

---

# Microsoft Pex Reference Manual

Version 0.93 – August 3, 2010

## Abstract

---

**Microsoft® Pex 2010** is a Visual Studio add-in that provides runtime code analysis for .NET code. With just a few mouse clicks, you can:

- Explore code-under-test to understand input/output behavior in the code.
- Save automated tests that increase your confidence that your code cannot crash—or point to errors that do cause crashes.
- Write powerful parameterized unit tests and generate suites of tests that ensure the code-under-test behaves as intended.

Microsoft Pex also contains the **Microsoft® Moles 2010** framework. With the Moles framework, you can replace any method that the code-under-test calls with a test implementation that uses a delegate. For documentation on the Microsoft Moles Framework, see the Moles Reference Manual.

**This manual is Technical Level 200 through 400.**

### Note:

- 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 .....	3
Running Example .....	3
Authoring Parameterized Unit Tests .....	3
PexMethodAttribute .....	3
Test Method Parameters .....	3
Test Classes .....	4
Generated Unit Tests .....	4
PexClassAttribute .....	5
PexAllowedExceptionAttribute .....	5
PexAllowedExceptionFromTypeUnderTestAttribute .....	5
PexAssert .....	6
PexAssume .....	6
PexAssumeNotNullAttribute .....	6
4A – Assume, Arrange, Act, Assert .....	7
PexInstrumentAssemblyAttribute .....	7
PexAssemblyUnderTestAttribute .....	8
PexObserve .....	8
PexFactoryMethodAttribute .....	8
PexPreparationMethodAttribute .....	9
Development Lifecycle .....	9
Explore, Check in and Execute .....	9
Continuous Integration + Microsoft Pex = Continuous Exploration .....	9
Generated Test Maintenance .....	10
Coding Recommendations .....	10
Resources and References .....	11

**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, Visual Studio, 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

Given a method, the Microsoft Pex generates inputs which exercise many different code paths. In other words, Microsoft Pex aims at generating a test suite that achieves maximum code coverage.

You can launch Microsoft Pex on any method in your projects or write specialized parameterized unit tests as part of your unit test projects. This document focuses on writing parameterized unit tests.

All Microsoft Pex attributes and helpers are contained in the **Microsoft.Pex.Framework.dll** assembly. The attributes are located under the `Microsoft.Pex.Framework` namespace or a subnamespace.

### Running Example

Throughout the Microsoft Pex reference manual, we will use a method that takes a string and capitalizes it, i.e. makes the first character upper case and leaves the rest unchanged.

```
public static class StringExtensions
{
    // capitalizes value and returns it
    public static string Capitalize(string value)
    { ... }
}
```

More advanced scenarios will also be added in later chapters.

## Authoring Parameterized Unit Tests

### PexMethodAttribute

In unit testing mode, Pex starts the code exploration from any method marked with the **PexMethodAttribute** attribute. The method should be visible, non-abstract and not a constructor. Static and instance methods may be used. The method may have any number of parameters.

```
using Microsoft.Pex.Framework;

[PexMethod]
public static void CapitalizedFirstCharIsUpper(string value)
{
    string result = StringExtensions.Capitalize(value);
}
```

### Test Method Parameters

Parameterized unit tests may have any number of parameters. Microsoft Pex can easily generate values of primitive types (int, short, long, string, char, etc...) but it can also generate instances of custom types with or without help from the user. If you need a collection of elements rather than a single instance, always prefer arrays, which are more efficient than other collection types for the Microsoft Pex exploration.

```
public class User {...}
```

```
[PexMethod]
public static void OtherTest(char c, string[] values)
{ ... }
```

Parameterized unit tests can also return values or have out or ref parameters. Those values will automatically be displayed in the **Exploration Results View** and asserted in the generated unit test.

```
[PexMethod]
public static int OtherTest(...)
{ ... }
```

## Test Classes

The parameterized methods are usually located in a type marked as a test class using your unit test framework notations. For example, with the Visual Studio Unit Test framework, the class should be marked with the `TestClassAttribute` attribute.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

