Parameterized Test Patterns for Microsoft Pex

Version 0.90 - March 2, 2010

Abstract

A parameterized unit test is the straightforward generalization of a unit test by allowing parameters. Parameterized unit tests make statements about the code's behavior for an entire set of possible input values, instead of just a single exemplary input value.

This document contains common patterns for parameterized unit tests. The patterns are written with automatic test input generation tools in mind, such as Microsoft Pex, which can determine all—or at least many—relevant test inputs that trigger the possible cases of the parameterized unit test and the code-under-test.

This guide is Technical Level 400. To take advantage of this content, you should be experienced with the concepts and capabilities discussed in these documents:

"Exploring Code with Microsoft Pex"
"Parameterized Unit Testing 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
- If you have a question, post it on the Pex forum.

Contents

Introduction to Parameterized Test Patterns

A unit test is a method without parameters that represents a test case that typically executes a method of a class-under-test with fixed arguments and verifies that it returns the expected result.

A parameterized unit test is the straightforward generalization of a unit test by allowing parameters. Parameterized unit tests make statements about the code's behavior for an entire set of possible input values, instead of just a single exemplary input value.

This document contains common patterns for parameterized unit tests. The patterns are written with automatic test input generation tools in mind, such as Microsoft Pex, which can determine all—or at least many—relevant test inputs that trigger the possible cases of the parameterized unit test and the code-under-test.

Writing good parameterized unit tests can be quite challenging. There are two core questions:

- **Coverage:** *What are good scenarios (sequences of method calls) to exercise the code-under-test?*
- **Verification:** *What are good assertions that can be stated easily without re-implementing the algorithm?*

A parameterized unit test is only useful if it provides answers for both questions:

- Without sufficient coverage—that is, if the scenario is too narrow to reach all the code-under-test, the extent of the parameterized unit test is limited.
- Without sufficient verification of computed results—that is, if the parameterized unit test does not contain enough assertions, the parameterized unit test does not check that the code is doing the right thing. All the parameterized unit test would check for is that the code-under-test does not crash.

Note: In traditional unit testing, the set of questions is slightly different. There is one more question: *What are relevant test inputs?* With parameterized unit tests, this question is taken care of by a tool such as Pex. However, the problem of finding good assertions is easier in traditional unit testing: the assertions tend to be simpler, because they are written for particular test inputs.

In this document:

Test Patterns

This section contains test patterns that illustrate how to write good parameterized unit tests.

Note: In support of early-adopters, Test Patterns retain their numbers from earlier versions of this document.

Arrange, Act, Assert

The ’AAA’ (Triple-A) is a well-known pattern for authoring unit tests. This pattern applies to parameterized unit tests as well.

Pattern 2.1

A (parameterized) unit test that is organized in three sections:

- **Arrange:** Set up the unit under test.
- **Act:** Exercise the unit under test, capturing any resulting state.
- **Assert:** Verify the behavior through assertions.

An example of this pattern in a traditional unit test:

```
[TestMethod]
void AddItem() {
    // arrange
    var list = new ArrayList();
    var item = new object();

    // act
    list.Add(item);

    // assert
    Assert.IsTrue(list.Count == 1);
}
```

The unit test is an instance method, annotated with the **TestMethodAttribute**.

If this method is placed in a class annotated with the **TestClassAttribute**, then the Visual Studio Unit Test unit test framework will be aware of the test case and include it during test execution.

An example of this pattern in a parameterized unit test:

```
[PexMethod]
void AddItem(object item) {
    // arrange
    var list = new ArrayList();

    // act
    list.Add(item);

    // assert
    Assert.IsTrue(list.Count == 1);
}
```

Note: The unit test is an instance method, annotated with the **PexMethodAttribute**. If this method is placed in a class annotated with the **PexClassAttribute**, then the Pex tool will be aware of the parameterized test case, and it will allow the exploration of this parameterized test case to determine relevant test inputs. Each test input will be persistent as a traditional unit test.

For example, the following two traditional unit test cases might be generated for the parameterized unit test **AddItem**. Each unit test fixes particular inputs and then calls the parameterized unit test.

```
[TestMethod]
void AddItem1() { AddItem(null); }

[TestMethod]
void AddItem2() { AddItem(new object()); }
```

Assume, Arrange, Act, Assert

This pattern is an extension of Arrange, Act, Assert as described in the previous section, with an Assumption section added at the beginning. An assumption restricts possible test inputs, acting as a filter.

Pattern 2.2

A parameterized unit test that is organized in four sections:

- **Assume:** Assume preconditions over the test inputs.
- **Arrange:** Set up the unit under test.
- **Act:** Exercise the unit under test, capturing any resulting state.
- **Assert:** Verify the behavior through assertions.

The following example tests that adding an element to any list increments the **Count** property. We use an assumption to filter out the case where **list** is a null reference.

```
[PexMethod]
void AssumeActAssert(ArrayList list, object item) {
    // assume
    PexAssume.IsNotNull(list);

    // arrange
    var count = list.Count;

    // act
    list.Add(item);

    // assert
    Assert.IsTrue(list.Count == count+1);
}
```

In practice, the distinction between the four sections might be less pronounced. For example, assumptions could be imposed on arranged values before acting.

Constructor Test

This pattern relates the arguments passed to a constructor to the constructed instance and its properties.

Pattern 2.3

A pattern to test instance constructors with three sections:

- **Create:** Initializes a new instance from parameters.
- **Assert Invariant:** Asserts the test invariant (Pattern 4.3) and the class invariant (Pattern 4.2), if available.
- **Assert:** Relates observable instance properties and the constructor parameters.

```
[PexMethod]
void Constructor(int capacity) {
    // create
    var list = new ArrayList(capacity);

    // assert invariant
    AssertInvariant(list);

    // assert
    Assert.AreEqual(capacity, list.Capacity);
}
```

The Pattern 2.10 can be used to deal gracefully with argument validation exceptions of the constructor. In this example, the parameterized unit test fails when the constructor rejects particular arguments.

Roundtrip

Pattern 2.4

This pattern applies to an API that transforms its inputs in a reversible way: When the API has a function **f** and an inverse function f_**1**, then it should hold that **f_1(f(x))=x** for all **x**.

