

Code Digging with Pex
Code Understanding and Automatic Testing
At Your Fingertips

(Rather Short Tutorial)

Nikolai Tillmann and Peli de Halleux,
Research in Software Engineering,
Microsoft Research.

Preliminary Draft
Copyright Microsoft Corporation.

October 21, 2008

Abstract

Pex is a new tool that helps in understanding the behavior of .NET code, debugging issues, and in creating a test suite that covers all corner cases – fully automatically. Through a context menu in the code editor, the user can invoke Pex to analyze an entire class or a single method. For any method, Pex computes and displays interesting input-output pairs. Pex systematically hunts for bugs – exceptions or assertion failures. As Pex discovers boundary conditions in code, Pex generates new tests that target these conditions. The result is a small test suite with high code coverage. Pex enables Parameterized Unit Testing, an extension of traditional unit testing that reduces test maintenance costs. Pex has been used in Microsoft to test core .NET components. Pex is developed at Microsoft Research and is integrated into Microsoft Visual Studio.

1 Introduction

Software developers strive for high-quality products that ships on time. Automated tools are essential for ensuring code quality. Pex is a new tool that helps in understanding the behavior of .NET code, debugging issues, and in creating a test suite that covers all corner cases – fully automatically.

This tutorial describes how to use Pex in Visual Studio:

- How to analyze existing code with a few clicks in the code editor.
- How to create test cases that reproduce issues that Pex finds.
- How to debug such issues.
- How to let Pex generate and save an entire test suite, which is small yet often achieves high code coverage.

Pex enables Parameterized Unit Testing, an extension to Unit Testing that reduces test creation and maintenance costs while uncovering more bugs.

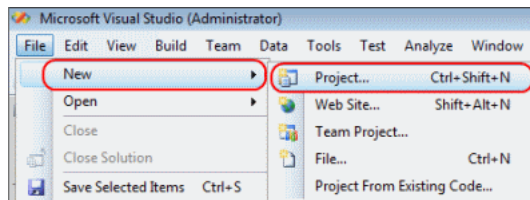
2 Running Pex for the first time

Note: We assume that you already installed Pex on your machine, and that you also have one of the Visual Studio 2010 Team System editions.

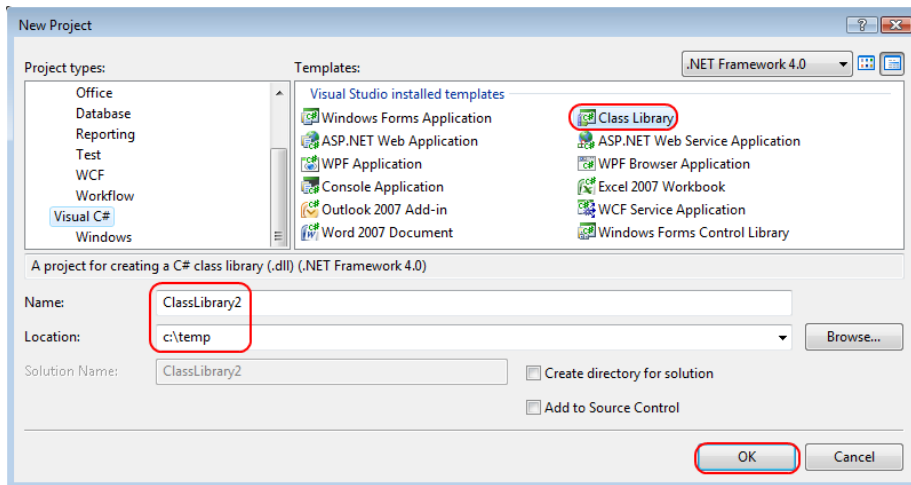
Let's start from scratch, and write some code. We will write a method that turns a string consisting of separate words into a capitalized identifier, where the leading characters of the original words have been turned into upper-case, and words are separated by underscores.

```
public static class StringExtensions {  
    // convert 'hello world' to 'HelloWorld'  
    // punctuation is turned into '_', others are ignored.  
    public static string Capitalize(string value) {  
        return null;  
    }  
}
```

First, we have to create a new project to hold our code. In Visual Studio, click on **File|New|Project...**



Select **Visual C#** on the left, and **Class Library** on the right, and press **OK**.