[TestClass]
public partial class StringExtensionsTest
{
    [PexMethod]
    public static void CapitalizedFirstCharIsUpper (string value)
    { ... }
}
```

## Generated Unit Tests

When Microsoft Pex discovers new interesting inputs, it saves them as source code, as a traditional unit test. Microsoft Pex uses the notation of your unit test framework so that the generated unit test can be executed by your unit test framework afterwards. With the Visual Studio Unit Test framework, the generated unit tests are marked with the `TestMethodAttribute`. Microsoft Pex also adds the `PexGeneratedByAttribute` attribute that specifies that this unit test was automatically generated by Microsoft Pex.

```
public partial class StringExtensionsTest
{
    [TestMethod]
    [PexGeneratedBy(typeof(StringExtensionsTest))]
    public static void CapitalizedFirstCharIsUpper001()
    {
        this.CapitalizedFirstCharIsUpper("\0");
    }
}
```

The generated unit tests are placed in a nested file under the file that holds the parameterized unit test. The nested file has a `.g` in the file name to denote that it contains automatically generated code and should not be edited.

---

**CAUTION: Do not edit the generated unit test methods.** Microsoft Pex is designed to maintain a minimal suite of generated unit tests. To do so, it may decide to delete any unit test marked with the `PexGeneratedByAttribute` attribute without notice.

---

## PexClassAttribute

The test class may be optionally marked with the `PexClassAttribute`. This attribute allows you to specify the **Type Under Test**, the type which the parameterized unit tests are supposed to be testing. Microsoft Pex uses this information as a hint to tune its exploration strategies and also to better decide when to emit a test case.

```
[TestClass]
[PexClass(typeof(StringExtensions))]
public partial class StringExtensionsTest
{ ... }
```

## PexAllowedExceptionAttribute

The exploration of a parameterized unit test may raise a number of different exceptions in the code under test. Some of those exceptions such as parameter validation exceptions may be expected and a negative test case should be emitted. With the Visual Studio Unit Test framework, the `ExpectedExceptionAttribute` is added to the generated unit test. The `PexAllowedExceptionAttribute` attribute may be used to specify that a particular exception type is allowed.

```
[PexMethod]
[PexAllowedException(typeof(ArgumentNullException))]
public static void CapitalizedFirstCharIsUpper(string value)
{ ... }

[TestMethod]
[PexGeneratedBy(typeof(StringExtensionsTest))]
[ExpectedException(typeof(ArgumentNullException))]
public static void CapitalizedFirstCharIsUpper000()
{
    this.CapitalizedFirstCharIsUpper(null);
}
```

---

**Tip: Allow Exceptions from Visual Studio.** The Pex Exploration Results View provides an easy way to add `PexAllowedExceptionAttribute` attributes. Select the failing test case in the results table, and click on the **Allow Exception** button on the lower right.

---

## PexAllowedExceptionFromTypeUnderTestAttribute

The `PexAllowedExceptionAttribute` attribute filters exceptions based on the type only. The `PexAllowedExceptionFromTypeUnderTestAttribute` attribute provides a more precise filter: it also requires that the exception be thrown from a method of the type under test (see `PexClassAttribute`). The rationale is that a component should enforce pre-conditions at its abstraction level. For example, the `StringExtensions.Capitalize` method should validate that the input string is not null, instead of relying on the low-level `String` methods to detect such errors.

```
[PexMethod]
```

```
[PexAllowedExceptionUnderTest(typeof(ArgumentNullException))]
public static void CapitalizedFirstCharIsUpper(string value)
{ ... }
```

---

**Tip: Allow exceptions at the type level.** An exception can be allowed for all parameterized unit test methods in a test class by marking the test class itself with a `PexAllowException` attribute. This is generally true for most attributes in Pex.

---

## PexAssert

The `PexAssert` type contains static methods to assert properties about the program. You should write an assertion, whenever you can think of a particular property that should hold at a particular program point. You may also use the assertion types provided by your unit test framework. In the following example, we assert that the first character of a capitalized string is upper case.

