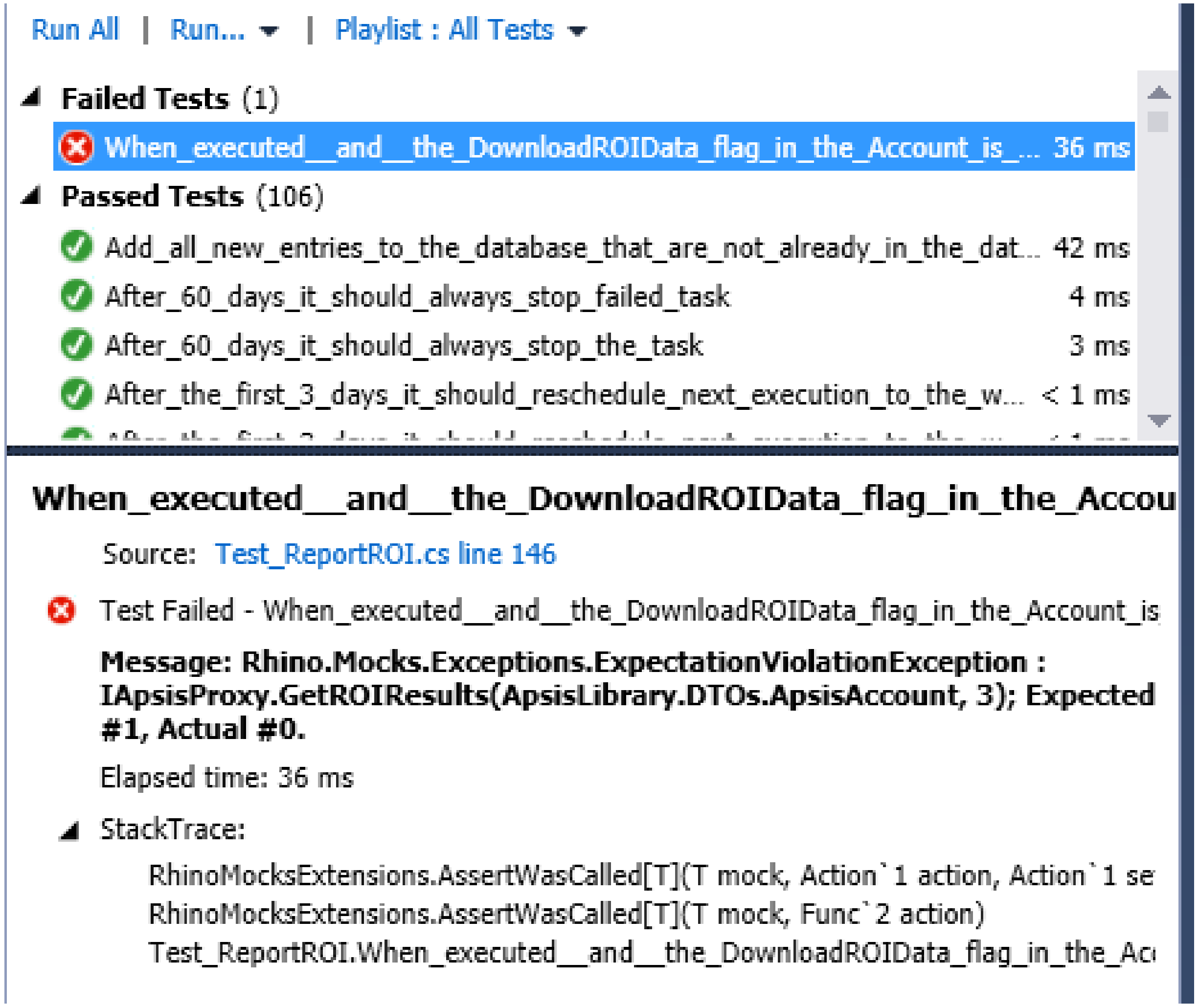
When_executed__and__the_DownloadROIData_fl...	30 ms

Passed Tests (106)

