
Parameterized Unit Testing with Microsoft Pex

Advanced Tutorial for Automated Whitebox Testing for .NET Applications

Version 0.93 - August 3, 2010

Abstract

This tutorial explores the principles of parameterized unit testing with Microsoft Pex 2010, which is a Microsoft® Visual Studio® add-in that provides a runtime code analysis tool for .NET Framework code.

This tutorial is Technical Level 400. This tutorial assumes that you are familiar with developing .NET applications. To take best advantage of this tutorial, first:

- Install Microsoft Pex 2010
- Practice with basic Pex capabilities, as described in “Exploring Code with Microsoft Pex”

Note:

- Most resources discussed in this paper are provided with the Pex software package. For a complete list of documents and references discussed, see “Resources and References” at the end of this document.
- For up-to-date documentation, Moles and Pex news, and online community, see <http://research.microsoft.com/pex>

Contents

Introduction to the Parameterized Unit Testing Tutorial	3
Unit Testing and Microsoft Pex	4
What Are Unit Tests?	4
Measuring Test Quality: Code Coverage and Assertions	7
Exercise 1: Creating Unit Tests in .NET	8
Task 1: Create a New Test Project.....	8
Task 2: Create a Passing Unit Test.....	8
Task 3: Create a Failing Unit Test	10
Task 4: Create a Negative Unit Test	11
Task 5: Enable Code Coverage	12
Exercise 2: Creating Unit Tests and Refactoring for Test-Driven Development	13
Task 1: Credit Card Number Validation Specification	14
Task 2: Add a Failing Test to an Empty Project	14
Task 3: Run the Unit Test and Refactor the Code	17
Exploring Parameterized Unit Testing	18
Coverage through Test Input Generation	19
Theory of Parameterized Unit Tests	20
Test-Driven Development by Parameterized Unit Testing	21
Exercise 3: Creating Parameterized Unit Tests in Visual Studio	21
Task 1: Add Pex to a Project and Create a Parameterized Unit Test	21
Task 2: Run the Parameterized Unit Test.....	22
Understanding How Pex Chooses Inputs	24
Exercise 4: Instrumenting Code for Parameterized Unit Testing.....	26
Task 1: Run the Pex Code-Generation Wizard	26
Task 2: Set Up the Code Instrumentation	26
Task 3: Explore the Instrumentation Configuration.....	28
Exercise 5: Using Pex from the Command Line	29
Task 1: Run Pex from the Command Line	30
Task 2: Use Pex HTML Report	32
Task 3: View the Pex HTML Report in Visual Studio.....	34
Experimenting with Expressions, Encoding, and RLE Algorithms	36
Isolating Tests from the Environment with Microsoft Moles	37
Isolating Tests with Mocks and Stubs	38
Manually Creating Parameterized Mock Objects	39
Parameterized Mock Objects with Assumptions	40
Using Dependency Injection to Build Mocks.....	41
Resources and References	44
Appendix: Pex Cheat Sheet.....	46

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

Introduction to the Parameterized Unit Testing Tutorial

Microsoft Pex 2010 is a testing tool that performs systematic code analysis, hunts for boundary conditions and flags exceptions and assertion failures.

Pex enables parameterized unit testing, an extension of unit testing that reduces test maintenance costs. A parameterized unit test is simply a method that takes parameters, calls the code under test, and states assertions. Pex analyzes the code in the parameterized unit test together with the code-under-test, attempting to determine interesting test inputs that might exhibit program crashes and assertion violations. Pex learns the program behavior by monitoring execution traces, using a constraint solver to produce new test cases with different behavior.

This tutorial presents a series of advanced concepts:

- An introduction to unit testing in .NET, and implementing the Luhn algorithm using test-driven development
- How to use test-driven development using parameterized unit tests.
- How Pex Chooses Inputs
- How to apply and experiment with parameterized unit testing
- How to use the Microsoft Moles framework to isolate your test code.

Prerequisites

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

- Microsoft® Visual Studio® 2010
- C# programming language
- .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 and Microsoft Moles 2010

See: [Moles and Pex download on Microsoft Research site.](#)

Getting Help

- For questions, see “Resources and References” at the end of this document.
- If you have a question, post it on the Pex and Moles forums.

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

Unit Testing and Microsoft Pex

Unit testing is increasingly popular among developers. A recent survey at Microsoft indicated that 79% of developers use unit tests. Developers write unit tests to document customer requirements, to reflect design decisions, to protect against changes. You can also use unit testing as part of the testing process, to produce a test suite with high code coverage and give confidence in the correctness of the tested code.

The growing adoption of unit testing is due to the popularity of methods like Extreme Programming, test-driven development, and test execution frameworks like JUnit, NUnit or MbUnit.

Extreme Programming does not specify how and which unit tests to write and test execution frameworks only automate test execution; they do not automate the task of creating unit tests. This leaves unit test creation as a manual and laborious undertaking for the developer. In many projects at Microsoft, there are more lines of unit test code than code under test. Are there ways to automate the generation of good unit tests? Parameterized unit testing is a possible answer and is the topic of the tutorial.

These tutorials describe how to design, implement and test software using the methodology of parameterized unit testing, supported by Microsoft Pex 2010.

Microsoft Pex is an automated test input generator that uses dynamic symbolic execution to check if the software under test agrees with its specification. This automation makes software development more productive and software quality increases. Pex produces a small test suite with high code coverage from parameterized unit tests.

The first part of this tutorial explains unit testing and compares traditional and parameterized unit testing. Following that, you will see how Microsoft Pex automates the process of creating inputs for parameterized unit tests, and runs tests using those inputs.

The latter part of the tutorial introduces you to the Microsoft Moles framework, which comes as part of Pex and as a separate download. The Moles framework allows you to isolate your code under test from external dependencies.

Note: The examples in this tutorial assume deterministic, single-threaded applications.

What Are Unit Tests?

A unit test is a self-contained program that checks an aspect of the implementation under test. A unit is the smallest testable part of the program. You can partition unit tests into three basic parts:

- **Arrange:** Data which might be considered as the test input that is passed to the methods as argument values.
- **Act:** A method sequence that works on the data provided in the Arrange step. You build a scenario that usually involves several method calls to the code-under-test.

- **Assert:** Assertions, which confirm the correctness of certain properties of the code under test. The test fails if any assertion fails or if the application throws an unhandled exception.

The following snippet is an example of a unit test that checks the interplay among some operations of the .NET **ArrayList** class. This example is a method written in C#, omitting the class context for brevity:

```
// A unit test for the ArrayList class
public void AddTest()
{
    // Arrange
    int capacity=1;
    object element=null;

    // Act
    ArrayList list = new ArrayList(capacity);
    list.Add(element);

    //assert
    Assert.IsTrue(list[0]==element);
}
```

How the test works:

- **Arrange :** Method state for AddTest is set by picking the values 1 and null for capacity and value.
- **Act:** The test method performs a sequence of method calls, starting by creating an instance of the ArrayList class with the selected capacity. An array list is a container whose size might change dynamically. Internally, it uses a fixed-length array as backing storage. The capacity of an array list is the allocated length of its current backing storage. Next, the test adds the element to the array list.
- **Assert:** Finally, the test uses an assertion to check that the array list at position 0 contains element. This is the test oracle.

There is often more than one way to partition a unit test into data, method sequence, and assertions. The following snippet shows a similar test where the input data is more complex, including:

- An object for the element.
- The initial instance of the array list itself.

```
// Another typical unit test
public void AddTest2()
{
    // Arrange
    Object element = new object();
    ArrayList list = new ArrayList(1);

    // Act
    list.Add(element);

    //assert
    Assert.IsTrue(list[0] == element);
}
```

A custom attribute such as **[TestMethod]** marks a parameterless unit test in a testing framework like Visual Studio Unit Test.

Each unit test usually explores a single aspect of the behavior of the class-under-test.

An attribute like `[TestClass]` marks the class containing the unit test.

Benefits of Unit Testing

Software developers and testers write unit tests for different purposes.

- **Design and specification:** Developers translate their understanding of the specification into unit tests and/or code. Developers who use test-driven development start by writing unit tests before they start to write code and therefore use the unit tests to drive the design. Of course, you write unit tests in all phases of software development. Unit tests can also capture and test exemplary customer scenarios.
- **Code coverage and regression testing:** Developers or testers might write unit tests to increase their confidence in the correctness of code that they have already written. A test suite that achieves high code coverage and checks many assertions is a good indicator of code quality. In this way, unit tests represent a safety net that developers use when refactoring the code.

Unit tests are usually small tests that run fast and give a quick feedback on effects of code changes. Several tools exist to automate the execution of a suite of unit tests on each code change that is committed to the source code repository.

- **Documentation.** Unit tests can serve as documentation of correct program behavior.

As mentioned above, developers usually write unit tests before or after writing the product code. When a unit test fails and exposes a bug, the feedback loop to get the bug fixed is very short.

A Critique of Unit Testing

Unit testing faces several challenges:

- **Quality of unit tests:** The quality of the unit tests is mostly dependent on the time the developer is willing to invest in them.
- **Quantity of unit tests:** Writing more unit tests does not necessarily increase the code coverage. Therefore, the size of the test suite is not an indicator of the code quality.
- **New code with old tests:** Developers maintain unit tests when they are implementing code; they rarely update these tests other than for syntactic corrections when refactoring APIs.

If the developer adds special cases in the code but does not update the unit tests, this might introduce a number of new untested behaviors.

- **Hidden integration test:** Ideally, a unit test should test the code in isolation. This means that the developer must hide all environment dependencies such as database and file I/O behind an abstraction layer. During testing, you customize the abstraction layer to provide a fake implementation, also referred as mocks. In practice, it is very easy to have code with hard coded dependencies on the environment. In such cases, the developer can use the Moles framework to replace those dependencies with a fake implementation.

Measuring Test Quality: Code Coverage and Assertions

What is a good test suite? When do you have enough unit tests to ensure a minimum level of quality? Developers face these hard questions. Experience within Microsoft and from the industry indicates that a test suite with high code coverage and high assertion density is a good indicator of code quality.

Code coverage alone is generally not a good enough measure of quality so use it with care. The lack of code coverage on the other hand clearly indicates a risk, as many behaviors are untested.

When at least one unit test executes a statement, the statement is considered covered. You compute code coverage by executing the entire unit test suite and computing the ratio of covered statements. Different notions of code coverage exist. Pex employs control flow coverage that follows these principles:

- **Basic Block Coverage.** A basic block representation of the control flow graph of the program is the foundation of basic block coverage. A "basic block" is a sequence of instructions, in this case, .NET MSIL instructions, that has one entry point, one exit point, and no branches within the block. Basic block coverage is common in the industry.
- **Branch Coverage.** You compute branch coverage by analyzing the coverage of explicit arcs. An arc is a control transfer from one basic block to another in the program control flow graph.
- **Implicit Branch Coverage.** This is an extension of the arc coverage where all explicit and implicit arcs are considered. Implicit arcs do not show up explicitly in the control flow of the code, but are still there. They occur for exceptional behavior of instructions. For example when accessing a field the arc that throws a null dereference exception is an implicit arc.

About Unit Testing in .NET

Several unit testing frameworks exist in .NET to help you effectively author and run unit tests. Although each framework has its own particularities, they all provide a core set of services:

- A custom attribute based system for tagging methods as unit tests
- Automatic detection and execution of unit tests
- A runner with reporting capabilities. The runner might be a simple console application or an integrated graphical user interface.

A later exercise shows to use Microsoft Pex to optimize and automate the generation of parameterized unit tests, but first you should know more about unit tests.

