
Getting Started with Microsoft Code Contracts and Microsoft Pex

Tutorial for Design by Contracts and Automated Whitebox Testing for .NET Applications

Pex Version 0.93 – August 3, 2010

Abstract

Microsoft® Code Contracts 2010 is a toolset that enables Design by Contracts for .NET Framework code. As a developer using Code Contracts, you can define:

Pre-conditions—what should hold entering a method.

Post-conditions—what should hold leaving a method.

Invariants—what should hold when an object is in a stable state.

Microsoft Pex 2010 is a Microsoft Visual Studio® add-in that provides a runtime code analysis tool for .NET Framework code. The output of the tool helps you understand the behavior of the code, and it also builds automated tests with high code coverage.

Together, the tools allow you to explore and specify methods directly from the editor. Through a context menu in the Visual Studio editor, you can invoke Pex to analyze an entire class or a single method. For any method, Pex computes and displays input-output pairs. Pex systematically hunts for Code Contracts failures or exceptions.

This document provides a step-by-step tutorial, to help you get started quickly with Code Contracts and Pex in your development practices.

This guide is Technical Level 200. To take advantage of this content, you should first install Microsoft Code Contracts and Pex as described in “Getting Started with Microsoft Pex and Moles.”

Note:

Most resources discussed in this paper are provided with the Code Contracts and Pex software packages. For a complete list of documents and references discussed, see “Resources and References” at the end of this document.

For up-to-date documentation and online community, see
<http://research.microsoft.com/pex>
<http://research.microsoft.com/contracts>

Contents

Introduction.....	4
Exercise 1: Running Pex for the First Time	5
Task 1: Create a Project.....	5
Task 2: Run Pex Explorations.....	6
Exercise 2: Annotating Code with Code Contracts	9
Task 3: Add Preconditions with Code Contracts	9
Task 4: Turning Code Contracts into Runtime Checks	10
Task 5: Add Preconditions from Pex	11
Task 6: Fixing the Remaining Issues	12
Task 7: Adding Post Conditions with Code Contracts	13
Resources and References.....	14

Disclaimer: This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, IntelliSense, Visual Studio, Windows Server, Windows Vista, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

Introduction

This tutorial introduces two toolset: Microsoft Code Contracts and Microsoft Pex. Microsoft Code Contracts is a toolset that you can use to write pre-condition, post-condition, and invariants in .NET code. Microsoft Pex is an automated testing tool that helps you to understand the behavior of .NET code, identify potential issues, and automatically create a test suite that covers corner cases. Together, these tools allow you to explore, test, and specify the code in an intuitive and lightweight fashion.

In this tutorial:

With just a few mouse clicks, you can explore code-under-test and save tests that prove your code doesn't crash—or point to errors that could cause crashes.

The Pex features and simple steps for exploring code are described in Exercise 1.

Use Code Contracts to specify the requirements on inputs, and the guarantees on the return values.

The Code Contracts features and simple steps for writing pre-conditions and post-conditions are described in Exercise 2.

Prerequisites

To take advantage of this tutorial, you should be familiar with the following:

Microsoft® Visual Studio® 2010

C# programming language (as all the samples will be written in C#)

.NET Framework

Basic practices for building, debugging, and testing software

Computer Configuration

These tutorials require that the following software components are installed:

Windows® 7, Windows Vista®, or Windows Server® 2008 R2 or later operating system

Visual Studio 2010 Professional

Microsoft Pex also works with Visual Studio 2008 Professional; this tutorial assumes that your edition supports the Visual Studio Unit Testing framework.

Microsoft Pex 2010

See: Moles and Pex download on Microsoft Research site.

Microsoft Code Contracts – Runtime Rewriter or Full version

See: Code Contracts download on Microsoft Research site.

Getting Help

For questions, see “Resources and References” at the end of this document.

If you have a question on Pex, post it on the Pex and Moles forums

If you have a question on Code Contracts, post it on the Code Contracts forums.

