

Stubs
Lightweight Test Stubs And Moles for .NET

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

Preliminary Draft
Copyright Microsoft Corporation.

September 15, 2009

Abstract

In software development, the notion of a *test stub* refers to dummy implementation that can replace a possibly complex component to facilitate testing. While the idea of stubs is simple, most existing frameworks that can help to create and maintain dummy implementations are actually quite complex themselves.

We have developed a new lightweight framework, which we simply call **Stubs**, that generates *stub types* for .NET interfaces and non-sealed classes. In this framework, a stub of type T provides a default implementation of each virtual member of T , and a mechanism to dynamically specify a custom implementation of each member (optionally, stubs can also be generated for non-abstract virtual members). The stub types are generated as C# code. The framework relies solely on delegates to dynamically specify the behavior of stub members.

While *stub types* work for virtual methods, it does not work for static methods or non-virtual methods. To address these cases, the **Stubs** framework also generates *mole types* for any .NET type. A mole of type T may provide an alternative implementation for each non-abstract member of T , and the framework will redirect the method of T to the mole implementation. Moles can redirect any public methods, including static methods and non-virtual methods.

Stubs supports .NET 2.0 and higher; it integrates with Visual Studio 2008 and higher.

Contents

1	Introduction	3
2	Stub Types	4
2.1	Example: Stubbing the file system	4
2.1.1	Comparison to existing frameworks	6
2.2	Stubs Basics	8
2.2.1	Methods	8
2.2.2	Properties	8
2.2.3	Events	9
2.2.4	Generic Methods	10
2.2.5	Partial Stubs	10
2.2.6	Recursive Stubs	11
2.3	Stubs Configuration	12
2.4	Code generation and naming conventions	12
2.5	Debugging Experience	13
2.6	Stubbing concrete classes and virtual methods	13
2.7	Limitations	13
2.8	Advanced Topics	13
2.8.1	Stubs with state	13
2.8.2	Changing the fallback behavior	14
2.8.3	Implementing a fallback behavior	15
3	Mole Types	16
3.1	Example: Stubbing A Sealed Class	16
3.2	Moles requirements	17
3.3	Attaching Moles	17
3.4	Code generation and naming conventions	18
3.5	Limitations	18
4	Examples of Stubs and Moles usage	18
4.1	Windows Communication Foundations	18
5	Code Generation Outside of Visual Studio	19
5.1	Command Line	19
5.2	MSBuildBuild Integration (Advanced)	19
A	.stubx Schema	20

1 Introduction

In software testing, it is often desirable to test individual components in isolation. It makes testing more robust and scalable. To this end, a common approach is to use dummy implementations – so called *test stubs* – for components that are not currently under test. While the idea of stubs is quite simple, most existing frameworks that can help to create and maintain dummy implementations are actually quite complex themselves. Some frameworks use dynamic code generation at test time to create dummy implementations on the fly. Other frameworks use remoting features to simulate interactions with stubbed components.

All those approaches have in common that instead of making testing simpler (which was the idea of stubs), they actually add an overhead that is often not negligible: Code generation at runtime and dynamic message interception impose significant runtime cost. And when an error occurs, debugging may be complicated, as it is sometimes not clear whether the error is an error in the infrastructure that supports stubbing, or whether it is an actual error of the user-written test or product code. Finally, dynamic white box test generation tools such as Pex [8], which rely on tracing control- and dataflow tracing at runtime, not only have to analyze the test and product code, but also the stub framework.¹

To address these issues, we have developed a new lightweight framework for .NET, which we aptly call **Stubs**. It generates *stub types* for .NET interfaces and abstract classes by straight-forward code generation at compile time.

Stubs is a framework for defining and generating *stubs* for interfaces and non-sealed classes. A *stub* of the type T provides a default implementation of each virtual member of T , i.e. virtual or abstract methods, properties, and events. The default behavior can be dynamically customized for each member. Such a stub is realized by a distinct type which is generated by the framework as C# code. As a result, all stubs are strongly typed.

The framework uses delegates to dynamically customize the behavior of individual stub members. It is a very lightweight framework; it does not provide advanced declarative verification features found in other mock frameworks, such as Moq [6], Rhino Mocks [1], NMock [7] or Isolator [9]. In that sense, our framework is really a *stub* framework, and not a *mock* framework, following the famous argument of Martin Fowler [5].

While *stub types* work for virtual methods, it does not work for static methods or non-virtual methods. To address these cases, the **Stubs** framework also generates *mole types* for any .NET type. A mole of type T may provide an alternative implementation for each non-abstract member of T , and the framework will redirect the method of T to the mole implementation. Under the cover, the **Stubs** framework uses a profiler to rewrite the method bodies to be able to redirect any public method, including static methods and non-virtual methods.