Add_all_new_entries_to_the_database_that_are_n...	35 ms
After_60_days_it_should_always_stop_failed_task	2 ms
After_60_days_it_should_always_stop_the_task	1 ms

Test Explorer

At the bottom of the Test Explorer we see the details about individual tests:


Run All | Run... ▾ | Playlist : All Tests ▾

◢ Failed Tests (1)

```
When_executed__and__the_DownloadROIData_flag_in_the_Account_is_...  36 ms
```

◢ Passed Tests (106)

```
Add_all_new_entries_to_the_database_that_are_not_already_in_the_dat...  42 ms
After_60_days_it_should_always_stop_failed_task                           4 ms
After_60_days_it_should_always_stop_the_task                              3 ms
After_the_first_3_days_it_should_reschedule_next_execution_to_the_w...  < 1 ms
```

When_executed__and__the_DownloadROIData_flag_in_the_Accou

Source: Test_ReportROI.cs line 146

Test Failed - When_executed__and__the_DownloadROIData_flag_in_the_Account_is

Message: Rhino.Mocks.Exceptions.ExpectationViolationException : IApsisProxy.GetROIResults(ApsisLibrary.DTOs.ApsisAccount, 3); Expected #1, Actual #0.

Elapsed time: 36 ms

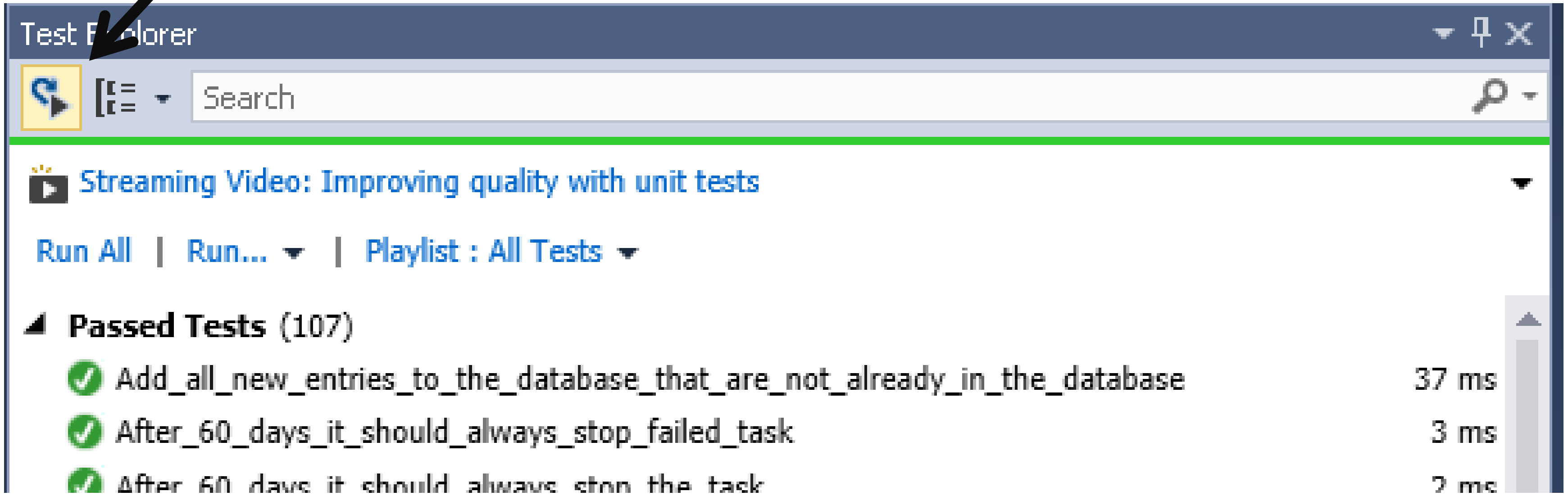
◢ StackTrace:

```
RhinoMocksExtensions.AssertWasCalled[T](T mock, Action`1 action, Action`1 se
RhinoMocksExtensions.AssertWasCalled[T](T mock, Func`2 action)
Test_ReportROI.When_executed__and__the_DownloadROIData_flag_in_the_Ac
```