Exercise 1: Running Pex for the First Time

In this exercise, you will start a project from scratch and create the basic code to be examined with Microsoft Pex. After creating this example code, you will use Pex to generate a series of tests for this code.

About the Example in This Tutorial. The example code for this tutorial implements a class containing a user-defined method— `TrimAfter`—with two parameters: an input string and a suffix string. In this example code:

The method cuts the input string at the first occurrence of the suffix string.

The method returns the original string if the suffix is not found.

Task 1: Create a Project

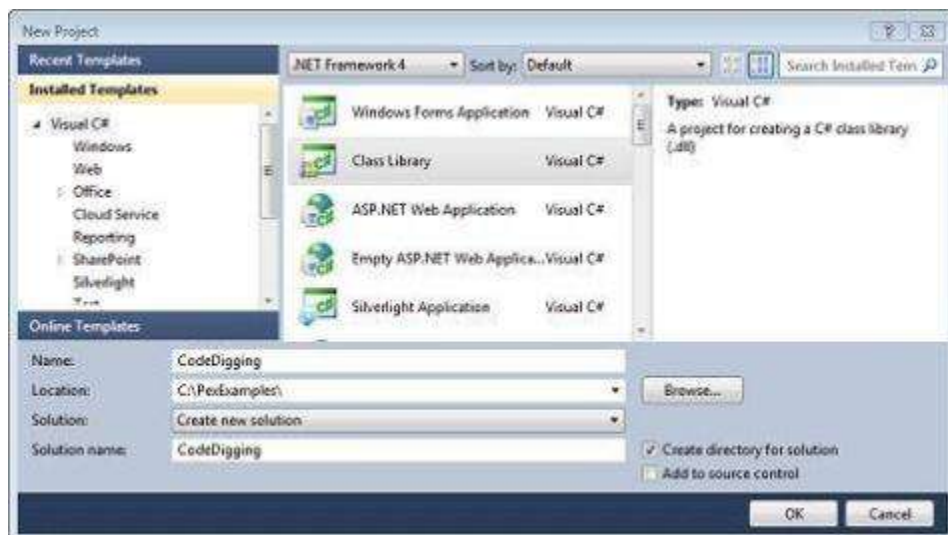
First, you must create a new project to hold your code and then create the basic code for the `StringExtensions` class, which is the user-defined class that you will be testing for the rest of this document.

If you already program in Visual Studio, you can review these steps briefly, following the details for naming classes and files.

If you are relatively new to programming in Visual Studio, these steps will guide you through the details of the Visual Studio interface.

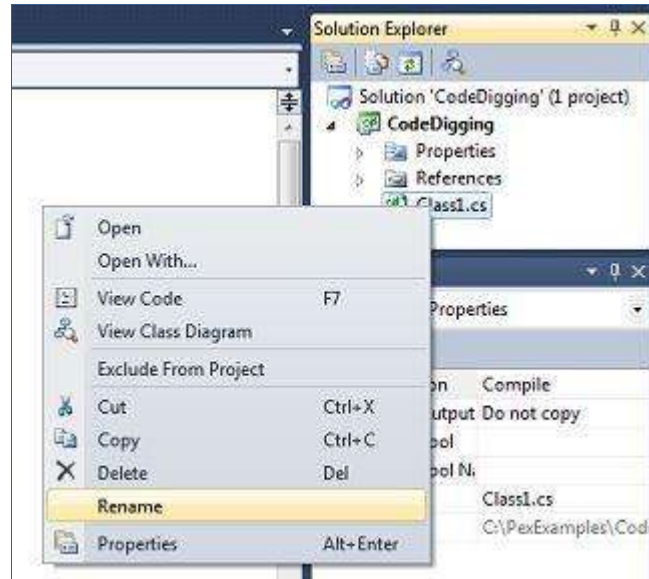
To create a project

1. In Visual Studio, click **File > New > Project**.
2. In the left pane of the **New Project** dialog box, click **Visual C#**. In the center pane, click **Class Library**.
3. Name the project **CodeDigging** and click **OK**.