Stubs supports .NET 2.0 and higher; **Stubs** integrates with Visual Studio 2008 and higher. Although the stub and mole types are generated as C# code, the framework can

¹It was the need for a stub framework that can be effectively used with Pex that initially motivated us to create a new, lightweight framework.

generate stubs and moles for types defined in any .NET language.

The idea using delegates to stub or mock types is not new; others have proposed it earlier [3, 4].

2 Stub Types

In this section, we will focus on the *stub types*.

2.1 Example: Stubbing the file system

Let's start this tutorial from a motivating example: an interface that creates an abstraction from the file system. To keep things simple, let's assume it has only one method.

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

In production code, this interface would be probably be implemented using the `System.IO.File` class.

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

For unit testing purposes, one wants to avoid the physical file system and simulate it through in-memory streams for example. A simple way to implement `IFileSystem` would be to provide a default implementation of each member and override the behavior of a subset of members. The downside of this approach is that each test would require a customized implementation of `IFileSystem` and a lot of members that are not used in the test would still have to be generated and clutter the code.

The approach selected by **Stubs** is to use *delegates*, i.e. managed function pointers, to provide custom implementations of interface members.

The following is an example of the code generated by **Stubs** for `IFileSystem`. The stubbed method calls a user-defined delegate, if provided, and otherwise falls back to a fallback behavior: to throw an exception that indicates that no custom behavior was defined. (This fallback behavior can be customized as well.)

```
class SIFileSystem
    : StubBase
    , IFileSystem {
    // the user can assign this delegate
    public Func<string, string> ReadAllText;
    // stub of IFileSystem.ReadAllText(string)
    string IFileSystem.ReadAllText(string fileName) {
        var fcn = this.ReadAllText;
        if (fcn != null)
            return fcn(fileName); // user-provided behaviors
    }
}
```

```

        else
            // throw!
            return
                this.FallbackBehavior
                    .Result<SIFileSystem, string>(this);
    }
}

```

Let's see how this works in a sample unit test. We are writing a simple helper method that reads the contents of a file, and checks if the file is empty (this is just an example, there are better ways to do this):

```

static class FileHelper {
    public static bool IsEmpty(IFileSystem fs, string f) {
        var content = fs.ReadAllText(f);
        return !String.IsNullOrEmpty(content);
    }
}

```

We start by writing a test that checks that if the content is "hello", then `IsEmpty` returns false. To simulate the behavior of `ReadAllText`, we set a custom `ReadAllText` delegate.

```

[TestMethod]
public void FileIsNotEmpty() {
    // arrange
    var fileSystem = new SIFileSystem();
    var file = "foo.txt";
    var content = "hello world";

    // ReadAllText should only be called for "foo.txt"
    // and always returns "hello",
    fileSystem.ReadAllText = fileName => {
        Assert.AreEqual(fileName, file);
        return content;
    };

    // act
    bool result = FileHelper.IsEmpty(fileSystem, file);

    // assert
    Assert.IsFalse(result);
}

```

The important point in this example is that we've attached a *behavior* to the `ReadAllText` method by assigning to the `ReadAllText` field:

```

fileSystem.ReadAllText = fileName => {
    Assert.AreEqual(fileName, file);
    return content;
};

```

In this example, we use the C# 3.0 lambda syntax (the `(...)=>...` expression) which provides an elegant way to write short anonymous methods that can be used as delegates. C# 2.0 also provides a way to define such anonymous delegates:

```
fileSystem.ReadAllText = delegate(string fileName) {  
    Assert.AreEqual(fileName, file);  
    return content;  
};
```

Remember that it is really just a short-hand notation for an implicitly created closure, similar to the following long form. This version makes it explicit that a delegate is really a managed function pointer.

```
var c = new Closure();  
c.file = file;  
c.content = content;  
fileSystem.ReadAllText = c.ReadAllText;
```

The closure class could be defined as follows.

```
class Closure {  
    public string file, content;  
    public string ReadAllText(string fileName) {  
        Assert.AreEqual(fileName, this.file);  
        return this.content;  
    }  
}
```

2.1.1 Comparison to existing frameworks

There are already many mocking frameworks for .NET [6, 1, 7]. The following snippet is taken from the Moq [6] home page; Moq is a popular mocking framework. The sample shows how expression trees can be used to specify behavior for a dynamically generated interface implementation.

```
public void MoqDemo() {  
    var mock = new Mock<ILoveThisFramework>();  
  
    // WOW! No record/reply weirdness?! :)  
    mock.Setup(  
        framework => framework.ShouldDownload(It.IsAny<Version>()))  
        .Callback(  
            (Version version) =>  
                Console.WriteLine("Someone wanted version {0}!!!", version))  
        .Returns(true)  
        .AtMostOnce();  
  
    // Hand mock.Object and exercise it,  
    // like calling methods on it...  
    ILoveThisFramework lovable = mock.Object;  
    bool download =
```

```

        lovable.ShouldDownload(new Version("2.0.0.0"));

mock.VerifyAll();

// You can also verify a single expectation
// you're interested in
// This checks that the given method
// was indeed called with that value
mock.Verify(
    framework
        => framework.ShouldDownload(new Version("2.0.0.0"))
    );
}

