




# Isolate code under test with Microsoft Fakes

Article • 05/24/2023

**Applies to:**  Visual Studio  Visual Studio for Mac  Visual Studio Code

Code isolation is a testing strategy often implemented with tools like Microsoft Fakes, where the code you're testing is separated from the rest of the application. This separation is achieved by replacing parts of the application that interact with the code under test with stubs or shims. These are small pieces of code controlled by your tests, which simulate the behavior of the actual parts they're replacing.

The benefit of this approach is that it allows you to focus on testing the specific functionality of the code in isolation. If a test fails, you know the cause is within the isolated code and not somewhere else. Additionally, the use of stubs and shims, provided by Microsoft Fakes, enables you to test your code even if other parts of your application aren't functioning yet.

## Requirements

- Visual Studio Enterprise
- A .NET Framework project
- .NET Core, .NET 5.0 or later, and SDK-style project support previewed in Visual Studio 2019 Update 6, and is enabled by default in Update 8. For more information, see [Microsoft Fakes for .NET Core and SDK-style projects](#).

### Note

Profiling with Visual Studio isn't available for tests that use Microsoft Fakes.

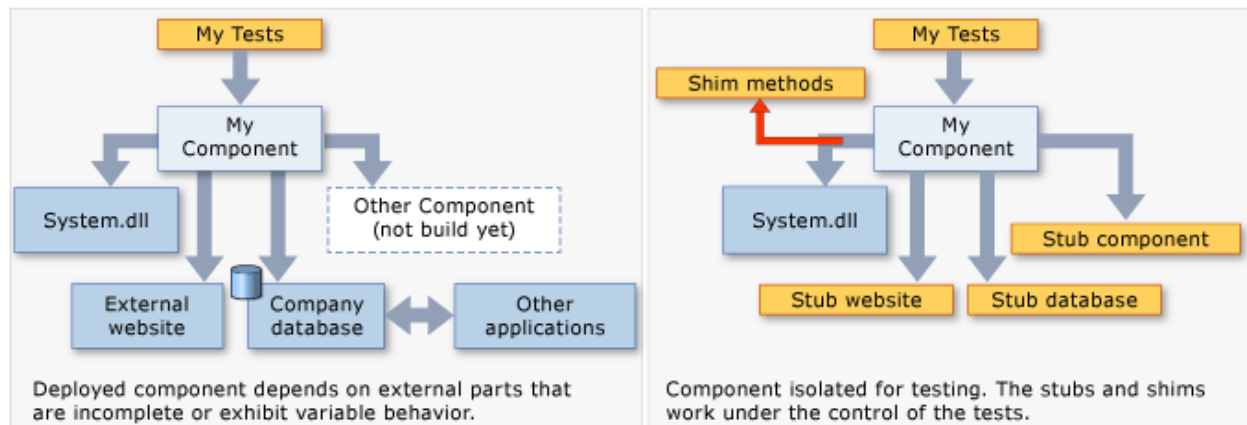
## The Role of Microsoft Fakes in Code Isolation

Microsoft Fakes plays a key role in code isolation by providing two mechanisms - stubs and shims.

- **Stubs:** These are used to replace a class with a small substitute that implements the same interface. This requires your application to be designed such that each component depends only on interfaces, not on other components.

- **Shims:** These are used to modify the compiled code of your application at runtime. Instead of making a specified method call, the application runs the shim code that your test provides. Shims can replace calls to assemblies that you can't modify, such as .NET assemblies.

Typically, stubs are used for calls within your Visual Studio solution, and shims for calls to other referenced assemblies. This is because within your solution, it's good practice to decouple the components by defining interfaces in the way that stubbing requires. However, external assemblies often don't come with separate interface definitions, so shims are used instead.



## Recommendations on When to Use Stubs

Stubs are typically used for calls within your Visual Studio solution because it's a good practice to decouple the components by defining interfaces in the way that stubbing requires. However, external assemblies, such as System.dll, typically aren't provided with separate interface definitions, so shims would be used in these cases instead.

Using stubs involves designing your application so that the different components are not dependent on each other, but only on interface definitions. This decoupling makes the application more robust and flexible, and allows you to connect the component under test to stub implementations of the interfaces for testing purposes.

In practice, you can generate stub types from the interface definitions in Visual Studio, then replace the real component with the stub in your test.

## Recommendations on When to Use Shims

While stubs are used for calls within your Visual Studio solution, shims are typically used for calls to other referenced assemblies. This is because external assemblies such as System.dll usually aren't provided with separate interface definitions, so shims must be used instead.

However, there are some factors to consider when using shims:

**Performance:** Shims run slower because they rewrite your code at runtime. Stubs don't have this performance overhead and are as fast as virtual methods can run.

**Static methods, sealed types:** You can only use stubs to implement interfaces. Therefore, stub types can't be used for static methods, non-virtual methods, sealed virtual methods, methods in sealed types, and so on.

**Internal types:** Both stubs and shims can be used with internal types that are made accessible by using the assembly attribute [InternalsVisibleToAttribute](#).

**Private methods:** Shims can replace calls to private methods if all the types on the method signature are visible. Stubs can only replace visible methods.

**Interfaces and abstract methods:** Stubs provide implementations of interfaces and abstract methods that can be used in testing. Shims can't instrument interfaces and abstract methods, because they don't have method bodies.

---

## Transitioning Microsoft Fakes in .NET Framework to SDK-Style Projects

### Transitioning your .NET Framework test projects that use Microsoft Fakes to SDK-style .NET Framework, .NET Core, or .NET 5+ projects.

You'll need minimal changes in your .NET Framework set up for Microsoft Fakes to transition to .NET Core or .NET 5.0. The cases that you would have to consider are:

- If you're using a custom project template, you need to ensure that it's SDK-style and builds for a compatible target framework.
- Certain types exist in different assemblies in .NET Framework and .NET Core/.NET 5.0 (for example, `System.DateTime` exists in `System/mscorlib` in .NET Framework, and in `System.Runtime` in .NET Core and .NET 5.0), and in these scenarios you need to change the assembly being faked.
- If you have an assembly reference to a fakes assembly and the test project, you might see a build warning about a missing reference similar to:

```
(ResolveAssemblyReferences target) ->  
warning MSB3245: Could not resolve this reference. Could not locate the  
assembly "AssemblyName.Fakes". Check to make sure the assembly exists  
on disk.  
If this reference is required by your code, you may get compilation  
errors.
```

This warning is because of necessary changes made in Fakes generation and can be ignored. It can be avoided by removing the assembly reference from the project file, because we now implicitly add them during the build.

## Running Microsoft Fakes tests

As long as Microsoft Fakes assemblies are present in the configured `FakesAssemblies` directory (The default being `$(ProjectDir)FakesAssemblies`), you can run tests using the [vstest task](#).

Distributed testing with the [vstest task](#) .NET Core and .NET 5+ projects using Microsoft Fakes requires Visual Studio 2019 Update 9 Preview `20201020-06` and higher.


## Compatibility and Support for Microsoft Fakes in Different .NET and Visual Studio Versions

### Microsoft Fakes in older projects targeting .NET Framework (non-SDK style).

- Microsoft Fakes assembly generation is supported in Visual Studio Enterprise 2015 and higher.
- Microsoft Fakes tests can run with all available Microsoft.TestPlatform NuGet packages.
- Code coverage is supported for test projects using Microsoft Fakes in Visual Studio Enterprise 2015 and higher.




### Microsoft Fakes in SDK-style .NET Framework, .NET Core, and .NET 5.0 or later projects

- Microsoft Fakes assembly generation previewed in Visual Studio Enterprise 2019 Update 6 and is enabled by default in Update 8.

- Microsoft Fakes tests for projects that target .NET Framework can run with all available Microsoft.TestPlatform NuGet packages.
- Microsoft Fakes tests for projects that target .NET Core and .NET 5.0 or later can run with Microsoft.TestPlatform NuGet packages with versions [16.9.0-preview-20210106-01](#)  and higher.
- Code coverage is supported for test projects targeting .NET Framework using Microsoft Fakes in Visual Studio Enterprise version 2015 and higher.
- Code coverage support for test projects targeting .NET Core and .NET 5.0 or later using Microsoft Fakes is available in Visual Studio 2019 update 9 and higher.