```
[PexMethod]
public static void CapitalizedFirstCharIsUpper(string value)
{
    string capitalized = StringExtensions.Capitalize(value);
    PexAssert.IsTrue(char.IsUpper(capitalized[0]));
}
```

---

**Tip:** `PexAssert` contains most of the assertion methods you would expect, such as `IsTrue`, `AreEqual`, etc... `PexAssert` also contains specialized assertion methods that are especially useful in the context of parameterized unit testing. Additionally, the implementations of the `PexAssert` methods are optimized for the Microsoft Pex exploration.

---

## PexAssume

Assumptions allow you to filter out irrelevant test inputs. When an input violates an assumption, Microsoft Pex silently discards the input.

An assumption is similar to an assertion, but also very different. It is similar, because it also checks at a particular program point whether a condition holds. It is different, because it treats cases where the check fails not as a test failure, but instead as a hint to the Microsoft Pex exploration that those cases don't matter.

The `PexAssume` type contains helper methods to specify assumptions. For example, you can use the `PexAssume.IsNotNull` method to specify that the input of the parameterized unit test method should not be null.

```
[PexMethod]
public static void CapitalizedFirstCharIsUpper(string value)
{
    PexAssume.IsNotNull(value);
    ...
}
```

## PexAssumeNotNullAttribute

Assuming that a test parameter is not null is such a common scenario that a specialized attribute, `PexAssumeNotNullAttribute`, can be used instead of a call to `PexAssume.IsNotNull`.

```
[PexMethod]
public static void CapitalizedFirstCharIsUpper(
    [PexAssumeNotNull] string value)
{ ... }
```

A more specialized version of this attribute is the `PexAssumeUnderTestAttribute` attribute. It specifies that an argument should not be null and should be precisely of the parameter type, e.g. not a subtype.

## 4A – Assume, Arrange, Act, Assert

Traditional unit tests are often written using the so-called “3A pattern”. The 3A pattern separates the unit test in three parts, **Arrange**, **Act** and **Assert**. The Arrange part prepares the data, the Act part exercises the code under test and the Assert part contains assertions.

```
[TestMethod]
public static void CapitalizedFirstCharIsUpperC()
{
    // Arrange
    string value = "c";
    // Act
    string result = StringExtensions.Capitalize(value);
    // Assert
    PexAssert.AreEqual('C', result[0]);
}
```

Parameterized unit tests extend this paradigm by adding the **Assume** part where assumptions about the input data are made. This pattern is simply called the “4A pattern”, **Assume**, **Arrange**, **Act** and **Assert**.

```
[PexMethod]
public static void CapitalizedFirstCharIsUpper(string value)
{
    // Assume
    PexAssume.IsTrue(value != null && value.Length > 0);
    // Arrange
    string firstCharUpper = char.ToUpper(value[0]);
    // Act
    string result = StringExtensions.Capitalize(value);
    // Assert
    PexAssert.AreEqual(firstCharUpper, result[0]);
}
```

## PexInstrumentAssemblyAttribute

In order to track the data flow and control flow of the program, Microsoft Pex injects several callbacks in the .NET method bodies at runtime. Because this instrumentation has significant performance implications, you have to specify which code needs to be

instrumented. The `PexInstrumentAssemblyAttribute` attribute specifies that an assembly should be instrumented by Microsoft Pex. Any number of this attribute may be added to your test project.

```
using Microsoft.Pex.Framework.Instrumentation;

// System.Extensions contains StringExtensions
[assembly: PexInstrumentAssembly("System.Extensions")]
```

## PexAssemblyUnderTestAttribute

Similarly to the `Type Under Test` in the `PexClassAttribute` attribute, the `PexAssemblyUnderTestAttribute` attribute can be added to a test project to specify which project is being tested. Microsoft Pex uses this hint in various ways to enhance its exploration heuristics, improve the decision on when to emit a test case and infer where to save the generated unit tests. This attribute also implicitly specifies that the assembly under test should be instrumented.