Test Explorer

We can let the test explorer to automatically run all the tests after each build:

Run tests after build button



Test Explorer

Search

Streaming Video: Improving quality with unit tests

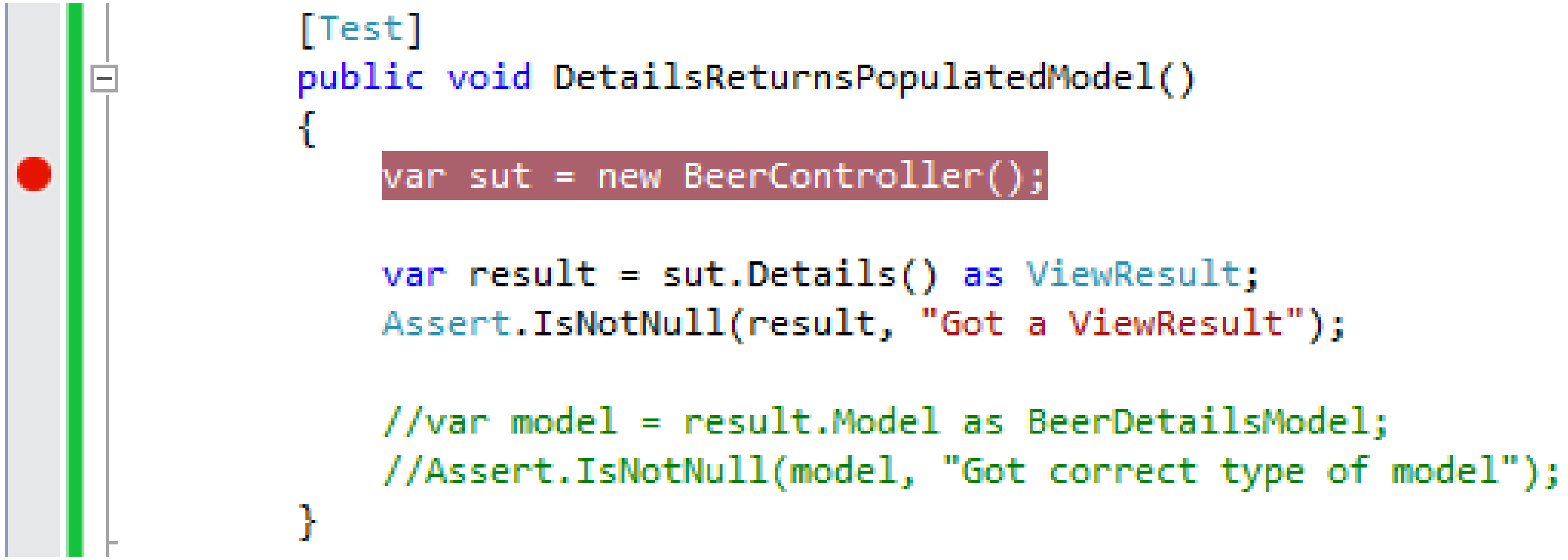
Run All | Run... | Playlist : All Tests

Passed Tests (107)

```
Add_all_new_entries_to_the_database_that_are_not_already_in_the_database   37 ms
After_60_days_it_should_always_stop_failed_task                             3 ms
After_60_days_it_should_always_stop_the_task                                2 ms
```

