

Getting Started With Stubs, Moles (and Pex)

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

Preliminary Draft
Copyright Microsoft Corporation.

October 14, 2009

Abstract

This document provides a short, step-by-step tutorial for the *Stubs* framework [2].

This tutorial assumes that the reader is familiar with some basic notions of unit testing [1].

Contents

1	The Application under Test	3
1.1	Exercise 1: Create the Project Under Test	3
2	Integration Test Disguised as a Unit Test	5
2.1	Exercise 2: Create the Test Project	6
3	Isolating the Unit Test	9
4	Stubs	9
4.1	Exercise 3: Using Stubs	11
5	Moles	13
5.1	Exercise 4: Using Moles	14
6	Intermezzo: Executing Tests with Pex	14
6.1	Exercise 5: Using The Pex runner	16
7	Stubs and Moles	16
7.1	Code Under Test	16
7.2	Trapping Environment Calls	17
7.3	Moling the <code>Bank</code> constructor	17
7.4	Returning a stub instance of <code>IAccount</code>	18
7.5	Observing values	18
7.6	The <code>DefaultValue</code> fallback behavior	18
7.7	The final Unit Test	19
7.8	Are we done yet?	19
8	Parameterized Unit Testing with Pex	20
9	Further Reading	22

1 The Application under Test

In this tutorial, we will test in various ways a simple component, `FooReader`, that loads the content of a file, checks that it starts with some pre-defined header and returns the content.

```
public class FooReader {
    public string Content { get; private set; }
    public void LoadFile(string fileName) {
        var content = FileSystem.ReadAllText(fileName);
        if (!content.StartsWith("foo"))
            throw new ArgumentException("invalid file");
        this.Content = content;
    }
}
```

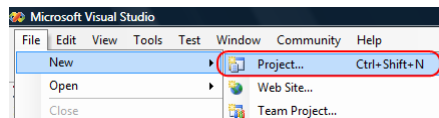
The `FooReader` implementation uses `FileSystem` which provides services to interact with the file system.

```
public static class FileSystem {
    public static string ReadAllText(string fileName) {
        return File.ReadAllText(fileName);
    }
}
```

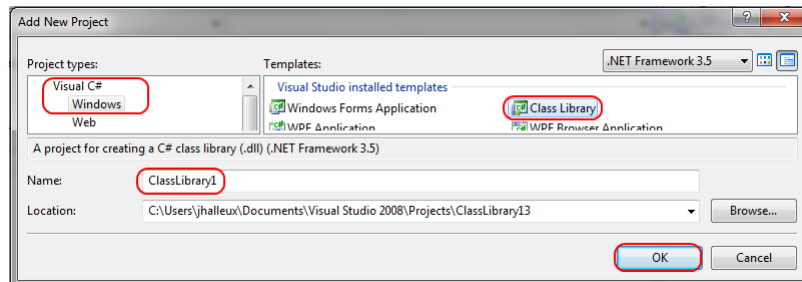
1.1 Exercise 1: Create the Project Under Test

Tip: Got a question? Use the forums! If you have any question along this tutorial, do not hesitate to post it on the Pex forums at <http://social.msdn.microsoft.com/Forums/en/pex/threads>.

1. Open Visual Studio
2. Go to **File|New|Project....**



3. On the left pane, select **Visual C#-Windows**, then select the **Class Library** item.



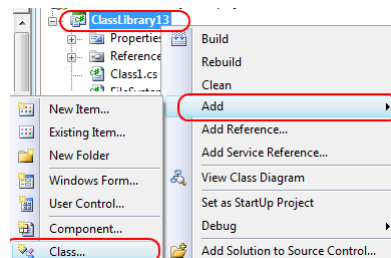
Select a location for the new project and click **Ok**. (The content of the window shown above might vary depending on your Visual Studio installation.)

4. Replace the content of the `Class1.cs` file with the following snippet. You can also rename it to `FooReader.cs` for clarity.

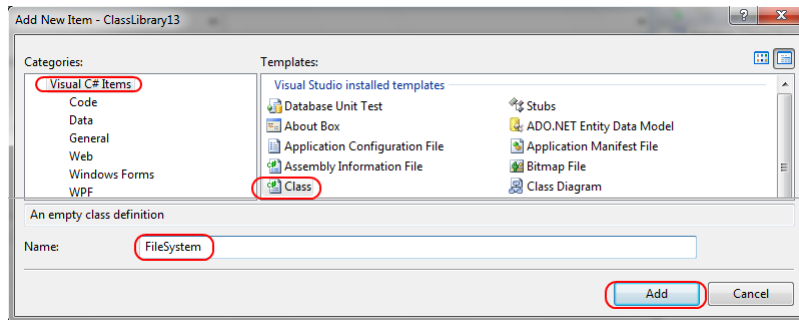
```
namespace StubsTutorial {
    using System;

    public class FooReader {
        public string Content { get; private set; }
        public void LoadFile(string fileName) {
            var content = FileSystem.ReadAllText(fileName);
            if (!content.StartsWith("foo"))
                throw new ArgumentException("invalid file");
            this.Content = content;
        }
    }
}
```

5. Right click on the project node and select **Add...|Add New Class...**,



rename the file to `FileSystem` and click **Ok**.