Exercise 1: Creating Unit Tests in .NET

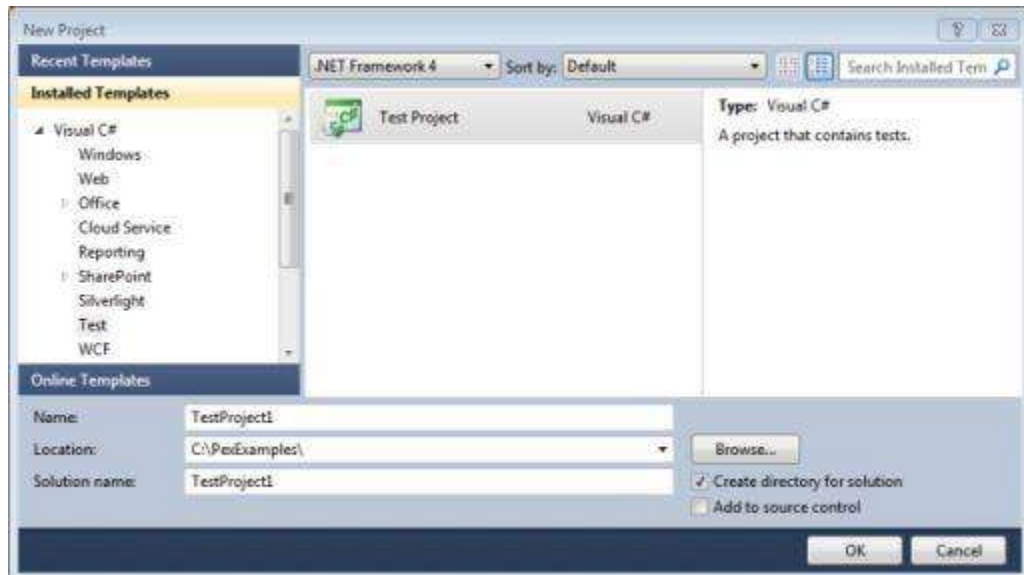
This section is for developers who are new to Visual Studio Unit Test projects in Visual Studio. This exercise shows how to create a new test project, author a unit test, and run the unit tests.

These tasks create some basic code that you'll use in later exercises to explore parameterized unit testing with Microsoft Pex.

Task 1: Create a New Test Project

To create the new test project

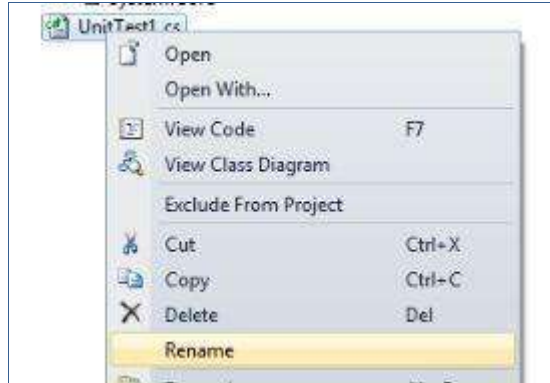
1. Run Visual Studio 2010.
2. In the Visual Studio menu bar, click **File > New > Project**.
3. In the left pane of the **New Project** dialog box, click **Visual C# > Test**. In the middle pane click **Test Project**.
4. To accept the defaults for the project, click **OK**.



Task 2: Create a Passing Unit Test

To create a passing unit test

1. In the Visual Studio Solution Explorer pane, right-click the UnitTest1.cs file and click **Rename**.



2. Rename UnitTest1 **HelloWorldTest.cs**.
3. Open **HelloWorldTest.cs** and delete the generated class so that you have an empty class definition.
4. Copy the following class definition to HelloWorldTest.cs.

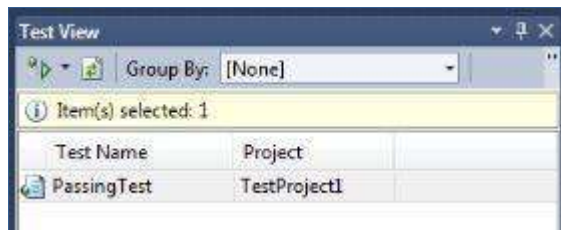
This is a test class, tagged with a **[TestClass]** attribute. You can also call such a test class a test fixture:

```
[TestClass] //this class contains unit tests
public class HelloWorldTest
{
}
```

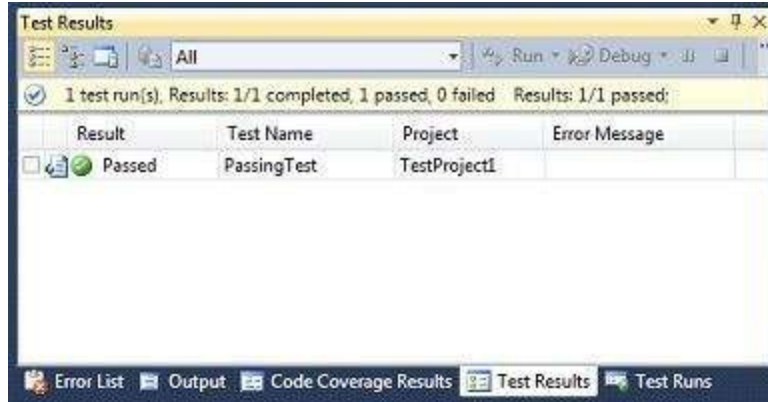
5. Add a new public method to the HelloWorldTest class that will write "HelloWorld" to the console. Tag the method with the **[TestMethod]** attribute, as follows:

```
[TestMethod]//this is a test
public void PassingTest()
{
    Console.WriteLine("hello world");
}
```

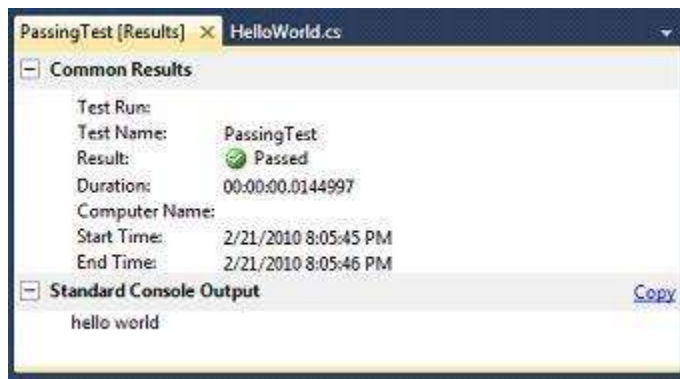
6. In the Visual Studio menu bar, click **Test > Windows > Test View** to display the **Test View** pane.
7. Click the **Refresh** icon in the pane toolbar, then click **PassingTest**, and click the **Run Selection** icon in the toolbar.



The test result pane displays the status of the current run. Each row in the report represents a test. In this case, **PassingTest** succeeded.



8. Double-click the row to display the details of that particular test in the **Results** pane.



The **Results** pane gives various statistics about the test run and shows the console output.

Task 3: Create a Failing Unit Test

In this task, you'll see how to add a test that fails by adding a new public instance method to the **HelloWorldTest** class that throws an exception and so fails as a test.

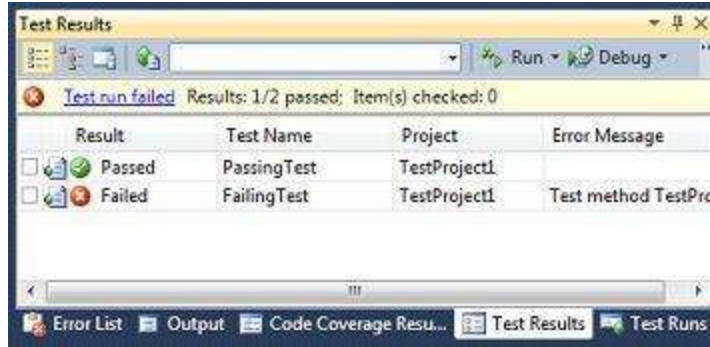
To create a failing test project

1. In the body of the HelloWorldTest class, add the following method.

```
[TestMethod]
public void FailingTest()
{
    throw new InvalidOperationException("boom");
}
```

2. Open the **Test View** pan, click the **Refresh** icon, and select both tests.
3. Execute both tests by clicking on the **Run Selected** icon.

The test result pane now contains two test results: one for **PassingTest** and one for **FailingTest**.



4. Click in the body of the **FailingTest** method and press **F9** to set a debugger breakpoint.
5. In the Test Results toolbar, click the **Debug all Tests in Test Results** icon to start debugging the failing tests.

The debugger automatically stops at the breakpoint.

If you are not familiar with the Visual Studio debugger, this is a good time to get some experience. A yellow line marks the statement that executes next.

6. Press **F11** to advance the debugger to the next line in your code.
Boom—you see the **InvalidOperationException** information box.



7. In Visual Studio, click **Debug > Stop Debugging** to exit the debugger.
8. Remove the breakpoint by clicking the red circle.

Task 4: Create a Negative Unit Test

Visual Studio Unit Test supports a special attribute **ExpectedException** that allows you to specify that the test must throw an exception of a particular type. Use it to write unit tests that check that parameter validation code works properly. This is a negative unit test.

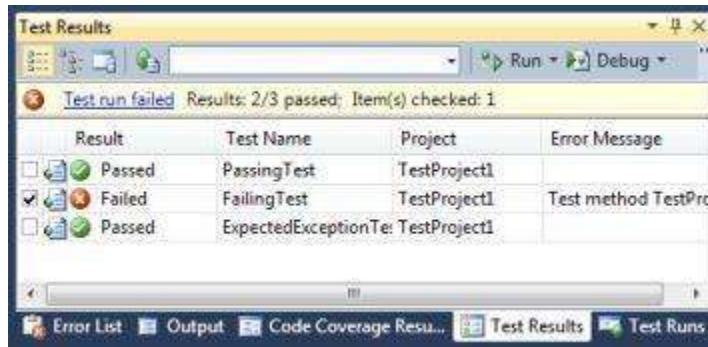
To create a Negative Unit Test

1. Add a new unit test to HelloWorldTest, and notice the **ExpectedException** attribute.

```
[TestMethod]
[ExpectedException (typeof( InvalidOperationException))]
public void ExpectedExceptionTest()
{
    throw new InvalidOperationException("boom");
}
```

2. Run all three tests from the **Test View** pane.

Notice that the **ExpectedExceptionTest** test was marked as a passing test, because it threw the expected exception.

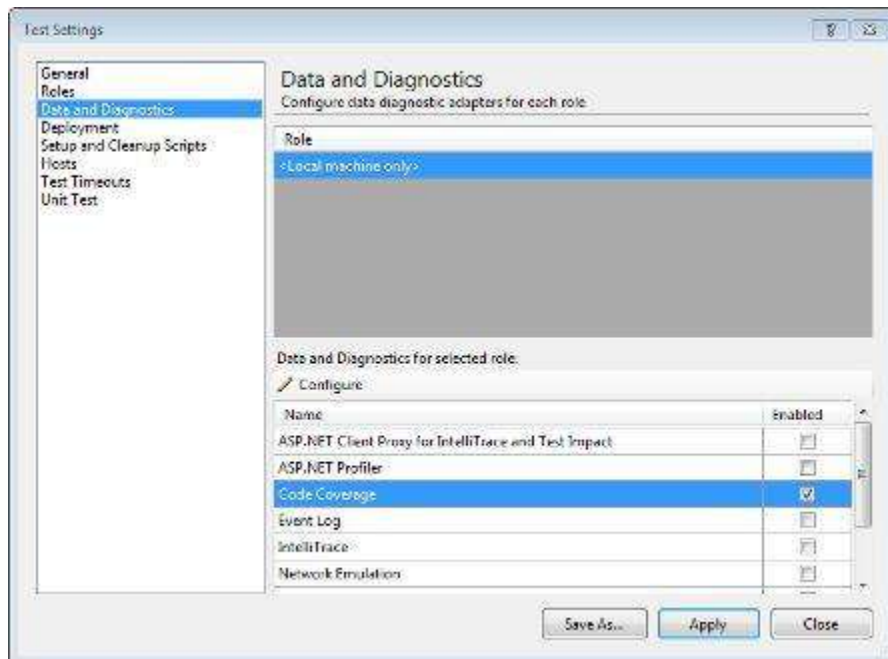


Task 5: Enable Code Coverage

Visual Studio Unit Test in Visual Studio comes with a built-in code coverage support. This task shows how to enable code coverage support for your test.