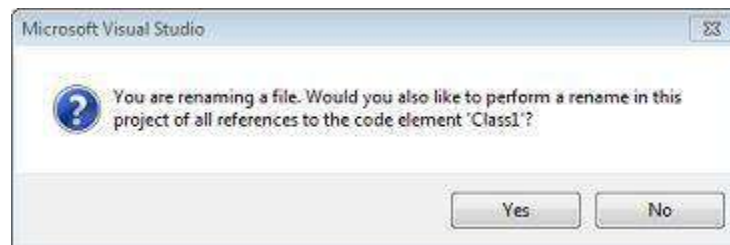


4. In **Solution Explorer**, rename the file `Class1.cs` to **StringExtensions.cs**.

Notice that when you created the CodeDigging project, Visual Studio automatically created Properties and References, and also created a generic class entry, which you are renaming in this step.



5. In the **Microsoft Visual Studio** message box, click **Yes** to confirm that you want to rename all references to this class.



6. In the class body for `StringExtensions`, add the following code to implement the user-defined `TrimAfter` method.

```
public static string TrimAfter(string value, string suffix)
{
    int index = value.IndexOf(suffix);
    return value.Substring(0, index);
}
```

7. In the **Build** menu, click **Build Solution**.

The code should compile with no errors. If you have errors, check your code for typos.

Task 2: Run Pex Explorations

When you run Pex on your code, Microsoft Pex runs your code many times with different inputs. This is referred to as "Pex explorations." The result appears as a table that shows each test attempted by Pex. Each line in the table

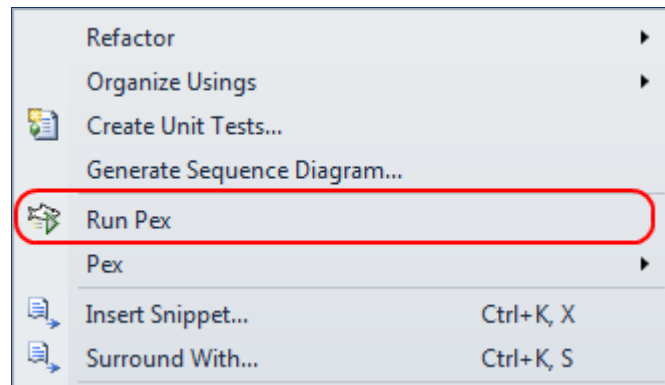
shows the input and the resulting output or exceptions generated by the method for that input.

In this task, you will run Pex explorations on your example code. For more background information about how Pex runs explorations, see “Pex Explorations: Under the Hood” later in this document.

CAUTION: Pex will exercise every path in your code. If your code is connected to external resources such as databases or controls physical machinery, make certain that these resources are disconnected before executing Pex; otherwise, serious damage to those resources might occur.

To run Pex explorations

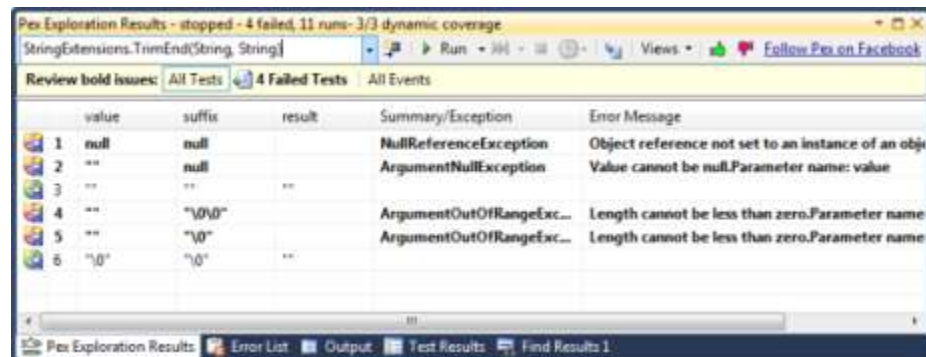
1. In the CodeDigging project created in Task 1, right-click in the body of the TrimAfter method, and click **Run Pex**.