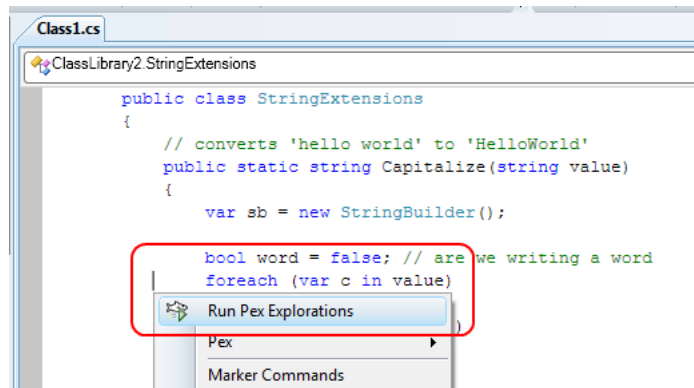
In the new project, rename `Class1` to `StringExtensions`. In this class, let's write the following implementation of `Capitalize`.

```
public static string Capitalize(string value)
{
    var sb = new StringBuilder();

    bool word = false; // are we writing a word
    foreach (var c in value) {
        if (char.IsLetter(c)) {
            if (!word) {
                sb.Append(char.ToUpper(c));
                word = true;
            }
            else
                sb.Append(c);
        }
        else {
            if (char.IsPunctuation(c))
                sb.Append('_');
            word = false;
        }
    }
    return sb.ToString();
}
```

Select **Build|Build Solution**, which should compile the code with no errors. Now we can invoke Pex.

Right-click into the code of the method, and select **Run Pex Explorations** in the context menu.



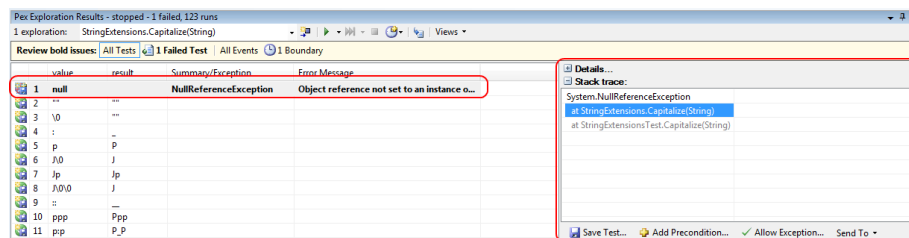
Background Information: What does it mean to Run Pex Explorations? Pex will run your code, possibly many times with different inputs. Don't run Pex on code that could launch real rockets!

After a brief moment of deep thinking, Pex shows the results of its analysis in a separate *Pex Exploration Results* window as a table.

	value	result	Summary/Exception	Error Message
1	null		NullReferenceException	Object reference not set to an instance o...
2	""	""		
3	\0	""		
4	:	-		
5	p	P		
6	\0	J		
7	Jp	Jp		
8	\0\0	J		
9	::	-		
10	ppp	Ppp		
11	p:p	P_P		

Each row in the table contains certain inputs for the analyzed method, and also the resulting output, and possibly more information about raised exceptions.

When we click on a row, we see more details on the right.