A classic example of such pattern is property setters and getters:

```
[PexMethod]
void PropertyRoundtrip(string value) {
    // arrange
    var target = new MyClass();

    // two-way roundtrip
    target.Name = value; // calls setter
    var roundtripped = target.Name; // calls getter

    // assert
    Assert.AreEqual(value, roundtripped);
}
```

The Pattern 2.10 can be used to gracefully deal with cases where some values are rejected by the property setter. In this example, the parameterized unit test fails when the setter rejects particular arguments.

Another example is serialization and deserialization of values:

```
[PexMethod]
void ToStringParseRoundtrip(int value) {
    // two-way roundtrip
    string s = value.ToString();
    int parsed = int.Parse(s);

    // assert
    Assert.AreEqual(value, parsed);
}
```

Normalized Roundtrip

The Pattern 2.4 showed how to test a method for which in inverse operation exists. For example, **int.Parse** is the inverse of **int.ToString**. This is not the case in the other direction: **int.ToString** is not exactly the inverse of **int.Parse**, because the parsing ignores whitespace:

```
int.Parse(" 5").ToString()  == "5"
```

The following pattern is a variation of the previous pattern, where testing starts from non-normalized data.

Pattern 2.5

This pattern applies to an API that transforms its inputs in a reversible way, performing some kind of data normalization. When the API has a function **f** and an inverse function **f_1**, then it should hold that **f_1(f(f_1(x)))=f_1(x)** for all **x** where **f_1(x)** is defined.

Examples:

- **Parse**, **ToString**,
- **Streams**
- Escaping A

```
[PexMethod]
void ThreeWayRoundtrip(string value) {
    // 'hello%20world' <= 'hello world'
    var normalized  =  Uri.EscapeDataString(value);

    // 'helloworld' <= 'hello%20world'
    var intermediate = Uri.UnescapeDataString(normalized);

    // 'hello%20world' <= 'hello world'
    var roundtripped  = Uri.EscapeDataString(intermediate);

    // assert
    Assert.AreEqual(normalized, roundtripped);
}
```

State Relation

Pattern 2.6

This pattern applies when an API call causes an internal state change that can be (partially) observed through other API calls. A classic example of such a pattern is the combination of **Insert** and **Contains** operation on any collection type:

```
[PexMethod]
void InsertContains(stringvalue) {
    var list = new List<string>();
    list.Add(value);
    Assert.IsTrue(list.Contains(value));
}
```

Many different API combinations fall into this pattern. For example, **Remove** and **Contains** can also be tested in a similar fashion:

```
[PexMethod]
void RemoveNotContains(List<string> list, string value) {
    // remove value from list (if present)
    list.Remove(value);
    // assert that list does not contain value anymore
    Assert.IsFalse(list.Contains(value));
}
```

Same Observable Behavior

Pattern 2.7

Given two methods **f(x)** and **g(x)**, and a method **b(y)** that observes the result or the exception behavior of a method, assert that **f(x)** and **g(x)** have the same observable behavior under **b**—that is, **b(f(x))=b(g(x))** for all **x**.

Let us assume that a developer wrote a method to format two strings as a pair, separated by a comma. His first implementation used **String.Format**. Then, the developer realized that it was more efficient to call the **String.Concat** method directly.

As a result, we now have two methods that should compute the same result: **Concat** and **ConcatNew**.

```
static class StringFormatter {
    public static string Concat(string left, string right) {
        return String.Format("{0},{1}", left, right);
    }
    public static string ConcatNew(string left, string right) {
        return string.Concat(left,",",right);
    }
}
```

Changing the API used to implement a method might introduce new behavior, such as a different kind of exception being thrown in exceptional cases. To ensure that both methods have the same observable behavior, one can write the following test:

```
[PexMethod]
public void ConcatsBehaveTheSame(string left, string right) {
    PexAssert.AreBehaviorsEqual(
         () =>StringFormatter.Concat(left, right),
         () =>StringFormatter.ConcatNew(left, right)
        );
}
```

PexAssert.AreBehaviorEqual checks that both delegates have the same behavior—that is, they return equal values or throw exceptions of equal types. It is equivalent to the following longer parameterized unit test:

```
[PexMethod]
public void ConcatsBehaveTheSame(string left, string right) {
    string returnValue1 = null, returnValue2 = null;

    Type exceptionType1 = null, exceptionType2 = null;
    try {
    returnValue1 = StringFormatter.Concat(left, right);
    }
    catch (Exception ex) {
        exceptionType1 = ex.GetType();
    }
    try {
        returnValue2 = StringFormatter.ConcatNew(left, right);
    }
    catch (Exception ex) {
        exceptionType2 = ex.GetType();
    }
    PexAssert.AreEqual(returnValue1, returnValue2);
    PexAssert.AreEqual(exceptionType1, exceptionType2);
}
```

Commutative Diagram

Pattern 2.8

This pattern applies when there are two implementations of the same algorithm, each possible requiring a different number of steps. More formally:

- If one way to compute a value is $f_1\left(f_2\left(\dots\left(f_n(x)\right)\right)\right)$
- And another is $g_1\left(g_2\left(\dots\left(g_m(x)\right)\right)\right)$

Then it should hold that, for all x,

$$f_1\left(f_2\left(\dots\left(f_n(x)\right)\right)\right) = g_1\left(g_2\left(\dots\left(g_m(x)\right)\right)\right).$$

Consider two methods that perform multiplication: one that is implemented over integers, and one that takes strings which represent integers:

```
string Multiply(string x, string y);
int Multiply(int x, int y);
```

In either case, we should get the same result, if we perform the appropriate conversions:

```
[PexMethod]
void CommutativeDiagram1(int x, int y)
{
    // compute result in one way
    string z1 = Multiply(x, y).ToString();
    // compute result in another way
    string z2 = Multiply(x.ToString(), y.ToString());
    // assert equality if we get here
    PexAssert.AreEqual(z1, z2);
}
```

We should also get the same result when we apply the conversion the other way around:

```
[PexMethod]
void CommutativeDiagram2(string x, string y)
{
    // compute result in one way
    int z1 = Multiply(int.Parse(x), int.Parse(y));
    // compute result in another way
    int z2 = int.Parse(Multiply(x, y));
    // assert equality if we get here
    PexAssert.AreEqual(z1, z2);
}
```

Note that most patterns that relate test inputs and program outputs can be characterized as commutative diagrams of some form.

