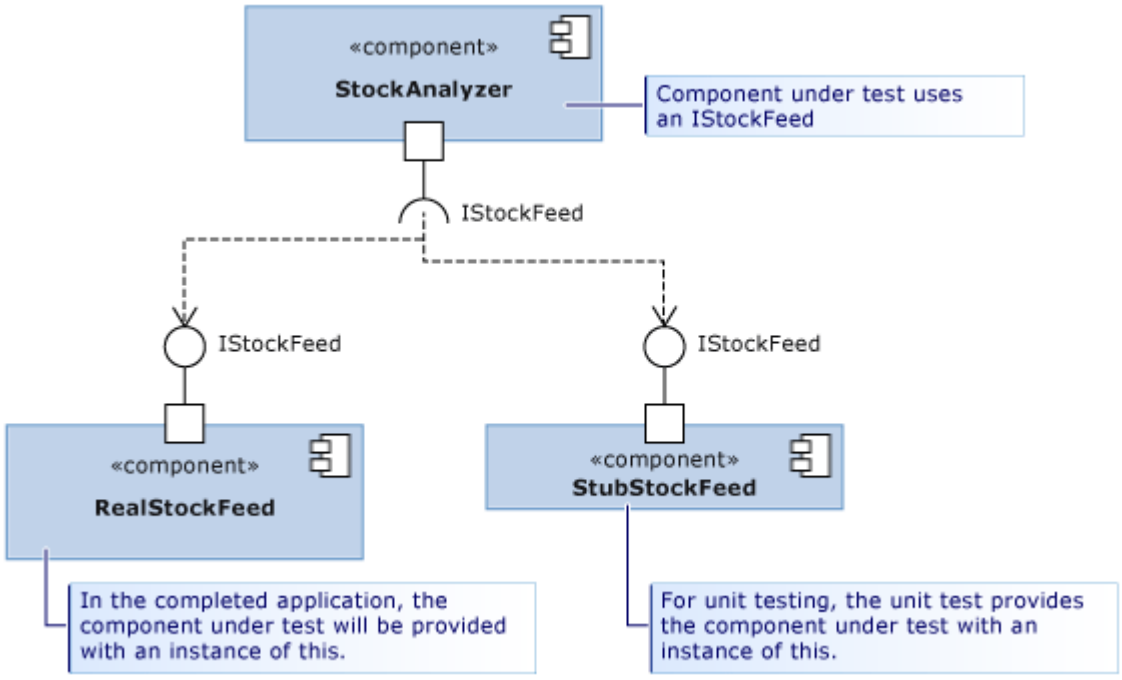
Using stubs to isolate parts of your application from each other for unit testing

Visual Studio 2015

Stub types are one of two technologies that the Microsoft Fakes framework provides to let you easily isolate a component you are testing from other components that it calls. A stub is a small piece of code that takes the place of another component during testing. The benefit of using a stub is that it returns consistent results, making the test easier to write. And you can run tests even if the other components are not working yet.

To use stubs, you have to write your component so that it uses only interfaces, not classes, to refer to other parts of the application. This is a good design practice because it makes changes in one part less likely to require changes in another. For testing, it allows you to substitute a stub for a real component.

In the diagram, the component *StockAnalyzer* is the one we want to test. It normally uses another component, *RealStockFeed*. But *RealStockFeed* returns different results every time its methods are called, making it difficult to test *StockAnalyzer*. During testing, we replace it with a different class, *StubStockFeed*.



Because stubs rely on your being able to structure your code in this way, you typically use stubs to isolate one part of your application from another. To isolate it from other assemblies that are not under your control, such as System.dll, you would normally use shims.

How to use stubs

Design for dependency injection

To use stubs, your application has to be designed so that the different components are not dependent on each other, but only dependent on interface definitions. Instead of being coupled at compile time, components are connected at run time. This pattern helps to make software that is robust and easy to update, because changes tend not to propagate across component boundaries. We recommend following it even if you don't use stubs. If you are writing new code, it's easy to follow the dependency injection pattern. If you are writing tests for existing software, you might have to refactor it. If that would be impractical, you could consider using shims instead.