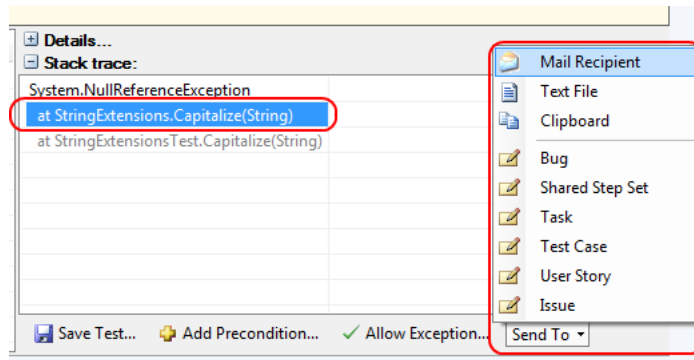


Background Information: Why did Pex choose these inputs? Are they special, or simply random? On the first sight, the inputs might appear to be just randomly chosen character sequences. But take a closer look: The inputs contain only the characters `' : ' , ' p ' , ' J ' ,` and `' \0 ' .` These characters are representative of punctuation, lower-case characters, upper-case characters, and everything else. Pex picked exactly one representative of these equivalence classes that the program distinguishes. You might see slightly different characters, depending on the mood the of the constraint solver that Pex uses under the hood.

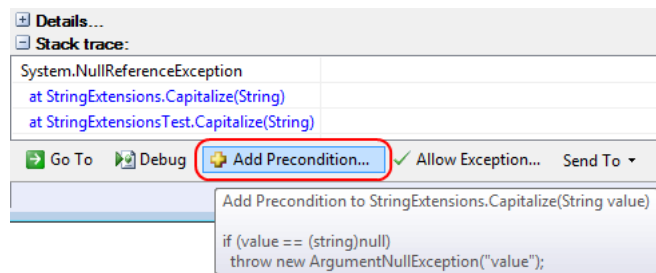
Think about the following: What would be the probability of randomly chosen inputs to contain punctuation characters, when there are $2^{16} = 65536$ different characters?

Here, we see a stack trace of the exception. We could click on the *Details* header to see details about the inputs that Pex chose for this row.

The .NET guidelines say that a method should never throw a `NullReferenceException`. We could fix this issue ourselves, or we could simply drop someone an email, or file a work item, to let someone else take care of the issue. (Assuming that we have someone else to take care of our issues.) That's what the **Send To** button is for.



But let's face reality, we have to fix our own code by ourselves. Actually, we don't – we can let Pex do it for us. Click on **Add Precondition...**, and then on the **Apply** button.



Pex adapts the beginning of our `Capitalize` method, similar to the following code snippet:

```

public static string Capitalize(string value) {
    if (value == null)
        throw new ArgumentNullException("value");
    ...
}

```

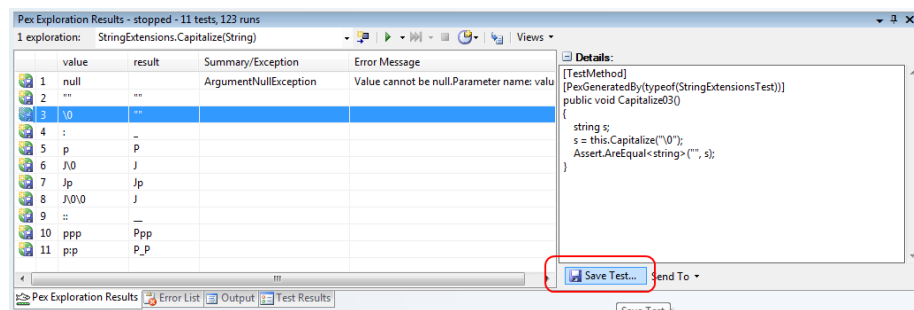
When we run Pex again, instead of seeing a failing `NullReferenceException`, we now see the acceptable behavior of an `ArgumentNullException`. The red row has turned to green.

	value	result	Summary/Exception	Error Message
1	null		ArgumentNullException	Value cannot be null.Parameter name: value
2	""	""		