Test Explorer

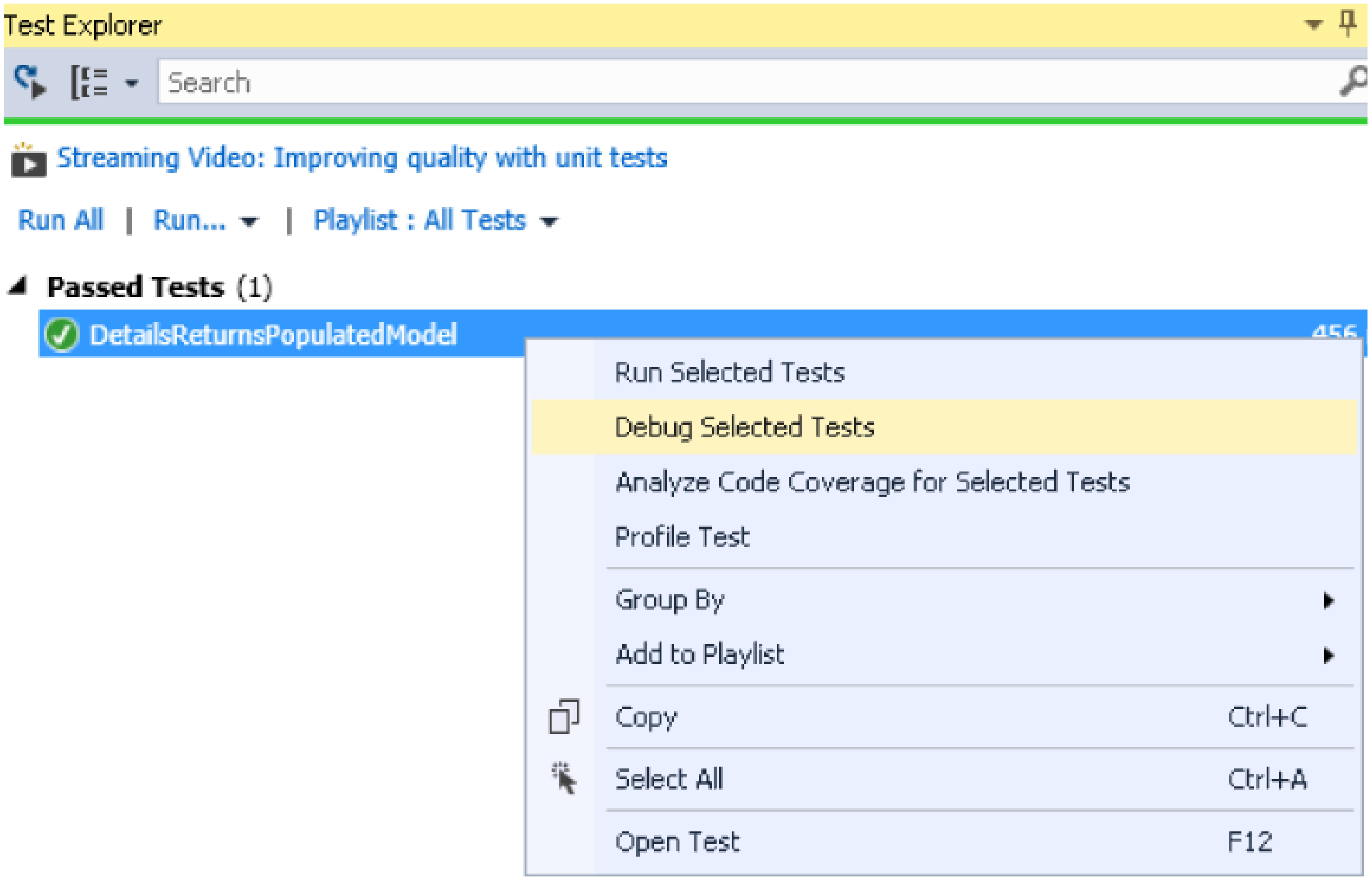
To debug our unit tests we can add a break point in our tests.



```
[Test]
public void DetailsReturnsPopulatedModel()
{
    var sut = new BeerController();

    var result = sut.Details() as ViewResult;
    Assert.IsNotNull(result, "Got a ViewResult");

    //var model = result.Model as BeerDetailsModel;
    //Assert.IsNotNull(model, "Got correct type of model");
}
```