```

Using **Stubs**, this example can be rewritten as follows:

```

public void StubsDemo() {
    var stub = new SLoveThisFramework();

    // Just attach a delegate to add a custom behavior
    int callCount = 0;
    stub.ShouldDownload = version => {
        Console.WriteLine("Someone wanted version {0}!!!", version);
        Assert.IsTrue(callCount++ < 1); // at most once
        return true;
    };

    // simply cast the stub to the interface
    ILoveThisFramework lovable = stub;
    bool download = lovable.ShouldDownload(new Version("2.0.0.0"));
}

```

The important difference between **Stubs** and other mock frameworks are

- Many mock framework rely on dynamic code generation (`Reflection.Emit`), transparent proxy objects (intended for remoting), or C# 3.0 expression trees to let users define custom behaviors at runtime. **Stubs** pre-generates straightforward source code for stubs beforehand, and it relies solely on delegates to customize behavior.
- Mock frameworks provide facilities to set expectation and verify them (`mock.Verify(...)`). **Stubs** provide no such facilities. However, with a few lines of delegate code together with closures to maintain state, one can often achieve a similar effect. In other words, while advanced mock framework often provide facilities for *declarative* specifications of behavior, **Stubs** doesn't, and it promotes a more *imperative* style of specifying mocked behavior.
- Some advanced mock frameworks support mocking of or static and non-virtual methods, either by requiring a custom profiler at runtime, or by rewriting the product `.dll` on disk. **Stubs** only supports mocking of interfaces and abstract methods.

2.2 Stubs Basics

2.2.1 Methods

As described in the `IFileSystem` example, methods may be stubbed by attaching a delegate with the same signature to their backing field.

For example, given the following `IFoo` interface and method `Bar`,

```
interface IFoo {  
    int Bar(string value);  
}
```

we attach a stub to `Bar` that always returns 1:

```
var stub = new SFoo();  
stub.Bar = (value) => 1;
```

Internally, the implementation of `Bar` simply calls the user-defined delegate if any, or executes the fallback behavior:

```
class SFoo  
    : StubBase  
    , IFoo {  
    public Func<string, int> Bar;  
    int IFoo.Bar(string value) {  
        var sh = this.Bar;  
        if (sh != null)  
            return sh(value);  
        else  
            return  
                this.FallbackBehavior  
                    .Result<SFoo, int>(this);  
    }  
}
```

2.2.2 Properties

Property getters and setters are exposed as separate delegates and can be stubbed separately.

For example, let us consider the `Bar` property of `IFoo`

```
interface IFoo {  
    int Bar { get; set; }  
}
```

We attach delegates to the getter and setter of `Bar` to simulate an auto-property:

```
var stub = new SFoo();  
int bar = 5;  
stub.BarGet = () => bar;  
stub.BarSet = (value) => bar = value;
```

When a property has a setter and a getter and their delegate fields have not been set, **Stubs** may automatically generate a backing field behavior. When **Stubs** generates the backing field delegates, it queries to fallback behavior for an initial value.

```
public void AttachBackingFieldToBar()
{
    int initialValue;
    if (this.BarGet == null &&
        this.BarSet == null &&
        this.FallbackBehavior.TryGetPropertyvalue(this, out initialValue))
    {
        var field = new StubValueHolder<int>(initialValue);
        this.BarGet = field.GetGetter();
        this.BarSet = field.GetSetter();
    }
}
```

2.2.3 Events

Events are exposed as delegate fields. As a result, any stubbed event can be raised simply by invoking the event backing field. Let us consider the following interface to stub,

```
interface IWithEvents {
    event EventHandler SomeEvent;
}
```

To raise the `SomeEvent` event, we simply invoke the backing delegate:

```
var withEvents = new SIWithEvents();
// raising SomeEvents
withEvents.SomeEvents(withEvents, EventArgs.Empty);
```

The implementation is quite small since it simply exposes the backing delegate field publicly:

```
public EventHandler SomeEvents;
event EventHandler IWithEvents.SomeEvent
{
    add {
        this.SomeEvents =
            (EventHandler)Delegate.Combine(this.SomeEvents, value);
    }
    remove {
        this.SomeEvents =
            (EventHandler)Delegate.Remove(this.SomeEvents, value);
    }
}
```

2.2.4 Generic Methods

It is possible to stub generic methods by providing a delegate for each desired instantiation of the method.