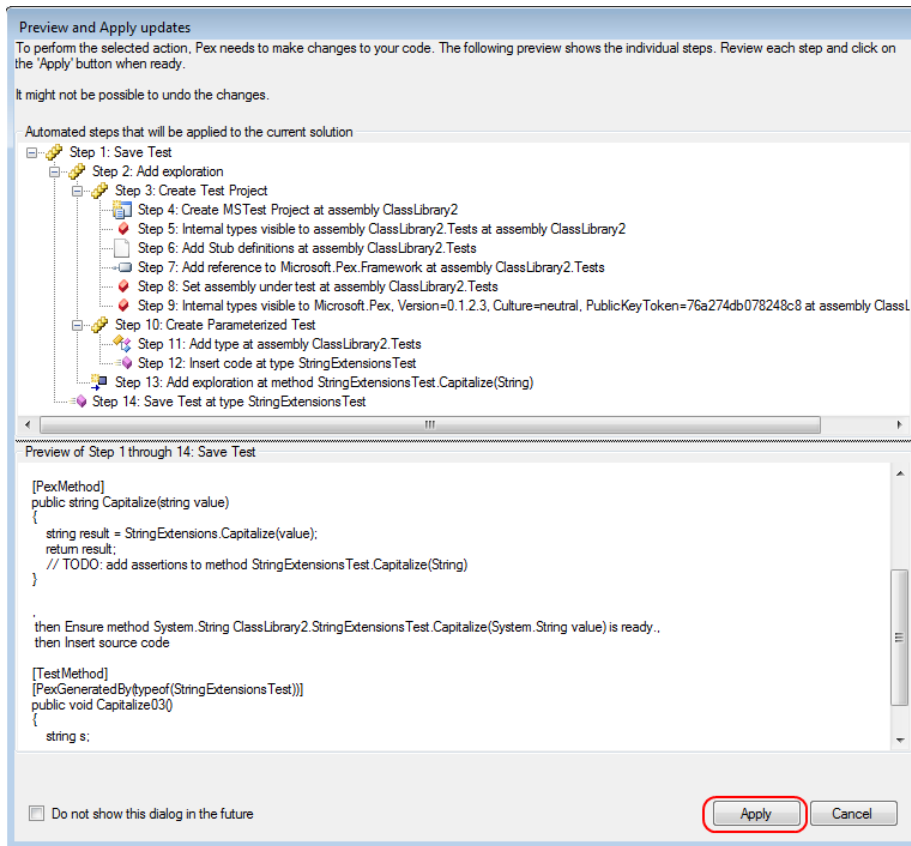
This concludes our first section. We have seen how to run Pex to create a table of interesting inputs and outputs, how to interpret the results, and how to fix an issue that Pex found.

3 Saving a Test, and Debugging an Issue

Some of the inputs that Pex lists in the table look strange. It would be nice if we could debug the code for such an input, and step through the code lines. Actually, that's easy with Pex! Select a row, and click on the **Save Test...** button.

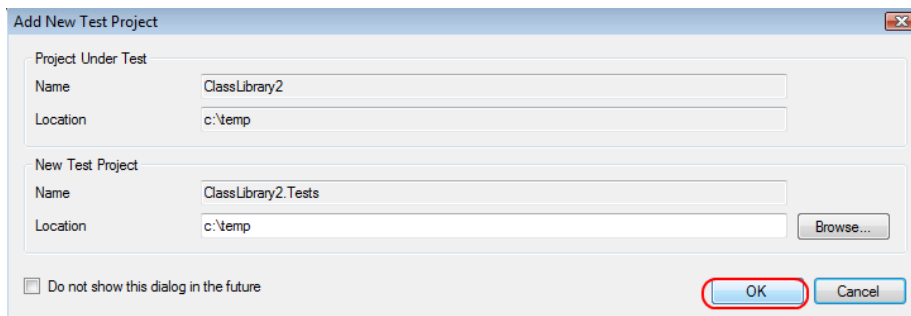


Pex shows a dialog that illustrates a sequence of steps that Pex will take: Pex will create a new test project, then perform many more little steps that we'll ignore for the moment, and finally Pex will create a unit test with the test inputs of the currently selected row.