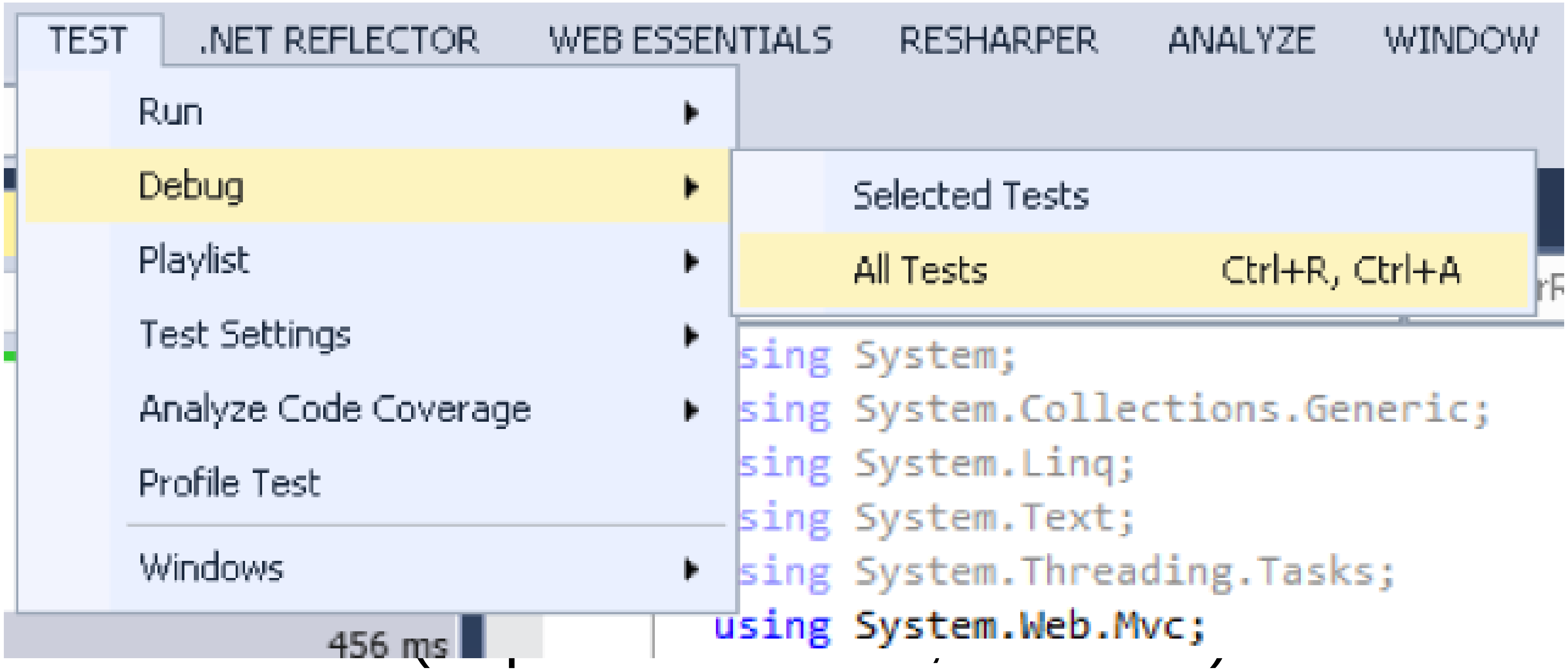
Test Explorer

Then we can right click on a test and choose “Debug Selected Tests”


Test Explorer
Search
Streaming Video: Improving quality with unit tests
Run All | Run... | Playlist : All Tests
Passed Tests (1)
DetailsReturnsPopulatedModel
Run Selected Tests
Debug Selected Tests
Analyze Code Coverage for Selected Tests
Profile Test
Group By
Add to Playlist
Copy Ctrl+C
Select All Ctrl+A
Open Test F12

Test Explorer

We can also debug all tests by selecting
Test ➪ Debug ➪ All Tests



```
sing System;
sing System.Collections.Generic;
sing System.Linq;
sing System.Text;
sing System.Threading.Tasks;
using System.Web.Mvc;
```

Test Explorer

Using different **test adapters**, the test explorer supports many different testing frameworks, including:

Test framework
Xunit
xRM
Abel
SpecFlow
Chutzpah
MbUnit
Qunit
Jasmine
….

Exercise #1

In this exercise, you will get started with setting up **NUnit** and writing your first unit test.

You will create **two** projects, one for tests and one for business logic.

The application that you'll write tests for (and develop) is a simple email validation engine.