For example, given the following interface containing a generic method,

```
interface IGenericMethod {  
    T GetValue<T>();  
}
```

one could write a test that stubs the `GetValue<int>` instantiation:

```
[TestMethod]  
public void GetValue() {  
    var stub = new SIGenericMethod();  
    stub.GetValue<int>(() => 5);  
  
    IGenericMethod target = stub;  
    Assert.AreEqual(5, target.GetValue<int>());  
}
```

If the code was to call `GetValue<T>` with any other instantiation, the stub would simply call the fallback behavior.

Internally, a dictionary of instantiation is stored and maintained by the stub instance:

```
class SIGenericMethod  
    : StubBase  
    , IGenericMethod {  
    // the set of implementations for GetValues  
    StubDictionary getValues;  
    // stores an instantiated stub delegate in the dictionary  
    public void GetValue<T>(Func<T> stub) {  
        this.getValues = StubDictionary.Concat(this.getValues, stub);  
    }  
    // stub of GetValue<T>  
    T IGenericMethod.GetValue<T>() {  
        Func<T> stub;  
        if (this.getValues.TryGetValue<Func<T>>(out stub))  
            return stub();  
        else  
            return this.FallbackBehavior.Result<SIGenericMethod, T>(this);  
    }  
}
```

2.2.5 Partial Stubs

Partial stubs occur when stubbing classes and allow to stub a part of the members only (they follow the same ideas of Partial Mocks ??). When a *virtual* member is not stubbed, the base method is called instead of the fallback behavior. The partial

stub mode can be turned on through the `CallBase` property, which stubs of classes implement (from the `IPartialStub` interface).

For example, given a class with a `GetName` *virtual* method,

```
class Base {
    public virtual string GetName () {
        return "joe";
    }
}
```

we specify to the stub that it should call the base implementation by setting the `CallBase` property:

```
var stub = new SBase() {
    CallBase = true
};

// calls the base implementation
Assert.AreEqual("joe", stub.GetName());
```

Internally, the stub implementation first uses a user-provided delegate if any, then the base implementation if `CallBase` is true, then call the fallback behavior:

```
public Func<string> GetNameStub;
public override string GetName ()
{
    var sh = this.GetNameStub;
    if (sh != null) return sh();
    else if (this.CallBase)
        return base.GetName();
    else
        return this.FallbackBehavior
            .Result<SBase, string>(this);
}
```

2.2.6 Recursive Stubs

A common scenario with stubs or mocks is to access a nested element, through a chain of property calls:

```
if (parent.Child.Name == "joe")
    throw new Exception();
```

In such case, it would be tedious to instantiate each stub at each property invocation. Therefore, **Stubs** provides helper methods, tagged as `AsStub` to instantiate the desired stub, store as the result of the getter and returns it.

```
var parent = new SIParent();
parent.ChildGetAsStub<SIChild>().NameGet = () => "joe";
```

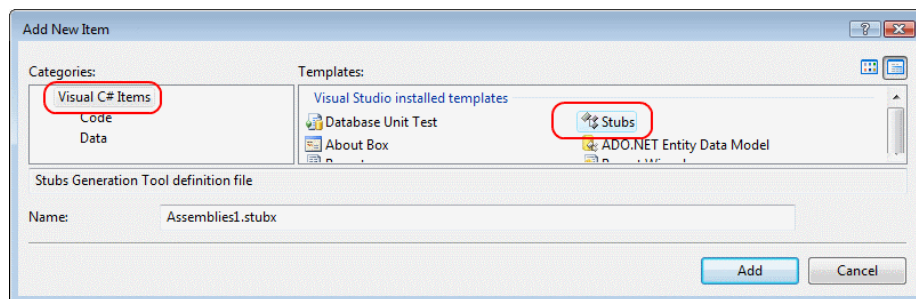
2.3 Stubs Configuration

The generation of stubs is configured in an XML file, that has the `.stubx` file extension. The framework has a file generator associated to this file extension, which runs when the project is loaded or whenever the `.stubx` file is saved.

The XML format is documented in Section A. The following examples illustrates how to stub types defined in `FileSystem.dll`.

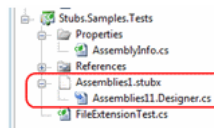
```
<?xml version="1.0"?>
<Stubs xmlns="http://schemas.microsoft.com/stubs/2008/">
  <!-- generate stubs from the project FileSystem -->
  <Assembly Name="FileSystem" />
</Stubs>
```

A new `.stubx` file can be created from the **Add New Item** dialog window.