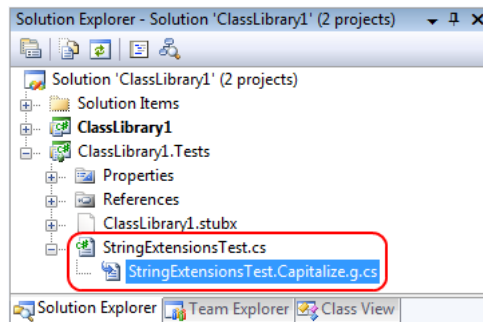
Background Information: What is a Unit Test anyway? A unit test is a method that takes no parameters, similar to the `Main` method of an application. When a unit test raises an unexpected exception, it *fails*, otherwise it *passes*.

Part of the process is to create a new test project. Just press **OK**.



The project contains several files. The C# source code of the generated saved test that corresponds to the current row in the table is stored in `StringExtensionsTest.Capitalize.g.cs`.

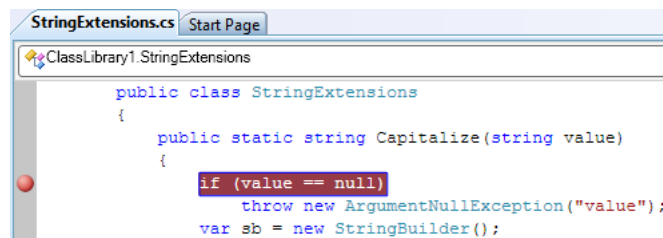
Tip: Little overlaid icons In the **Pex Exploration Results** window, did you notice that the green check-box icon of the test we just saved no longer has the little overlaid disk symbol?



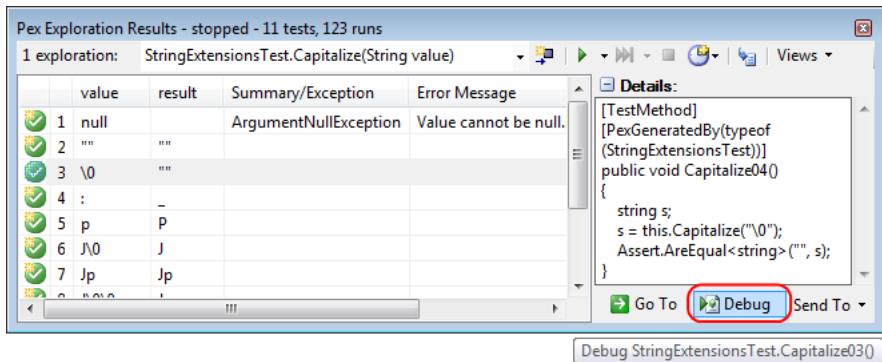
We should now see the following C# code of the generated saved test case.

```
[TestMethod]
[PexGeneratedBy(typeof(StringExtensionsTest))]
public void Capitalize03()
{
    string s;
    s = this.Capitalize("\0");
    Assert.AreEqual<string>("", s);
}
```

Go back to the source code of `Capitalize`, and set a breakpoint on the first line by pressing **F9**.



We can now press the **Debug** button in the **Pex Exploration Results** window under the details of the selected row, in order to debug the issue, starting from the generated test.

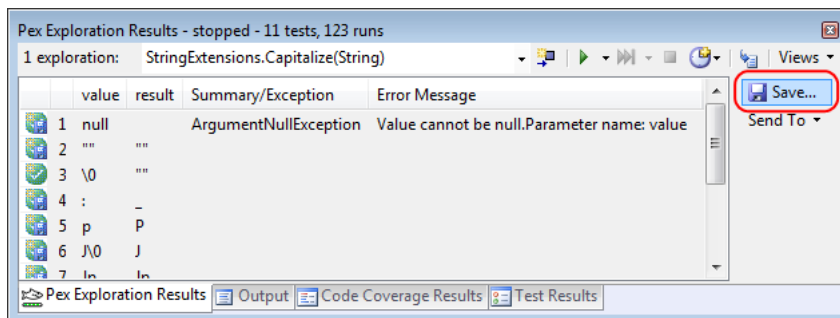


We can now step through the code line by line.