2. If this is your first time running Pex, the **Pex: Select a Test Framework** dialog box appears. Make sure that **Visual Studio Unit Test** is selected, and click **OK**.

This dialog box will not appear again after you select the test framework.

After a brief pause, Pex shows the results of its analysis in the **Pex Exploration Results** window.



The Pex results are displayed as a table. Each row in the table contains:

An input **value** for the analyzed method.

The output **results** of the analyzed method.

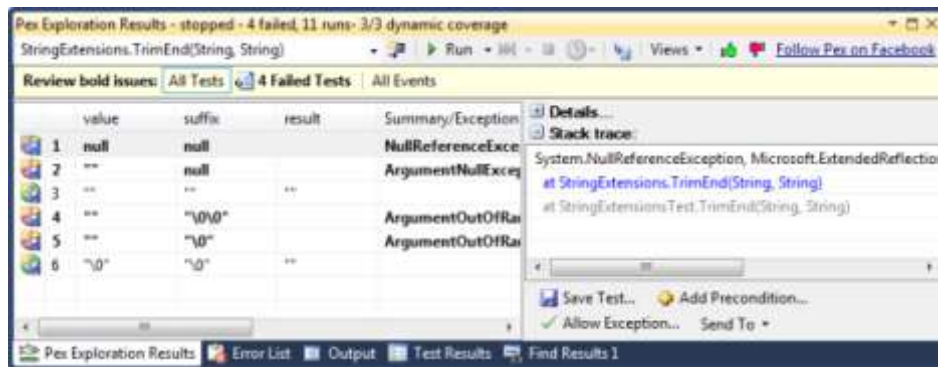
Summary information about raised **exceptions**, if any were raised during Pex explorations.

To learn more about why Pex chose the particular input values, see the “Pex Explorations: Under the Hood” at the end of this procedure.

3. In the **Pex Exploration Results** window, click one row in the table on the left to see details in the right pane.

The following illustration shows the detail view of an exception. In the right pane, you see a stack trace of the exception.

To see details about the inputs that Pex chose for the selected test, click the **Details** header.



Pex Explorations: Under the Hood

How does Pex choose input values when generating tests?

Microsoft Pex:

- Does not simply throw random test inputs at the code.

- Does not exhaustively enumerate all possible values.

- Does not require you to write test input generators.

Instead, Pex generates test cases by analyzing the program code that gets executed. This is called whitebox test generation (as opposed to blackbox test generation). For every statement in the code, Pex will eventually try to create a test input that will reach that statement. Pex will do a case analysis for every conditional branch in the code—for example, **if** statements, assertions, and all operations that can throw exceptions.

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: it executes the code multiple times and learns about the program behavior by monitoring the control and data flow.

After each run, Pex does the following:

- Chooses a branch that was not covered previously.

- Builds a constraint system that describes how to reach that branch.

- Uses a constraint solver to determine new test inputs that fulfill the constraints, if any exist.

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.

For more information, specifically created for the experienced developer, see “Parameterized Unit Testing with Microsoft Pex.”

Exercise 2: Annotating Code with Code Contracts

In this exercise you will use Microsoft Code Contracts to add preconditions and postconditions. You will use Microsoft Pex to ensure that the contracts hold for any input.

Task 3: Add Preconditions with Code Contracts

The .NET guidelines say that a method should never throw a **NullReferenceException** like the one you observed at the end of Task 2. Therefore, Pex has identified an error in your example code.