Let's start this discussion with a motivating example, the one in the diagram. The class *StockAnalyzer* reads share prices and generates some interesting results. It has some public methods *GetContosoPrice()* (see a definition below), which we want to test. To keep things simple, let's just look at one of those methods, a very simple one that reports the current price of a particular share. We want to write a unit test of that method. Here's the first draft of a test:

```
 [TestMethod]
public void TestMethod1()
{
    // Arrange:
    var analyzer = new StockAnalyzer();
    // Act:
    var result = analyzer.GetContosoPrice();
    // Assert:
    Assert.AreEqual(123, result); // Why 123?
 }
```

One problem with this test is immediately obvious: share prices vary, and so the assertion will usually fail.

Another problem might be that the *StockFeed* component, which is used by the *StockAnalyzer*, is still under development. Here's the first draft of the code of the method under test:

```
 public int GetContosoPrice()
{
    var stockFeed = new StockFeed(); // NOT RECOMMENDED
    return stockFeed.GetSharePrice("COOO");
}
```

As it stands, this method might not compile or might throw an exception because work on the *StockFeed* class is not yet complete.

Interface injection addresses both of these problems.

Interface injection applies the following rule:

- The code of any component of your application should never explicitly refer to a class in another component, either in a declaration or in a new statement. Instead, variables and

parameters should be declared with interfaces. Component instances should be created only by the component's container.

By "component" in this case we mean a class, or a group of classes that you develop and update together. Typically, a component is the code in one Visual Studio project. It's less important to decouple classes within one component, because they are updated at the same time.

It is also not so important to **decouple** your components from the classes of a relatively stable platform such as System.dll. Writing interfaces for all these classes would clutter your code.

The *StockAnalyzer* code can therefore be improved by decoupling it from the *StockFeed* by using an interface like this:

```
public interface IStockFeed
{
    int GetSharePrice(string company);
}

public class StockAnalyzer
{
    private IStockFeed stockFeed;
    public Analyzer(IStockFeed feed)
    {
        stockFeed = feed;
    }
    public int GetContosoPrice()
    {
        return stockFeed.GetSharePrice("COOO");
    }
}
```

In this example, *StockAnalyzer* is passed an implementation of an *IStockFeed* when it is constructed. In the completed application, the initialization code would perform the connection:

```
analyzer = new StockAnalyzer(new StockFeed())
```

There are more flexible ways of performing this connection. For example, *StockAnalyzer* could accept a factory object that can instantiate different implementations of *IStockFeed* in different conditions.

Generate stubs

You've decoupled the class you want to test from the other components that it uses. As well as making the application more robust and flexible, the decoupling allows you to connect the component under test to stub implementations of the interfaces for test purposes.

You could simply write the stubs as classes in the usual way. But Microsoft Fakes provides you with a more dynamic way to create the most appropriate stub for every test.

To use stubs, you must first generate stub types from the interface definitions.

Adding a Fakes Assembly

1. In Solution Explorer, expand your unit test project's References.
   - If you are working in Visual Basic, you must select Show All Files in the Solution Explorer toolbar, in order to see the References list.
2. Select the assembly that contains the interface definitions for which you want to create stubs.
3. On the shortcut menu, choose Add Fakes Assembly.