4 Saving Test Suite For Team Test

At this point, we can not only save individual tests, but we can save the entire table as a test suite. This test suite can serve as a regression test suite in the future, or as a fast-running build verification test (BVT) suite, a test suite that can be executed every time a code change is submitted to the source depository.

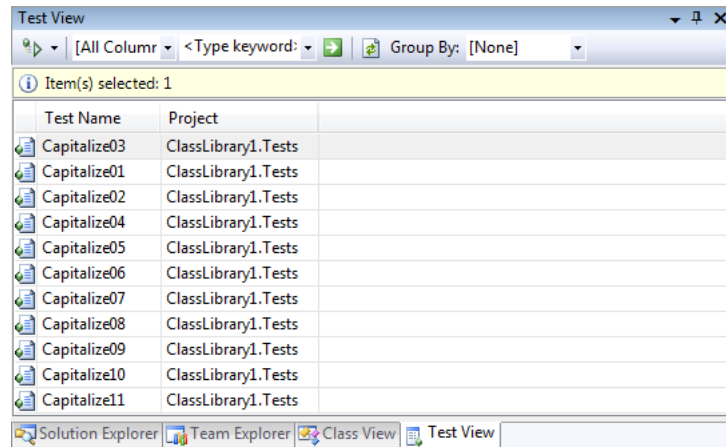
To save the entire table, select a row in the table, and press **Ctrl-A**. Then click on the **Save...** button to the right of the selected rows.



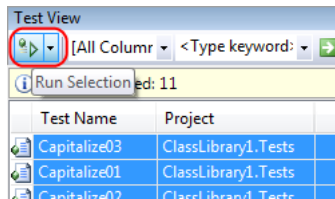
Pex will show a dialog that details all the individual steps that Pex performs to save the tests as C# code. Press **Apply**.

We now have an entire test suite that we can execute without Pex.

The generated tests are recognized by the Team Test, the unit test framework built into Visual Studio. Select **Test|Windows|Test View**.



Select a test, and select all by pressing **Ctrl-A**, and press on the *Run Selection* button in the **Test View** window to run all tests:



Running the tests at this time will simply reproduce the same results that Pex reported earlier. However, running the tests in the future might discover breaking changes in the program behavior.

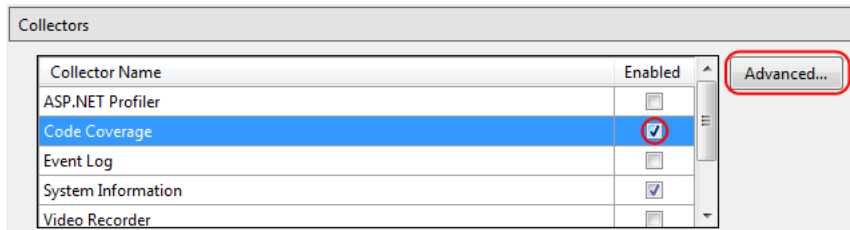
5 Code Coverage with Team Test

Let's check the code coverage achieved by the generated test suite. Visual Studio Team Test can measure code coverage of a test suite. In the following, we will go through the steps necessary to enable measuring code coverage.