About Preconditions. The error occurs because the developer does not ensure that the input string is not null, which subsequently leads to the null dereference. Following a defensive programming principle, the developer should first validate the inputs, and then act upon them. Typically in the .NET framework, developers tend to write the following validation code where a **NullReferenceException** is thrown when the input string is null:

```
public static string TrimEnd(string value, string suffix)
{
    if (value == (string)null)
        throw new ArgumentNullException("value");
}
```

The problem with this approach is that the caller of the TrimAfter method cannot infer this requirement from the method signature, which only specifies constraints on the input and output types, as the following shows:

```
string TrimAfter(string value, string suffix)
```

Microsoft Code Contracts provide a structured mechanism for expressing preconditions in the code. Code Contracts is exposed as a static class Contract in the **System.Diagnostics.Contracts** namespace. This class contains **Requires** methods to express pre-conditions, as follows:

```
using System.Diagnostics.Contracts;
public static string TrimAfter(string value, string suffix)
{
    Contract.Requires(value != null);
}
```

To add Code Contracts preconditions to fix an error

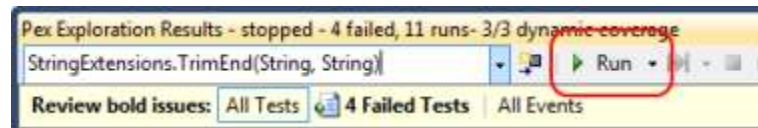
1. At the beginning of the TrimAfter method, write the word **crn** and then press TAB twice.
2. The **crn** snippet expanded to the Contract.Requires method call. In the snippet box, type value and then press Enter.

3. Place your cursor in the name, and a small box should show up under the name. Place your cursor over the box, or click **CTRL+.** (that is, the period character). This displays an IntelliSense® dialog.
4. To update your code automatically, click the first item in the IntelliSense dialog to add the missing namespace that contains the reference.

Tip: Remember this trick!

CTRL+. is the shortcut to the IntelliSense menu, and it can save you lots of time when coding in C#.

5. To run Pex again, click the Run button in the Pex Exploration Results view. Observe that the NullReferenceException is still present.

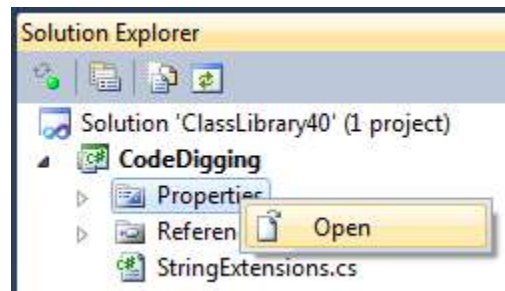


Task 4: Turning Code Contracts into Runtime Checks

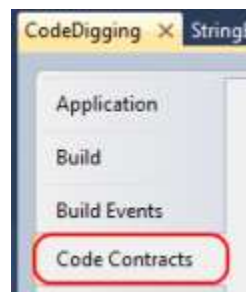
By default, the Contract methods are removed by the C# compiler, because those methods are conditional. In this task, you will enable the Runtime Rewriting feature of Code Contracts. In that mode, the contracts are transformed into runtime checks that will be executed as part of the program flow. This also means that Pex will automatically be aware of the contracts.

To enable runtime checking of Code Contracts

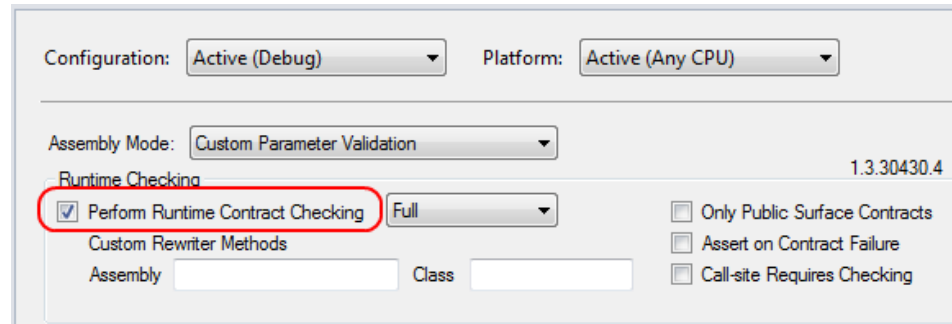
1. In the **Solution Explorer**, go to the project node, right-click the **Properties** node, and select **Open**.