To enable code coverage

1. In the Visual Studio menu bar, click **Test > Edit Test Settings > Local Test Settings**.
2. In the left pane of the **Test Settings** dialog box, click **Data and Diagnostics**.
3. In the table in the lower right of the dialog, click **Code Coverage**.

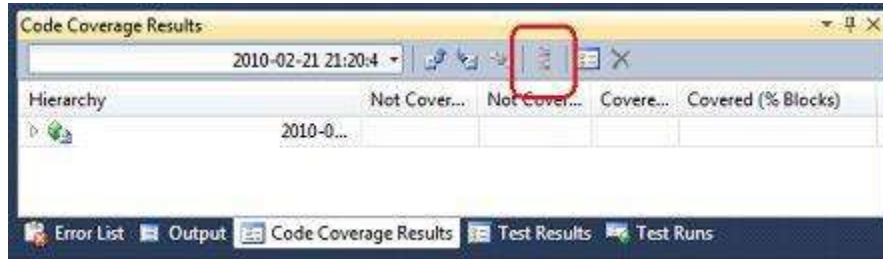


4. Click **Configure**, and in the **Code Coverage Detail** dialog box, click **TestProject1.dll** and click **OK**.

This tells Visual Studio to instrument your test project.

5. Click **Apply**, and then click **Close**.

6. In the **Test View** pane, run **PassingTest** and **FailingTest**—leaving out **ExpectedExceptionTest**.
7. In the toolbar of the **Test Results** pane, click the **Show Code Coverage Results** button. This button is the last one on the right of the toolbar, and might be hidden if the pane is not large enough.
8. In the **Code Coverage Results** pane, enable **source code coloring**.



Covered code is colored in light blue, and uncovered code is colored in red. In this example, you did not run the **ExpectedExceptionTest**, which is why this method is colored in red, as in the following example:

```
[TestMethod]
public void FailingTest()
{
    throw new InvalidOperationException("boom");
}

[TestMethod]
[ExpectedException(typeof(InvalidOperationException))]
public void ExpectedExceptionTest()
{
    throw new InvalidOperationException("boom");
}
```

Summary of Exercise 1

In this exercise, you created a new test project and saw how to author, execute, and debug unit tests, and how to enable code coverage and analyze the results.

Exercise 2: Creating Unit Tests and Refactoring for Test-Driven Development

Test Driven Development (TDD) is a development process where writing tests that specify intended functionality *precedes* writing code. In TDD, the main reason to write tests is to drive the design of the API of the code. It is a side effect that the result is a test suite that also serves as documentation and specification of the intended behavior of the code.

The test-driven development cycle consists of the following short steps:

1. Add a test.
2. Run it and watch it fail.

3. Change the code as little as possible to make the test pass.
4. Run the test again and see it succeed.
5. Refactor the code, if needed.

In this exercise, you implement the Luhn validation algorithm using a TDD approach. These tasks create code that you'll use in later exercises to explore parameterized unit testing with Microsoft Pex.

Task 1: Credit Card Number Validation Specification

The first step in TDD is to create a specification for the application. Your application is going to be a program to test the validity of a credit card number structured according to the Luhn algorithm.

Most credit card companies use a check digit encoding scheme. You add a check digit to the original credit card number at the beginning or the end, and use this check digit to validate the authenticity of the number.

The most popular encoding algorithm is the Luhn algorithm. The following steps compute the Luhn algorithm.

To create a credit card number validation specification

1. Double the value of alternate digits of the primary account number beginning with the second digit from the right. The first digit on the right is the check digit.
2. Add the individual digits comprising the products obtained in Step 1 to each of the unaffected digits in the original number.
3. To be validated, the total obtained in Step 2 must be a number ending in zero—for example 30, 40, or 50 for the account number.

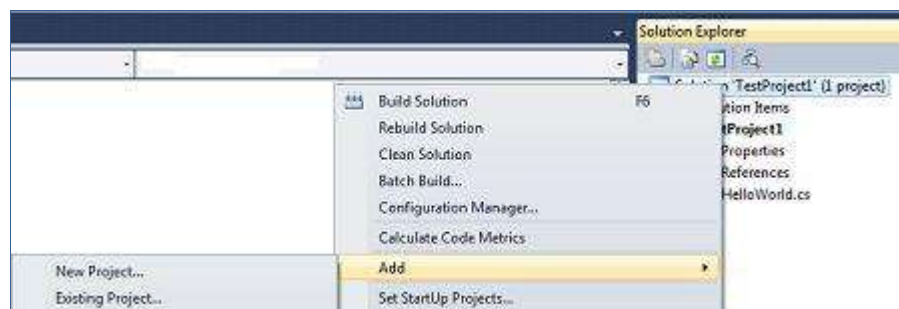
For more information about the Luhn algorithm, see the specification in Annex B of ISO/IEC 7812-1.

Task 2: Add a Failing Test to an Empty Project

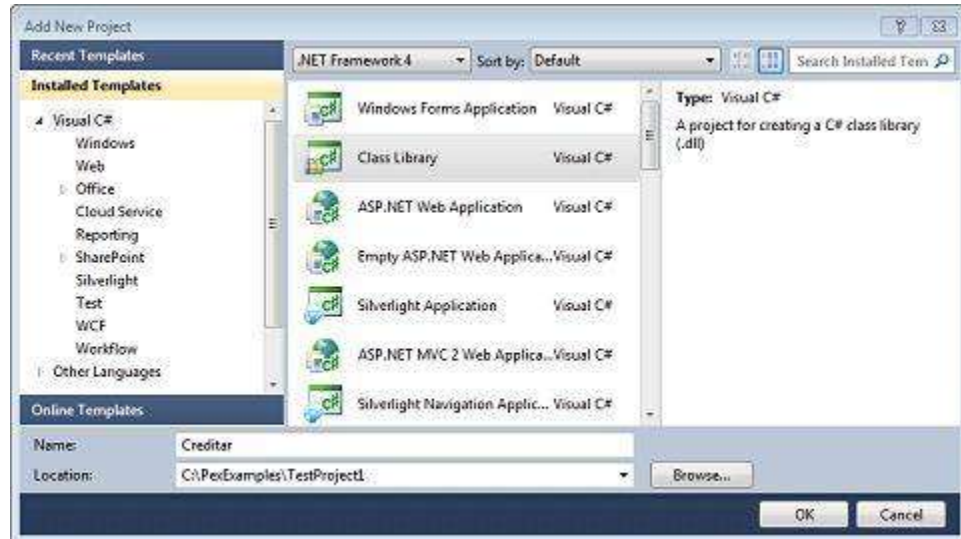
Now that you have a specification for the algorithm, start working on the implementation.

To create a failing test in an empty project

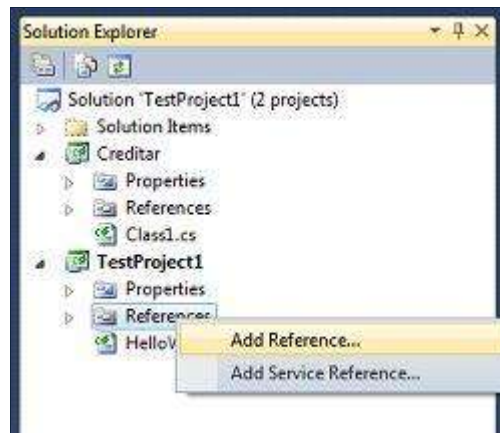
1. In Visual Studio, open TestProject1, which is the solution you created in Exercise 1.
2. In the Solution Explorer, right-click the solution node and click **Add > New Project**. The solution node is the top-level node in the Solution Explorer.



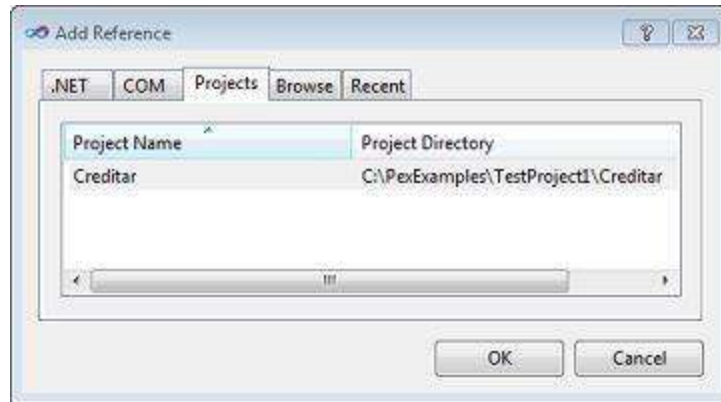
3. In the left pane of the **New Solution** dialog box, click **Visual C# > Windows**. In the center pane, click **Class Library**.
4. In the **Project Name** field, change the name to **Creditar** and click **OK** to add the Creditar class library project to your solution.



5. In the Solution Explorer pane, under the TestProject1 project (not the solution), right-click **References** and click **Add References** from the context menu.



6. In the **Add Reference** dialog box, click the **Projects** tab and double-click the **Creditar** project name to add it as a reference.



Start by writing a unit test for the **Validate** method before writing or declaring the **Validate** method itself, as shown in the following steps.

7. In HelloWorldTest .cs, add a new unit test that verifies that the **Validate** method throws **ArgumentException** when it receives a **null** reference, as follows:

```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void NullNumberThrowsArgumentNullException()
{
    LuhnAlgorithm.Validate(null);
}
```

8. Add a using Creditar statement:

```
using Creditar;
```

9. Right-click Class1.cs in the Creditar project node and click **Rename**.
10. Rename the class to **LuhnAlgorithm**. Click **Yes** in the prompt to rename all references to class1.
11. In the class body of LuhnAlgorithm, add a minimal implementation of the Validate method to make the test fail, as follows:

```
public static class LuhnAlgorithm
{
    public static bool Validate(string number)
    {
        return false;
    }
}
```

12. In the Visual Studio menu bar, click **Build > Build Solution** to compile the project and confirm that it passes.
13. Click **Test > Windows > Test View** to display the **Test View** pane.
14. In the **Test View** pane, click **NullNumberThrowsArgumentNullException**, and click **Run Selection**.
Confirm that the unit test fails.

Task 3: Run the Unit Test and Refactor the Code

Make a minimal change to the **Validate** method to make the test pass.

To run the unit test and watch it pass

1. In the body of the LuhnAlgorithm class, replace the Validate method with code from the following snippet:

```
public static class LuhnAlgorithm
{
    public static bool Validate(string number)
    {
        if(number == null)
            throw new ArgumentNullException("number");
        return false;
    }
}
```

2. Compile the project.
3. Execute the unit test, and confirm that when a non-digit character is passed to the **Validate** method, the implementation throws an **ArgumentException**, as follows:

```
[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void AThrowsArgumentException()
{
    LuhnAlgorithm.Validate("a");
}
```

The minimum change to get this test to pass is not a “correct” implementation or even a useful implementation; however, it is a starting point:

```
public static class LuhnAlgorithm
{
    public static bool Validate(string number)
    {
        if (number == null)
            throw new ArgumentNullException("number");
        if(number=="a")
            throw new ArgumentException("a");
        return false;
    }
}
```

To implement code in this way really means following the incremental idea of the TDD methodology. That is, you continue writing more unit tests that describe the behavior of the specification and fail, and then improve your implementation in order to pass the test.