Cases

Pattern 2.9

Split possible outcomes into several cases, each consisting of a precondition and an implied post-condition. For more complex relationships, Pex provides a way to build a decision rule table.

For example, let us consider the following business rules for salaries in a fictitious company:

Age	Job title	Salary
Younger than 30	Manager	Less than 1000 peanuts
Older than 35	Technician	Greater than 15000 peanuts
Younger than 20	Manager	Too young to be manager

This table can be translated into executable code using **PexAssert.Case**:

```
[PexMethod]
void BusinessRules(int age, Job job) {
    var salary = SalaryManager.ComputeSalary(age, job);
    PexAssert
        .Case(age < 30)
            .Implies(() =>salary < 10000)
        .Case(job == Job.Manager && age > 35)
            .Implies(() =>salary > 10000)
        .Case(job == Job.Technician && age > 45)
            .Implies(() =>salary > 15000)
        .Case(job == Job.Manager && age < 20)
            .Implies(() => false);

}
```

Allowed Exception

In parameterized unit testing, different arguments might trigger exceptional behavior, which can sometimes be tolerated.

Pattern 2.10

An allowed exception for a parameterized unit test describes an exception that might occur during the execution. When such exception is raised, the resulting generated test case does not fail, but is tagged with an expected-exception annotation.

Traditional unit test frameworks support the concept of expected exception, where a test case or API call is expected to throw an exception. If the test does not throw the exception or throws an exception that does not match the criteria, the execution fails.

The allowed-exception concept extends this concept in the context of parameterized unit testing, where a single test yields many different behaviors.

```
[PexMethod]
[PexAllowedException(typeof(ArgumentNullException))]
void Constructor(string value) {
    // throws ArgumentNullException if value is null
    var myClass = new MyClass(value);
}
// generated expected exception test
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
void Constructor01() {
    this.Constructor(null);
}
```

Reachability

Sometimes, test assumptions can be so complicated that it is not clear whether there is any test input that fulfills them. You should make sure that there is at least one.

Pattern 2.11

Indicate which portions of a parameterized unit test should be reachable.

A parameterized unit test fails when Pex does not find a way to reach a goal, indicated by calling the **PexAssert.Reached** method in a parameterized unit test annotated with the **PexAssertReachEventuallyAttribute**.

```
[PexMethod]
[PexAssertReachEventually("passed", StopWhenAllReached = true)]
public void ParsingSuccesful(string input)
{
    // complicated parsing code
    DateTime date;
    if (DateTime.TryParse(input, out date))
    {
        // and we want to see at least one case
        // where parsing is successful.
        PexAssert.ReachEventually("passed");
    }
}
```

Multiple goals can be combined in a single parameterized unit test for more advanced scenarios by passing a list of goal identifiers in the constructor of the **PexAssertReachEventuallyAttribute**.

Each goal identifier must be reached and notified in order for the parameterized unit test to succeed.

```
[PexMethod]
[PexAssertReachEventually("passed", "y2k", StopWhenAllReached = true)]
public void ParsingSuccesfulWithMoreGoals(string input)
{
    // complicated parsing code
    DateTime date;
    if (DateTime.TryParse(input, out date))
    {
        // and we want to see at least one case
        // where parsing is successful.
        PexAssert.ReachEventually("passed");
    }

    if (date.Year == 2000)
        // we want to see at least one test
        // with the year 2000
        PexAssert.ReachEventually("y2k");
}
```

Seed Values for Fuzzing and to Help Pex

A parameterized unit test needs concrete input data to be executed. Although Pex can often automatically generate relevant input data via code analysis, it might sometimes be desirable or even necessary to provide manually chosen seed values to Pex to guide the automated code exploration. In effect, Pex will fuzz the provided values in ways that cause alternative execution paths to be taken.

Pattern 2.12

Providing seed values for Parameterized Unit Tests. Most traditional unit test frameworks now support parameterized unit tests where the user binds a data source to the method parameters. The data source might be a text or XML file, a database query, or primitive values encoded in attributes.

The test framework reads the data source, deserializes data structures if necessary, and feeds tuples to the parameterized unit test. Pex has a similar mechanism. The **PexArgumentsAttribute** can be used to provide primitive data. Each instance of this attribute gives a list of values that must match the parameter types of the parameterized unit test:

```
[PexMethod]
[PexArguments("var i = 0;", 0)]
[PexArguments("class MyClass {}", 12)]
public void ParseTest(string text, int line)
{
    var parser = new Parser();
    parser.SetLine(line);
    var node = parser.Parse(text);
    ...
}
```

Before analyzing the branch conditions in the code, Pex will first execute the parameterized unit test with the provided values. This way, Pex acquires knowledge about the code reachable from a parameterized unit test, and during the subsequence code exploration Pex will try to further increase code coverage by slightly modifying the values to trigger different execution paths. In effect, Pex will fuzz the provided values. Input data can also be encoded as unit tests that simply call the parameterized unit test.

Having the following methods in the test class is equivalent to attaching instances of the **PexArgumentsAttribute** to the parameterized unit test:

```
[TestMethod]
public void ParseTextVar() {
    var text = "var i = 0;";
    this.ParseTest(text, 0);
}
[TestMethod]
public void ParseTextClass() {
    var text = "class MyClass{}";
    this.ParseTest(text, 12);
}
```

Pex will pick up values from handwritten unit tests, and Pex will also pick up values from previously generated unit tests that were stored in the test class as code. As a result, Pex will not start from scratch when exploring the same parameterized unit test again later, but continue from where it stopped earlier.

State Machine

Pattern 2.13

This pattern applies for stateful component **x** that expose members that might transition the state.

For each member **f(x)**, one defines a transition type **Tf(x;o)** that contains a method to invoke **f(x)** and where **o** is the test oracle.