2. Click the **Code Contracts** pane that sits at the bottom of the panes list on the left of the view.

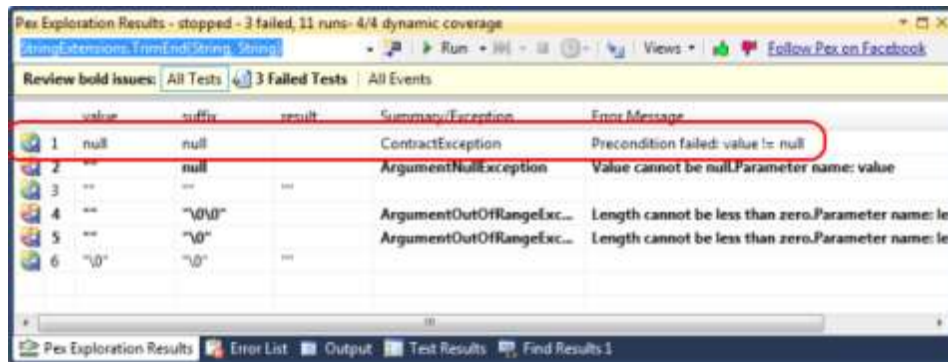


3. In the Code Contracts pane, check the **Perform Runtime Contract Checking** checkbox.



4. When you run Pex again, instead of seeing a failing **ArgumentNullException**, you see the acceptable behavior of a **ContractException**.

The formerly red row is now green, as shown in the following illustration.



Task 5: Add Preconditions from Pex

You have fixed the first failing test case, but Microsoft Pex still reports three sets of inputs that lead to **ArgumentNullException** or **ArgumentOutOfRangeException**. In this task, you will fix the **ArgumentNullException** issue by using a precondition that Microsoft Pex adds for you automatically.

There are usually many ways to fix an error. Ultimately, a developer or tester must decide what the intended meaning of the program is, and then fix the error accordingly. However, one kind of error fixing can be automated:

If a higher-level component passes malformed data to a lower-level component, which the lower-level component rejects, then the higher-level component should be prevented from doing so in the first place.

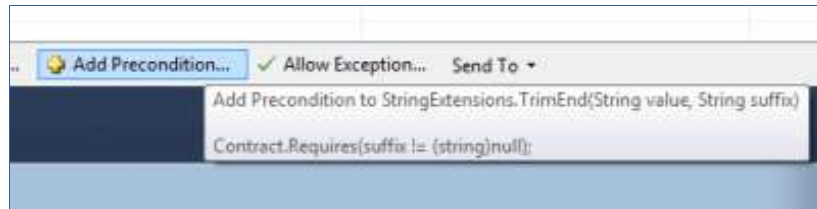
One way to achieve this is to promote the failure condition of the lower-level component to the abstraction level of the higher-level component. This is what the **Add Precondition** feature of Pex does: It expresses a low-level failure condition at the abstraction level of the code under test.

When added to the code-under-test as argument validation code—also called a precondition—then this failure condition effectively prevents the code under

test from causing a failure somewhere in its execution. In other words, the Add Precondition feature doesn't completely fix an error; rather, it promotes the failure condition to a higher, more appropriate abstraction level.

To add preconditions to fix an error

1. In the results from Task 2, click the row of the table that contains the **ArgumentNullException**.
2. Click **Add Precondition**, as shown in the following illustration.



3. In the **Preview and Apply Updates** dialog box, you can preview the contents of the adapted code that Pex creates. Click **Apply** to accept the change.