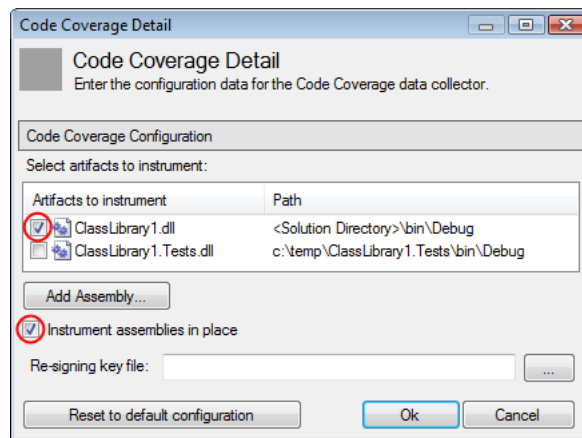
Background Information: Code Coverage and Pex. Pex internally tracks code coverage while exploring code to generate better test inputs. It is not necessary to enable the Visual Studio Team Test Code Coverage collector if we just want to run Pex. However, Pex only has a *local* coverage knowledge, which Pex refers to as *dynamic coverage*. The Visual Studio Code Coverage collector will give you *global* coverage information on all classes and methods.

Click on **Test|Edit Test Run Configurations|Test Settings (local.testsettings)**.

In the dialog that appears, select **Execution Criteria** on the left, and scroll down to the **Collectors** group on the right. Click on the check box next to **Code Coverage**.

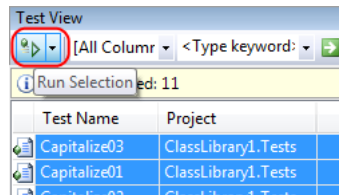


Then click on the **Advanced...** button, and click on the check box left to the project name which contains the `Capitalize` implementation, here `ClassLibrary1.dll`. Also make sure that **Instrument assemblies in place** is selected.

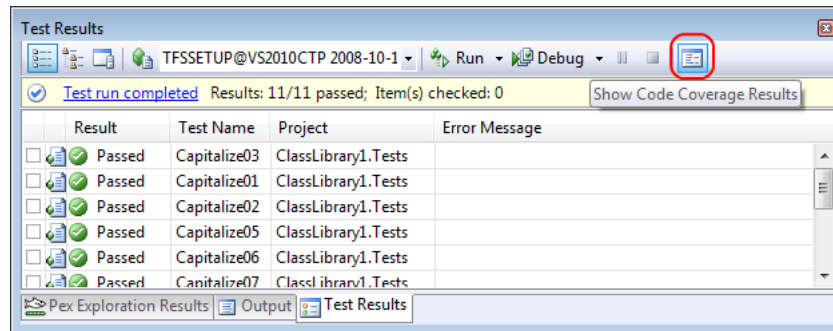


Then click on **OK**, **Apply**, and **Close**. That was easy! We have now configured Visual Studio to collect code coverage data.

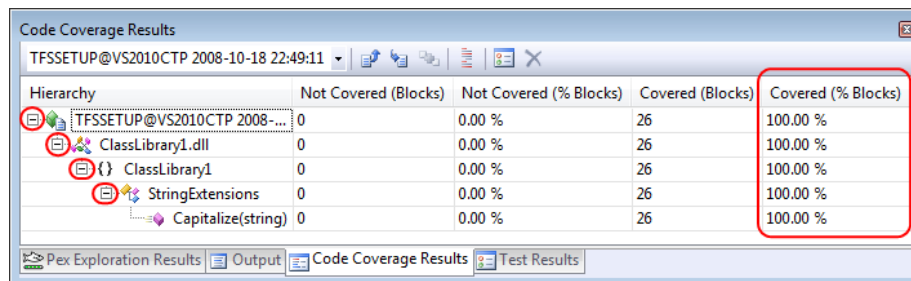
We have to run the test suite once more to actually collect code coverage information. Click again on the *Run Selection* button in the **Test View** window.



The results of running the test suite are shown in separate **Test Results** view. We can now take a look at the collected code coverage data view by clicking the **Show Code Coverage Results** button in the **Test Results** view.



We now see a window called **Code Coverage Results**. Open all the nodes of the tree view. We see that the test suite achieved 100% code coverage.