# Use stubs to isolate parts of your application from each other for unit testing

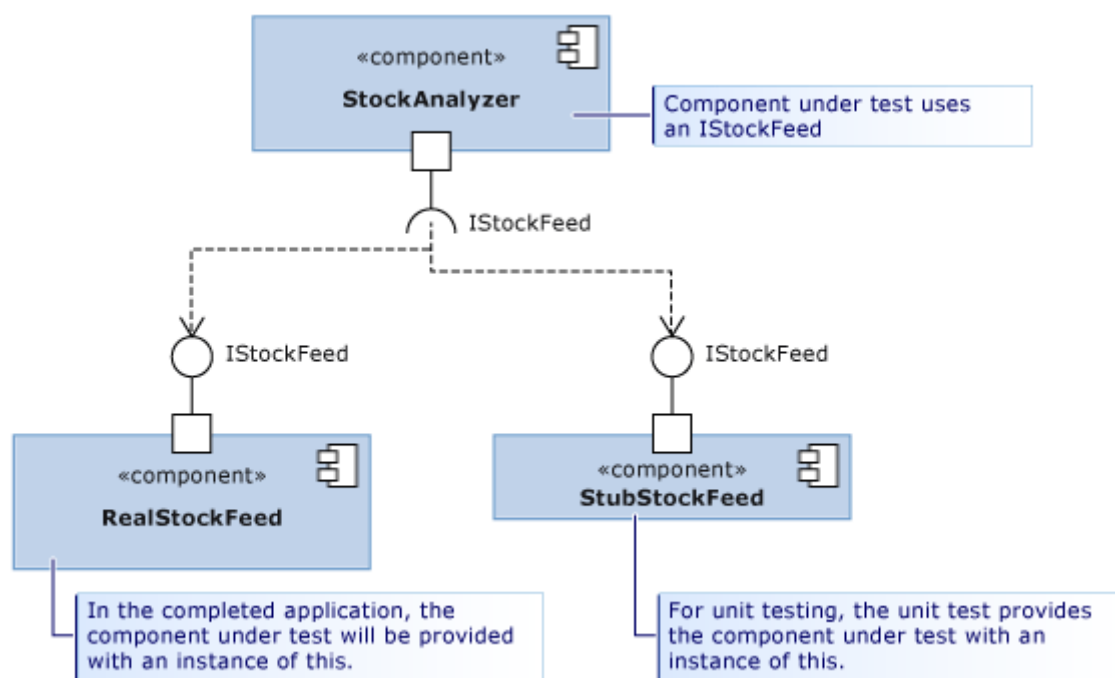
Article • 10/30/2023

Applies to:  Visual Studio  Visual Studio for Mac  Visual Studio Code

*Stub types* are an important technology provided by the Microsoft Fakes framework, enabling easy isolation of the component you're testing from other components it relies on. A stub acts as a small piece of code that replaces another component during testing. A key benefit of using stubs is the ability to obtain consistent results to make test writing easier. Even if the other components aren't yet fully functional, you can still execute tests by using stubs.

To apply stubs effectively, it's recommended to design your component in a way that it primarily depends on interfaces rather than concrete classes from other parts of the application. This design approach promotes decoupling and reduces the likelihood of changes in one part requiring modifications in another. When it comes to testing, this design pattern enables substituting a stub implementation for a real component, facilitating effective isolation and accurate testing of the target component.

For example, let's consider the diagram that illustrates the components involved:



In this diagram, the component under test is **StockAnalyzer**, which typically relies on another component called **RealStockFeed**. However, **RealStockFeed** poses a challenge

for testing because it returns different results each time its methods are called. This variability makes it difficult to ensure consistent and reliable testing of `StockAnalyzer`.

To overcome this obstacle during testing, we can adopt the practice of [dependency injection](#). This approach involves writing your code in such a way that it doesn't explicitly mention classes in another component of your application. Instead, you define an interface that the other component and a stub can implement for test purposes.

Here's an example of how you can use dependency injection in your code:

C#

```
C#  
  
public int GetContosoPrice(IStockFeed feed) =>  
    feed.GetSharePrice("C000");
```

## Stub limitations

Review the following limitations for stubs.

- Method signatures with pointers aren't supported.
- Sealed classes or static methods can't be stubbed using stub types because stub types rely on virtual method dispatch. For such cases, use shim types as described in [Use shims to isolate your application from other assemblies for unit testing](#)

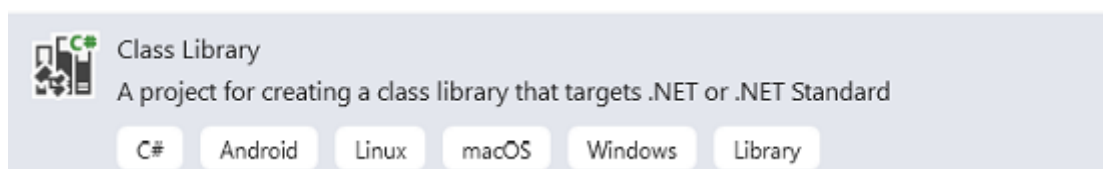
## Creating a Stub: A Step-by-Step Guide

Let's start this exercise with a motivating example: the one shown in the preceding diagram.

### Create a Class Library

Follow these steps to create a class library.

1. Open Visual Studio and create a **Class Library** project.



2. Configure the project attributes:

- Set the **Project name** to *StockAnalysis*.
- Set the **Solution name** to *StubsTutorial*.
- Set the project **Target framework** to **.NET 8.0**.

3. Delete the default file *Class1.cs*.

4. Add a new file named *IStockFeed.cs* and copy in the following interface definition:

C#

```
C#  
  
// IStockFeed.cs  
public interface IStockFeed  
{  
    int GetSharePrice(string company);  
}
```

5. Add another new file named *StockAnalyzer.cs* and copy in the following class definition:

C#

```
C#  
  
// StockAnalyzer.cs  
public class StockAnalyzer  
{  
    private IStockFeed stockFeed;  
    public StockAnalyzer(IStockFeed feed)  
    {  
        stockFeed = feed;  
    }  
    public int GetContosoPrice()  
    {  
        return stockFeed.GetSharePrice("C000");  
    }  
}
```

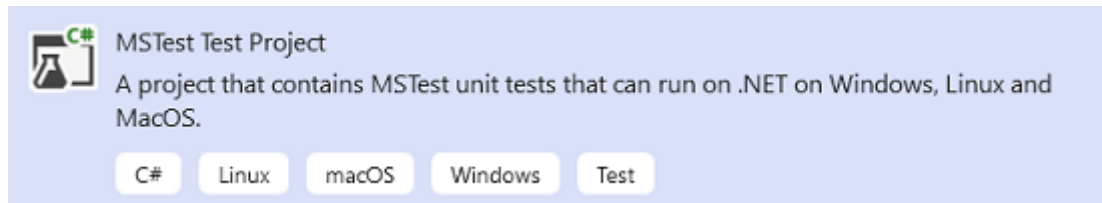
## Create a Test Project

Create the test project for the exercise.