2.4 Code generation and naming conventions

The generated code gets added to a designer file below the `.stubx` file.



The **Stubs** Visual Studiopackage monitors builds events and automatically forces the regeneration of stubs for project that have been updated. For efficiency reasons, this automatic update only works for `.stubx` files that are in the top level folder of the project, to avoid walking large nested project structures. This automatic update creates a smooth experience when writing code in a test driven development style [2].

Generated stubs are nested in sub-namespaces, i.e. interfaces from the `System` namespace will be generated in the `System.Stubs` namespace. The name of generated stubs is constructed by prefixing the basic type name with capital `S`.

The generated file should not be edited.

2.5 Debugging Experience

The stubs generated by **Stubs** are designed to provide a smooth debugging experience. By default, the debugger is instructed to step over any generated code, and thus it should step directly into the custom member implementations that were attached to the stub.

2.6 Stubbing concrete classes and virtual methods

By default, only interfaces and abstract classes are stubbed. This behavior can be changed, through the stub configuration file, to stub non-sealed classes.

```
<?xml version="1.0"?>
<Stubs xmlns="http://schemas.microsoft.com/stubs/2008/">
  ...
  <StubGeneration>
    <TypeFilter NonSealedClasses="true" />
  </StubGeneration>
</Stubs>
```

2.7 Limitations

The current implementation of **Stubs** has several limitations. These limitations are not inherent to the approach, and might be resolved in future releases.

- The code generator only emits C# code. However, you can create stubs for types defined in any .NET language. Just use a separate C# project in which you generate the stubs, and reference it in your project.
- **Stubs** only supports a limited number of method signature; up to 10 arguments, where the last argument can be an `out` or `coref` argument. Method signatures with pointers are not supported.
- Sealed classes or static methods cannot be stubbed since it relies on virtual method dispatch. To such cases, use *moles* as described in section 3.

2.8 Advanced Topics

2.8.1 Stubs with state

Closures can encapsulate mutable state. Let us consider a test for the file `Copy` method, which reads from a file, using `ReadAllText` and writes to a target file using `WriteAllText`.

```
[TestMethod]
public void CopyAndCheckContent() {
    var fs = new SFileSystem();
    string source = "source.txt";
    string target = "target.txt";
    string content = "foo";
```

```

// prepare stub scenario :
// copying content from source to target
fs.Exists = f => f == source;
fs.ReadAllText = f => {
    Assert.AreEqual(f, source);
    return content;
};
string actualContent = null;
fs.WriteAllText = (f, c) => {
    Assert.AreEqual(f, target);
    actualContent = c;
};
// act
FileHelper.Copy(fs, source, target);
// assert contents are equal
Assert.AreEqual(content, actualContent);
}

```

Here, the custom `WriteAllText` stub sets the `actualContent` variable when someone writes to `target`, and at the end of the test, we assert that the correct content was written.

```

string actualContent = null;
fs.WriteAllText = (f, c) => {
    Assert.AreEqual(f, target);
    actualContent = c;
};
// act
FileHelper.Copy(fs, source, target);
// assert contents are equal
Assert.AreEqual(content, actualContent);

```

(Remember that all variables used in anonymous delegates and lambda expressions are put on the heap in a closure objects; that's why the stub can easily communicate state changes with the test code.)

2.8.2 Changing the fallback behavior

Each generated stub type holds an instance of the `IStubBehavior` interface (through the `IStub.FallbackBehavior` property). The fallback behavior is called whenever a client calls a member with no attached custom delegate. If the fallback behavior has not been set, it will use the instance returned by the `StubFallbackBehavior.Current` property. By default, this property returns a behavior that throws a `StubNotImplementedException` exception.

The fallback behavior can be changed at any time by setting the `FallbackBehavior` property on any stub instance. For example, the following snippet changes to a fallback behavior that does nothing or returns the default value of the return type (i.e. `default(T)`):

```

var stub = new SFileSystem();

```

```
// return default(T) or do nothing
stub.FallbackBehavior = StubFallbackBehavior.DefaultValue;
```

The fallback behavior can also be changed globally for all stub objects for which the fallback behavior has not been set by setting the `StubFallbackBehavior.Current` property:

```
// change default stub for all instantiations of
// SFileSystem where the default stub has not been set
StubFallbackBehavior.Current =
    StubFallbackBehavior.DefaultValue;
```

Pex also provides a fallback behavior (in `Microsoft.Pex.Stubs.dll`) that makes a 'choice' whenever a new value is needed:

```
// set Pex on this particular instance
stub.FallbackBehavior = PexChooseStubBehavior.Instance;
// set Pex for all future stub instances
StubFallbackBehavior.Current = PexChooseStubBehavior.Instance;
```