To refactor the code and continue the iteration

- Now that you have a passing test, refactor the code into a smarter implementation that checks for any non-digit character as shown in the following snippet:

```
public static class LuhnAlgorithm
{
    public static bool Validate(string number)
    {
        if( number == null)
            throw new ArgumentNullException ("number");
        foreach(var c in number)
            if(!Char.IsDigit(c))
                throw new ArgumentException ("number");
    }
}
```

```

        return false;
    }
}

```

You can perform the rest of the unit testing of the Validate method as an exercise:

- Do not forget to use the code coverage view to ensure that the unit test reaches a minimum level of basic block code coverage.
- Executing the unit test suite should yield to a certain percentage of basic block coverage. In the case of this exercise, 80 percent is a reasonable goal.

Summary of Exercise 2

This exercise demonstrated Test Driven Development (TDD) and how it works in unit testing. The following exercises will build on this example by creating parameterized unit tests with Microsoft Pex.

Exploring Parameterized Unit Testing

In “What Are Unit Tests,” you examined a unit test of the .NET **ArrayList** class. Strictly speaking, this unit test only says that by adding a new object to an empty array list, this object becomes the first element of the list. If you want to test for other objects such as array lists using traditional unit tests that do not take inputs, you have to write many tests to cover all possible inputs.

A straightforward extension is to allow parameters, which serve as the test input. The result is a parameterized unit test. You can partition parameterized unit tests into four parts:

- **Assume:** Assume preconditions over the test inputs.
- **Arrange:** Set up the unit under test—determine which parameters are used to shape legal test inputs.
- **Act:** Exercise the unit under test by adding a method sequence, which specifies a scenario, and capturing any resulting state.

You use parameters for argument values when calling other methods in the Act stage.

- **Assert:** Verify the behavior by adding assertions that encode the rules of a unit test.

Here is a parameterized version of the array list unit test that describes the normal behavior of the Add method with respect to two observers, the property **Count**, and the indexing operator **[]**. Under the condition that a given array list is not **null**, this parameterized unit test asserts that after adding an element to the list, the element is indeed present at the end of the list:

```

public void AddSpec2(
    // parameters
    ArrayList list, object element)
{
    // arrange
    PexAssume.IsTrue(list != null);
    // act
    int len = list.Count;
    list.Add(element);
    // assert
    PexAssert.IsTrue(list[len] == element);
}

```

This test is more general than the original test. You can call parameterized unit tests like this one with various input values, perhaps drawn from an attached database. Unit testing frameworks that support parameterized unit tests sometimes refer to them as data-driven tests. Instead of stating the data values from the Arrange step explicitly, a parameterized unit test might state assumptions about what valid input data must look like.

You can assume that the list is not null. Parameterized unit tests are more general specifications than traditional unit tests: parameterized unit tests state the intended program behavior for entire classes of program inputs, and not just for one input. Yet parameterized unit tests are still easy to write, because they merely state what the system is supposed to do and not how to accomplish the goal.

Unlike many other forms of specification documents, you write parameterized unit tests on the level of the actual software APIs, in the programming language of the software project. This allows parameterized unit tests to evolve naturally with the code that they test.

To split the specification and test cases for parameterized unit testing:

- First, you specify the intended external behavior of the software as parameterized unit tests. Only human beings can perform this specification task.
- Second, a tool such as Pex automatically creates test suites with high code coverage by determining test inputs that exercise different execution paths of the implementation.

Coverage through Test Input Generation

Adding parameters to a unit test improves its expressiveness as a specification of intended behavior, but you lose concrete test cases. You no longer execute a parameterized test by itself. You need actual parameters, and you need to decide what values must be provided to ensure sufficient and comprehensive testing. Which values should you choose?

Consider the following code that implements **Add** and the indexing operator in the .NET base class library:

```
// ArrayList implementation in .NET
public class ArrayList
{
    ...
    private Object[] _items = null;
    private int _size, _version;
    ...
    public virtual int Add(Object value)
    {
        if (_size == _items.Length) EnsureCapacity(_size + 1);
        _items[_size] = value;
        _version++;
        return _size++;
    }
}
```

In this method, there are two points of interest:

- *When the internal capacity of the array list is large enough to add another element.*

- *When the internal capacity of the array list must increase before adding another element.*

Assume that the library methods invoked by the **ArrayList** implementation are themselves correctly implemented (**EnsureCapacity** guarantees that the `_items` array is resized so its length is greater or equal `_size+1`), and do not consider possible integer overflows.

With these assumptions, you only need to run two test cases to check that the assertion embedded in `Add` is true for all array lists and all objects given the existing .NET implementation. You need two test cases, as there are only two execution paths through the `Add` method; this means you can partition all inputs into two equivalence classes:

- One where `_size == _items.Length`.
- One where it does not.

Each of the following two test cases represents one of the two equivalence classes:

```
[TestMethod]
public void TestAddNoOverflow()
{
    AddSpec2(new ArrayList(1), new object());
}

[TestMethod]
public void TestAddWithOverflow()
{
    AddSpec2(new ArrayList(0), new object());
}
```

Every input will execute one or the other of the behaviors. Because of this, you do not need any other inputs beyond these two.

Theory of Parameterized Unit Tests

This section is advanced and is not required to understand how to use Pex.

By adding parameters, you turn a closed unit test into a universally quantified conditional axiom that must hold for all inputs under specified assumptions. Intuitively, the **AddSpec2(;;)** method asserts that for all array lists *a* and all objects *o*, the following holds:

$$\forall \text{ArrayList } a, \text{ object } o, \\ (a \neq \text{null}) \rightarrow \text{let } i = a.\text{Count} \text{ in } a.\text{Add}(o) ; a[i] == o$$

Where “;” represents the sequence composition from left to right: $(f ; g)(x) == g(f(x))$. The axiom becomes more complicated when we specify side effects of sequential code precisely.

For information about how to model references to objects on the heap, see the section “Symbolic State Representation” in “Advanced Concepts: Parameterized Unit Testing with Microsoft Pex.”

Tip: Apply patterns for parameterized unit testing

As parameterized unit tests are really just a way to write algebraic specifications as code, you can apply many standard patterns for algebraic specifications in the context of parameterized unit testing. You can find a collection of such patterns for Pex in “Parameterized Test Patterns for Microsoft Pex”.

Test-Driven Development by Parameterized Unit Testing

Exercise 2 introduced Test Driven Development (TDD). It is straightforward to extend test-driven development to parameterized unit tests. In fact, it is usually more expressive to state the intended properties of an API with parameterized unit tests instead of closed unit tests with static data.

When you write parameterized unit tests, the TDD process is as follows:

1. Write or change a parameterized unit test, or code that implements the behavior described by already existing parameterized unit tests.
2. Run Pex on the parameterized unit test.
3. If Pex finds errors, go back to step 1 to change the parameterized unit test or fix the code—possibly with the **Add Preconditions** feature in Pex.
4. Keep the generated tests for future regression testing.

Exercise 3: Creating Parameterized Unit Tests in Visual Studio

The previous section described how parameterized unit tests improve on traditional unit tests. This exercise shows how to enable Pex for the TestProject1 project and how to create a unit test with Pex.

Task 1: Add Pex to a Project and Create a Parameterized Unit Test

To add Pex to a project

1. In the **Solution Explorer**, right-click the **References** folder under TestProject1, and click **Add Reference**.
2. In the **Add Reference** dialog box, click the **.NET** tab, and click **Microsoft.Pex.Framework**. Click **OK** to add this reference to your project.

To create a parameterized unit test

1. Open the class HelloWorldTest for editing.
2. In the HelloWorldTest class, add a new public method ParameterizedTest that takes an **int** parameter. Mark this method with [PexMethod], as follows:

```
[PexMethod]
public void ParameterizedTest(int i)
{
    if (i == 123)
        throw new ArgumentException("i");
}
```

Notice that Visual Studio does not recognize [PexMethod] yet. You could add the following using statement to your code:

```
using Microsoft.Pex.Framework
```

Or you could use the following steps to let Visual Studio add it for you.

To let Visual Studio add a using statement

1. In Visual Studio, move the cursor over the **[PexMethod]** attribute.
Notice the small red rectangle located at the bottom right of **[PexMethod]**.
2. Click the red rectangle to display a context menu with two entries:
 - The first adds a using clause to your source code for the namespace that contains Pex
 - The second adds the fully qualified namespace to this occurrence of **PexMethod**.
3. Click **Enter** to insert the **using** clause.
The keyboard shortcut to open the IntelliSense context menu is **CTRL+.** (the period character).

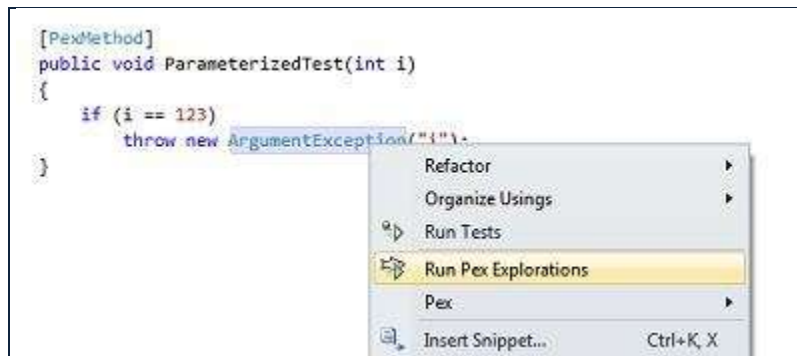
Tip: Using snippets

Pex also provides snippets to create a new parameterized unit test skeleton. In the editor, write **pexm** and then press **TAB**. This expands the snippet to a new blank unit test.

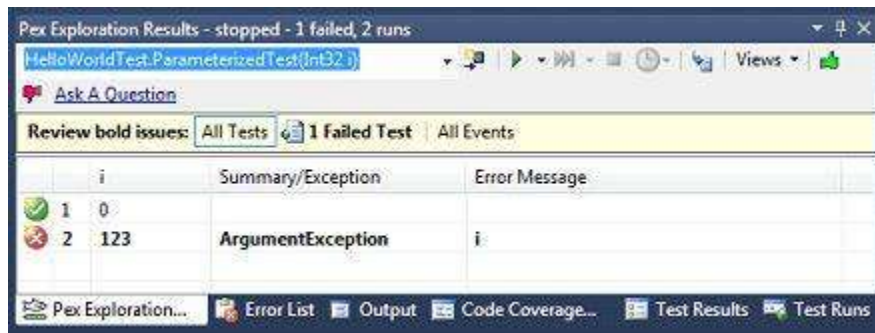
Task 2: Run the Parameterized Unit Test

To run the parameterized unit test

1. Right-click inside the ParameterizedTest method, and click **Run Pex Exploration** in the context menu.



Pex automatically displays the **Pex Results** pane at the bottom of the Visual Studio window. Most of your interactions with Pex are through this pane.



Each row in the table corresponds to a generated test for the current exploration. Each row contains:

- An icon describing the status of the test: passing or failing.
- A number indicating how often Pex had to execute the parameterized unit test with different input values in order to arrive at this result.
- A summary of any exceptions that occurred.

Pex also logs the values of the input parameters of the test.