Let us illustrate this pattern with the **XmlWriter** class from the **System.Xml** library. This class contains a number of methods—**Write...**—that have to be called in a particular order to build valid XML documents. The writer also exposes a **WriteState** property that gives a partial view on the state of the writer:

```
public abstract class XmlWriter
{
    public abstract void WriteStartElement(string elementName);
    public abstract void WriteEndElement();
    public abstract WriteState WriteState{get;}
    ...
}
```

We start by defining an abstract **Transition** type that will be used to define each possible transition of the **XmlWriter**:

```
public abstract class Transition
{
    public abstract void Execute(

        XmlWriter writer,
        XmlWriterOracle oracle);
}
```

The second parameter of **Execute** holds a test oracle that will be used to assert additional properties of the writer:

```
public sealed class XmlWriterOracle
{
    public int ElementDepth = 0;

    public void Invariant()
    {
        PexAssert.IsTrue(this.ElementDepth > -1);
    }
}
```

For each method in **XmlWriter**, a transition type inherited from **Transition** can be defined:

```
[DebuggerDisplay("WriteEndElement")]
public class WriteEndElementTransition
    :Transition
{
    public override void Execute(
        XmlWriter writer,
        XmlWriterOracle oracle)
    {
        writer.WriteEndElement();
        oracle.ElementDepth--;
    }
}
```

Transitions can also embed assertions and assumptions:

```
[DebuggerDisplay("WriteStartElement({Name})")]
public class Write StartElementTransition
    :Transition
{
    public string Name;
    public override void Execute(
        XmlWriter writer,
        XmlWriterOracle oracle
    )
{
    writer.WriteStartElement(this.Name);
    PexAssert.IsNotNull(this.Name);
    PexAssert.AreEqual(writer.WriteState,WriteState.Element);
    oracle.ElementDepth++;
    }
}
```

Finally, the parameterized unit test simply takes a sequence of transitions and executes them:

```
[PexMethod]
[PexUseType(typeof(WriteStartElementTransition))]
[PexUseType(typeof(WriteEndElementTransition))]
public void WriteXml([PexAssumeNotNull]Transition[]transitions)
{
    PexAssume.AreElementsNotNull(transitions);

    using (var writer = new StringWriter())
    using (var xwriter = XmlWriter.Create(writer))
    {
        var oracle = new XmlWriterOracle();
        for (int i = 0; i< transitions.Length; ++i)
        {
            var transition = transitions[i];
            // apply transition
            transition.Execute(xwriter, oracle);
            // assert invariant still holds
            oracle.Invariant();
        }
    }
}
```

The assertions and assumptions embedded in the transitions and product code will drive Pex to instantiate different sequence of transitions. The **PexUseTypeAttribute** annotation on the test specify to Pex that those types should be used when a **Transition** type is needed.

Parameterized Stub

Sometimes, the code-under-test already contains many assertion statements that verify its behavior. In this case, an effective parameterized unit test might be quite simple in itself, because it can leverage the assertions in the code.

Pattern 2.14

A parameterized unit test that calls an API without any kind of assertion or specialized scenario:

```
[PexMethod]
public void Add(
    [PexAssumeUnderTest]ArrayList list,
    object item) {
    list.Add(item);
}
```

The attribute **PexAssumeUnderTest** is a short-hand notation to make sure that the parameter is not null, and has exactly the type indicated by its declaration (as opposed to a subtype). The parameterized unit test above equivalent to the following parameterized unit test:

```
[PexMethod]
public void Add(
    ArrayList list,
    object item) {
    PexAssume.IsTrue(list != null);
    PexAssume.IsTrue(list.GetType() == typeof(ArrayList));
    list.Add(item);
}
```

This test pattern relies solely on assertions in the product code, parameter validation, or runtime checks.

Manual Output Review

Pattern 2.15

Log the result of API calls, which you can then review. This pattern is similar to Pattern 2.14; the difference is that values that are computed as part of the parameterized unit test are logged for manual review. The output can be explicitly logged by calling **PexObserve.ValueForViewing**:

```
[PexMethod]
void Add(int a, int b)
{
    var result = a + b;
    PexObserve.ValueForViewing<int>("result", result);
}
```

Regression Tests

Pattern 2.16

Persist a computed value in the generated test. When the generated test is executed in the future, the test verifies that the (possibly changed) code-under-test still computes the same value.

This pattern is similar to Patterns 2.14 and 2.15; the difference is that values are logged, and future runs must compute exactly the same values. There are several ways how outputs can be logged. For a single output value, one can use the return value of the parameterized unit test:

```
[PexMethod]
int Add(int a, int b)
{
    return a + b;
}
```

Pex will recursively traverse the observable properties and fields of the value and add assertions in the generated test for each one of them:

```
void Add01()
{
    int result = this.Add(0, 1);
    Assert.AreEqual(1, result);
}
```

For multiple values, use **out** parameters in the parameterized unit test:

```
[PexMethod]
void Add(int a, int b, out int result)
{
    result = a + b;
}
```

If the number of values might be dynamic, you can also observe values dynamically using the **PexObserve.ValueAtEndOfTest** method:

```
[PexMethod]
void Add(int a, int b)
{
    int result = a * b;
    PexObserve.Value<int>("result", result);
}
```

Differential Regression Test Suite

Pattern 2.17

A test suite, generated from parameterized unit test stubs (Pattern 2.14), is used as a baseline (Pattern 2.16) for behavior changes:

- Using the Pex Wizard, generate an entire parameterized unit test suite of stubs (or reuse a manual parameterized unit test, if available) on product versionn_1.
- Using Pex to generate a traditional unit test suite from the parameterized unit tests obtained above.

The test suite can be seen as a baseline on the observable behavior of the code under-test. Apply the generated test suite on product version n.

Parameterized Models Patterns

Pex provides an infrastructure to implement parameterized models. We refer to parameterized models as they build on top of Pex infrastructure to generate new parameters on the fly, which we usually refer as choices. Parameterized models can be used to replace traditional mocked-based testing as a single model captures all possible behavior.

For a modeled component, instead of defining a single input/output pair as with mock objects, a parameterized model can specify a general input/output relationship, and it can use test parameters to act in various ways. In unit testing, mock objects are used to simulate the behavior of external components in order to test each component in isolation.

Although mock object frameworks have greatly improve the usability in recent years, mock-based testing is still a tedious task. Note that the term mock object is used for somewhat different concepts by developers practicing unit testing. The meaning ranges from very simple (empty) **stubs** to complex and precise behavior with expected inputs and correctly computed outputs.

Martin Fowler discusses this in details (see entry in “Resources and References” at the end of this document). In this sense, the first parameterized model patterns we present start out as simple stubs, but the patterns allow sophisticated models that assert expected inputs and restrict possible outputs.

There are many frameworks that make it easy to write mock objects—for example, for .NET. Similar to how NUnit relates to Pex, these existing frameworks make it easy to manage mock objects—for example, by reducing the amount of code that must be written—but they do not help in exploring different behaviors. Note that Pex comes with a simple stubs framework. This framework was designed to be friendly with the kind of code analysis Pex does.