```
using Microsoft.Pex.Framework.Instrumentation;

// this test project is for System.Extensions
[assembly: PexAssemblyUnderTest("System.Extensions")]
```

## PexObserve

The `PexObserve` type can be used to observe intermediate values. It behaves similarly to returning values but may be used anywhere in the test method.

```
[PexMethod]
public static void OtherTest(...)
{
    int result = ...;
    PexObserve.Value<int>("result", result);
}
```

## PexFactoryMethodAttribute

When a parameterized unit test takes a custom type as a test parameter, Pex faces the problem of creating instances of that type, and bringing such instances into interesting states. Pex has a number of heuristics to generate a factory that can create instances of the type but in some cases, it is not good enough and the user needs to update it.

Object factory methods are static methods marked with the `PexFactoryMethod` in a static type as shown in the following:

```
// code under test
public class Node
{
    public int Value { get; set; }
}
// in test project
public static class NodeFactory
{
    [PexFactoryMethod(typeof(Node))]
    public static Node Create(int value)
    {
        var node = new Node
```

```

        node.Value = value;
        return value;
    }
}

```

When exploring the code, Pex will explore the `NodeFactory.Create` method to instantiate the `Node` type.

Object factories are automatically mined from the test assembly. It is possible to specify additional assemblies that contains object factories by using `PexExplorableFromFactoriesFromAssemblyAttribute` attribute as follows

```

using Microsoft.Pex.Framework.Explorables;
[assembly: PexExplorableFromFactoriesFromAssembly("MyFactories")]

```

## PexPreparationMethodAttribute

Pex allows you to register methods to be called before any member of a given type is executed. Those methods are called ‘preparation methods’ as they are usually used to prepare (or isolate) the environment that the type requires. You can see preparation methods as lazy, on demand setup methods. Preparation methods are static parameterized methods marked with the `PexPreparationMethodAttribute` located in a static type. The following example shows a preparation method that ensures that `DateTime.Now` does not rely on the system time; this example uses Moles to replace the implementation of `DateTime.Now`.

```

using System.Moles;
public static class DateTimePreparation
{
    [PexPreparationMethod(typeof(DateTime))]
    public static void Prepare(DateTime now)
    {
        MDateTime.NowGet = () => now;
    }
}

```

Multiple preparation methods maybe registered for the same type. They will all be executed, but their ordering is not guaranteed.

## Development Lifecycle

### Explore, Check in and Execute

A common use case scenario for Microsoft Pex is the following: a developer writes a parameterized unit tests and explores it with Microsoft Pex. Microsoft Pex generates a test suite of unit tests. The generated unit tests are checked into the source control as source and integrated as part of the regression suite. Since Microsoft Pex generates unit tests that are executable by traditional test frameworks, the generated unit tests will automatically integrate in any existing continuous integration process.

One of the benefits of this approach is that it does not force other team members to execute the generated unit tests as long as the Microsoft Pex public assemblies are stored along in the source control system.

## Continuous Integration + Microsoft Pex = Continuous Exploration

The Microsoft Pex command line can be used to automate the exploration of parameterized unit tests, or even the creation of such unit tests in any continuous integration environment. By exploring your code with Microsoft Pex in your continuous integration, you will leverage the full power of automated test generation.

### Exploring Parameterized Unit Tests

The following command line explores the parameterized unit tests from an assembly **MyApp.Tests.dll**.

```
pex.exe MyApp.Tests.dll /nor /ftf
```

At the end of the execution, Microsoft Pex generates a full XML report, stored in a file called `report.per`, and an HTML report of the run. The `.per` report file can be imported in Visual Studio through the Pex Exploration Results View. Microsoft Pex also stores all the sources of the generated unit tests in files.

There are many command line options you can use to configure Microsoft Pex. In an automated continuous integration setting, the options `/nor` and `/ftf` are especially useful. They indicate that Pex should not automatically open the generated HTML report in your default browser, and that Microsoft Pex should indicate via its exit code whether any failing unit test was generated.