Note: Pex often runs the parameterized unit test several times until Pex outputs the next test row. The rationale behind this behavior is that Pex explores different execution paths of the program, but it only logs a new test when the test increases the coverage. Many execution paths might result in the same coverage.

Coverage here refers to arc coverage. For more information on test coverage, see “Advanced Concepts: Parameterized Unit Testing with Microsoft Pex.”

2. When exploring the `ParameterizedTest` method that you wrote earlier, Pex generates two unit tests. To access each unit test, click the corresponding row and click **Go to generated test**.

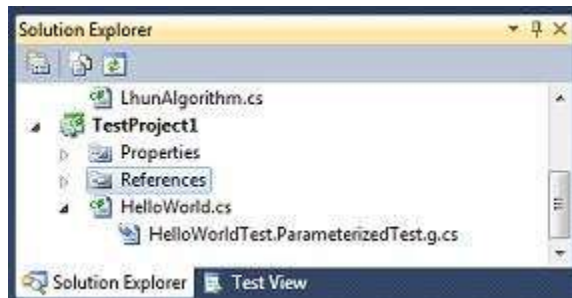
```
[TestMethod]
[PexGeneratedBy (typeof(TestClass))]
public void ParameterizedTest01()
{
    this.ParameterizedTest(0);
}

[TestMethod]
[PexGeneratedBy(typeof(TestClass))]
[PexRaisedException(typeof(ArgumentException))]
public void ParameterizedTest02()
{
    this.ParameterizedTest(123);
}
```

The **TestMethod** attribute indicates that the generated methods are unit tests. The **PexGeneratedBy** attribute indicates Pex generated the test by exploring parameterized unit tests, in a particular test class, and the **PexRaisedException** indicates that this test raised an unexpected exception.

3. Pex writes the generated unit tests to a separate file that is located in the same directory as **HelloWorldTest.cs**, the file that contains the parameterized unit test.

In the Visual Studio **Solution Explorer** pane, the generated file shows up as a sub item of **HelloWorldTest.cs**.



Summary of Exercise 3

In this section, you built on what you learned previously about parameterized unit tests and how they improve on traditional unit tests. You learned how to enable Pex and to use it on your unit test.

Understanding How Pex Chooses Inputs

Classic unit tests are methods without parameters, parameterized unit tests are methods with parameters. Many unit testing frameworks now support parameterized unit tests, including the following:

- Visual Studio Unit Test in Visual Studio—referred to as data-driven tests.
- MbUnit—referred to as row tests.
- JUnit—referred to as theories.

At one time, you had to provide the input parameters for those tests as a range, a spreadsheet, or database of some sort. Improper choice of inputs would lead to missed corner cases or a hugely redundant test suite. With Pex, things change for the better and you do not have to provide any input to the parameterized unit tests. By analyzing the program behavior at runtime Pex generates inputs that matter, and those inputs increase the coverage of the test suite.

Choosing meaningful test inputs requires a certain understanding of the code under test, which in turn requires an understanding of what the relevant parts of the (potentially huge) code under test are.

Whitebox Testing is an incremental process during which the tester learns more and more about the actual behavior of the code. Another characterization of whitebox testing is that it is test design and test execution at the same time. Whitebox testing allows the developer to construct better tests, and more of them.

Pex uses dynamic symbolic execution, a technique that works in a way similar to whitebox testing. The tool executes code multiple times and learns about the program behavior by monitoring the control and data flow. After each run, it picks a branch that was not covered previously, builds a constraint system (a predicate over the test inputs) to reach that branch, then uses a constraint solver to determine new test inputs, if any.

You execute the test again with the new inputs and this 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. This is Automated Whitebox Testing.

For the advanced reader, “Advanced Concepts: Parameterized Unit Testing with Microsoft Pex” discusses the theoretical foundations and technical details of dynamic symbolic execution.

In this section, you learn how to use whitebox testing to test a simple method that takes two integers as an input and prints different strings to the console based on those values. Pex would perform a similar analysis, only fully automatically, but in this section, you will manually go through all the steps involved in the analysis, so that you can understand how Pex works.

In the next code sample, the numbered comments identify the parts of the code that are discussed after this program listing:

```
void SomeMethod(int i,int j){           // Line 1
    if(i < 0){                          // Line 2
        Console.WriteLine("line3");     // Line 3
        if(j == 123)                    // Line 4
            Console.WriteLine("line5"); // Line 5
    }                                     // Line 6
    else                                  // Line 7
        Console.WriteLine("line8");     // Line 8
    }                                     // Line 9
}
```

One way to explore the possible behaviors of this method is to throw different values at SomeMethod and analyze the output to see what is happening.

Iteration 1: Pick an arbitrary value

Create a unit test that does exactly that and step into the debugger. Because you do not really know anything about the SomeMethod method yet, you simply pick 0 for i and j.

```
[TestMethod]
void Zero()
{
    SomeClass.SomeMethod (0, 0);
}
```

When you reach the statement on line 8:

```
Console.WriteLine("line8");
```

This branch executes because the condition $i < 0$ on line 2 evaluated to false. If you let the execution continue, the test finishes successfully.

Iteration 2: Flip the last condition

You might have noticed in the previous iteration that execution did not reach some code on line 3. You also know that this code path was not covered because the condition $i < 0$ evaluated to false. At this point, you can usually figure out a value of i to make this condition true. In this case, you need to solve for i such that $i < 0$. Pick -1.

```
[TestMethod]
void MinusOne()
{
    SomeClass.SomeMethod (-1, 0);
}
```

Run the test under the debugger. As expected on line 2, the condition now evaluates to true and the program takes the other branch.

The program continues and reaches the **if** statement on line 4. The condition $j = 123$ evaluates to false, so remember the following: Line 4, $j \neq 123$, uncovered branch. The program continues to run and finishes.

Iteration 3: Path condition + flipped condition

There are still some uncovered branches to cover in the method, guarded by the condition at line 4. To be able to cover this code, you need to be able to do two things:

- Reach line 4: $i < 0$
- Make the condition in line 4 evaluate to true: $j = 123$

So, to cover the last statement in the method, you need to find parameter values such that $i < 0 \wedge j = 123$. This means pick $i = -1$ and $j = 123$, as follows:

```
[TestMethod]
void MinusOneAndOneTwoThree()
{
    SomeClass.SomeMethod (-1, 123);
}
```

The test executes and prints **line5** as expected. At this point, you have fully covered the behavior of `SomeMethod`.

Exercise 4: Instrumenting Code for Parameterized Unit Testing

In this exercise, you revisit the `LuhnAlgorithm` class, write a parameterized unit test, and use Pex to create stubs for the public API of the class. This section builds on the code you created in Exercise 3.

Task 1: Run the Pex Code-Generation Wizard

Pex provides a code-generation wizard that automatically produces parameterized test stubs for the entire public API of a class. These stubs are used as a starting point to write more elaborate scenarios.

To use the wizard

1. Right-click inside the body of the `LuhnAlgorithm` class.
2. In the context menu, click **Pex > Create Parameterized Unit Test**.

The Pex wizard compiles and analyzes the `LuhnAlgorithm` class to produce the test stubs. Pex automatically adds the test files to the test project using the information provided by the `PexAssemblyUnderTest` attribute.

Task 2: Set Up the Code Instrumentation

To set up code instrumentation

1. Add a parameterized unit test to your `HelloWorldTest.cs` class that specifies that a valid credit card number, according to the Luhn algorithm, and does not contain any non-digit characters.

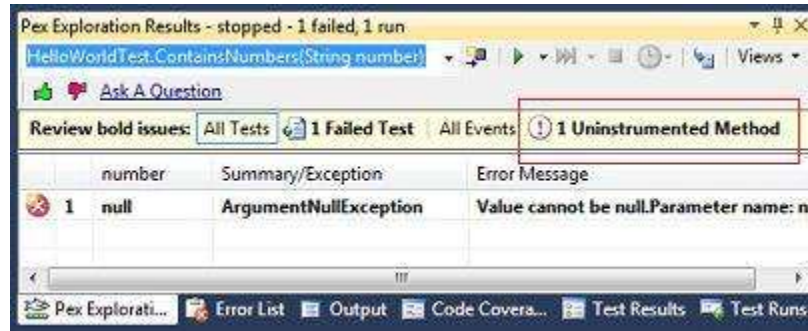
```
[PexMethod]
public void ContainsNumbers(string number)
{
    PexAssume.IsTrue(LuhnAlgorithm.Validate(number));
    PexAssert.TrueForAll(number, delegate(char c)
    {
        return char.IsDigit(c);
    });
}
```

2. Run the test and analyze the results using what you have learned so far. Pex only generated a single trivial test with a **null** reference as the number string.

The problem is that Pex did not analyze the `Validate` method. Instrument your code for Pex to generate relevant test inputs. Because instrumentation is quite expensive, Pex does not instrument all code by default.

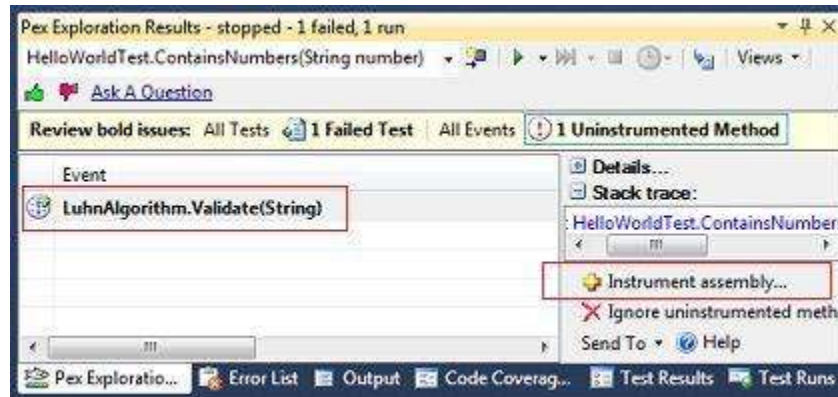
You need to specify which assemblies or types Pex instruments. You usually do this process once. Pex cannot build the constraints to explore the behavior of the `Validate` method, because the product assembly is uninstrumented. This eventually leads to poor code coverage in the generated test suite.

Pex shows important issues such as noninstrumented calls to methods in a status bar toward the top of the **Pex Exploration Results** pane.



3. Click **Uninstrumented Method** to see the list of uninstrumented methods.

When you click a method in the list, Pex offers several actions to get rid of the warning in the Details pane. Because you want to test the `Validate` method, click **Instrument assembly**.



4. This adds a custom attribute in the test project that tells Pex that the product assembly should be instrumented:

```
[assembly:PexInstrumentAssembly("Creditar")]
```

Pex adds a special file **PexAssemblyInfo.cs** to hold the assembly level attributes. Pex persists all project-specific settings as attributes.

5. Run Pex again and analyze the results. The coverage should be much better.

Binding the Tests and the Code under Test

You should specify the code under test so that Pex focuses its analysis on that code and shows a relevant dynamic coverage graph. Pex provides several ways to provide this information.

- **[PexClass]** has a special constructor that takes a type.

You use this constructor to specify the type under test of a particular test fixture. Pex uses this information to prioritize the exploration. Update the **LuhnAlgorithmTest** class to specify that it is testing the **LuhnAlgorithm** class.

```
[TestClass, PexClass(typeof(LuhnAlgorithm))]
public partial class LuhnAlgorithmTest{
```

- Because you know that the project **Creditor.Tests** is the test project for **Creditor**, add a custom attribute that provides this information to Pex.

```
[assembly: PexAssemblyUnderTest("Creditor")]
```

Place this assembly-level attribute before the namespace declaration in C #.