Pex adapts the beginning of the TrimAfter method in your example code, similar to the following code snippet:

```
public static string TrimAfter(string value, string suffix)
{
    // <pex>
    Contract.Requires(suffix != (string)null);
    // </pex>
    Contract.Requires(value != null);
}
```

Pex uses the xml comment to position properly future pre conditions.

4. When you run Pex again, instead of seeing a failing **ArgumentNullException**, you see the acceptable behavior of a **ContractException**.

The formerly red row is now green, as shown in the following illustration.

	value	suffix	result	Summary/Exception	Error Message
1	null	null		ContractException	Precondition failed: suffix != (string)null
2	null	""		ContractException	Precondition failed: value != null
3	""	""			
4	""	"\0"		ArgumentOutOfRangeException	Length cannot be less than zero.Parameter name: length
5	""	"\0\0"		ArgumentOutOfRangeException	Length cannot be less than zero.Parameter name: length
6	"\0"	"\0"			

Task 6: Fixing the Remaining Issues

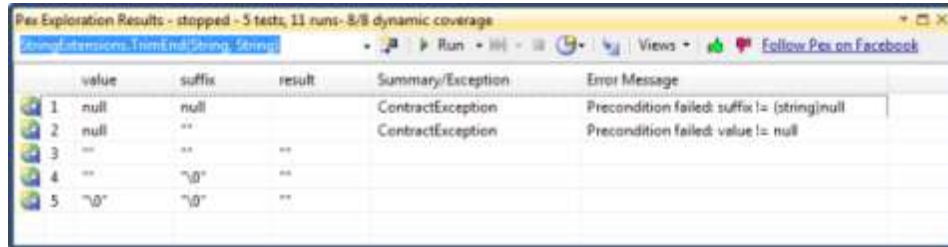
Microsoft Pex still reports two inputs that lead to an **ArgumentOutOfRangeException**. In this task, you will fix the code and run Pex until no more failing tests are reported.

To use Pex and Contract.Requires to fix the remaining issues

1. When a suffix is not found, the `IndexOf` returns -1. You handle this case in the method by returning the original string in that case, as follows:

```
int index = value.IndexOf(suffix);  
if (index < 0)  
    return value;
```

2. Run Pex again. It should report all passing tests as shown in the following example.



	value	suffix	result	Summary/Exception	Error Message
1	null	null		ContractException	Precondition failed: suffix != (string)null
2	null	""		ContractException	Precondition failed: value != null
3	""	""	""		
4	""	"0"	""		
5	"0"	"0"	""		

Task 7: Adding Post Conditions with Code Contracts

So far, you have specified how the input string should be defined, and you have ensured that the method does not crash. In this task, you will use post-conditions to specify properties about the result value. In this example, you will want to ensure that the return value does end with the suffix.

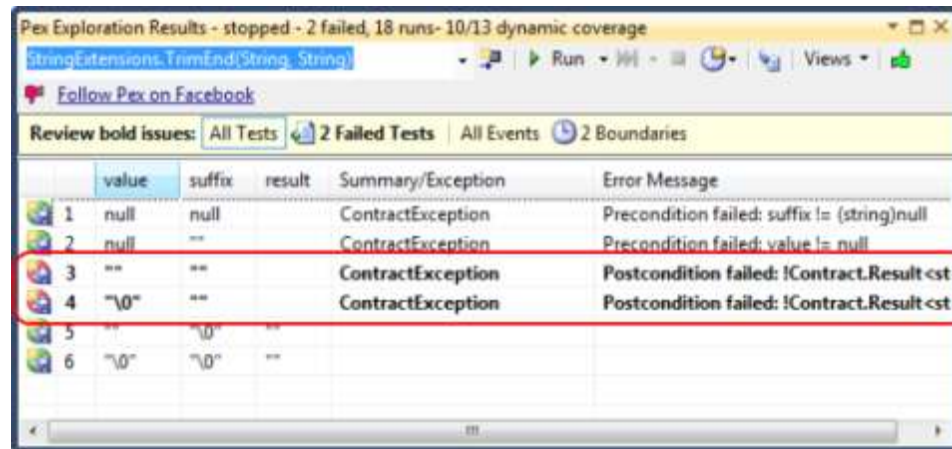
To use Pex and Contract.Requires to fix the remaining issues