Replace the content of the file with the following snippet:

```
namespace StubsTutorial {
    using System;
    using System.IO;

    public static class FileSystem {
        public static string ReadAllText(string fileName) {
            return File.ReadAllText(fileName);
        }
    }
}
```

6. Build the solution through the menu **Build|Build Solution**. The solution should build without errors.

In this example, one could argue that the `FileSystem` class is useless and we could directly call into the `System.IO.File` methods. The purpose of this class is to keep this tutorial simple and provide the foundation to make the application more testable. In practice, `System.IO.File` methods could also be *moled*.

2 Integration Test Disguised as a Unit Test

The current implementation of `FooReader` is problematic for testing: it relies on a static method `FileSystem.ReadAllText` that interacts with the external environment. This means that this code cannot be tested in isolation. In this simple example, it is just a file and may seem benign. In reality, the dependency might be external servers, network connections, devices, etc...

One way to work around this environment dependency is to write an integration test. In this case, that would mean creating a file in the course of the unit test and pass it into the `FooReader` implementation. The following unit test does exactly this in three steps:

- arrange: we set up the data necessary for the test,
- act: we invoke the program under test with the inputs,

- assert: we assert the correct expected behavior

```
[TestMethod]
public void CheckValidFile() {
    // arrange
    var fileName = "foo.txt";
    var content = "foo";
    File.WriteAllText(fileName, content);

    // act
    var foo = new FooReader();
    foo.LoadFile(fileName);

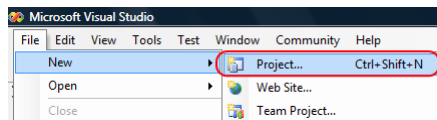
    // assert
    Assert.AreEqual(content, foo.Content);
}
```

Although this test is written as a unit test for the Visual Studio Unit Test test framework, it is truly an integration test in disguise. It might fail when the disk is full, or when we don't have the right access permissions. In the following section, we will see how *Stubs* can be used to alleviate this issue so that we can ignore sporadic catastrophic environment failures.

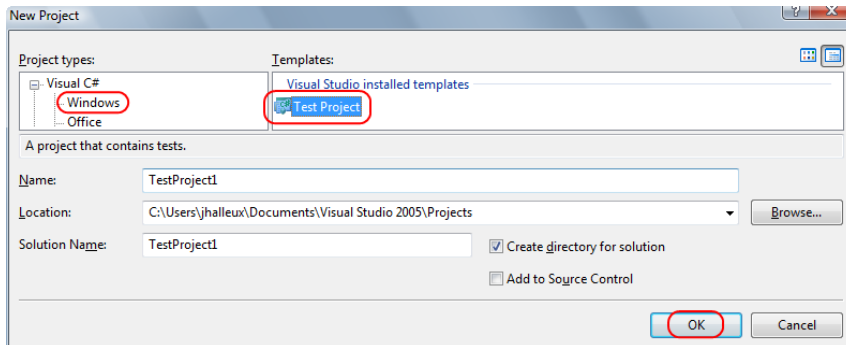
2.1 Exercise 2: Create the Test Project

This exercise shows how to create a Visual Studio Unit Test project and add a new unit test. The Visual Studio Unit Test framework is part of the Visual Studio Pro, Team Engineering or Team Test Suite. The screenshots of this tutorial were taken on Visual Studio2008, the appearance of menus might have changed in Visual Studio2010.

1. Go to **File|New|Project...**

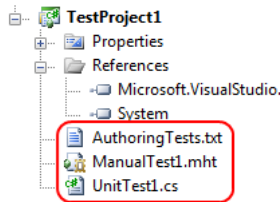


2. On the left pane, select **Visual C#|Test**, then select the **Test Project** item. Select a location for the new project and click **Ok**.

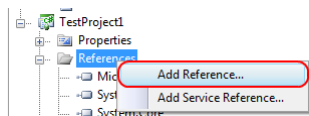


(The content of the window shown above might vary depending on your Visual Studio installation.)