<https://learn.microsoft.com/enus/visualstudio/test>



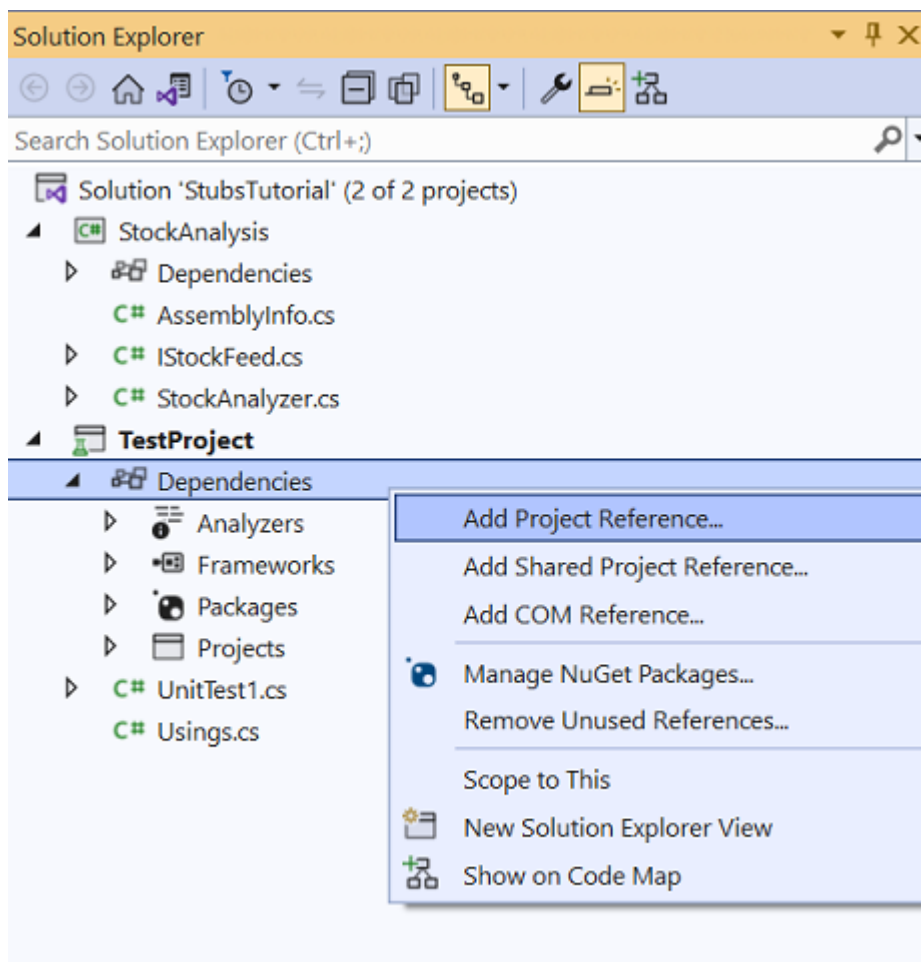
1. Right-click on the solution and add a new project named *MSTest Test Project*.
2. Set the project name to *TestProject*.
3. Set the project's target framework to **.NET 8.0**.



## Add Fakes assembly

Add the Fakes assembly for the project.

1. Add a project reference to `StockAnalyzer`.

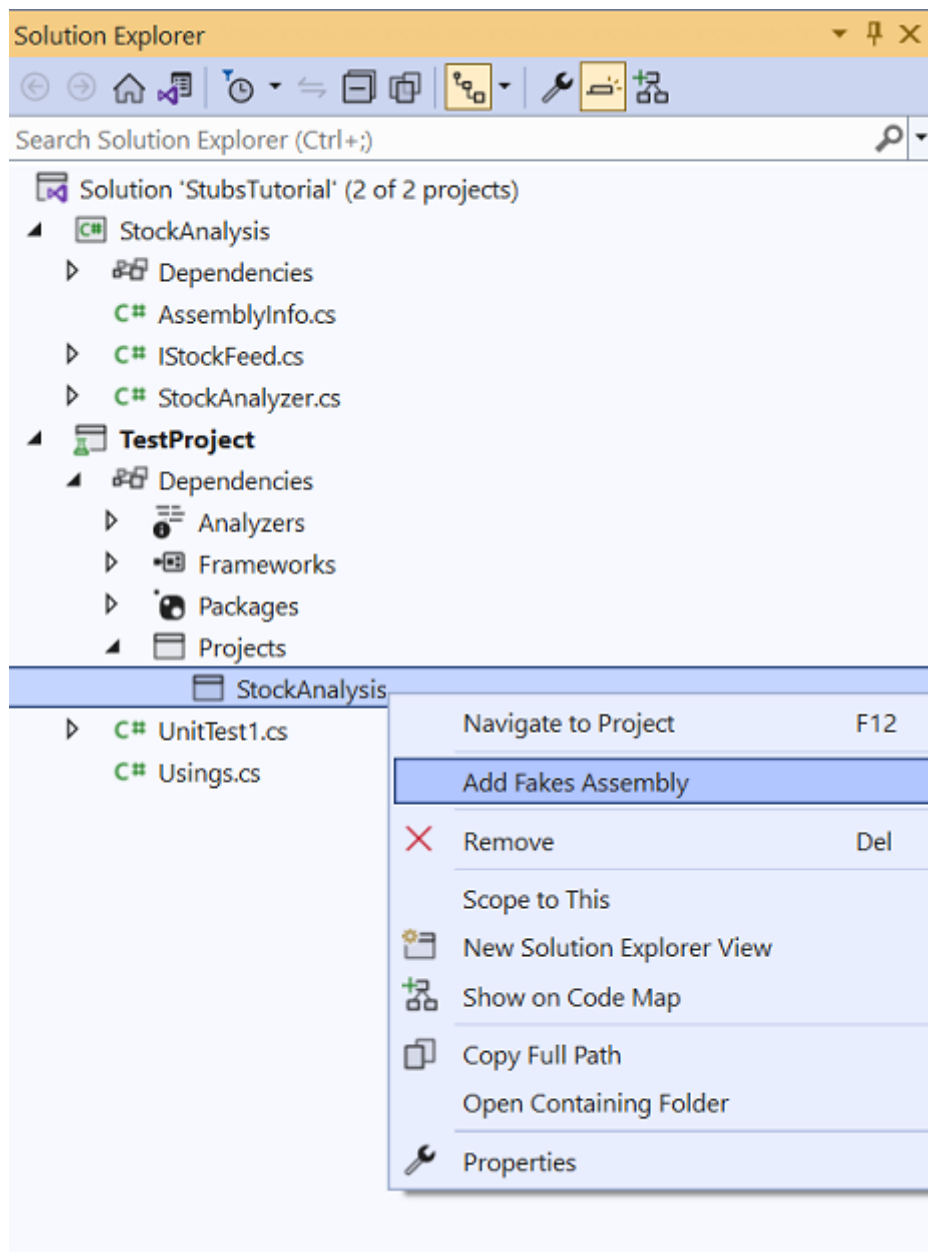


2. Add the Fakes Assembly.

a. In **Solution Explorer**, locate the assembly reference:

- For an older .NET Framework Project (non-SDK style), expand your unit test project's **References** node.

- For an SDK-style project targeting .NET Framework, .NET Core, or .NET 5.0 or later, expand the **Dependencies** node to find the assembly you would like to fake under **Assemblies**, **Projects**, or **Packages**.
  - If you're working in Visual Basic, select **Show All Files** in the **Solution Explorer** toolbar to see the **References** node.
- b. Select the assembly that contains the class definitions for which you want to create stubs.
- c. On the shortcut menu, select **Add Fakes Assembly**.



## Create a unit test

Now create the unit test.