2.8.3 Implementing a fallback behavior

Other fallback behaviors can be achieved by implementing the `IFallbackBehavior` interface. For example, the following snippet implements a fallback behavior that returns the default value of a given type.

```
[Serializable]
[DebuggerNonUserCode]
[__Instrument]
class DefaultValueStub
    : IStubBehavior {
    public TResult Result<TStub, TResult>(TStub me)
        where TStub : IStub {
        return default(TResult);
    }
    public void VoidResult<TStub>(TStub me)
        where TStub : IStub {}
    public void ValueAtReturn<TStub, TValue>(
        TStub me, out TValue value)
        where TStub : IStub {
        value = default(TValue);
    }
    public void ValueAtEnterAndReturn<TStub, TValue>(
        TStub stub, ref TValue value)
        where TStub : IStub { }
    public bool TryGetPropertyValue<TStub, TValue>(
        TStub stub, out TValue value)
        where TStub : IStub {
        value = default(TValue);
        return true;
    }
}
```

```
}
```

It is advised to avoid keeping state in the fallback behavior and, for performance reasons, to store it as a singleton.

3 Mole Types

In this section, we will focus on the *mole types*.

3.1 Example: Stubbing A Sealed Class

Let us consider a sample of code that needs to be tested.

```
public class User {
    public int ID { get; set; }
}
public static class Warehouse {
    public static User CreateUser() {
        var id = Singleton.GetInstance().NextID();
        if (id < 0)
            return null;
        return new User { ID = id };
    }
}
```

The warehouse allocates users and uses a singleton, `Singleton`, to provide unique identifiers.

```
public sealed class Singleton {
    // there shall be only one
    public static Singleton GetInstance() {
        ...
    }
    // returns some shared resource
    public int NextID() {
        ...
    }
}
```

The logic of `CreateUser` relies on `Singleton.NextID()` which is a non-virtual method from a sealed class. Typically, this method cannot easily be stubbed because one simply cannot override it. In many cases, the `Singleton` type cannot be refactored as it is part of a 3-rd party framework over which the developer has no control. In order to test this method, we need to be able to force `NextID()` to return precise values.

This is where *moles* should be used. *Mole types* provide a mechanism to detour any .NET method to a user defined delegate. Their usage is quite similar to *stub types*: *mole types* get code-generated by the **Stubs** generator and they use delegates, which we call *mole*, to specify the new method implementations. The test below shows how to use the *mole type* of `Singleton`, i.e. `MSingleton`, to provide a custom implementation `Singleton.NextID`:

```

// instantiate a mole of Singleton
var mole = new MSingleton();

// hook to the method to redirect
int expectedID = 101;
mole.NextID = (singleton) => expectedID;

// turn on the mole redirections
using (mole.Attach()) {
    // calls to Singleton.NextID are redirected
    var user = Warehouse.CreateUser();
    Assert.AreEqual(expectedID, user.ID);
}

```

It is important to note that each *mole* needs to be 'attached' in order to activate its mole. The `Attach` method returns a disposable instance that defines the lifetime of the mole.

3.2 Moles requirements

Under the hood, *moles* use callbacks that were injected at runtime in the method MSIL bodies by the Pex profiler. Therefore, *moles* require to be run under the Pex profiler. With Visual Studio Unit Test one can simply add the `[HostType("Pex")]` attribute to achieve this:

```

[TestMethod]
[HostType("Pex")] // run under Pex
public void OfficesClosedOnSaturday() {
    ...
}

```

When generating unit tests from a Pex method, Pex will automatically detect that *moles* are used and will emit the attribute accordingly.

Moreover, the type being *moled* needs to be instrumented. The instrumentation can be set up using the `Pexinstrumentation` attributes. When using *moles* with Pex, this process happens naturally.

3.3 Attaching Moles

In order to activate the redirection of a *mole type*, you need to call the `Attach` method. This method returns a disposable object that defines the lifetime of the redirection (it expires when the object is disposed).

```

using (mole.Attach()) {
    ...
}

```

Multiple moles may be attached as once for convenience using the `MoleContext` helper class:

```

// will hold moles
var context = new MoleContext();
// adding moles
var singleton = context.AddMole<MSingleton>();
...
var other = new MOtherSingleton();
context.AddMole(other);
...
// attach all moles
using (context.Attach()) {
    ...
}

```

3.4 Code generation and naming conventions

Generated moles are nested in sub-namespaces, i.e. the mole of `System.DateTime` will be generated in the `System.Moles` namespace. The name of generated stubs is constructed by prefixing the basic type name with capital `M`.