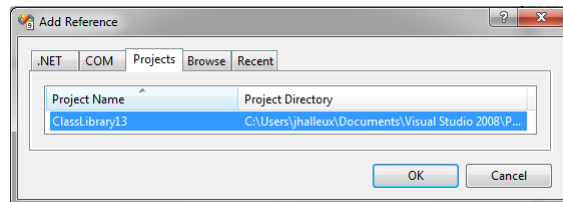
3. Delete the sample files that were generated by the project wizard (`AuthoringTests.txt`, `ManualTest1.mht`, `UnitTest1.cs`). (You can right-click a file name, and select **Delete** in the context menu.)



4. Right click on the **References** node under the project node,



select the **Projects** tab and select the project containing `FooReader`,



5. Add a new class and replace the content with the following snippet,

```
namespace StubsTutorial {
    using System.IO;
    using Microsoft.VisualStudio.TestTools.UnitTesting;
```

```

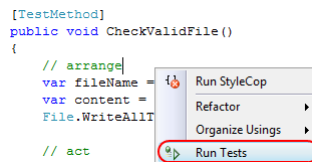
[TestClass]
public partial class FooReaderTest {
    [TestMethod]
    public void CheckValidFile() {
        // arrange
        var fileName = "foo.txt";
        var content = "foo";
        File.WriteAllText(fileName, content);

        // act
        var foo = new FooReader();
        foo.LoadFile(fileName);

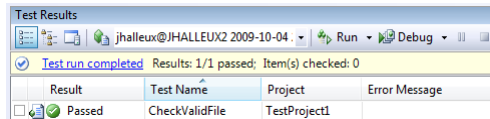
        // assert
        Assert.AreEqual(content, foo.Content);
    }
}

```

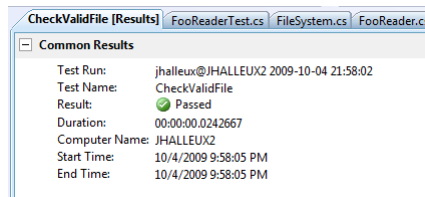
- Right click inside the method body of CheckValidFile and click on **Run Tests**.



- CheckValidFile succeeded and you can review the details of that particular test by double-clicking on the row.



- The test details view gives various metrics about the test run (duration, etc.) as well as the console output.



3 Isolating the Unit Test

There are two ways to isolate the unit test.

The preferred solution should be to refactor the code and introduce an abstract layer between `FooReader` and `FileSystem`, usually as an interface. When unit testing `FooReader`, one can use a fake implementation of `FileSystem` that does not rely on the environment. The **Stubs** generator will generate *stub types* for any .NET interfaces that can be used for this purpose. This approach is detailed in section 4.

In some cases, refactoring is not possible. For example, even after introducing abstractions, some low-level code will still have to call some environment-facing API which might be part of the runtime libraries which cannot be refactored. In that case, one has to use runtime instrumentation to inject hooks to detour methods¹. Using those hooks, it is possible to intercept any environment facing API and fake its behavior. This approach is detailed in section 5.

4 Stubs

The preferred solution to deal with testability issues is to introduce abstraction layers, e.g. interfaces, between the environment and the project. In this example, this would mean introducing an `IFileSystem` interface representing the services of `FileSystem`,

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

Then, we refactor the `FooReader` class to receive an instance of `IFileSystem` in the constructor. This pattern is commonly called *dependency injection* citefowler04:inversionOfControl.

```
public class FooReader {
    IFileSystem fs;
    public FooReader(IFileSystem fs) {
        this.fs = fs;
    }
    public void LoadFile(string fileName) {
        var content = this.fs.ReadAllText(fileName);
        ...
    }
    public string Content { get; private set; }
}
```

When writing the unit test, we can simply pass a fake implementation of `IFileSystem`, called `MockFileSystem`, instead of using the real file system. In this case, we create a simple implementation that returns a single, hand-picked content for any file.

¹A common approach to inject hooks into method is to implement a rewriting .NET profiler, which is exactly how our *moles* are realized. Before any method is JIT compiled by the runtime, the profiler has the opportunity to rewrite the method body MSIL and inject the hooks.

```

[TestMethod]
public void CheckValidFile() {
    // arrange
    var fileName = "foo.txt";
    var content = "foo";
    var fs = new MockFileSystem() {
        Content = content
    };

    // act
    var foo = new FooReader(fs);
    foo.LoadFile(fileName);

    // assert
    Assert.AreEqual(content, foo.Content);
}

```

The problem with the approach above is that we had to write manually the fake file system implementation `MockFileSystem` which increases the amount of code to write and maintain.

```

class MockFileSystem : IFileSystem {
    public string Content;
    string IFileSystem.ReadAllText(string fileName) {
        return this.Content;
    }
}