1. Modify the default file *UnitTest1.cs* to add the following **Test Method** definition.

<https://learn.microsoft.com/enus/visualstudio/test>

C#

```

[TestClass]
class UnitTest1
{
    [TestMethod]
    public void TestContosoPrice()
    {
        // Arrange:
        int priceToReturn = 345;
        string companyCodeUsed = "";
        var componentUnderTest = new StockAnalyzer(new
StockAnalysis.Fakes.StubIStockFeed()
        {
            GetSharePriceString = (company) =>
            {
                // Store the parameter value:
                companyCodeUsed = company;
                // Return the value prescribed by this test:
                return priceToReturn;
            }
        });

        // Act:
        int actualResult = componentUnderTest.GetContosoPrice();

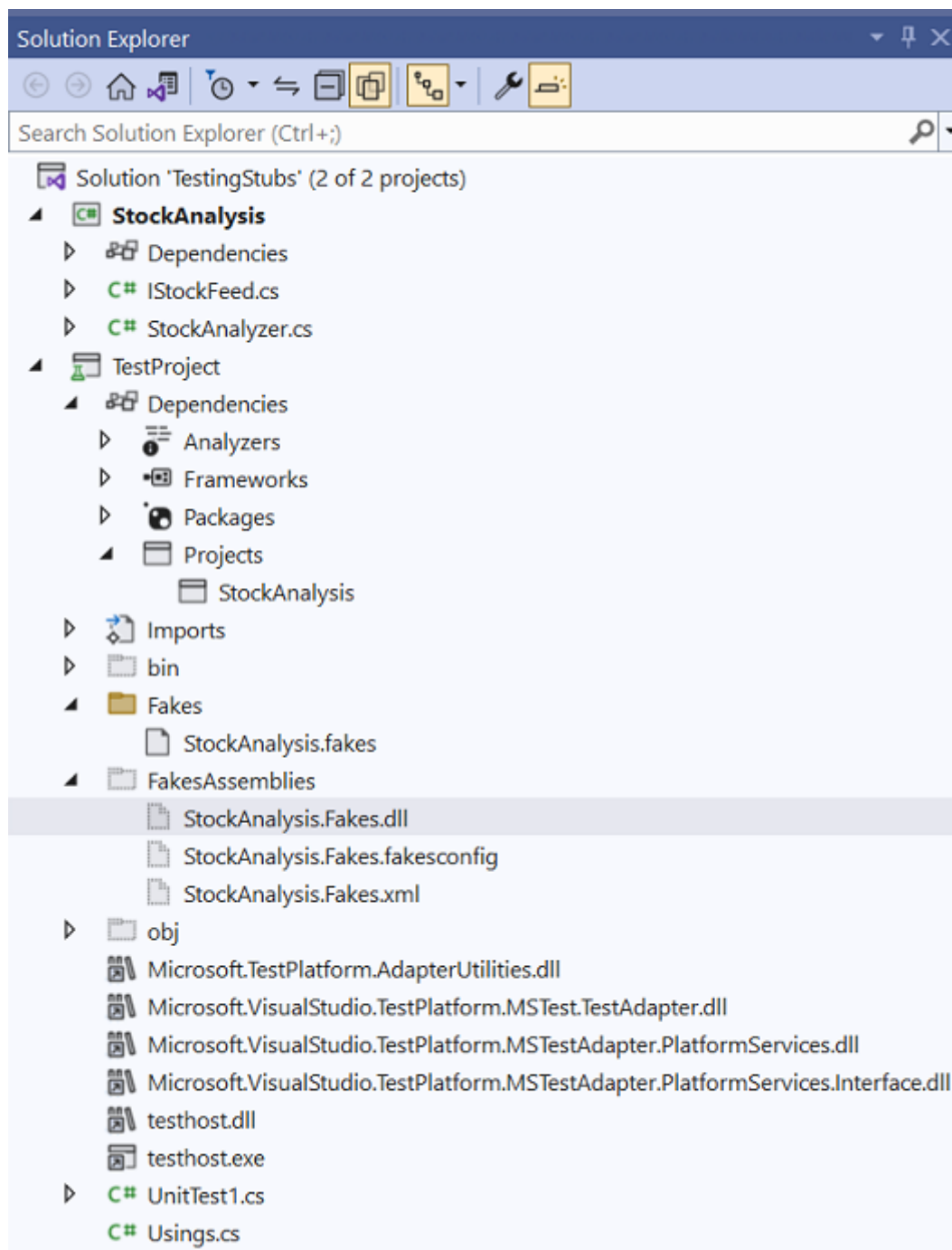
        // Assert:
        // Verify the correct result in the usual way:
        Assert.AreEqual(priceToReturn, actualResult);

        // Verify that the component made the correct call:
        Assert.AreEqual("C000", companyCodeUsed);
    }
}

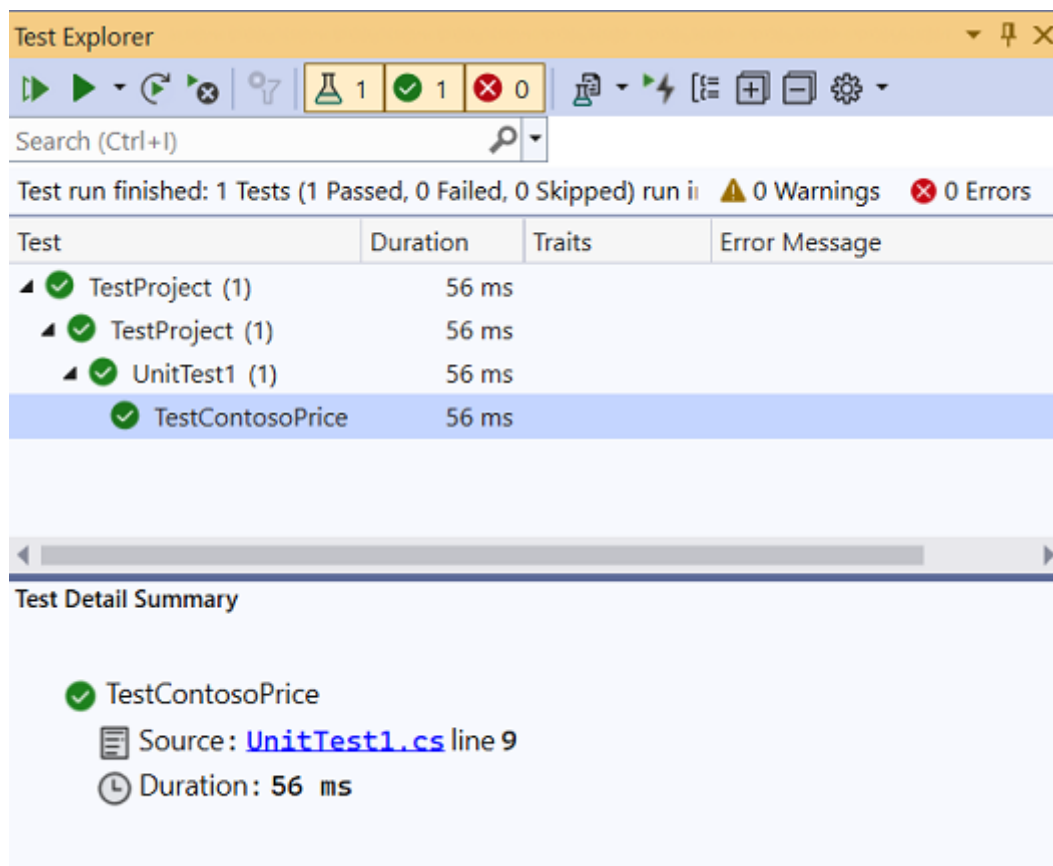
```

The special piece of magic here's the `StubIStockFeed` class. For every interface in the referenced assembly, the Microsoft Fakes mechanism generates a stub class. The name of the stub class is derived from the name of the interface, with "`Fakes.Stub`" as a prefix, and the parameter type names appended.

Stubs are also generated for the getters and setters of properties, for events, and for generic methods. For more information, see [Use stubs to isolate parts of your application from each other for unit testing](#).



2. Open Test Explorer and run the test.



## Stubs for different kinds of type members

There are stubs for different kinds of type members.

### Methods

In the provided example, methods can be stubbed by attaching a delegate to an instance of the stub class. The name of the stub type is derived from the names of the method and parameters. For example, consider the following `IStockFeed` interface and its method `GetSharePrice`:

C#

```
// IStockFeed.cs
interface IStockFeed
{
    int GetSharePrice(string company);
}
```

We attach a stub to `GetSharePrice` by using `GetSharePriceString`:

C#

```
// unit test code
var componentUnderTest = new StockAnalyzer(new
    StockAnalysis.Fakes.StubIStockFeed()
    {
        GetSharePriceString = (company) =>
        {
            // Store the parameter value:
            companyCodeUsed = company;
            // Return the value prescribed by this test:
            return priceToReturn;
        }
    });
```

If you don't provide a stub for a method, Fakes generates a function that returns the default value of the return type. For numbers, the default value is 0. For class types, the default is `null` in C# or `Nothing` in Visual Basic.