3.5 Limitations

- Moles cannot be used to rewrite constructors or external methods. This feature is in our roadmap.
- Moles cannot be used with some `microsoft` types.
- Moles require to be run under the Pex instrumentation. Currently, Pex provides a host type for the Visual Studio Unit Test framework only.

4 Examples of Stubs and Moles usage

4.1 Windows Communication Foundations

The following example shows how *mole types* are used to set up the return value of the `OperationContext.Current.HasSupportingTokens` method to `true`.

```

var moles = new MoleContext();
moles.GetMole<MOperationContext>()
    .CurrentGetAsMole<MOperationContext>()
    .HasSupportingTokensGet = (me) => true;

using (moles.Attach()) {
    if (OperationContext.Current.HasSupportingTokens)
        Console.WriteLine("ok!");
}

```

Moles can also be recursively called together. In the follow method, we access the `Channel` property of the operation context which returns an interface, `IContextChannel`.

The **Stubs**code generator automatically generated a method to return a stub of that interface:

```
var moles = new MoleContext();
moles.GetMole<MOperationContext>()
    .CurrentGetAsMole<MOperationContext>()
    .ChannelGetAsStub<SIContextChannel>()
    .StateGet = () => CommunicationState.Created;

using (moles.Attach()) {
    if (OperationContext.Current.Channel.State ==
        CommunicationState.Created)
        Console.WriteLine("ok!");
}
```

5 Code Generation Outside of Visual Studio

This section describes how to use the command line version of **Stubs**and the MSBuild-targets.

5.1 Command Line

Stubscomes with a command line application that can generate the code from existing `.stubx` files or directly from assemblies. Here are example of typical usage:

- generate stubs and moles for a particular assembly:

```
stubs.exe Assembly.dll
```

- generate stubs over an existing `.stubx` file

```
stubs.exe Assembly.stubx
```

- generate stubs under a particular namespace

```
stubs.exe Assembly.dll /nf:MyNamespace
```

- a detailed help about the usage of `stubs.exe` can be found by executing

```
stubs.exe help
```

5.2 MSBuildBuild Integration (Advanced)

Stubsalso provides custom MSBuildtargets to integrate stub generation before and after the build occurs. **Stubs**does not provide yet an integration with the property pages so the rest of this section assume that you are familiar with the MSBuildfile format.

The documentation of the MSBuildtargets is available in the `MSBuildMicrosoft.Stubs.targets` file in the installation directory.

A .stubx Schema

The framework uses an XML file to configure which and how assemblies should be stubbed. A custom tool that processes these XML files is automatically registered for the .stubx file extension.

Stubs Schema

Stubs files (.stubx) declare a list of assemblies to stub. The files must conform to the following XML schema. A new .stubx file can be created from the **Add New Item** menu.

Root Element

- Stubs : StubsElement

Target Namespace

`http://schemas.microsoft.com/stubs/2008/`

Example

```
<?xml version="1.0"?>
<Stubs xmlns="http://schemas.microsoft.com/stubs/2008/">
  <Assembly Name="..." />
</Stubs>
```

StubsElement

Description

This is the top-level container element for every `.stubx` file. The `Assembly` children define which assemblies should be stubbed.

Parents

None

Inner Text

None

Children

Sequence (min: 1, max: 1)

- `Assembly : AssemblyElement, (min: 1, max: 1)`
- `CodeStyle : CodeStyleElement, (min: 0, max:1)`
- `StubGeneration : StubGenerationElement, (min: 0, max:1)`
- `MoleGeneration : MoleGenerationElement, (min: 0, max:1)`

Attributes

Disable Disable code generation for the current file.

DisableGenerationOnBuild Disable re-generating the stubs before each build.

AssemblyElement

Description

Defines an assembly whose interfaces should be stubbed.

Parents

Stubs

Inner Text

None

Children

None

Attributes

Name (required) The assembly name for which stubs should be generated. If the name refers to a referenced project, the tool will resolve the location of the assembly automatically. You can use the `Location` attribute to specify a path to the assembly to stub.

Location Full path name of the assembly file.

Remarks

In Visual Studio, the **Stubs** addin will automatically resolve the assembly location, if the assembly is a project of the solution. The addin uses the assembly name to find the project, from which it can query the location of the assembly.

CodeStyleElement

Description

Defines formatting parameters for the code generation

Parents

Stubs

Inner Text

None

Children

FileHeader : FileHeaderElement (min:0, max:1)

Attributes

None

Example

```
<CodeStyle>  
  <FileHeader>Do not edit!</FileHeader>  
</CodeStyle>
```

FileHeaderElement

Description

Defines the header text to be added to the generated C# file.

Parents

`CodeStyle`

Inner Text

Header text that will be added as a comment to the generated file.