```

The **Stubs** framework automatically generates similar implementations of interfaces. The code generation is controlled by a `.stubx` file that can be added through the **New Item** menu. Each method of the interface has an associated field whose type is a delegate type that matches the signature of the method. When calling the stub method, the user delegate is called if set. Each *stub type* is named by prepending with `s` the stubbed type name in a sub-namespace `.Stubs`, e.g. the stub type of `Foo.IBar` would be `Foo.Stubs.SIBar`.

```

class SIFileSystem : IFileSystem {
    public Func<string, string> ReadAllTextString;
    string IFileSystem.ReadAllText(string fileName) {
        var stub = this.ReadAllTextString;
        if (stub != null)
            return stub(fileName);
        else
            ...
    }
}

```

Using `SIFileSystem`, we can get rid of the manually written `MockFileSystem` class, and instead attach a lambda expression to the delegate field of the generated stub type.

```

[TestMethod]

```

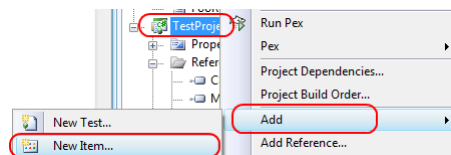
```

[HostType("Pex")]
public void CheckValidFileWithStubs() {
    // arrange
    ...
    var fs = new SIFileSystem() {
        ReadAllTextString = f => {
            Assert.IsTrue(f == fileName);
            return content;
        }
    };
    ...
}

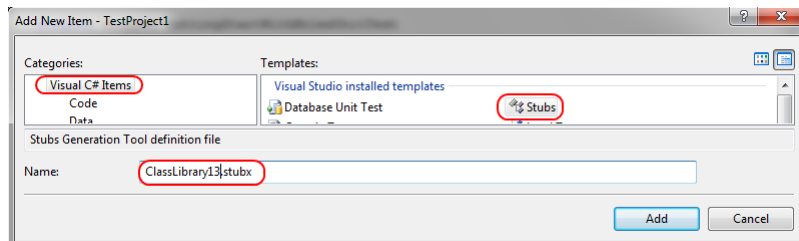
```

4.1 Exercise 3: Using Stubs

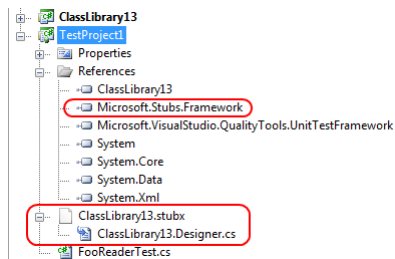
1. Right click on the project node and select **Add...|Add New Item...**,



2. On the left pane, select the **Stubs** item, **rename the file name to the project under test name and click Add**,



3. As soon as the `.stubx` is added, the **Stubs** code generator will process it and generate the *mole types* and *stub types*. The code generator also adds the necessary references automatically, such as the `Microsoft.Stubs.Framework.dll`.



4. The `.stubx` file is a simple XML file that specifies which assembly should be moled and stubbed. It also provides a fine grained control over the code generation that is detailed in the reference manual.

```
<Stubs xmlns="http://schemas.microsoft.com/stubs/2008/">
  <Assembly Name="ClassLibrary13" />
</Stubs>
```

5. Add a new file in the project that will hold the `IFileSystem` interface,

```
namespace StubsTutorial {
    using System;
    public interface IFileSystem {
        string ReadAllText(string fileName);
    }
}
```

6. Refactor the `FooReader` class to take an `IFileSystem` instance in the constructor and use it in the `Load` method,

```
namespace StubsTutorial {
    using System;
    public class FooReader {
        IFileSystem fs;
        public FooReader(IFileSystem fs) {
            this.fs = fs;
        }
        public void LoadFile(string fileName) {
            var content = this.fs.ReadAllText(fileName);
            ...
        }
        public string Content { get; private set; }
    }
}
```

7. Refactor the unit test to use the generated stub type.

```
[TestMethod]
public void CheckValidFileWithStubsFull() {
    // arrange
    var fileName = "foo.txt";
    var content = "foo";
    var fs = new SIFileSystem() {
        ReadAllTextString = f => {
            Assert.IsTrue(f == fileName);
            return content;
        }
    };
    // act
```

```

var foo = new FooReader(fs);
foo.LoadFile(fileName);

// assert
Assert.AreEqual(content, foo.Content);
}

```

8. Execute the unit test with or without the debugger, possibly setting breakpoints to understand how the stubs work.

5 Moles

When refactoring is not possible, runtime instrumentation may be used.