```
pex.exe MyApp.Tests.dll /nor /ftf
```

### Exploring APIs without Parameterized Unit Tests

The following command line explores any visible API from the **MyApp.dll** assembly,

```
pex.exe MyApp.dll /nor /ftf /erm:wizard
```

Similarly to the execution of parameterized unit tests, Microsoft Pex generated a full XML and HTML report. It also generates a fully compilable test project that contains the generated parameterized unit tests and generated unit tests.

### Exploring APIs with Parameterized Unit Tests

It is possible to explore a mix of existing (hand-written) parameterized unit tests and (boilerplate) automatically generated parameterized unit test stubs. Microsoft Pex will go through the API under test and look if any parameterized unit tests already exist for each visible API. If a parameterized unit test exists, it will execute it; otherwise Microsoft Pex will generate a new parameterized unit test stub and explore it as well. The key to enable this feature is to specify the assembly that contains the parameterized unit tests as a *settings* assembly:

```
pex.exe MyApp.dll /nor /ftf /erm:wizard /sa:MyApp.Tests.dll
```

## Coding Recommendations

The following section covers various tips and tricks to ensure that your code gets explored efficiently by Pex. These are general guidelines that you can try to apply to your development habits.

## Write efficient code

Pex analyzes every MSIL instruction executed on the .NET virtual machine. Therefore, writing efficient code is critical to enable efficient exploration. For example, be aware of the hidden complexity cost of XML parsing, regular expression matching, etc... as they often involve the execution of thousands or even millions of instructions.

## Write assertions

Pex will find interesting bugs if you encode correctness as assertions, in the product code or the test code. If your code contains no assertion, Pex will only find runtime errors such as null references, index out of range, etc... Code Contracts provide an elegant way in .NET 4.0 to specify assertions as well.

## Refactor to functional helper with simple types

Pex is very efficient analyzing primitive types, e.g. integer, long, string, and arrays thereof. Whenever code contains complicated branching logic, you can try to refactor that code in such a way that it becomes a static pure method taking primitive types as input. This creates a great starting point to let Pex explore that method.

## Stay close to the code under test

Try starting Pex as close as possible to the code under test. Starting from the Main of your application usually leads to extremely poor results. Use the ability to execute private methods with Pex to test private helper methods directly.

## Distinguish preconditions from assertions

Use different exception to distinguish a method precondition from an internal assertion violation. This is critical to efficiently filter exceptions during the Pex exploration. Code Contracts provide a great way to achieve this goal.

## Test Isolated Code

Write isolated code or use Moles to isolate your code from environment dependencies. Pex cannot reason about the file system, database, web services or any other environment facing API.

## Use the ASCII encoding for Testing

Whenever a test input takes a stream, prefer the ASCII encoding. Other encodings such as UTF-8 translates into hundreds of range constraint that put a large burden on the Pex analysis.

## Avoid floating point, decimal values

Whenever possible, avoid using floating point or decimal values. They introduce rounding problems, which are difficult to understand for humans as well as the symbolic analysis engine of Microsoft Pex.

## Appendix: Pex Cheat Sheet

### Getting Started

<b>Microsoft.Pex.Framework.dll</b>	Add Pex reference
<b>[assembly:PexAssemblyUnderTest("UnderTest")]</b>	Bind test project

### Custom Attributes

<b>PexClassAttribute</b>	Marks a class containing a parameterized unit test
<b>PexMethodAttribute</b>	Marks a parameterized unit test
<b>PexAssumeNotNullAttribute</b>	Marks a non-null parameter
<b>PexAssumeUnderTestAttribute</b>	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

<b>PexAssume</b>	Evaluates assumptions (input filtering)
<b>PexAssert</b>	Evaluates assertions
<b>PexObserve</b>	Logs live values to the report and/or generated tests
<b>PexChoose</b>	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

<b>PexInstrumentAssemblyAttribute</b>	Specifies to instrument an assembly
<b>PexInstrumentTypeAttribute</b>	Specifies to instrument a type
<b>PexAssemblyUnderTestAttribute</b>	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.

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

Behavior Equality (PexAssert only)	
returns the result or exception resulting from the execution of f	Catch<T> (Func<T> f)

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

## 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 <a href="#">(II)</a>	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

## References