Children

None

Attributes

None

CodeGenerationElementBase

Description

Abstract base class for code generation parameters of stubs and moles.

Inner Text

None

Attributes

NamespaceSuffix The string suffix that will be appended to each stub namespace. The default is `.Stubs`. For example, stubs for the interfaces of the `System` namespace will be generated in the `System.Stubs` namespace.

TypeNameFormatString The format string used to create stubbed type names. Default is `S{0}`. The name of generated stubs is built from the type name by stripping a leading capital `I` (for interfaces), or by stripped a leading or trailing `Base` (for abstract classes), then applying the format string.

StubGenerationElement

Description

Defines parameters for the code generation of *stub types*.

Extends

`CodeGenerationElementBase`

Parents

`CodeGeneration`

Inner Text

None

Children

`TypeFilter : StubTypeFilterElement (min:0, max:unbounded)`

Attributes

NamespaceSuffix inherited from `CodeGenerationElementBase`. Default value is `".Stubs"`.

TypeNameFormatString inherited from `CodeGenerationElementBase`. Default value is `"S0"`

SkipVirtualMethods Disables the generation of stubs for virtual, non-abstract methods.

Remarks

All `TypeFilter` element are combined into a single conjunction.

MoleGenerationElement

Description

Defines parameters for the code generation of *mole types*.

Extends

CodeGenerationElementBase

Parents

CodeGeneration

Inner Text

None

Children

TypeFilter : MoleTypeFilterElement (min:0, max:unbounded)

Attributes

NamespaceSuffix inherited from CodeGenerationElementBase. Default value is ".Moles".

TypeNameFormatString inherited from CodeGenerationElementBase. Default value is "M0"

Remarks

All `TypeFilter` element are combined into a single conjunction.

TypeFilterElementBase

Description

Abstract base class for type filters of stubs and moles.

Inner Text

None

Children

None

Attributes

Namespace Filter to select stubbed interfaces based on their namespace.

TypeName Filter to select stubbed interfaces based on their type name.

ExcludedNamespace Filter to exclude stubbed interfaces based on their namespace.

ExcludedTypeName Filter to exclude stubbed interfaces based on their type name.

SkipObsolete Disables generation of obsolete types, i.e. marged with the `ObsoleteAttribute`, or types containing obsolete members.

Remarks

By default the filters perform a case-insensitive substring matching. More specializing matching can be done by adding a `!` (case-sensitive and full match) or `*` (case-insensitive, starts with) at the end of the filter.

For example, `fo` matches `Foo` and `FooBar`, `Foo!` matches `Foo` but not `FooBar`, `Foo*` matches `Foo` and `FooBar` but not `BarFoo`.

StubTypeFilterElement

Description

Defines a filter over the types of the assembly that would be suitable to generate stubs.

Parents

Stubs : StubGenerationElement

Inner Text

None

Children

None

Attributes

Namespace inherited from TypeFilterElementBase.

TypeName inherited from TypeFilterElementBase.

ExcludedNamespace inherited from TypeFilterElementBase.

ExcludedTypeName inherited from TypeFilterElementBase.

SkipObsolete inherited from TypeFilterElementBase.

SkipInterfaces Disables generation of stubs for interfaces.

SkipClasses Disables generation of stubs for classes (non-interfaces).

NonSealedClasses Enables generation of stubs for all non-sealed classes.

MoleTypeFilterElement

Description

Defines a filter over the types of the assembly that would be suitable to generate moles.

Parents

Moles : MoleGenerationElement

Inner Text

None

Children

None

Attributes

Namespace inherited from TypeFilterElementBase.

TypeName inherited from TypeFilterElementBase.

ExcludedNamespace inherited from TypeFilterElementBase.

ExcludedTypeName inherited from TypeFilterElementBase.

SkipObsolete inherited from TypeFilterElementBase.

References

- [1] Ayende Rahiem. Rhino Mocks. <http://ayende.com/projects/rhino-mocks.aspx>.
- [2] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.
- [3] D. Cazzuolino. Mocks, stubs and fakes: it's a continuum, Dec 2007. [accessed 7-October-2008].
- [4] S. Lambla. Why mock frameworks s..., and how to write delegate-based test doubles, Dec 2007. [accessed 19-January-2009].
- [5] Martin Fowler. Mocks aren't stubs. <http://www.martinfowler.com/articles/mocksArentStubs.html>. [accessed 11-September-2008].
- [6] Moq Team. Moq. <http://code.google.com/p/moq/>.
- [7] NMock Development Team. NMock. <http://nmock.org>.

- [8] Pex development team. Pex. <http://research.microsoft.com/Pex>, 2008.
- [9] TypeMock Development Team. Isolator. http://www.typemock.com/learn_about_typemock_isolator.html.