Mole types are strongly typed wrappers that allow to redirect any .NET method to a user defined delegate. They are generated by the *Stubs* framework code generator in addition to the regular interface and non-sealed class stubs. They are useful when dealing with APIs with static methods, sealed types or non-virtual methods.

The code generation is defined through the same `.stubx` file and follows the same idea as *stub types*: Each *mole type* is named after the *moled* type prepending a `M` to its name and placing it into a `.Stubs` namespace: `Bar.Foo` \rightarrow `Bar.Stubs.MFoo`. Each static method in the moled type has an associated property². By setting the property, one can redirect the method to a user defined delegate (the property type is a delegate that matches the signature of the moled method).

Let us rollback to the code in section 1.

In this example, the *mole type* looks like this (the generated code is actually more complicated):

```

namespace StubsTutorial {
    using System;
    public static class MFileSystem {
        public static Func<string, string> ReadAllTextString {
            set {
                ...
            }
        }
    }
}

```

Using this mole type, we can attach a delegate to the `FileSystem.ReadAllText` by assigning it to the `MFileSystem.ReadAllText` property. Under the hood, the **Stubs** runtime will take care of rerouting any future call to `FileSystem.ReadAllText` to that delegate.

```

[TestMethod]
[HostType("Pex")]
public void CheckValidFileWithMoles() {

```

²All its argument types and the return type must be visible. It is a current implementation restriction that the signature must match one of the predefined set of delegate signatures supported by the code generator

```

    // arrange
    ...
    MFileSystem.ReadAllTextString = f => {
        Assert.IsTrue(f == fileName);
        return content;
    };
    ...
}

```

Moles must be executed under the Pex host type because they require runtime instrumentation. This is specified through the `[HostType("Pex")]` attribute. The host type is currently only support for Visual Studio Unit Test 2008.

```

[TestMethod]
[HostType("Pex")]

```

5.1 Exercise 4: Using Moles

1. Add the following unit test to the `FooReaderTest` class and execute it,

```

[TestMethod]
[HostType("Pex")]
public void CheckValidFileWithMole() {
    // arrange
    var fileName = "foo.txt";
    var content = "foo";
    MFileSystem.ReadAllTextString = f => {
        Assert.IsTrue(f == fileName);
        return content;
    };

    // act
    var foo = new FooReader();
    foo.LoadFile(fileName);

    // assert
    Assert.AreEqual(content, foo.Content);
}

```

2. To understand better how the mole redirection works, place a breakpoint in the delegate by pressing the `F9` key and execute the test under the debugger by clicking on the `Ctrl + R + T` keys.

6 Intermezzo: Executing Tests with Pex

Before we move on, let us introduce Pex [3, 4]. At its core, Pex is a test input generation tool which enables *parameterized unit testing* [5]: Starting from a unit test with

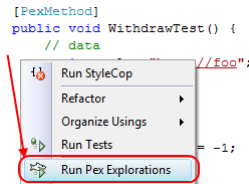
parameters, Pex automatically analyzes the code under test, using a constraint solver to compute test inputs that will exercise a diverse set of execution paths. The result is a small generated test suite with high code coverage.

The **Stubs** framework was designed to work efficiently with Pex. For now, let us introduce you to the Pex test runner in Visual Studio, we will talk more about leveraging Pex to generate tests in section 8.

After adding a reference to the `Microsoft.Pex.Framework` assembly, replace the `TestMethod` and `HostType` attribute with the `PexMethod` attribute. We can now

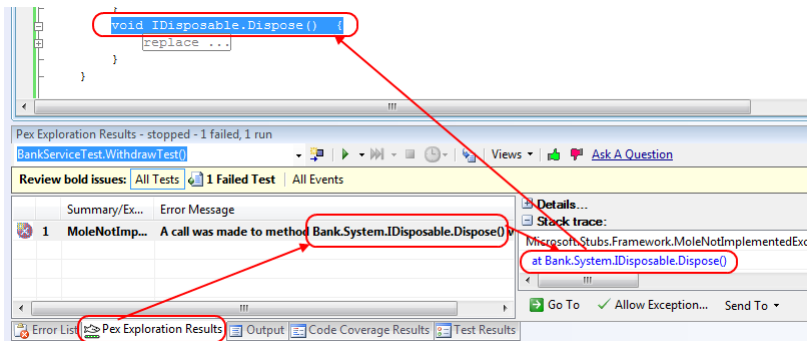
Tip: Ctrl + . shortcut After adding the `PexMethod` attribute, a small red rectangle will appear under the last letter of the word. This is a shortcut to the intellisense menu. Click on the rectangle and select the first item using `Microsoft.Pex.Framework`; which will add the appropriate namespace import at the top of the file. You can also use the `Ctrl + .` shortcut which will save you a lot of time.

run the test through the Pex runner by right-clicking in the test method and selecting **Run Pex Explorations**.