Choice Provider

Pattern 3.1

In the context of parameterized models, we use a so-called choice provider as a source of test inputs: each choice made by the choice provider can be seen as an additional parameter to the test.

Each generated test names particular value for choices, just as each generated test fixes particular argument values of the parameterized unit test. The values for the choices are not directly passed to the test, but they are “recorded” at the beginning of the generated test, so that they can later be “replayed” while the test is running.

```
[PexMethod]
void AskOracle(int i) {
    // query for a new test parameter
    var j = PexChoose.Value<int>("j");

    // same as usual,
    Assert.IsTrue(I + j == 123);
}
```

The choices made at runtime are saved in the generated test as the record section. The record section specifies for each expected call (**OnCall**) the value that should be returned by the **Choose** methods (**ChooseAt**). Once this behavior has been set, it will be replayed in the test.

```
[TestMethod]
void AskOracle01() {
    // record
    var record = PexChoose.Replay.Setup();
    record.DefaultSession
        .At(0, "j", 123);
    // replay
    this.AskOracle(0);
}
```

The values returned by the choice provider can be filtered using assumptions, just as regular test parameters.

```
[PexMethod]
void AskOracle(int i) {

    var j = PexChoose.Value<int>("j");

    // it’s not just any’j’
    PexAssume.IsTrue(j > 200);

    ...
}
```

Parameterized Models

Pattern 3.2

A parameterized model is an implementation of an interface that:

- Might state assertions on how the interface can be used by clients.
- Obtains observable results from a Choice Provider. See "Choice Provider" earlier in this section).
- Might state assumptions on the results from the point of view of clients.

The **IFileSystem** interface is abstraction of the physical file system. Let's assume that it contains a single method that reads the content of a file:

```
interface IFileSystem{
    string ReadAllText(string fileName);
}
```

In traditional unit testing, you could create a simple stub for the **IFileSystem** interface that unconditionally returns a predefined content:

```
class SFileSystem : IFileSystem {
    public string Content = null;
    public string ReadAllText(string fileName) {
        return this.Content;
    }
}
```

You can then assign a different value to **Content** and use the mock in the test. In parameterized unit testing, **ReadAllText** can obtain its content through a Choice Provider:

```
class PFileSystem : IFileSystem {
    public string ReadAllText(string fileName) {
        // the file could contain anything,
        return PexChoose.Value<string>("result");
    }
}
```

Based on the usage of the return value of **ReadAllText**, Pex will generate different contents for the file. In that sense, choices can simply be seen as additional test parameters, added on the fly. In the following example, the **Parser** type uses dependency injection to gather the necessary services, in particular **IFileSystem**.

```
class Parser {
    IFileSystem fileSystem;

        public Parser(IFileSystem fileSystem) {
            this.fileSystem = fileSystem;
        }

        public void Parse(string fileName) {
            var content = this.fileSystem.ReadAllText(fileName);
            ...
    }
}
```

Dependency injection makes it easy to inject the mock implementation of **IFileSystem** to test **Parser**, bypassing the physical file system:

```
[PexMethod]
void ReadAllText(string fileName) {
    // creating the mock file system
    var fileSystem = new PFileSystem();
    // inject mock in instance under test
    var parser = new Parser(fileSystem);
    // act
    parser.Parse(fileName);
}
```

Parameterized Models with Behavior

Pattern 3.3

A parameterized model with behavior is a parameterized mock that checks for correct arguments, and/or imposes additional assumptions on the choices.

```
class PFileSystem : IFileSystem {
    public string ReadAllText(string fileName) {
        PexAssert.IsTrue(fileName != null);
        var content = PexChoose.Value<string>(fileName);
        // file always starts with 'hello'
        PexAssume.IsTrue(content != null);
        PexAssume.IsTrue(content.StartsWith("hello"));
        return content;
    }
}
```

This implementation of **IFileSystem.ReadAllText** checks that the argument is not **null**, and it constrains the returned file content to always start with “hello”.

Parameterized Models with Negative Behavior

Pattern 3.4

A parameterized mock with negative behavior is a parameterized model that might choose to throw exceptions:

```
class PFileSystem : IFileSystem {
    public string ReadAllText(string fileName) {

        ...
        // throw documented exceptions
        if (PexChoose.Value<bool>("throw"))
            throw new IOException();
        ...
    }
}
```

The Choice Provider might make the choice to throw one of the specified exceptions.

Behaved Value

Pattern 3.5

A behaved value is an undefined value that lazily requests and caches its value from the instance behavior. Pex provides a generic helper method—**BehavedValue<T>**—that can be used to implement a behaved value:

```
interface INamed {
    string Name { get; set; }
}

class MNamed : BehavedBase, INamed  {
    readonly BehavedValue<string> name;
    public MNamed() {
        this.name = new BehavedValue<string>(this, "Name");
    }

    public string Name {
        get { return this.name.Value; }
        set { this.name.Value = value; }
    }
}
```

The implementation of the **Value** property of **BehavedValue<T>** initializes its value lazily, caching the choice once it has been made:

```
private bool hasValue;
private T value;
public TV alue
{
    get
    {
        if (!this.hasValue)
        {
            this.value = PexChoose.Value<T>("Value");
            this.hasValue = true;
        }
        return this.value;
    }
}
```

Behaved Collection

Pattern 3.6

A behaved collection is a collection that lazily requests and caches its content from the instance behavior. Pex provides a generic helper type—**BehavedCollection<T>**—that can be used to implement an indexed cached choice.

```
interface IMap {
    string this[int index] { get; set; }
}

class MMap : BehavedBase, IMap{
    BehavedCollection<string> items;
    public MMap() {
        this.items = new BehavedCollection<string>(this, "Items");
    }
    public string this[int index] {
        get { return this.items[index]; }
        set {this.items[index] = value; }
    }
}
```

Test Ingredients

This section contains practices and patterns that can be combined together to author parameterized unit tests.

Code Contracts

Code Contracts are a new API of the .NET 4.0 runtime; the library and tools is also available separately for earlier .NET versions. The API comprises methods to express pre-conditions, post-conditions, and object invariants.