1. Add a postcondition to the method using the `Ensures` method. To refer to the result, use the `Contract.Result<string>()` method, as follows:

```
public static string TrimAfter(string value, string suffix)  
{  
    ...  
    Contract.Ensures(  
        !Contract.Result<string>().EndsWith(suffix));  
}
```

Notice that post-condition have to be written at the beginning of the method. The Code Contracts rewriter will take care of moving the runtime checks at the exit of the method.

2. Run Pex and observe that two new failures were found.



The failure is due to the fact that all strings end with the empty string.

3. Strengthen the precondition on the suffix element by specifying that it should not be null or empty, as follows:

```
public static string TrimAfter(string value, string suffix)
{
    Contract.Requires(!String.IsNullOrEmpty(suffix));
    Contract.Requires(value != null);
    Contract.Ensures(
        !Contract.Result<string>().EndsWith(suffix));
}
```

4. Run Pex and observe that all failures have been fixed.

Summary of Exercise 2

In this exercise, you have seen how to write preconditions and postconditions using Code Contracts and turn them into runtime checks.

Resources and References

Pex Resources, Publications, and Channel 9 Videos

Pex and Moles at Microsoft Research

<http://research.microsoft.com/pex/>

Pex Documentation Site

Pex and Moles Tutorials

Technical Level:

Getting Started with Microsoft Pex and Moles	200
Getting Started with Microsoft Code Contracts and Pex	200
Unit Testing with Microsoft Moles	200
Exploring Code with Microsoft Pex	200
Unit Testing ASP.NET applications with Microsoft Pex and Moles	300
Unit Testing SharePoint Foundation with Microsoft Pex and Moles	300
Unit Testing SharePoint Foundation with Microsoft Pex and Moles (II)	300
Parameterized Unit Testing with Microsoft Pex	400

Pex and Moles Technical References

Microsoft Moles Reference Manual	400
Microsoft Pex Reference Manual	400
Microsoft Pex Online Documentation	400
Parameterized Test Patterns for Microsoft Pex	400
Advanced Concepts: Parameterized Unit Testing with Microsoft Pex	500

Community

Pex Forum on MSDN DevLabs

Pex Community Resources

Nikolai Tillmann's Blog on MSDN

Peli de Halleux's Blog

Terminology

code coverage

Code coverage data is used to determine how effectively your tests exercise the code in your application. This data helps you to identify sections of code not covered, sections partially covered, and sections where you have complete coverage. With this information, you can add to or change your test suites to maximize their effectiveness. Visual Studio Team System helps measure code coverage. Microsoft Pex internally measures coverage knowledge of specific methods under test (called "dynamic coverage"). Pex generates test cases that often achieve high code coverage.

explorations

Pex runs the code-under-test, using different test inputs and exploring code execution paths encountered during successive runs. Pex aims to execute every branch in the code-under-test, and will eventually execute all paths in the code. This phase of Pex execution is referred to as "Pex explorations."

integration test

An integration test exercises multiple test units at one time, working together. In an extreme case, an integration test tests the entire system as a whole.

unit test

A unit test takes the smallest piece of testable software in the application, isolates it from the remainder of the code, and determines whether it behaves exactly as you expect. Each unit is tested separately. Units are then integrated into modules to test the interfaces between modules. The most common approach to unit testing requires drivers and stubs to be written, which is simplified when using the Moles framework.

whitebox testing

Whitebox testing assumes that the tester can look at the code for the application block and create test cases that look for any potential failure scenarios. During whitebox testing, you analyze the code of the application block and prepare test cases for testing the functionality to ensure that the class is behaving in accordance with the specifications and testing for robustness.