The **Pex Exploration Results** will appear at the bottom of the IDE. It contains two main components: a table where each row represents a generated test. Each input or output of the test is mapped to a column, and exceptions are displayed on the last two columns. In this example, we still have a closed unit test so the table looks pretty empty.

Whenever an exception occurs, one can click on the test case row. A details pane will open on the right side of the **Pex Exploration Results** view. The details pane shows a clickable stacktrace of the exception (simply click on the (blue) frames to move the editor to that location). The full error message is also available by hovering over the exception type in the stacktrace view. We will use Pex as the test runner in the following exercises.



6.1 Exercise 5: Using The Pex runner

1. Add a reference to the `Microsoft.Pex.Framework.dll` assembly through the **Add References** dialog window.
2. Replace the `[TestMethod]` and `[HostType("Pex")]` attributes with the `[PexMethod]` attribute from the `Microsoft.Pex.Framework` namespace.
3. Right-click inside the test method and click on **Run Pex Explorations** item.
4. Inject an exception somewhere in the `withdraw` method and execute Pex again. Drill through the exception message and stacktrace to familiarize yourself with the result view.

7 Stubs and Moles

This section shows how *mole types* and *stub types* can be used together to stub a sequence of calls. While mole types can deal with concrete types, stubs might be needed to provide default interface implementations.

This section also gives the methodology to build unit test involving environment dependencies. There are two key ideas behind the methodology:

1. Identify the environment facing APIs and attach fallback behaviors to them.
2. Follow the flow of the program to write the mole redirections.

7.1 Code Under Test

The purpose of this exercise is to test the following method.

```
class BankService {
    public void Withdraw(string url, int accountId, int amount) {
        using (var bank = new Bank(url)) {
            var account = bank.GetAccountById(accountId);
            account.Withdraw(amount);
            account.Save();
        }
    }
}
```

This method has several testability issues: it allocates an `Bank` instance instead of taking it as a parameter, `Bank` is a sealed class with no public constructor. Moreover, we will need an implementation of the `IAccount` interface if we want to introduce a fake bank for testing purpose. Although interfaces makes the system testable, it does not relieve from the burden of providing a fake implementation for testing.

```
sealed class Bank : IDisposable {
    public Bank(string url) {
        ...
    }
}
```

```

    public IAccount GetAccountById(int id) {
        ...
    }
    void IDisposable.Dispose() {
        ...
    }
}

interface IAccount {
    int Balance { get; }
    void Withdraw(int value);
    void Save();
}

```

7.2 Trapping Environment Calls

It is not always obvious which externally facing API might be invoked in the program under test. An easy way to tackle with this is to mole all the methods from an assembly (or a type) so that they throw a `MoleNotImplementedException`. `MoleNotImplementedException` is a special exception type that is used to flag a method that was not moled. Once the trap is in place, we simply call the method under test.

```

[PexMethod]
public void WithdrawTest() {
    MBank.FallbackBehavior = MoleFallbackBehavior.NotImplemented;

    new BankService().Withdraw("http://foo", 10, 1);
}

```

We launch Pex on the test method and it generates a single test case which fails with a `MoleNotImplementedException`. With the message of the exception and the stack trace, we can precisely identify which method needs to be moled and in which context. In this case, the constructor of `Bank` was invoked and needs to be moled.

7.3 Moling the Bank constructor

Each instance constructor has an associated static method `New`. In that method, one can wrap a newly runtime instance, which will be passed in as the first `me` parameter, into a mole.

```

...
MBank.NewString = (me, url) => {
    new MBank(me);
};

new BankService().Withdraw("http://foo", 10, 1);

```

Since we've added a mole, we execute the test again and investigate the failure to find the next missing mole. In this case, the `GetAccountById` method needs to be *moled* next.

7.4 Returning a stub instance of `IAccount`

Since this method returns the `IAccount` interface, we return an instance of `SIAccount` which is the stub type of `IAccount`.

```
...
MBank.NewString = (me, url) => {
    new MBank(me) {
        GetAccountByIdInt32 = id => new SIAccount()
    };
};
...
```

Again, we execute the test and analyze the results to find out that the `Account.Withdraw` setter was not stubbed. In this case, the exception returned was the `StubNotImplementedException` which is the dual of `MoleNotImplementedException` but for stub types. This iterative process should continue until the test actually goes through.