Pattern 4.1

The following snippet shows how to use the static **CodeContract** class to express the pre-conditions and post-conditions of a method that replaces the first character of a string:

```
static string Capitalize(strings) {
    // pre-condition: s is neither null nor empty
    Contract.Requires(s != null && s.Length > 0);
    // post-condition: result[0] is an upper-case character
    Contract.Ensures(
        char.IsUpper(Contract.Result<string>()[0])
        );
    // post-condition: result[1,..] = s[1,...]
    Contract.Ensures(
        Contract.Result<string>().Substring(1) ==
                                                s.Substring(1)

    );
    ...
}
```

Code Contracts come with an assembly rewriter that transforms the code to insert the post-condition calls at each method exit point. For more information, see the Code Contracts documentation on MSDN.

Class Invariant

Pattern 4.2

A class invariant is an instance method that asserts properties of the instance under test that should hold before and after any public API call. This invariant can use private state to assert properties. Using conditional compilation, this invariant method can only be present at test time, and it can be removed when building releases for customers.

Because the class invariant refers to private state, it is usually co-located with the implementation, and not the test code:

```
public class ArrayList
{
    ...
     [Conditional("DEBUG")]
    public void Invariant() {
        // item array is not null
        Debug.Assert(this.items!=null);
        // count in [0, items.Length]
        Debug.Assert(0 <- this.count);
        Debug.Assert(this.count <= this.items.Length);
    }
}
```

A call to the invariant can be added in the test at various places:

```
[PexMethod]
void AddTest(ArrayList list, object item)
{
    ...
    // act
    list.Add(item);

    // assert invariant
    list.Invariant();
}
```

We can also leverage the Code Contracts library, as described in “Code Contracts” earlier in this section. The Code Contracts rewriter will insert calls to designated class invariant methods automatically at the end of all public methods when runtime checking of contracts is enabled.

To this end, you have to annotate the **Invariant** method with the **ContractInvariantMethod** attribute and call **Contract.Invariant** instead of **Debug.Assert**:

```
[ContractInvariantMethod]
internal void Invariant() {
    // items array is not null
    Contract.Invariant(this.items != null);
    // count in [0, items.Length]
    Contract.Invariant(
        0 <= this.count &&
        this.count <= this.items.Length);
}
```

As a side effect of defining a class invariant with the **ContractInvariantMethod** attribute, Pex will configure instances of this class by directly setting the (private) fields and making sure that the invariant holds.

Test Invariant

Pattern 4.3

A test invariant is a static method that asserts properties of an instance under test that should hold before and after any public API call. This invariant should only inspect the publicly available state—for example, by accessing state only through the public API, in order to assert the properties.

You can use Code Contract invariants for this purpose, or you can define your own test invariant method, which can then be located in the test class.

```
class ArrayListTest
{
    ...
    static void AssertInvariant(ArrayList list)
    {
        Assert.IsTrue(0 <= list.Capacity);
        Assert.IsTrue( 0<= list.Count & list.Count <= list.Capacity);
    }
}
```

Assert Invariant

Pattern 4.4

Assert the invariant of instances under test that have been mutated in the test.

```
[PexMethod]
void AddTest(ArrayList list, object item)
{
    ...
    // act
    list.Add(item);

    // assert invariant
    AssertInvariant(list);
}
```

The invariant can be located in the test project (Pattern 4.3) or directly in the code under test (Pattern 4.2). When the invariant is realized by Code Contract and when you use the Code Contracts rewriter, then you do not have to assert the invariant manually. The Code Contracts rewriter will automatically cause the invariant to be checked at the end of all public methods.

Assume Invariant: Test Inputs Generation by Exploration of Invariants

Pattern 4.5

Create a valid instance of the type under test by setting its fields to arbitrary values and then assuming the class invariant. If your implementation has a method that checks the class invariant, you can expose a public constructor in DEBUG builds that allows you to set all fields of the object freely, provided that the invariant holds afterward.

Pex will then use this constructor to create instances of this class, provided that no other factory method has been specified:

```
class ArrayList {
    ...
    #if DEBUG
    public ArrayList(int[] items, int count) {
        this.items = items;
        this.count = count;
        this.Invariant(); // make sure that invariant holds
    }
    #endif
     [Conditional("DEBUG")]
    public void Invariant() {
        // items array is not null
        Debug.Assert(this.items != null);
        // count in [0, items.Length]
        Debug.Assert(0 <= this.count);
        Debug.Assert(this.count <= this.items.Length);
    }
}
...
[PexMethod]
void AddTest([PexAssumeUnderTest]ArrayList list, object item) {
    // at this point, we will have a properly initialized list
    list.Add(item);
    ...
}
```

When the invariant is realized by a Code Contracts invariant, and runtime checking of contracts is enabled, then you do not need to provide such a DEBUG-only constructor. Pex will simply set all object fields directly, and make sure that the class invariant holds.

Array as Inputs

Pattern 4.6

A parameterized unit test that takes a set of inputs, encoded as an array.

```
[PexMethod]
void AddRange([PexAssumeNotNull]int[] items) {
    ...
    foreach (var item in items)
        list.Add(item);
    ...
}
```

Arrays are the elementary objects in .NET that allow the encoding of a sequence or a set of values. All other data structure involve an overhead, even the rather simple **ArrayList** type. Pex can generate arrays (of any .NET type) quite efficiently, whereas test input generation for parameters of complex data types is much more expensive. Therefore, we strongly recommend that you use arrays when a parameterized unit test requires a collection of values as input.

Generic Test

Pattern 4.7

A parameterized unit test with generic method parameters that takes inputs of a generic type. The parameterized unit test is instantiated at runtime with user-given type arguments.

```
[PexMethod]
[PexGenericArguments(typeof(int))]
void AddTest<T>(List<T> list, T item) {
    ...
    // act
    list.Add(item);
    ...
}
```

From a generic parameterized unit test, Pex instantiates the method with the types provided by the **PexGenericArguments** attribute. When a generic parameter has no constraints, it is most effective to choose the **int** type.

Object Factory

Pattern 4.8

A static method that creates a new instance under test.

The method has enough parameters to be able to set the entire state of the instance under test. Pex only knows precisely how to create and configure very simple classes, where all private fields can be set directory via a constructor or a public property setter, and those classes that have a Code Contracts invariant method.

For all other classes, Pex uses Object Factories, which can construct complex objects from simple values. Pex uses the attributes **PexFactoryMethodAttribute** to identify object factories in **static** classes (**Module** in VisualBasic.NET).