## Properties

Property getters and setters are exposed as separate delegates and can be stubbed individually. For example, consider the `Value` property of `IStockFeedWithProperty`:

C#

```
interface IStockFeedWithProperty
{
    int Value { get; set; }
}
```

To stub the getter and setter of `Value` and simulate an auto-property, you can use the following code:

C#

```
// unit test code
int i = 5;
var stub = new StubIStockFeedWithProperty();
stub.ValueGet = () => i;
stub.ValueSet = (value) => i = value;
```

If you don't provide stub methods for either the setter or the getter of a property, Fakes generates a stub that stores values, making the stub property behave like a simple variable.

# Events

Events are exposed as delegate fields, allowing any stubbed event to be raised simply by invoking the event backing field. Let's consider the following interface to stub:

```
C#  
  
interface IStockFeedWithEvents  
{  
    event EventHandler Changed;  
}
```

To raise the `Changed` event, you invoke the backing delegate:

```
C#  
  
// unit test code  
var withEvents = new StubIStockFeedWithEvents();  
// raising Changed  
withEvents.ChangedEvent(withEvents, EventArgs.Empty);
```

## Generic methods

You can stub generic methods by providing a delegate for each desired instantiation of the method. For example, given the following interface with a generic method:

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

You can stub the `GetValue<int>` instantiation as follows:

```
C#  
  
[TestMethod]  
public void TestGetValue()  
{  
    var stub = new StubIGenericMethod();  
    stub.GetValueOf1<int>(() => 5);  
  
    IGenericMethod target = stub;  
    Assert.AreEqual(5, target.GetValue<int>());  
}
```

If the code calls `GetValue<T>` with any other instantiation, the stub executes the behavior.

## Stubs of virtual classes

In the previous examples, the stubs have been generated from interfaces. However, you can also generate stubs from a class that has virtual or abstract members. For example:

C#

```
// Base class in application under test
public abstract class MyClass
{
    public abstract void DoAbstract(string x);
    public virtual int DoVirtual(int n)
    {
        return n + 42;
    }

    public int DoConcrete()
    {
        return 1;
    }
}
```

In the stub generated from this class, you can set delegate methods for `DoAbstract()` and `DoVirtual()`, but not `DoConcrete()`.

C#

```
// unit test
var stub = new Fakes.MyClass();
stub.DoAbstractString = (x) => { Assert.IsTrue(x>0); };
stub.DoVirtualInt32 = (n) => 10 ;
```

If you don't provide a delegate for a virtual method, Fakes can either provide the default behavior or call the method in the base class. To have the base method called, set the `CallBase` property:

C#

```
// unit test code
var stub = new Fakes.MyClass();
stub.CallBase = false;
// No delegate set - default delegate:
Assert.AreEqual(0, stub.DoVirtual(1));
```



```
stub.CallBase = true;  
// No delegate set - calls the base:  
Assert.AreEqual(43, stub.DoVirtual(1));
```

## Change the default behavior of stubs

Each generated stub type holds an instance of the `IStubBehavior` interface through the `IStub.InstanceBehavior` property. This behavior is called whenever a client calls a member with no attached custom delegate. If the behavior isn't set, it uses the instance returned by the `StubsBehaviors.Current` property. By default, this property returns a behavior that throws a `NotImplementedException` exception.

You can change the behavior at any time by setting the `InstanceBehavior` property on any stub instance. For example, the following snippet changes the behavior so that the stub either does nothing or returns the default value of the return type `default(T)`:

C#

```
// unit test code  
var stub = new StockAnalysis.Fakes.StubIStockFeed();  
// return default(T) or do nothing  
stub.InstanceBehavior = StubsBehaviors.DefaultValue;
```

The behavior can also be changed globally for all stub objects where the behavior isn't set with the `StubsBehaviors.Current` property:

C#




```
// Change default behavior for all stub instances where the behavior has not  
// been set.  
StubBehaviors.Current = BehavedBehaviors.DefaultValue;
```

## See also

- [Isolate code under test with Microsoft Fakes](#)

# Use shims to isolate your app for unit testing

Article • 05/24/2023

Applies to:  Visual Studio  Visual Studio for Mac  Visual Studio Code

**Shim types**, one of the two key technologies utilized by the Microsoft Fakes Framework, are instrumental in isolating the components of your app during testing. They work by intercepting and diverting calls to specific methods, which you can then direct to custom code within your test. This feature enables you to manage the outcome of these methods, ensuring the results are consistent and predictable during each call, regardless of external conditions. This level of control streamlines the testing process and aids in achieving more reliable and accurate results.

Employ **shims** when you need to create a boundary between your code and assemblies that do not form part of your solution. When the aim is to isolate components of your solution from each other, the use of **stubs** is recommended.

(For a more detailed description for stubs, see [Use stubs to isolate parts of your application from each other for unit testing](#).)

## Shims limitations

It is important to note that shims do have their limitations.

Shims cannot be used on all types from certain libraries in the .NET base class, specifically **mscorlib** and **System** in the .NET Framework, and in **System.Runtime** in .NET Core or .NET 5+. This constraint should be taken into account during the test planning and design stage to ensure a successful and effective testing strategy.

## Creating a Stub: A Step-by-Step Guide

Suppose your component contains calls to `System.IO.File.ReadAllLines`:

C#

```
// Code under test:  
this.Records = System.IO.File.ReadAllLines(path);
```

# Create a Class Library

1. Open Visual Studio and create a **Class Library** project

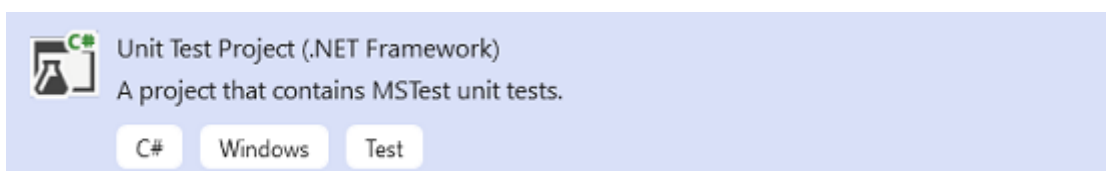


2. Set project name **HexFileReader**
3. Set solution name **ShimsTutorial**.
4. Set the project's target framework to *.NET Framework 4.8*
5. Delete the default file **Class1.cs**
6. Add a new file **HexFile.cs** and add the following class definition:



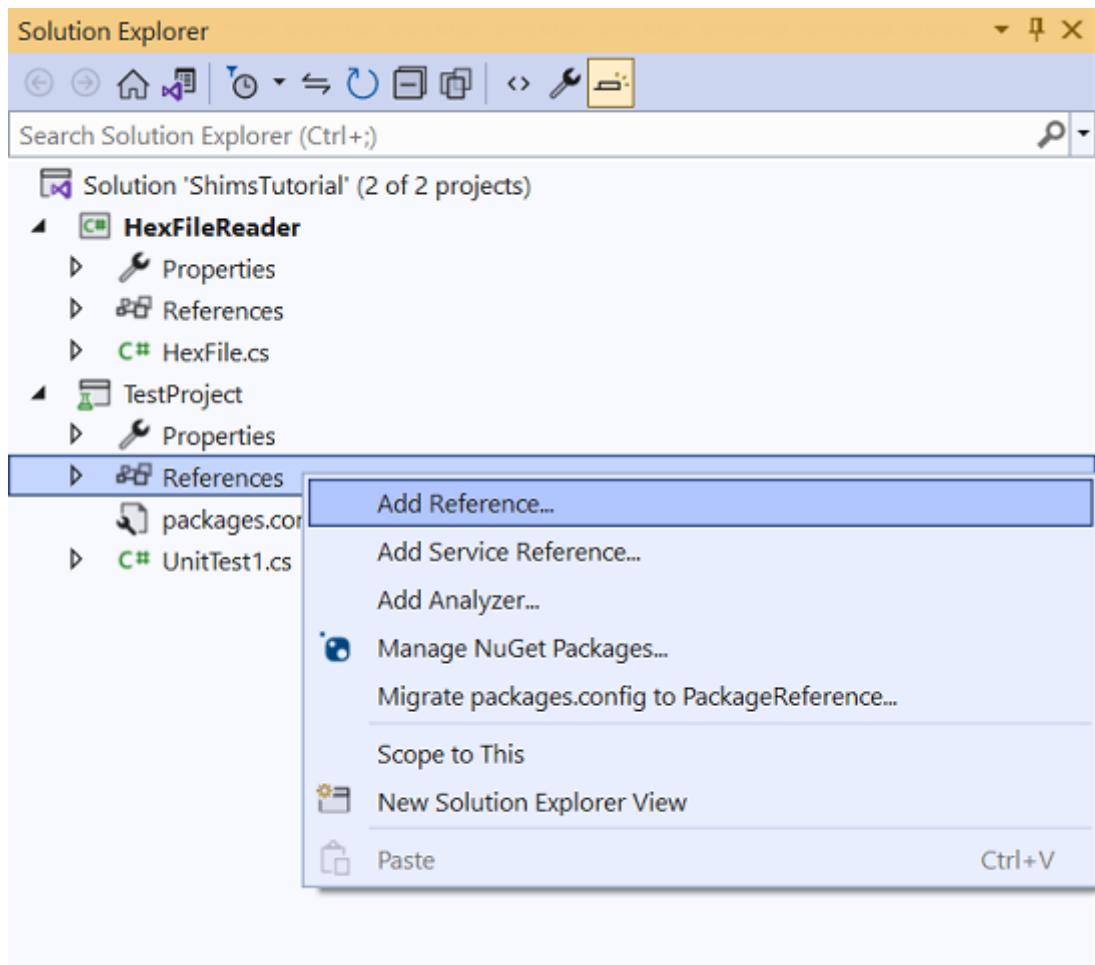
# Create a Test Project

1. Right-click on the solution and add a new project **MSTest Test Project**
2. Set project name **TestProject**
3. Set the project's target framework to *.NET Framework 4.8*



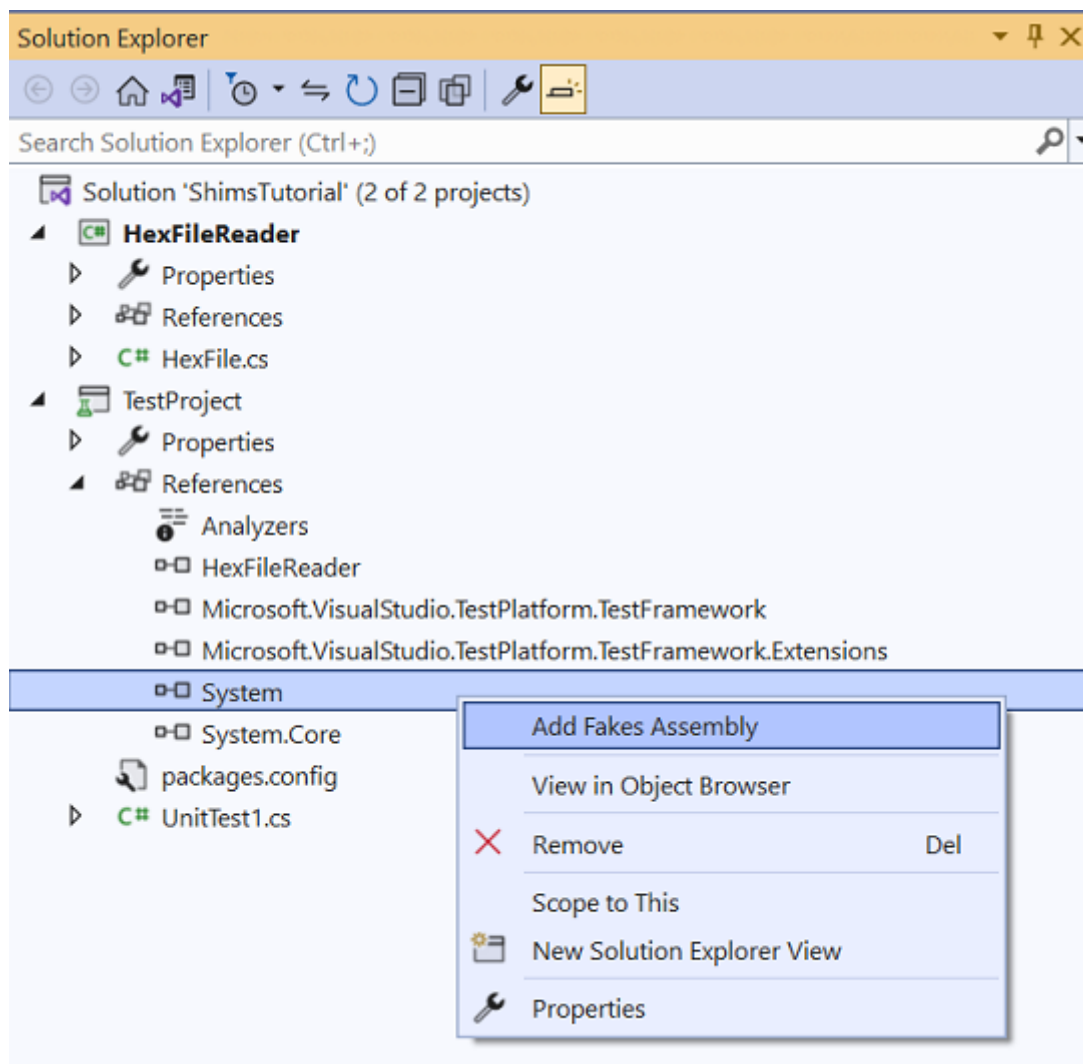
# Add Fakes Assembly

1. Add a project reference to `HexFileReader`



2. Add Fakes Assembly

- In **Solution Explorer**,
  - For an older .NET Framework Project (non-SDK style), expand your unit test project's **References** node.
  - For an SDK-style project targeting .NET Framework, .NET Core, or .NET 5+, expand the **Dependencies** node to find the assembly you would like to fake under **Assemblies**, **Projects**, or **Packages**.
  - If you're working in Visual Basic, select **Show All Files** in the **Solution Explorer** toolbar to see the **References** node.
- Select the assembly `System` that contains the definition of `System.IO.File.ReadAllLines`.
- On the shortcut menu, select **Add Fakes Assembly**.



Since building results in some warnings and errors because not all types can be used with shims, you will have to modify the content of Fakes\mscorlib.fakes to exclude them.

XML

```
<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/" Diagnostic="true">
  <Assembly Name="mscorlib" Version="4.0.0.0"/>
  <StubGeneration>
    <Clear/>
  </StubGeneration>
  <ShimGeneration>
    <Clear/>
    <Add FullName="System.IO.File"/>
    <Remove FullName="System.IO.FileStreamAsyncResult"/>
    <Remove FullName="System.IO.FileSystemEnumerableFactory"/>
    <Remove FullName="System.IO.FileInfoResultHandler"/>
    <Remove FullName="System.IO.FileSystemInfoResultHandler"/>
    <Remove FullName="System.IO.FileStream+FileStreamReadWriteTask"/>
    <Remove FullName="System.IO.FileSystemEnumerableIterator"/>
  </ShimGeneration>
</Fakes>
```