Task 3: Explore the Instrumentation Configuration

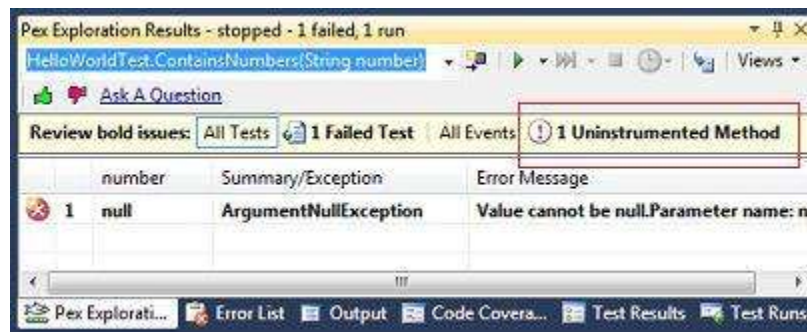
You learned in Exercise 4 how to tag your code so that Pex could instrument it.

Pex can generate a test suite with high code coverage only if it monitors the relevant parts of the code. Therefore, it is important to configure which types Pex should instrument. Consider the following parameterized unit test:

```
[PexMethod(MaxBranches = 2000)]
public void Test(string s)
{
    DateTime dt = DateTime.Parse(s);
    PexObserve.ValueForViewing("dt", dt);
}
```

The **MaxBranches** setting makes sure that Pex does not stop too early. For information, see “Advanced Concepts: Parameterized Unit Testing with Microsoft Pex.”

When Pex generates tests, it only generates a single test at first. However, you do get a warning that some methods were not instrumented.



When you click **Uninstrumented Methods**, the log view shows the list of uninstrumented methods. These are internal implementation methods from the .NET **DateTime** class.

Use what you learned in the last tutorial to tell Pex that it should instrument this type in the future. If code is irrelevant to the test, click **Ignore uninstrumented method**. Pex inserts custom attributes such as the following for you.

```
using Microsoft.Pex.Framework.Instrumentation;
[assembly: PexInstrumentType("mscorlib", "System.DateTimeParse")]
[assembly: PexInstrumentType("mscorlib", "System.__DTString")]
```

After you instruct Pex to instrument a type, you have to re-run Pex to see the results for the instrumentation. In turn, you might get more uninstrumented method warnings because Pex is now reaching new code.

To explore coverage filtering

1. Can you determine the set of types needed for **DateTime** that Pex must instrument in order to generate a valid **DateTime** string?

Hint: Work in an iterative loop, where you add one **DateTime** type at a time to the list of instrumented types.

2. Re-run Pex to see what changed.

DateTime is a .NET type defined in mscorlib. Pex normally does not report coverage data achieved in this basic library. You can tell Pex to include coverage data with the following assembly-level attribute:

```
using Microsoft.Pex.Framework.Coverage;
using Microsoft.ExtendedReflection.Coverage;
[assembly: PexCoverageFilterAssembly(PexCoverageDomain.UserCodeUnderTest, "mscorlib")]
```

Summary of Exercise 4

In this procedure, you have learned to control what parts of your code Pex instruments. Because instrumentation is expensive, you need to enable instrumentation only when needed. You also used Pex to generate stubs for public API of the Luhn Algorithm class.

Exercise 5: Using Pex from the Command Line

The previous exercise showed how to use Pex in the Visual Studio environment. This exercise introduces the ability to run Pex from the command line and read the HTML reports generated by Pex.

Before running Pex, you should create a .NET assembly—that is, a .NET executable with a filename ending in **.dll** or **.exe**—that contains a class annotated with the **PexClassAttribute**, which in turn contains a public instance method annotated with the **PexMethodAttribute**. If you do not have access to one, you can compile the following example:

```
//Self-contained parameterized unit test code
[PexClass]
public partial class TestClass
{
    [PexMethod]
    public void ParameterizedTest(int i)
    {
        if (i == 123)
            throw new ArgumentException("i");
    }
}
```

You can build this code with the standalone C#-compiler **csc.exe**, Visual C# Express Edition, or any other C# development environment. You will need to reference the **Microsoft.Pex.Framework** assembly.

Tip: You can use Pex with other test frameworks

Pex works best with the unit test framework of Visual Studio, but Pex can also generate tests for other unit test frameworks. Pex detects the intended unit test framework by inspecting the referenced assemblies in your test project.

Task 1: Run Pex from the Command Line

This section shows how to use Pex from the command line.

To run Pex from the command line

1. On the desktop, click **Start** and type **cmd** in the search box.
2. In the command prompt window, navigate to the build folder of the test project.
3. Run the Pex command-line program—**pex.exe**—on the test project assembly by typing the following command:

```
pex.exe bin\Debug\TestProject1.dll
```

This runs all the parameterized unit tests in that assembly, although you might need to change name of the .dll file as appropriate.

Pex runs and logs its activity to the console. At the end of the run, it automatically opens a detailed HTML report.

Understanding the Command Line Output

The console output reports different aspects of the test. The precise output might vary, depending on your version of Pex and .NET.

1. Pex inspects the test project and launches a second process.

In the second process, Pex acts as a profiler in order to instrument code so that Pex can precisely monitor execution path.

```
1 Microsoft Pex -- http://research.microsoft.com/pex
2 Copyright (c) Microsoft Corporation
3 All rights reserved.
4
5 instrumenting...
```

2. At this point, the instrumented process is running. Pex is loading the parameterized tests. Any issue in the test metadata is logged during this phase.

```
6 Launched Pex x86 Edition on .NET v2.0.50727
7 [reports] report path:
8 reports\TestProject1.71115.170912.pex
9 00:00:00.0> starting execution
10 00:00:00.1> reflecting tests
```

3. In the next phase, Pex explores all parameterized unit tests:

```
11 00:00:02.8> TestProject1
12 00:00:02.8> TestProject1.TestClass
13 00:00:12.3> TestClass.ParameterizedTest(Int32)
```

4. At each new test, Pex writes a line that describes the test.

On the second run, the test `ParameterizedTest02` was generated and raised an **ArgumentException**, as follows:

```
14 [test] (run 1) ParameterizedTest01 (new)
15 [coverage]coverage increased from 0 to 2 blocks
16 [test] (run 2) ParameterizedTest02 (new),
17 Exception: Exception of type 'System.Argument.Exception' was
   thrown.
18 [coverage] coverage increased from 2 to 4 blocks
```

5. At the end of the exploration, Pex also gives a basic block coverage summary:

```
19 [dynamic coverage] 4/4 block (100.00 percent)
```

6. After Pex processes all parameterized unit tests, Pex displays some statistics, and then renders the HTML report.

In this run, seven tests were generated, one of which was considered a failing test:

```
20 00:00:13.2> [finished]
21 -- 2 generated tests, 1 failing, 2 new
```

7. Pex logged no errors or warnings:

```
22 -- 0 critical errors, 0 errors, 0 warnings
```

Every time you run Pex, it generates a new folder in the **reports** directory to hold all the generated files. By default the folders get recycled after a while; in other words, the generated reports have only a limited lifetime and you should copy a report that you want to preserve. Pex constructs the generated folder name from the test assembly name and a time stamp.

```
23 [reports] generating reports...
24 [reports] html report:
25 reports\TestProject1.71115.170912.pex\pex.html
```

8. Finally, Pex displays the overall verdict, success or failure.

```
26 EXPLORATION SUCCESS
```

Tip: Where are the generated tests?

Pex creates a subdirectory that contains the generated tests as code. In this example, the directory would be:

```
reports\TestProject1.71115.170912.pex\tests
```

Using Command-Line Filters

The Pex command line supports a rich set of filters to select which exploration to run:

By namespace: **pex/nf:TestProject**

By type name: **pex/tf>HelloWorld**

By method name: **pex/mf:Add**

By test suite: **pex/sf:checkin**

Combine all those filters together: **pex/nf:TestProject/tf>HelloWorld/mf:Add**

By default, Pex uses a partial case-insensitive string match. To have a precise match, add a bang (!) at the end of the filter—for example:

pex/tf>HelloWorldTest!

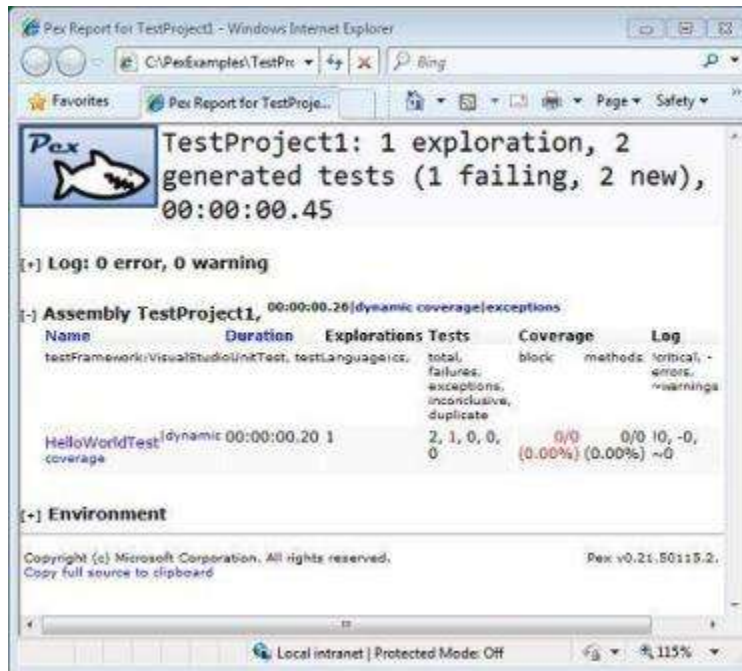
Task 2: Use Pex HTML Report

The HTML report appears after every run of Pex from the command line. It gives a tabular view of each assembly, fixture, and exploration. The title highlights general statistics such as the number of fixtures and generated tests.

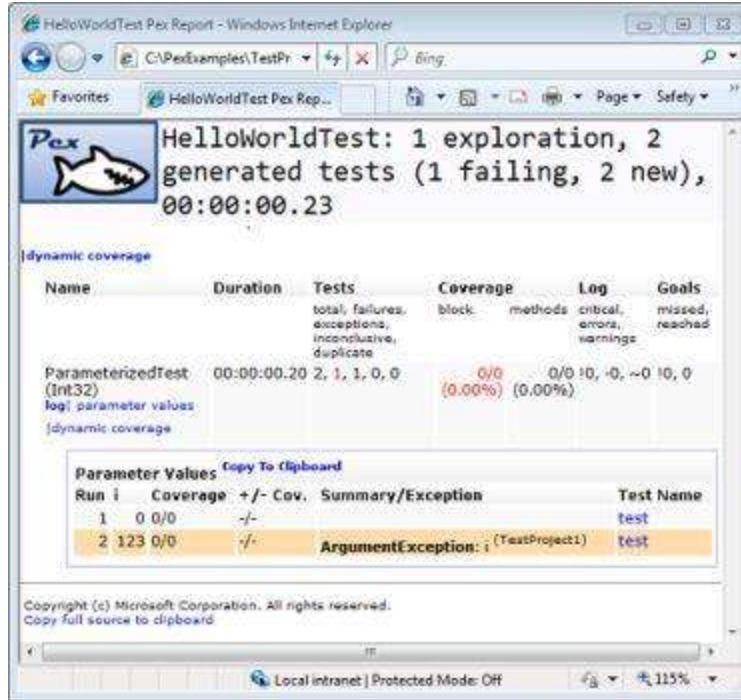
The following describes the report Pex generated from the test code used in the early exercise that showed how to use Pex in Visual Studio.