More than one object factory can be defined for a type; Pex will use all of them. In order to explore all corner cases of a parameterized unit test, it is important that the object factories can configure the returned object is all—or at least most—possible ways, controlled only by the parameters given to the object factories.

To this end, the object factory can include some control-flow—for example, loops—as shown in the following example:

```
public static class ArrayListFactory
{
    [PexFactoryMethod(typeof(ArrayList))]
    public static ArrayList Create(int capacity, object[] items)
    {
        var list = new ArrayList(capacity);
        foreach (var item in items)
            list.Add(item);
        return list;
    }
}
```

The **PexFactoryMethodAttribute** must state the exact type of which this factory method creates instances. If the type is not public, the type can be identified by its name. It is not allowed to state an abstract type or an interface as the type of the created instances. The reason for this is that when solving constraint systems to reach particular branches, the constraint solver will determine a particular non-abstract type, and then Pex will use all the factory methods that can return that particular non-abstract type.

In contrast, the formal return type of the factory method can be any super type, or even just **object**. Pex will ignore cases in which the factory method might throw exceptions; Pex only uses successful invocations of the factory method when building method sequences to create complex objects that serve as arguments to a parameterized unit test.

Object Factory Test

When Pex uses a factory method to create objects that are required as arguments to other factory methods or a parameterized unit test, Pex ignores exceptions that might be thrown by a factory method. The reason for this is that in that scenario, the goal for Pex is to exercise a parameterized unit test and not to test the factory methods. This makes it necessary to test the factory methods separately.

Pattern 4.9

Each factory method should be tested by a parameterized unit test that makes sure that at least one instance can be successfully created.

```
[PexClass]
public class ArrayListFactoryTest
{
   [PexMethod]
    // make sure at least one valid instance can be created
    [PexAssertReachEventually(StopWhenAllReached = true)]
    public void ArrayListCreateTest(int capacity, object[] items)
    {
        PexAssume.IsTrue(capacity >- 0);

        PexAssume.IsNotNull(items);
        var list = ArrayListFactory.Create(capacity, items);
        PexAssume.IsNotNull(list);
        PexAssert.ReachEventually();
    }
}
```

Instance Under Test as Parameter

Pattern 4.10

The Instance Under Test is a parameter of the parameterized unit test.

Pex supports any .NET type as input for the parameterized test. To instantiate types, Pex has built-in heuristics that look for public constructor or fields. More complex instantiation scenarios can be further customized using the Object Factory, as described previously in the "Object Factory" section:

```
[PexMethod]
void AddTest<T>(
    [PexAssumeUnderTest]List<T> target, // the instance under test
    T item) {
        target.Add(item);
        ...
}
```

In the example above, we add an item to a **List<T>** instance. A list is usually implemented as an array, which gets resized when the number of elements in the list has reached the capacity of the array.

```
void Add(T item) {
    if (this.count == this.items.Length) // needs resize?
        this.ResizeArray();
    this.items[this.count++] = item;
}
```

This means that depending on the state of the list, the test might trigger the resizing of the array or not. Assuming that the list has a constructor where the initial capacity can be specified, Pex will be able to cover both cases by passing lists with different states to the test method.

The **PexAssumeUnderTestAttribute** specifies that **target** is the instance under test. Using this information, Pex ensures that:

- **target** is not null.
- The type of **target** matches precisely the type of the parameter.

Distinguishable Argument Validation

Pattern 4.11

The pre-conditions on the input of an API are validated using distinguishable exceptions.

```
class Parser {
    public void Parse(string input) {
        // pre-conditions

        if (input == null)
            throw new ArgumentNullException("input");
        ...
        // regular assertion
        Debug.Assert(condition, "this should not happen");
    }
}
```

To enable an efficient use of the Pattern 2.10, the exception thrown by argument validation—that is, preconditions—should be distinguishable from other violations in the code. The following test only allows **ArgumentNullException** when it is raised by methods of the **Parser** class, whereas all other sources of this exception type would still indicate failures.

```
[PexMethod]
[PexAllowedExceptionFromType(typeof(ArgumentNullException),
typeof(Parser))]
void Parse(Parser parser, string input) {
    parser.Parse(input);
}
```

Type Under Test

Pattern 4.12

It is not uncommon that a test class **MyClassTest** targets a specific type **MyClass** from the code under test. In that case, we say that **MyClass** is the Type Under Test (TUT) of **MyClassTest**.

The TUT can be leveraged by automated test generators to tune the various search knobs of the exploration engine. The TUT is also useful to identify the code under test from the rest of the code—that is, system—test. This knowledge can be used to build smarter exception filters or provide more targeted coverage information.

The TUT can be specified in the constructor of the **PexClassAttribute** attribute:

```
class MyClass{
    ...
}

// User Code is the type under test.
[TestClass, PexClass(typeof(UserCode))]
class MyClassTest {
    ...
}
```

Anti-Patterns

This section contains a list of patterns that should be avoided.

Parsed Data Anti-Pattern

Anti-Pattern 5.1

A test creates structured data by parsing unstructured input and only uses the structured data during the test. For example, when a parameterized test builds structured data from an input xml string, then invokes methods that use the data.

The major problem with this approach is that the test generator tool will spend a great amount of computation to “understand” the xml parser. In fact, it might happen that the tool never manages to generate an interesting valid input:

```
[PexMethod]
void ParseAndTest(string xml) {
    // parse
    Employee e = Employee.Deserialize(xml);
    // test logic
    EmployeeCollection c = new EmployeeCollection();
    c.Add(e);
    Assert.IsTrue(c.Contains(e));
}
```

Instead, the test parameter should be the **Employee** itself. Then a more efficient Pattern 4.8 can be used instead of XML parsing:

```
[PexMethod]
void Test(Employee e)
{
    // test logic
    EmployeeCollection c = new EmployeeCollection();
    c.Add(e);
    Assert.IsTrue(c.Contains(e));
}
```

Hidden Integration Test

Anti-Pattern 5.2

A parameterized unit test outcome depends on the state of the environment:

```
[PexMethod]
void FileExists(string fileName) {
    if (!File.Exists(fileName))
        throw new FileNotFoundException();
    ...
}
```

The definition of unit test is fairly clear: if it depends on the state of the environment, it is not a unit test. This includes the file system, network streams, dates and time, random integer or user interaction in general. In practice, it requires a lot of discipline to avoid falling into one of the above traps. Who has never used **DateTime.Now**? There is no magic bullet solution to this problem. It usually requires inserting an additional layer of abstraction between the environment and the code, sufficient to enable mocking the environment.