# Create a unit test

1. Modify the default file `UnitTest1.cs` to add the following `TestMethod`

C#

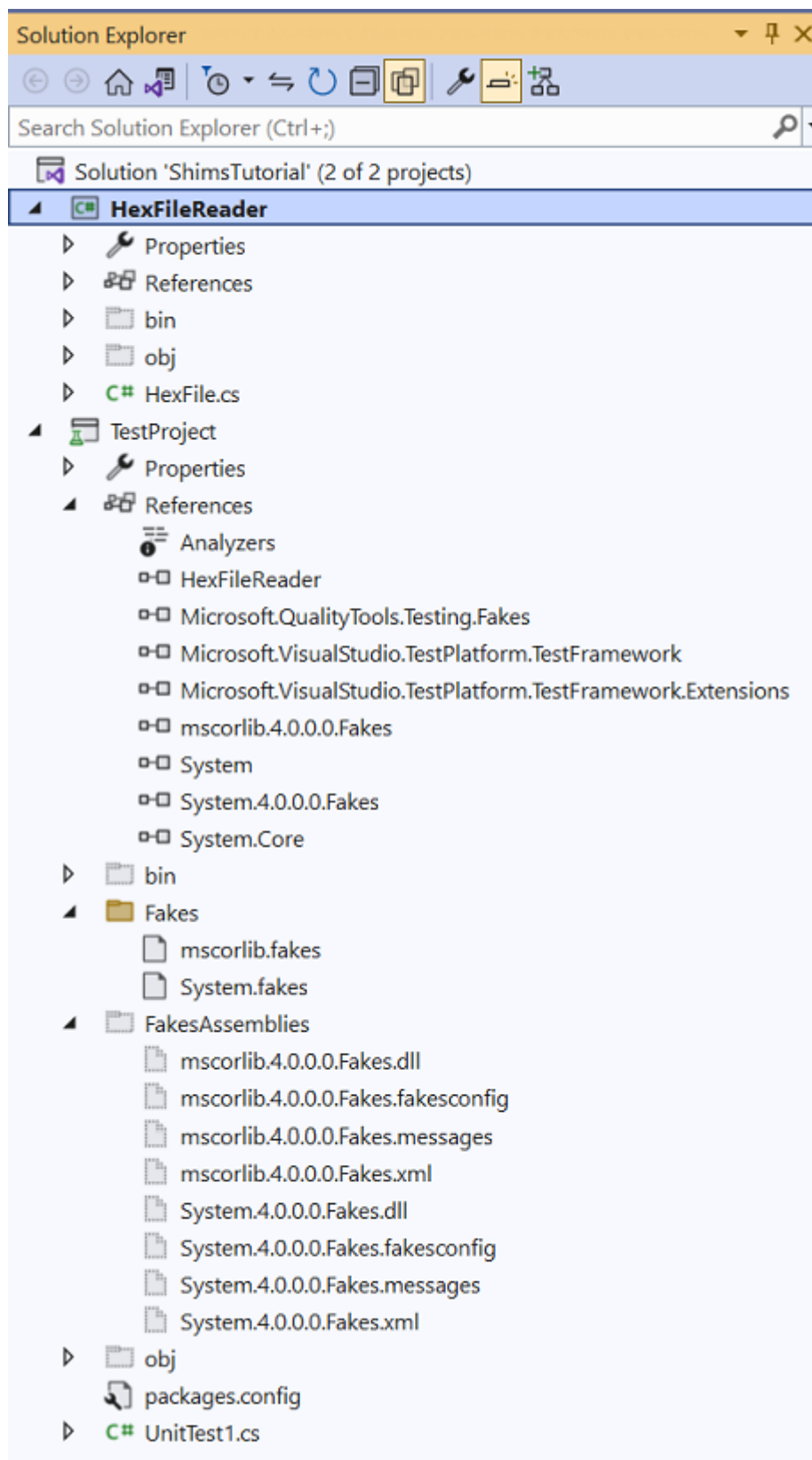
C#

```
[TestMethod]
public void TestFileReadAllLine()
{
    using (ShimsContext.Create())
    {
        // Arrange
        System.IO.Fakes.ShimFile.ReadAllLinesString = (s) => new
string[] { "Hello", "World", "Shims" };

        // Act
        var target = new HexFile("this_file_doesnt_exist.txt");

        Assert.AreEqual(3, target.Records.Length);
    }
}
```

Here is the Solution Explorer showing all the files



2. Open Test Explorer and run the test.

It's critical to properly dispose each shim context. As a rule of thumb, call the `ShimsContext.Create` inside of a `using` statement to ensure proper clearing of the registered shims. For example, you might register a shim for a test method that replaces the `DateTime.Now` method with a delegate that always returns the first of January 2000. If you forget to clear the registered shim in the test method, the rest of the test run would

always return the first of January 2000 as the `DateTime.Now` value. This might be surprising and confusing.

---

## Naming Conventions for Shim Classes

Shim class names are made up by prefixing `Fakes.Shim` to the original type name. Parameter names are appended to the method name. (You don't have to add any assembly reference to `System.Fakes`.)

C#

```
System.IO.File.ReadAllLines(path);
```

C#

```
System.IO.Fakes.ShimFile.ReadAllLinesString = (path) => new string[] {  
    "Hello", "World", "Shims" };
```

## Understanding How Shims Work

Shims operate by introducing *detours* into the codebase of the application being tested. Whenever there's a call to the original method, the Fakes system intervenes to redirect that call, causing your custom shim code to execute instead of the original method.

It's important to note that these detours are created and removed dynamically at runtime. Detours should always be created within the lifespan of a `ShimsContext`. When the `ShimsContext` is disposed, any active shims that were created within it are also removed. To manage this efficiently, it's recommended to encapsulate the creation of detours within a `using` statement.

---

## Shims for different kinds of methods

Shims support various types of methods.

### Static methods

When shimming static methods, properties that hold shims are housed within a shim type. These properties only possess a setter, which is used to attach a delegate to the



targeted method. For instance, if we have a class called `MyClass` with a static method `MyMethod`:

C#

```
//code under test
public static class MyClass {
    public static int MyMethod() {
        ...
    }
}
```

We can attach a shim to `MyMethod` such that it constantly returns 5:

C#

```
// unit test code
ShimMyClass.MyMethod = () => 5;
```

## Instance methods (for all instances)

Just like static methods, instance methods can also be shimmed for all instances. The properties that hold these shims are placed in a nested type named `AllInstances` to prevent confusion. If we have a class `MyClass` with an instance method `MyMethod`:

C#

```
// code under test
public class MyClass {
    public int MyMethod() {
        ...
    }
}
```

We can attach a shim to `MyMethod` so that it consistently returns 5, regardless of the instance:

C#

```
// unit test code
ShimMyClass.AllInstances.MyMethod = () => 5;
```

The generated type structure of `ShimMyClass` would appear as follows:

C#

```
// Fakes generated code
public class ShimMyClass : ShimBase<MyClass> {
    public static class AllInstances {
        public static Func<MyClass, int> MyMethod {
            set {
                ...
            }
        }
    }
}
```

In this scenario, Fakes passes the runtime instance as the first argument of the delegate.

## Instance methods (Single Runtime Instance)

Instance methods can also be shimmed using different delegates, depending on the call's receiver. This allows the same instance method to exhibit different behaviors per instance of the type. The properties that hold these shims are instance methods of the shim type itself. Each instantiated shim type is linked to a raw instance of a shimmed type.