In the new assembly, we can find the Stub class for our *IStockFeed* interface.

```
[DebuggerNonUserCode]
[DebuggerDisplay("Stub of IStockFeed")]
[StubClass(typeof (IStockFeed))]
public class StubIStockFeed : StubBase<IStockFeed>, IStockFeed
{
  /// <summary>
  /// Sets the stub of IStockFeed.GetSharePrice(String company)
  /// </summary>
  public FakesDelegates.Func<string, int> GetSharePriceString;
  private StubDelegateMap ___getValueOf1s;

  int IStockFeed.fd2013\u003A\u003AFakesDemo2013\u002EIStockFeed\u002EGetSharePrice(string company)
  {
    IStubObserver instanceObserver = this.InstanceObserver;
    if (instanceObserver != null)
    {
      FakesDelegates.Func<string, int> func = new FakesDelegates.Func<string, int>(((IStockFeed) this).GetSharePrice);
      instanceObserver.Enter(typeof (IStockFeed), (Delegate) func, (object) company);
    }
    FakesDelegates.Func<string, int> func1 = this.GetSharePriceString;
    if (func1 != null)
      return func1(company);
    else
      return this.InstanceBehavior.Result<StubIStockFeed, int>(this, "FakesDemo2013.IStockFeed.GetSharePrice");
  }

  T IStockFeed.fd2013\u003A\u003AFakesDemo2013\u002EIStockFeed\u002EGetValue<T>()
  {
    IStubObserver instanceObserver = this.InstanceObserver;
    if (instanceObserver != null)
    {
      FakesDelegates.Func<T> func = new FakesDelegates.Func<T>(((IStockFeed) this).GetValue<T>);
      instanceObserver.Enter(typeof (IStockFeed), (Delegate) func);
    }
    FakesDelegates.Func<T> @delegate;
    if (this.___getValueOf1s.TryGetValue<FakesDelegates.Func<T>>(out @delegate))
      return @delegate();
    else
      return this.InstanceBehavior.Result<StubIStockFeed, T>(this, "FakesDemo2013.IStockFeed.GetValue");
  }

  /// <summary>
  /// Sets stubs of GetValue()
  /// </summary>
  public void GetValueOf1<T>(FakesDelegates.Func<T> stub)
  {
    this.___getValueOf1s = StubDelegateMap.Concat(this.___getValueOf1s, (Delegate) stub);
  }
}
```

Write your test with stubs

```
[TestClass]
class TestStockAnalyzer
{
    [TestMethod]
    public void TestContosoStockPrice()
    {
      // Arrange:

      // Create the fake stockFeed:
      IStockFeed stockFeed =
           new Fakes.StubIStockFeed() // Generated by Fakes.
      {
          // Define each method:
          // Name is original name + parameter types:
          GetSharePriceString = (company) => { return 1234; }
       };

       // In the completed application, stockFeed would be a real one:
       var componentUnderTest = new StockAnalyzer(stockFeed);

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

      // Assert:
      Assert.AreEqual(1234, actualValue);
    }
    ...
}
```

The special piece of magic here is the class *StubIStockFeed*. For every public type in the referenced assembly, the Microsoft Fakes mechanism generates a stub class. The name of the stub class is the 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.

Verifying parameter values

You can verify that when your component makes a call to another component, it passes the correct values. You can either place an assertion in the stub, or you can store the value and verify it in the main body of the test. For example:

```
[TestClass]
class TestMyComponent
{

    [TestMethod]
    public void TestVariableContosoPrice()
    {
     // Arrange:
        int priceToReturn;
        string companyCodeUsed;
        var componentUnderTest = new StockAnalyzer(new StubIStockFeed()
            {
               GetSharePriceString = (company) =>
```

```
                {
                    // Store the parameter value:
                    companyCodeUsed = company;
                    // Return the value prescribed by this test:
                    return priceToReturn;
                };
            };
        // Set the value that will be returned by the stub:
        priceToReturn = 345;

    // 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("COOO", companyCodeUsed);
    }
...}
```

Stubs for different kinds of type members

Properties

Property getters and setters are exposed as separate delegates and can be stubbed separately. For example, consider the *Value* property of *IMyInterface*:

```
// code under test
interface IMyInterface
{
    int Value { get; set; }
}
```

We attach delegates to the getter and setter of *Value* to simulate an auto-property:

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

If you do not provide stub methods for either the setter or the getter of a property, Fakes will generate a stub that stores values, so that the stub property works like a simple variable.