7.5 Observing values

When attaching delegates to moles, it is also possible to observe values. In this example, we use the `observedValue` local to record which value if any was passed to the `IAccount.Withdraw` method. The `observedAmount` local can be used to verify that the `IAccount.Withdraw` was called with the proper arguments.

```
...
int? observedAmount = null;
new MBank(me) {
    GetAccountByIdInt32 = id => new SIAccount() {
        Withdraw = x => observedAmount = x
    }
};
...
Assert.IsTrue(1 == observedAmount);
```

7.6 The `DefaultValue` fallback behavior

Another common scenario is that a number of methods are called on the moled type that do not actually matter. Instead of attaching 'empty' delegates to each member, one can attach the `DefaultValue` behavior which creates delegates that do nothing or return the default value. This behavior exists for both stub types (`StubFallbackBehavior.DefaultValue`) and mole types (`MoleFallbackBehavior.DefaultValue`).

In this example, we attach the `DefaultValue` behavior to the `SIAccount` instance to ignore the call to `Save`.

```
GetAccountByIdInt32 = id => new SIAccount() {
    Withdraw = x => observedAmount = x,
    FallbackBehavior = StubFallbackBehavior.DefaultValue
},
```

7.7 The final Unit Test

After a couple more iterations and missing moles, we finally have a test that passes, observes values and asserts expected behaviors.

```
[PexMethod]
public void WithdrawTest() {
    // data
    string url = "http://foo";
    int account = 10;
    int amount = 1;

    // observers
    int observedAmount = -1;
    bool saved = false;

    // arrange
    MBank.FallbackBehavior = MoleFallbackBehavior.NotImplemented;

    MBank.NewString = (me, _url) => {
        new MBank(me) {
            GetAccountByIdInt32 = _id => new SIAccount() {
                Withdraw = x => observedAmount = x,
                Save = () => saved = true
            },
            SystemIDisposable.Dispose = () => { }
        };
    };

    // act
    var service = new BankService();
    service.Withdraw(url, account, amount);

    // assert
    Assert.IsTrue(amount == observedAmount);
    Assert.IsTrue(saved);
}
```

7.8 Are we done yet?

After all this work, we have successfully executed a test that covered all the code path in the method under test... since there is only a single execution path when we run the code with fixed input values. Of course things are rarely that simple in reality and the method under test might contain parameter validation and loops that will need to be tested. For example, let us make the `Withdraw` method more interesting and realistic by adding validation on the amount being transferred.

```
void Withdraw(string url, int accountId, int amount) {
    if (String.IsNullOrEmpty(url))
        throw new ArgumentNullException("url");
}
```

```

if (accountId < 0)
    throw new ArgumentOutOfRangeException("accountId");
if (amount < 0 || amount > 1000)
    throw new ArgumentOutOfRangeException("amount");

using (var bank = new Bank(url))
{
    var account = bank.GetAccountById(accountId);
    var balance = account.Balance;
    if (balance < amount) // not enough funds
        throw new InvalidOperationException("no funds");
    account.Withdraw(amount);
    account.Save();
}
}

```

To cover the different behaviors of the new version of `Withdraw`, we would have to write at least 5 more unit tests, which would all involve some number of moles or stubs and a careful understanding of the program semantics to craft the relevant scenarios.

Although, *mole types* and *stub types* have successfully worked around the testability issues of the API, the developer is still facing a huge task of writing (and later maintain) the necessary unit tests. Pex is a tool that can help reduce that burden.

8 Parameterized Unit Testing with Pex

The unit test of `Withdraw` contained a data section with three *magic* values:

```

// data
string url = "http://foo";
int account = 10;
int amount = 1;

```

The choice of these values is critical to test the implementation of the `Withdraw` method, as different values cause the application to take different code paths. Before Pex, the developer would have to make educated guesses to determine the right values for testing, or a random test input generator could be employed to try thousands or millions of values. Pex is a tool that automates the process of finding relevant test inputs in a systematic way. Pex executes the code under test while monitoring every instruction executed by the .NET code. In particular, Pex records all the conditions that the program checks along the executed code path, and how they relate to the test input. Pex gives this information to a constraint solver, which then crafts new inputs that will trigger different code paths. Explaining how Pex works in more detail is beyond the scope of this tutorial, so we refer the reader to the Pex documentation for further details [3, 4].