For example, given a class `MyClass` with an instance method `MyMethod`:

C#

```
// code under test
public class MyClass {
    public int MyMethod() {
        ...
    }
}
```

We can create two shim types for `MyMethod` such that the first consistently returns 5 and the second consistently returns 10:

C#

```
// unit test code
var myClass1 = new ShimMyClass()
{
    MyMethod = () => 5
};
var myClass2 = new ShimMyClass { MyMethod = () => 10 };
```

The generated type structure of `ShimMyClass` would appear as follows:

```
C#

// Fakes generated code
public class ShimMyClass : ShimBase<MyClass> {
    public Func<int> MyMethod {
        set {
            ...
        }
    }
    public MyClass Instance {
        get {
            ...
        }
    }
}
```

The actual shimmed type instance can be accessed through the Instance property:

```
C#

// unit test code
var shim = new ShimMyClass();
var instance = shim.Instance;
```

The shim type also includes an implicit conversion to the shimmed type, allowing you to use the shim type directly:

```
C#

// unit test code
var shim = new ShimMyClass();
MyClass instance = shim; // implicit cast retrieves the runtime instance
```

## Constructors

Constructors are no exception to shimming; they too can be shimmed to attach shim types to objects that will be created in the future. For instance, every constructor is represented as a static method, named `Constructor`, within the shim type. Let's consider a class `MyClass` with a constructor that accepts an integer:

```
C#

public class MyClass {
    public MyClass(int value) {
https://learn.microsoft.com/enus/visualstudio/test
    }
```

```

        this.Value = value;
    }
    ...
}

```

A shim type for the constructor can be set up such that, irrespective of the value passed to the constructor, every future instance will return -5 when the Value getter is invoked:

C#

```

// unit test code
ShimMyClass.ConstructorInt32 = (@this, value) => {
    var shim = new ShimMyClass(@this) {
        ValueGet = () => -5
    };
};

```

Each shim type exposes two types of constructors. The default constructor should be used when a new instance is needed, whereas the constructor that takes a shimmed instance as an argument should only be used in constructor shims:

C#

```

// unit test code
public ShimMyClass() { }
public ShimMyClass(MyClass instance) : base(instance) { }

```

The structure of the generated type for `ShimMyClass` can be illustrated as follows:

C#

```

// Fakes generated code
public class ShimMyClass : ShimBase<MyClass>
{
    public static Action<MyClass, int> ConstructorInt32 {
        set {
            ...
        }
    }

    public ShimMyClass() { }
    public ShimMyClass(MyClass instance) : base(instance) { }
    ...
}

```

## Accessing Base members

Shim properties of base members can be reached by creating a shim for the base type and inputting the child instance into the constructor of the base shim class.

For instance, consider a class `MyBase` with an instance method `MyMethod` and a subtype `MyChild`:

```
C#  
  
public abstract class MyBase {  
    public int MyMethod() {  
        ...  
    }  
}  
  
public class MyChild : MyBase {  
}
```

A shim of `MyBase` can be set up by initiating a new `ShimMyBase` shim:

```
C#  
  
// unit test code  
var child = new ShimMyChild();  
new ShimMyBase(child) { MyMethod = () => 5 };
```

It's important to note that when passed as a parameter to the base shim constructor, the child shim type is implicitly converted to the child instance.

The structure of the generated type for `ShimMyChild` and `ShimMyBase` can be likened to the following code:

```
C#  
  
// Fakes generated code  
public class ShimMyChild : ShimBase<MyChild> {  
    public ShimMyChild() { }  
    public ShimMyChild(MyChild child)  
        : base(child) { }  
}  
public class ShimMyBase : ShimBase<MyBase> {  
    public ShimMyBase(MyBase target) { }  
    public Func<int> MyMethod  
    { set { ... } }  
}
```

## Static constructors

Shim types expose a static method `StaticConstructor` to shim the static constructor of a type. Since static constructors are executed once only, you need to make sure that the shim is configured before any member of the type is accessed.

## Finalizers

Finalizers are not supported in Fakes.

## Private methods

The Fakes code generator creates shim properties for private methods that only have visible types in the signature, that is, parameter types and return type visible.

## Binding interfaces

When a shimmed type implements an interface, the code generator emits a method that allows it to bind all the members from that interface at once.

For example, given a class `MyClass` that implements `IEnumerable<int>`:

C#

```
public class MyClass : IEnumerable<int> {  
    public IEnumerator<int> GetEnumerator() {  
        ...  
    }  
    ...  
}
```

You can shim the implementations of `IEnumerable<int>` in `MyClass` by calling the `Bind` method:

C#

```
// unit test code  
var shimMyClass = new ShimMyClass();  
shimMyClass.Bind(new List<int> { 1, 2, 3 });
```

The generated type structure of `ShimMyClass` resembles the following code:

C#

```
// Fakes generated code  
public class ShimMyClass : ShimBase<MyClass> {  
  
    https://learn.microsoft.com/enus/visualstudio/test
```

```
public ShimMyClass Bind(IEnumerable<int> target) {  
    ...  
}  
}
```

## Change Default Behavior

Each generated shim type includes an instance of the `IShimBehavior` interface, accessible via the `ShimBase<T>.InstanceBehavior` property. This behavior is invoked whenever a client calls an instance member that has not been explicitly shimmed.

By default, if no specific behavior has been set, it uses the instance returned by the static `ShimBehaviors.Current` property, which typically throws a `NotImplementedException` exception.

You can modify this behavior at any time by adjusting the `InstanceBehavior` property for any shim instance. For instance, the following code snippet alters the behavior to either do nothing or return the default value of the return type—i.e., `default(T)`:

C#

```
// unit test code  
var shim = new ShimMyClass();  
//return default(T) or do nothing  
shim.InstanceBehavior = ShimBehaviors.DefaultValue;
```

You can also globally change the behavior for all shimmed instances—where the `InstanceBehavior` property has not been explicitly defined—by setting the static `ShimBehaviors.Current` property:

C#

```
// unit test code  
// change default shim for all shim instances where the behavior has not  
// been set  
ShimBehaviors.Current = ShimBehaviors.DefaultValue;
```

## Identifying Interactions with External Dependencies

To help identify when your code is interacting with external systems or dependencies (referred to as the `environment`), you can utilize shims to assign a specific behavior to all  
<https://learn.microsoft.com/enus/visualstudio/test>

members of a type. This includes static methods. By setting the `ShimBehaviors.NotImplemented` behavior on the static `Behavior` property of the shim type, any access to a member of that type that hasn't been explicitly shimmed will throw a `NotImplementedException`. This can serve as a useful signal during testing, indicating that your code is attempting to access an external system or dependency.

Here's an example of how to set this up in your unit test code:

C#

```
// unit test code
// Assign the NotImplementedException behavior to ShimMyClass
ShimMyClass.Behavior = ShimBehaviors.NotImplemented;
```

For convenience, a shorthand method is also provided to achieve the same effect:

C#

```
// Shorthand to assign the NotImplementedException behavior to ShimMyClass
ShimMyClass.BehaveAsNotImplemented();
```

## Invoking Original Methods from Within Shim Methods

There could be scenarios where you might need to execute the original method during the execution of the shim method. For instance, you might want to write text to the file system after validating the file name passed to the method.

One approach to handle this situation is to encapsulate a call to the original method using a delegate and `ShimsContext.ExecuteWithoutShims()`, as demonstrated in the following code:

C#

```
// unit test code
ShimFile.WriteAllTextStringString = (fileName, content) => {
    ShimsContext.ExecuteWithoutShims(() => {

        Console.WriteLine("enter");
        File.WriteAllText(fileName, content);
        Console.WriteLine("leave");
    });
};
```



Alternatively, you can nullify the shim, call the original method, and then restore the shim.

C#

```
// unit test code
ShimsDelegates.Action<string, string> shim = null;
shim = (fileName, content) => {
    try {
        Console.WriteLine("enter");
        // remove shim in order to call original method
        ShimFile.WriteAllTextStringString = null;
        File.WriteAllText(fileName, content);
    }
    finally
    {
        // restore shim
        ShimFile.WriteAllTextStringString = shim;
        Console.WriteLine("leave");
    }
};
// initialize the shim
ShimFile.WriteAllTextStringString = shim;
```

## Handling Concurrency with Shim Types

Shim types operate across all threads within the AppDomain and do not possess thread affinity. This property is crucial to keep in mind if you plan to utilize a test runner that supports concurrency. It's worth noting that tests involving shim types cannot run concurrently, although this restriction is not enforced by the Fakes runtime.

## Shimming System.Environment

If you wish to shim the `System.Environment` class, you'll need to make some modifications to the `mscorlib.fakes` file. Following the Assembly element, add the following content:

XML

```
<ShimGeneration>
  <Add FullName="System.Environment"/>
</ShimGeneration>
```

Once you have made these changes and rebuilt the solution, the methods and properties in the `System.Environment` class are now available to be shimmed. Here's an

<https://learn.microsoft.com/enus/visualstudio/test>

example of how you can assign a behavior to the `GetCommandLineArgsGet` method:

C#

```
System.Fakes.ShimEnvironment.GetCommandLineArgsGet = ...
```