Generic methods

It's 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:

```
// code under test
interface IGenericMethod
{
    T GetValue<T>();
}
```

you could write a test that stubs the *GetValue<int>* instantiation:

```
// unit test code
[TestMethod]
public void TestGetValue()
{
    var stub = new StubIGenericMethod();
    stub.GetValueOf1<int>(() => 5);

    IGenericMethod target = stub;
    Assert.AreEqual(5, target.GetValue<int>());
}
```

If the code were to call *GetValue<T>* with any other instantiation, the stub would simply call the behavior.

How to use shims

Let's consider a method that throws an exception on January 1st of 2000:

```
// code under test
public static class Y2KChecker {
    public static void Check() {
        if (DateTime.Now == new DateTime(2000, 1, 1))
            throw new ApplicationException("y2kbug!");
    }
}
```

Testing this method is particularly problematic because the program depends on *DateTime.Now*, a method that depends on the computer's clock, an environment-dependent, non-deterministic method. Furthermore, the *DateTime.Now* is a static property so a stub type can't be used here. This problem is symptomatic of the isolation issue in unit testing: programs that directly call into database APIs, communicate with web services, and so on are hard to unit test because their logic depends on the environment.

This is where shim types should be used. Shim types provide a mechanism to detour any .NET method to a user defined delegate. Shim types are code-generated by the Fakes generator, and they use delegates, which we call shim types, to specify the new method implementations.

The following test shows how to use the shim type, *ShimDateTime*, to provide a custom implementation of *DateTime.Now*:

```
//unit test code
// create a ShimsContext cleans up shims
```

```
using (ShimsContext.Create()
    // hook delegate to the shim method to redirect DateTime.Now
    // to return January 1st of 2000
    ShimDateTime.NowGet = () => new DateTime(2000, 1, 1);
    Y2KChecker.Check();
}
```

How to use Shims

Add Fakes Assemblies

1. In Solution Explorer, expand your unit test project's References.
2. Select the assembly that contains the classes definitions for which you want to create shims. For example, if you want to shim DateTime, select System.dll
3. On the shortcut menu, choose Add Fakes Assembly.

Use ShimsContext

When using shim types in a unit test framework, you must wrap the test code in a *ShimsContext* to control the lifetime of your shims. If we didn't require this, your shims would last until the *AppDomain* shut down. The easiest way to create a *ShimsContext* is by using the static *Create()* method as shown in the following code:

```
//unit test code
[Test]
public void Y2kCheckerTest() {
  using(ShimsContext.Create()) {
    ...
  } // clear all shims
}
```

It is critical to properly dispose each shim context. As a rule of thumb, always 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 1 January 2000. If you forget to clear the registered shim in the test method, the rest of the test run would always return 1 January 2000 as the *DateTime.Now* value. This might be surprising and confusing.

Write a test with shims

In your test code, insert a detour for the method you want to fake. For example:

```
[TestClass]
public class TestClass1
{
        [TestMethod]
        public void TestCurrentYear()
        {
            int fixedYear = 2000;
```

```
            using (ShimsContext.Create())
            {
              // Arrange:
                // Detour DateTime.Now to return a fixed date:
                System.Fakes.ShimDateTime.NowGet =
                () =>
                { return new DateTime(fixedYear, 1, 1); };

                // Instantiate the component under test:
                var componentUnderTest = new MyComponent();

              // Act:
                int year = componentUnderTest.GetTheCurrentYear();

              // Assert:
                // This will always be true if the component is working:
                Assert.AreEqual(fixedYear, year);
            }
        }
}
```

Shim class names are made up by prefixing *Fakes.Shim* to the original type name.

Shims work by inserting detours into the code of the application under test. Wherever a call to the original method occurs, the *Fakes* system performs a detour, so that instead of calling the real method, your shim code is called.

Notice that detours are created and deleted at run time. You must always create a detour within the life of a *ShimsContext*. When it is disposed, any shims you created while it was active are removed. The best way to do this is inside a using statement.

You might see a build error stating that the Fakes namespace does not exist. This error sometimes appears when there are other compilation errors. Fix the other errors and it will vanish.