Since we want to test the `Withdraw` method for all possible input values, we simply refactor them as parameters.

```

[PexMethod]
public void WithdrawTest(string url, int account, int amount) {

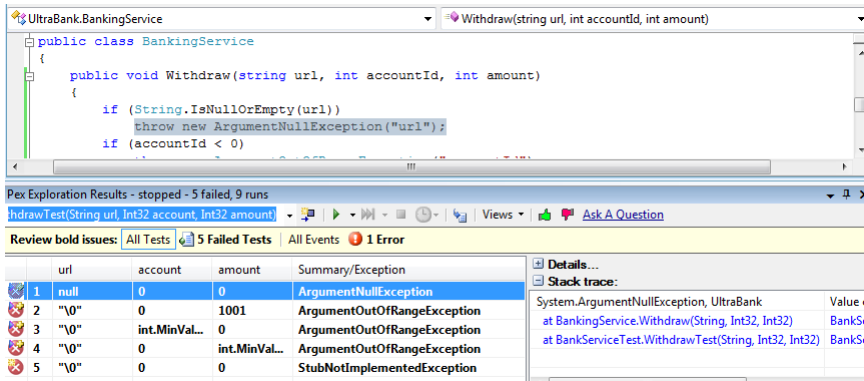
```

```

    ...
}

```

Note that since we changed the signature of the `Withdraw` method, we need to delete the generated unit tests. Pex provides a menu item to delete previously generated unit tests. Simply right-click in the class and select **Pex|Delete Generated Unit Tests In Class**. We then run Pex on the test method.



As expected, Pex generated multiple test cases to cover the different code paths in the `Withdraw` method. Since Pex does not make any assumption about correctness, it flagged the execution that triggered the parameter validation code as failures. Let us annotate the test so that Pex automatically filters those cases as *negative unit tests*. We do this in two steps:

1. we add a `[PexClass(typeof(BankService))]` attribute to the test class that specifies that we are testing the `BankService` class. Pex uses this 'hint' in various places.
2. for acceptable exception types, we add a `[PexAllowedExceptionFromTypeUnderTest(...)]` attribute. This attribute specifies that if an exception is raised from the type under test and with the specified type, it should be flagged as an expected exception.

```

[TestClass]
[PexClass(typeof(BankService))]
public partial class BankServiceTest
{
    [PexMethod]
    [PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentNullException))]
    [PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentOutOfRangeException))]
    public void WithdrawTest(string url, int account, int amount)
    {

```

We run Pex again. The tests covering the argument validation code are now flagged as passing test, but we still have a `StubNotImplementedException`.

	url	account	amount	Summary/Exception
1	null	0	0	ArgumentNullException
2	"\0"	0	1001	ArgumentOutOfRangeException
3	"\0"	int.MinValue	0	ArgumentOutOfRangeException
4	"\0"	0	int.MinValue	ArgumentOutOfRangeException
5	"\0"	0	0	StubNotImplementedException

In the newer version of the test, we added a call to the `IAccount.Balance` property getter that still needs to be stubbed. An account could have any positive balance, so we want to add a parameter to test for the balance and tell Pex that it should be positive. The input data can be 'shaped' through the use of the `PexAssume.IsTrue` method; Pex will ensure that the inputs do not violate assumptions.

```
public void WithdrawTest(string url, int account, int amount,
    int balance)
{
    PexAssume.IsTrue(balance >= 0);
    ...
    GetAccountByIdInt32 = _id => new SIAccount()
    {
        BalanceGet = () => balance,
    ...
}
```

We finally run Pex again on the test and end up with the final test suite (the `InvalidOperationException` needs to be allowed too).

	url	account	amount	balance	Summary/Exception	Error Message
1	null	0	0	0	ArgumentNullException	Value cannot be
2	"\0"	0	0	0		
3	"\0"	0	8	0	InvalidOperationException	no funds
4	"\0"	0	1001	0	ArgumentOutOfRangeException	Specified argume
5	"\0"	0	int.MinValue	0	ArgumentOutOfRangeException	Specified argume
6	"\0"	int.MinValue	0	0	ArgumentOutOfRangeException	Specified argume

9 Further Reading

In this tutorial, we've given a glimpse at how to use *stub types* and *mole types* to isolate unit tests and use Pex to write parameterized unit tests that achieve high code coverage. Further tutorials are available at <http://research.microsoft.com/stubs> and <http://research.microsoft.com/pex>.

References

- [1] A. Hunt, D. Thomas, and M. Hargett. *Pragmatic Unit Testing in C# with NUnit*. Pragmatic Bookshelf, 2 edition, August 2007.
- [2] Jonathan de Halleux and Nikolai Tillmann. Stubs, Lightweight Test Stubs and De-tours for .NET. <http://research.microsoft.com/pex/articles/stubs.pdf>.

- [3] Pex development team. Pex. <http://research.microsoft.com/Pex>, 2008.
- [4] 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.
- [5] 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.