To navigate the Pex HTML report

1. In Visual Studio, click **HelloWorldTest** to go to a Pex report details page.

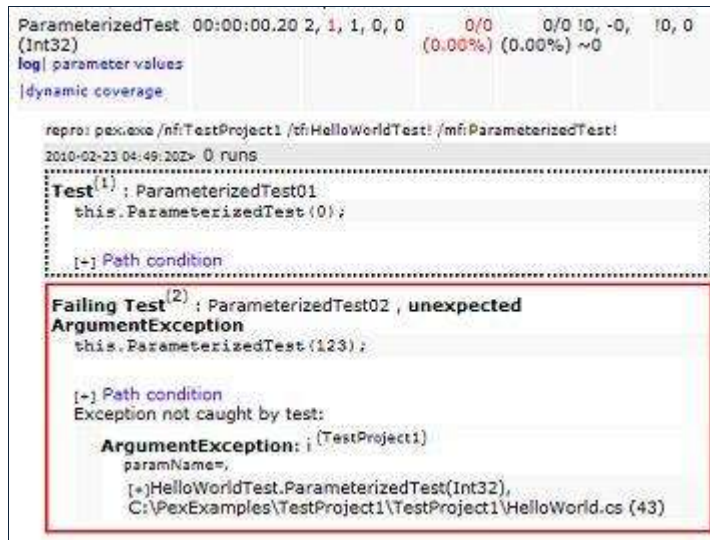


2. Below the test entry for Parameter values, click **Parameter** values.



Parameter Values displays a table where each row contains the values of the parameters for a particular test. This table is similar to the **Pex Results** view in Visual Studio.

- To open a detailed activity log of the exploration process, click the **log** link.



The log of this test starts with a repro command that Pex uses to execute that particular exploration from the command line. Logs for the generated tests appear after the repro command.

- Click **Dynamic Coverage** to display the **Dynamic Code Coverage Summary** page.

Dynamic Code Coverage Summary			
Modules/Types/Methods	Dynamic Coverage (blocks)		
	User Code Under Test	User Or Test Code	
{+;TestProject1.dll(1 methods)	4/4	(100.00%)	
Sources	Methods	Dynamic Coverage (blocks)	
		User Code Under Test	User Or Test Code
C:\PexExamples\TestProject1\HelloWorld.cs	1	4/4	(100.00%)
<small>Copyright (c) Microsoft Corporation. All rights reserved. Copy full source to clipboard</small>			

- Browse through the coverage data by assemblies, fixture, and methods.
- To view the coverage by source file, click the **html** link.

Each source file's color follows the legend at the top of the page.

- **User code under test (covered)**—Covered statements of the code-under-test.
- **User code under test (uncovered)**—Statements of the code-under-test that were not covered.
- **User code or test (covered)**—Statements that were not part of the code-under test that were covered.
- **User code or test (uncovered)**—Statements that were not covered or under test.
- **Tagged**—The statements are tagged with interesting information from the exploration process. Hover with the mouse over a tagged statement to see more information.

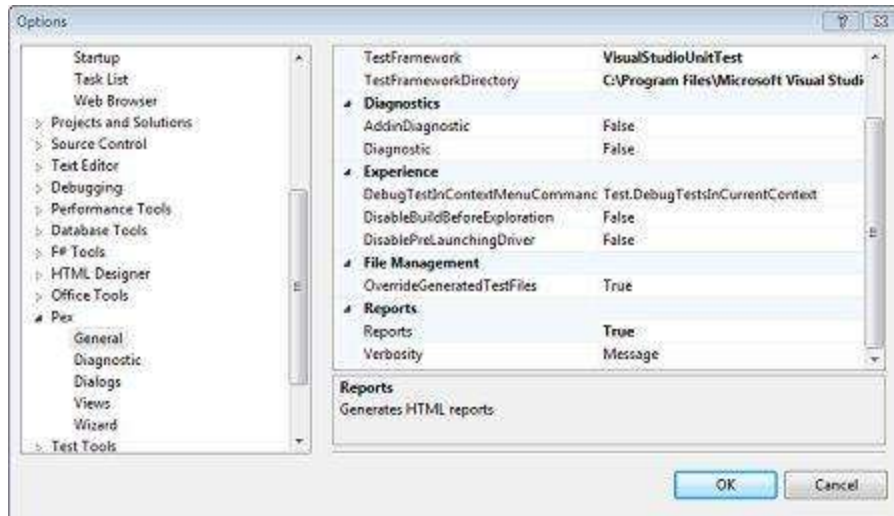
The following section shows how to control code coverage.

Task 3: View the Pex HTML Report in Visual Studio

You can also view the Pex HTML report in Visual Studio. Use the following procedure to enable HTML report generation in Visual Studio.

To navigate the Pex HTML report

- Click **Tools > Options** in the Visual Studio toolbar to open the **Options** dialog box.
- In the left pane of the **Options** dialog box, click on **Pex > General**. In the right pane, click the dropdown list next to **Reports** and choose **True**.



Once you enable reports, run Pex again to actually generate a report.

3. In the **Pex Exploration Results** pane, click the **Views** dropdown box, and then click **Open Report** to open the HTML report.

The report will open in the Visual Studio editing pane.



Summary of Exercise 5

In this tutorial, you learned how to use Pex from the command line, how to interpret console output, how to view HTML reports in both a browser and in Visual Studio, and how to understand those reports.

Experimenting with Expressions, Encoding, and RLE Algorithms

This section suggests some experiments for the experienced programmer.

Try creating a parameterized unit test for regular expressions

Try out your favorite regular expression and see if Pex generates an input string that matches that expression:

```
[PexMethod]
public void MatchRegex([PexAssumeNotNull]string value)
{
    if (Regex.IsMatch(value, @"^[0-9a-fA-F]+[\r\n]*$"))
        throw new Exception("found it!");
}
```

The [PexAssumeNotNull] attribute indicates that Pex should pass null for the value parameter. This is a simple case of an assumption on test inputs. For more details, see “Advanced Concepts: Parameterized Unit Testing with Microsoft Pex.”

Try creating a parameterized unit test for binary encoding

The .NET **Encoding** class is a reader-writer between byte arrays and char arrays. A simple example of encoding copies the bits of a char to two byte and vice versa.

The **Encoding** class is defined in the System.Text namespace and has four abstract methods that need to be overloaded.

Using the following parameterized unit test, write an implementation for GetChars and GetCharCount:

```
[PexMethod]
public void CharCountMatching (
    [PexAssumeNotNull]Encoding encoding,
    byte[] bytes,
    int index,
    int count)
{
    int charCount = encoding.GetCharCount(bytes, index, count);
    char[] chars = encoding.GetChars(bytes, index, count);
    Assert.AreEqual(charCount, chars.Length);
}
```

Start from the following program that contains an implementation of those methods. There are errors in the implementation of this code that you can use Pex to find and resolve:

```
public class SloppyBinaryEncoding : Encoding
{
    const int BytesPerChar = 2;
    public override int GetCharCount(byte[] bytes, int index, int count)
    {
        return count / 2;
    }

    public override int GetChars(
        byte[] bytes, int byteIndex, int byteCount,
        char[] chars, int charIndex)
```

```

{
    int j = charIndex;
    for (int i = 0; i < byteCount; i += 2)
    {
        chars[j++] =
            (char)(bytes[byteIndex + i] << 8 |
                bytes[byteIndex + i + 1]);
    }
    return j - charIndex;
}

```

Try creating a parameterized unit test for RLE compression and decompression

Write a run-length encoding (RLE) compression and decompression algorithm that takes arrays of bytes as input and output. The following snippet describes the signature of the RLE implementation:

```

public class RleCompressor
{
    public static byte[] Compress(byte[] input)
    {
        throw new NotImplementedException();
    }

    public static byte[] Decompress(byte[] input)
    {
        throw new NotImplementedException();
    }
}

```

The parameterized unit test should simply state the roundtrip property: Compression followed by decompression should yield the original sequence of bytes.

For example, write the following parameterized unit test:

```

[PexMethod]
public void Roundtrip(byte[] data)
{
    PexAssume.IsNotNull(data, "data");
    byte[] compressed = RleCompressor.Compress(data);
    byte[] uncompressed = RleCompressor.Decompress(compressed);
    // assertions
    PexAssert.IsNotNull(uncompressed, "uncompressed");
    Assert.AreEqual(data.Length, uncompressed.Length);
    for (int i = 0; i < data.Length; i++)
        PexAssert.AreEqual(data[i], uncompressed[i]);
}

```

Isolating Tests from the Environment with Microsoft Moles

Each unit test, whether parameterized or not, should test a single feature so that a failing unit test identifies the broken feature as concisely as possible

In practice, it is often difficult to test features in isolation: The code might take a file name as its input and use the operating system to read in the contents of the file. Alternatively, the code might need to connect to another machine to fulfill its purpose.

The first step towards making the code testable is to introduce abstraction layers. For example, the following Parse method is not testable in isolation, because it interacts with the file system. When you cannot add an abstraction layer due to the code being inaccessible, Microsoft Moles offers a solution:

```
public void Parse(string fileName)
{
    StreamReader reader = File.OpenText(fileName);
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        ..
    }
}
```

In the next snippet, you will decouple the actual parsing logic from the implementation of the abstract **StreamReader** class. This makes the class more testable and you can pass any implementation of the abstract **StreamReader** class into the method:

```
public void Parse(string fileName)
{
    this.Parse(File.OpenText(fileName));
}

public void Parse(StreamReader reader)
{
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        ...
    }
}
```

Abstraction from the environment is necessary if you want to have a Pex-testable design.

Isolating Tests with Mocks and Stubs

When you write code with abstraction layers, you use mock objects to hide parts of the program that are irrelevant to a tested feature. Mock objects answer queries with fixed values similar to those that the substituted program would have computed.

Today, developers usually define the behavior of mock objects by hand. Behavior means the return values of mocked methods and what exceptions they should throw among other things.

Stubs are trivial implementations of all methods of an interface or a class. Stubs do not perform any actions by themselves and usually just return some default value. The developer must still program the behavior of the stubs.

You can distill the actual behavior of a program into mock objects using capture-replay, where the code is iteratively tested and new mocks provided as existing calls into the program trigger errors. For an example of this approach, see “Unit Testing SharePoint Foundation with Microsoft Pex and Moles.”

Tip: The Moles framework does more than isolate tests.

The Microsoft Moles framework makes writing stubs easier. In addition to creating stubs, the Moles framework also allows detouring legacy code that communicates with the environment and does not provide an encapsulated abstraction layer. For more information, see “Unit Testing with Microsoft Moles.”

In the remainder of this section, the examples will not use any framework for mocks or stubs, not even the one that comes with Pex, but will define everything required by hand. This is done to illustrate the general mechanisms that the Moles framework uses.

Manually Creating Parameterized Mock Objects

When manually writing mock objects, some of the main questions are: What values should the mock object return? How many versions of a mock object do I need to write to test my code thoroughly?

You now know that parameterized unit tests are a way to write general tests that do not have to state particular test inputs. In a similar way, parameterized mock objects are a way to write mock objects that do not have just one particular, fixed behavior.

Consider the method **AppendFormat** of the **StringBuilder** class in the .NET base class library. Given a string with formatting instructions and a list of values to be formatted, it computes a formatted string. For example, formatting the string "Hello {0}!" with the single argument "World" yields the string "HelloWorld!"

```
public StringBuilder AppendFormat(
    IFormatProvider provider,
    string format,
    object[] args)
{
    ...
}
```

The first parameter of type **IFormatProvider** provides a mechanism for retrieving an object to control formatting:

```
public interface IFormatProvider
{
    object GetFormat(Type fmtType);
}
```

A non-trivial test calling **AppendFormat** needs an object that implements **IFormatProvider**. Although the Stubs framework that comes with Pex generates implementations for interfaces automatically, the following code illustrates how to write such an implementation manually, but leaves how it should behave to Pex:

```
public class MockFormatProvider : IFormatProvider
{
    public object GetFormat(Type fmtType)
    {
        return PexChoose.Value<object>("format");
    }
}
```

The mock method **GetFormat** obtains from the global **PexChoose** a handle called **call** that represents the current method call. **PexChoose** provides the values that define the behavior of the mocked methods such as their return values.