Branch to Assumption Anti-Pattern

Anti-Pattern 5.3

An assumption is conditionally executed.

```
[PexMethod]
void Test(int i, int j) {
    if (I < 0)
        PexAssume.IsTrue(j > 0);
    ...
}
```

The meaning of the above parameterized unit test is difficult to understand, and its exploration by Pex is somewhat inefficient. Instead, use specialized methods to state conditional assumptions:

```
[PexMethod]
void Test(int i, int j) {
PexAssume.ImpliesIsTrue(I < 0, () => j > 0);
...
}
```

Asserting Precondition and Invariants

Anti-Pattern 5.4

Using the same API to express preconditions—that is, argument validation—post-conditions, invariants, and assertions.

```
public void Parse(string input) {
    // precondition
    Debug.Assert(input != null, "invalid argument");
    ...
    // invariant
    Debug.Assert(condition, "this should not happen");
}
```

There are several disadvantages on using the same API to express preconditions, post-conditions, invariants, and assertions: It makes it impossible to distinguish undesirable inputs from program failures. The Pattern 2.10 cannot be applied.

Although internal invariants are usually removed from the production build, argument validation at the level of public APIs might still be required in production. Using the same API, it is not possible to take advantage of the **ConditionalAttribute** capabilities.

Hidden Complexity

Anti-Pattern 5.5

A piece of code that uses an innocent-looking API that in fact performs a lot of computation.

This pattern defeats automated tools like Pex because the tool spends a lot of time understanding the “hidden” code. Complexity should be treated as a dependency and refactored appropriately. Examples of such patterns are:

- Using regular expressions for simple string validation.
- Parsing and processing xml streams, when the test only cares about well-formed data.
- Sophisticated mocking framework that make writing mocks easy by postponing code-generation to the runtime, and/or realize mock behavior by interpretation instead of executing code.

.NET Patterns

This section describes test patterns that apply to any .NET type that implements core interfaces or methods such as **Object.Equals** or **IDisposable**.

Equals Itself

Pattern 6.1

An instance should be equal to itself.

```
[PexMethod]
void EqualsItself<T>(T target) {
    Assert.IsTrue(target.Equals(target));
}
```

Different Hash Code Implies Not Equal

Pattern 6.2

Instances with difference hash code are not equal.

```
[PexMethod]
void HashCodeNotEqual<T>(T left, T right) {
    // assume different hashcodes
    PexAssume.IsTrue(left.GetHashCode() != right.GetHashCode());
    // should not be equal
    Assert.IsTrue(!object.Equals(left,right));
}
```

Dispose Twice

Pattern 6.3

A disposable instance—that is, implementing **IDisposable**—can be disposed twice.

```
[PexMethod]
void DisposeTwice<T>(T target)
    where T : IDisposable {
    target.Dispose();
    target.Dispose(); // should behave nicely
}
```

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
Unit Testing with Microsoft Moles	200
Exploring Code with Microsoft Pex	200
Unit Testing SharePoint Foundation with Microsoft Pex and Moles	300
Parameterized Unit Testing with Microsoft Pex	400

Pex and Moles Technical References	
Microsoft Moles Reference Manual	400
Microsoft Pex Reference Manual	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

[1] Ayende Rahiem. Rhino Mocks. http://ayende.com/projects/rhino-mocks.aspx.

[2] M. Andersen, M. Barnett, M. Fähndrich, B. Grunkemeyer, K. King, and F. Logozzo. http://research.microsoft.com/projects/contracts, 2008.

[3] Martin Fowler. Mocks aren't stubs. http://www.martinfowler.com/articles/mocksArentStubs.html. [accessed 11-September-2008].

[4] Moq Team. Moq. http://code.google.com/p/moq/.

[5] NMock Development Team. NMock. http://nmock.org.

[6] Pex development team. Stubs, Lightweight Test Stubs and Detours for .NET. http://research.microsoft.com/Stubs, 2009.

[7] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In Proc. of Tests and Proofs (TAP'08), volume 4966 of LNCS, pages 134–153, Prato, Italy, April 2008. Springer.

[8] N. Tillmann and W. Schulte. Parameterized unit tests. In Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 253–262. ACM, 2005.

[9] TypeMock Development Team. Isolator. http://www.typemock.com/learn_about_typemock_isolator.html.

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()`
c is true	`IsTrue(bool c)`
c is false	`IsFalse(bool c)`
Treats test as inconclusive	`Inconclusive()`
Implication	
p holds if *c* holds	`ImpliesIsTrue(bool c, Predicate p)`
p holds if *c* holds (case-split)	`Case(bool c).Implies(Predicate p)`
Nullarity	
o is not null	`IsNotNull(object o)`
a is not null or empty	`IsNotNullOrEmpty<T>(T[] a)`
a elements are not null	`AreElementsNotNull<T>(T[] a)`
Equality	
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)`
Reference Equality	
expected is same as *actual*	`AreSame(expected, actual)`
expected is not the same as *actual*	`AreNotSame(expected, actual)`
Type Equality	
o is an instance of *t*	`IsInstanceOfType(object o, Type t)`
o is not an instance of *t*	`IsNotInstanceOfType(object o, Type t)`
Over collections	
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)`
Exceptional Behavior (PexAssert only)	
action throws an exception of type **TException**	`Throws<TException>(Action action)`
action succeeds or throws an exception of type **TException**	`ThrowAllowed<TException>(Action action)`
Behavior Equality (PexAssert only)	
l and *r* behave the same	`AreBehaviorsEqual<T>(Func<T> l, Func<T> r)`
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 **T**	`PexChoose.Value<T>(string description)`
any non-null value of type **T**	`PexChoose.ValueNotNull<T>(string description)`
any valid **enum** value of type **T**	`PexChoose.EnumValue<T>(string description)`

Range Choices	
any value from **a**	`PexChoose.ValueFrom<T>(string description, T[] a)`
any integer value within a **(min; max)**	`PexChoose.ValueFromRange(string description, int min, int max)`
any index for **a**	`PexChoose.IndexValue<T>(string description, T[] a)`