6 A Glimpse of Parameterized Unit Tests

We didn't point it out yet, but when we saved tests, Pex did not only create (traditional) unit tests, but Pex also created a *Parameterized Unit Test* (PUT) for the `Capitalize` method:

```
[TestClass]
[PexClass(typeof(StringExtensions))]
[PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentException), true)]
public partial class StringExtensionsTest
{
    [PexMethod]
    public string Capitalize(string value)
    {
        string result = StringExtensions.Capitalize(value);
        return result;
        // TODO: add assertions to method StringExtensionsTest.Capitalize(String)
    }
}
```

This method lives in the file `StringExtensionsTest.cs`, while the individual (traditional) unit tests were saved in `StringExtensionsTest.Capitalize.g.cs`.

The `...g.cs` file should never be modified by hand. If we want to customize the way Pex generates tests, or if we want to add checks that verify that the tested code works correctly, then we should always edit the PUT in the top-level file.

For example, here we could adapt the PUT to check that `Capitalize` does not change the number of letters:

```
[PexMethod]
public void CapitalizeMaintainsLettersCount(string input)
{
    string output = StringExtensions.Capitalize(input);
    PexAssert.AreEqual(
        LettersCount(input),
        LettersCount(output));
}

private static int LettersCount(string s)
{
    return Enumerable.Count(s,
        c => char.IsLetter(c) || c == '_');
}
```

Another PUT could check that `Capitalize` is idempotent, i.e. if we capitalize an already capitalized string, then we get back an equal string:

```
[PexMethod]
public void CapitalizeIsIdempotent(string input)
{
    string output = StringExtensions.Capitalize(input);
    string output2 = StringExtensions.Capitalize(output);
    PexAssert.AreEqual(output, output2);
}
```

We can state that the result of `Capitalize` only contains letters and underscores:

```
[PexMethod]
public void CapitalizeReturnsOnlyLettersAndUnderscores(string input)
{
    string output = StringExtensions.Capitalize(input);
    PexAssert.TrueForAll(output,
        c => char.IsLetter(c) || c == '_');
}
```

You get the idea. You can find more pointers on how to write PUTs in Section 8.

7 Pex Under The Hood

Pex is *not* a *black box* test generation tool. In other words, Pex does *not*

- simply throw random test inputs at the code,
- exhaustively enumerate all possible values,

- require you to write test input generators.

Pex is a *white box* test generation tool. Pex generates test cases by analyzing the program code.

In a nutshell: For every statement in the code, Pex will eventually try to craft a test input that will reach that statement. For every conditional branch in the code (e.g. `if` statements, but also assertions and all operations that can throw exceptions), Pex will do a case analysis. In other words, the number of test inputs that Pex generates depends on the number and possible combinations of conditional branches in the code.

Pex operates in a feedback loop: Pex executes the code multiple times and learns about the program behavior by monitoring the control and data flow. After each run, Pex picks a branch that was not covered previously, builds a constraint system that describes how to reach that branch, then uses a constraint solver to determine new test inputs that fulfill the constraints. The test is executed again with the new inputs, and the process repeats. On each run, Pex might discover new code and dig deeper into the implementation. In this way, Pex *explores* the behavior of the code.

8 Further Reading

In the `Documentation` folder of the Pex installation, as well as on the Pex website <http://research.microsoft.com/Pex/Documentation.aspx>, there is further documentation:

- **Reference Manual.**
- **Parameterized Unit Testing with Pex (Tutorial).** This is an extensive tutorial on the topic of parameterized unit testing, with in-depth information about the inner workings of Pex and many of its configuration options.
- **Parameterized Test Patterns For Effective Testing with Pex** An extensive list of test patterns and other useful recipes for writing parameterized unit tests.
- **Stubs – A Simple Framework For .NET Test Stubs.** This document describes how to test methods whose parameters are interfaces or abstract classes.