When you execute the test case, ChooseResult initially returns some simple value, for example, null for reference types. The Pex symbolic analysis tracks how the program uses the value obtained from ChooseResult just as Pex tracks all other test inputs. Pex checks the conditions on the value obtained from ChooseResult, and executes the test case multiple times, trying other values that are different from null.

Now you might change the code of the mock type to allow behavior that is more diverse. For example adding the choice to throw an exception, perform a callback, or change some accessible state. To do this, insert the following lines in the GetFormat method:

```
if (PexChoose.Value<bool>("throw"))
    throw new Exception();
return PexChoose.Value<object>("format");
```

Because there is a choice to throw an exception or not, Pex considers two execution paths. If the caller of GetFormat uses several catch statements to distinguish different exception types, Pex might explore even more execution paths.

As mentioned before, Pex tracks the usage of the values obtained from ChooseResult, and might execute the program with several different values. The following call to GetFormat occurs in AppendFormat after checking that provider != null:

```
cf=(ICustomFormatter)provider.GetFormat(typeof(ICustomFormatter));
```

Depending on the result of GetFormat, the cast to ICustomFormatter might fail.

Pex understands this type constraint and generates a test case with the following mock object behavior:

```
var m = new MockFormatProvider();
PexChoose.Replay.Setup()
    .DefaultSession
    .At(0, "format", m);
```

Here, Pex creates a mock object and instructs the oracle that during the execution of a unit test the first call to m.GetFormat should return the mock object itself.

The test cases that Pex generates are always minimal. This is an example of how Pex tries to use as few objects as possible to trigger a particular execution path. This particular mock object behavior causes the cast to fail, because MockFormatProvider does not implement ICustomFormatter.

Parameterized Mock Objects with Assumptions

Unconstrained mock objects might cause the code to behave in unexpected ways. Just as you state assumptions on the arguments of parameterized unit tests, you state assumptions on the results of mock object calls. For example, the author of the **IFormatProvider** interface probably had the following contract in mind:

```
public object GetFormat(Type fmtType)
{
    object result = PexChoose.Value<object>("format");
    // constraining result
    PexAssume.IsTrue(result != null);
    PexAssume.IsTrue(fmtType.IsAssignableFrom(result.GetType()));
    return result;
}
```

Using Dependency Injection to Build Mocks

Dependency Injection, also known as Inversion of Control, is a design pattern that helps to build mockable components. For more information on Dependency Injection, see the MSDN Patterns and Practices topic: Dependency Injection.

The steps in this section build on the example created in Exercise 4.

The following application illustrates dependency injection using the **Credit Card Validation Client**. This program is a client for the LuhnAlgorithm class. It has a simple graphical user interface that lets the user of the program input a credit card number, checks the input for correctness and queries the LuhnAlgorithm class to determine the numbers validity.

To write the validator—a first attempt

The dialog box uses **ShowWindow** to display up dialog box, a Number property to access to the number string value and a Status property to display the result:

```
public class CreditarDialog
{
    public bool Show()
    ...
    public string Number {get;set;}
    public string Status {get;set;}
}
```

Here is a first attempt at writing the validator:

```
public class CreditarClient
{
    public void Execute()
    {
        CreditarDialog dialog = new CreditarDialog();
        if (dialog.Show())
        {
            bool valid = LuhnAlgorithm.Validate(dialog.Number);
            if (valid)
                dialog.Status = "validated";
            else
                dialog.Status = "invalidated";
        }
    }
}
```

The implementation of the Execute method causes several testing problems. The control flow depends on the Show method of the CreditarDialog dialog box:

- The test displays a dialog box that would need complex automated tools to click the buttons and test. This means you would spend a large amount of effort to test functionality not directly related to the CreditarClient method.

```
CreditarDialog dialog = new CreditarDialog();
if (dialog.Show())
```

- A similar problem arises with the validation of the credit card number. You directly call the Validate method of the LuhnAlgorithm type:

```
bool valid = LuhnAlgorithm.Validate(dialog.Number);
```

To work around these problems, you need to add an abstract layer between the CreditarClient class and its dependencies:

- The user interface
- The validator service

To extract an interface for each dependency

- In Visual Studio, right-click in the body of LuhnAlgorithm, and click **Refactor > Extract Interface** to create an interface for this class.
- Name this interface **ICreditCardValidator**.

The following snippet provides an interface for the user interface dialog:

```
public interface ICreditarDialog
{
    bool Show();
    string Number { get; }
    string Status { get; set; }
}
```

Each dependency is injected into the CreditarClient:

```
public partial class CreditarClient
{
    ICreditarDialog dialog;
    ICreditCardValidator validator;
    public void Execute()
    {
        if (this.dialog.Show())
        {
            bool valid =
                this.validator.Validate(dialog.Number);
            if (valid)
                this.dialog.Status = "validated";
            else
                this.dialog.Status = "invalidated";
        }
    }
}
```

To inject dependencies

The last problem that remains is that you need a way to set the dialog and validator fields in the CreditarClient class. There are many ways to implement this feature. The following are common patterns:

- **Constructor Injection.** Each dependency is passed in the class constructor, thus the inversion of control name of the pattern:

```
public class CreditarClient
{
    ICreditarDialog dialog;
    ICreditCardValidator validator;
    public CreditarClient(
        ICreditarDialog dialog,
        ICreditCardValidator validator)
    {
    }
}
```

```
{
    this.dialog = dialog;
    this.validator = validator;
}
...
}
```

- **Service Locator.** The class instance is hosted in a container that is queried through a message for particular services. In .NET, this pattern is implemented in the **System.ComponentModel** namespace. A **Component** element queries for a service using the **GetService** method:

```
public class CreditarClient : Component
{
    public void Execute()
    {
        ICreditarDialog dialog =
            (ICreditarDialog) this.GetService(typeof(ICreditarDialog));
        ICreditCardValidator validator =
            (ICreditCardValidator)
            this.GetService(typeof(ICreditCardValidator));
        ...
    }
}
```

Learn more about Microsoft Pex

To continue to deepen your understanding of how Microsoft Pex works and how it can be applied in your testing practices, see:

“Parameterized Unit Testing with Microsoft Pex” and

“Advanced Concepts: Parameterized Unit Testing with Microsoft Pex”

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.

Appendix: Pex Cheat Sheet

Getting Started	
Microsoft.Pex.Framework.dll	Add Pex reference
[assembly:PexAssemblyUnderTest("UnderTest")]	Bind test project

Custom Attributes	
PexClassAttribute	Marks a class containing a parameterized unit test
PexMethodAttribute	Marks a parameterized unit test
PexAssumeNotNullAttribute	Marks a non-null parameter
PexAssumeUnderTestAttribute	Marks a non-null and precise type parameter

```
using Microsoft.Pex.Framework;
[PexClass(typeof(MyClass))]
public partial class MyClassTest {
    [PexMethod]
    public void MyMethod([PexAssumeNotNull]MyClass target, int i) {
        target.MyMethod(i);
    }
}
```

Static Helpers	
PexAssume	Evaluates assumptions (input filtering)
PexAssert	Evaluates assertions
PexObserve	Logs live values to the report and/or generated tests
PexChoose	Generates new choices (additional inputs)

```
[PexMethod]
void StaticHelpers(MyClass target) {
    PexAssume.IsNotNull(target);
    int i = PexChoose.Value<int>("i");
    string result = target.MyMethod(i);
    PexObserve.ValueForViewing("result", result);
    PexAssert.IsNotNull(result);
}
```

Instrumentation	
PexInstrumentAssemblyAttribute	Specifies to instrument an assembly
PexInstrumentTypeAttribute	Specifies to instrument a type
PexAssemblyUnderTestAttribute	Binds a test project to a project

```
[assembly:PexAssemblyUnderTest("MyAssembly")]
[assembly:PexInstrumentAssembly("Lib")]
[assembly:PexInstrumentType(typeof(MyClass))]
```

PexAssume and PexAssert

PexAssume filters the input; **PexAssert** checks the behavior. Each method can have a number of overloads.

Basic	
Fails unconditionally	Fail()
<i>c</i> is true	IsTrue(bool <i>c</i>)
<i>c</i> is false	IsFalse(bool <i>c</i>)
Treats test as inconclusive	Inconclusive()
Implication	
<i>p</i> holds if <i>c</i> holds	ImpliesIsTrue(bool <i>c</i> , Predicate <i>p</i>)
<i>p</i> holds if <i>c</i> holds (case-split)	Case(bool <i>c</i>).Implies(Predicate <i>p</i>)
Nullarity	
<i>o</i> is not null	IsNotNull(object <i>o</i>)
<i>a</i> is not null or empty	IsNotNullOrEmpty<T>(T[] <i>a</i>)
<i>a</i> elements are not null	AreElementsNotNull<T>(T[] <i>a</i>)
Equality	
<i>expected</i> is equal to <i>actual</i>	AreEqual<T>(T <i>expected</i> , T <i>actual</i>)
<i>l</i> and <i>r</i> elements are equal	AreElementsEqual<T>(T[] <i>l</i> , T[] <i>r</i>)
<i>expected</i> is not equal to <i>actual</i>	AreNotEqual<T>(T <i>expected</i> , T <i>actual</i>)
Reference Equality	
<i>expected</i> is same as <i>actual</i>	AreSame(<i>expected</i> , <i>actual</i>)
<i>expected</i> is not the same as <i>actual</i>	AreNotSame(<i>expected</i> , <i>actual</i>)
Type Equality	
<i>o</i> is an instance of <i>t</i>	IsInstanceOfType(object <i>o</i> , Type <i>t</i>)
<i>o</i> is not an instance of <i>t</i>	IsNotInstanceOfType(object <i>o</i> , Type <i>t</i>)
Over collections	
<i>p</i> holds for all elements in <i>a</i>	TrueForAll<T>(T[] <i>a</i> , Predicate<T> <i>p</i>)
<i>p</i> holds for at least one element in <i>a</i>	TrueForAny<T>(T[] <i>a</i> , Predicate<T> <i>p</i>)
Exceptional Behavior (PexAssert only)	
<i>action</i> throws an exception of type TException	Throws<TException>(Action <i>action</i>)
<i>action</i> succeeds or throws an exception of type TException	ThrowAllowed<TException>(Action <i>action</i>)
Behavior Equality (PexAssert only)	
<i>l</i> and <i>r</i> behave the same	AreBehaviorsEqual<T>(Func<T> <i>l</i> , Func<T> <i>r</i>)
returns the result or exception resulting from the execution of <i>f</i>	Catch<T>(Func<T> <i>f</i>)

PexChoose

Make **choices**, effectively adding new test parameters on the fly. Choices get serialized in the generated test code.

Value Choices	
any value of type T	<code>PexChoose.Value<T>(string description)</code>
any non-null value of type T	<code>PexChoose.ValueNotNull<T>(string description)</code>
any valid enum value of type T	<code>PexChoose.EnumValue<T>(string description)</code>
Range Choices	
any value from a	<code>PexChoose.ValueFrom<T>(string description, T[] a)</code>
any integer value within a (min; max)	<code>PexChoose.ValueFromRange(string description, int min, int max)</code>
any index for a	<code>PexChoose.IndexValue<T>(string description, T[] a)</code>