Unit Testing

Mock Objects to the Rescue! Test Your .NET Code with NMock

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifContents

[Understanding the Unit Testing Problem](http://msdn.microsoft.com/en-us/magazine/cc163904.aspx#S1)  
[Mock Objects to the Rescue](http://msdn.microsoft.com/en-us/magazine/cc163904.aspx#S2)  
[The Shopping Sample Library](http://msdn.microsoft.com/en-us/magazine/cc163904.aspx#S3)  
[NMock to the Rescue](http://msdn.microsoft.com/en-us/magazine/cc163904.aspx#S4)  
[Setting Up Results](http://msdn.microsoft.com/en-us/magazine/cc163904.aspx#S5)  
[Setting Expectations](http://msdn.microsoft.com/en-us/magazine/cc163904.aspx#S6)  
[Let the Mock Participate in the Test](http://msdn.microsoft.com/en-us/magazine/cc163904.aspx#S7)  
[Designing for Testability](http://msdn.microsoft.com/en-us/magazine/cc163904.aspx#S8)  
[Conclusion](http://msdn.microsoft.com/en-us/magazine/cc163904.aspx#S9)

Have you ever considered implementing unit tests for a project but dismissed the idea because the module you wanted to test either had too many dependencies or it was so difficult to isolate the unit itself that the tests started to look like integration tests? Or perhaps setting up and configuring all the external dependencies was impractical or downright impossible to achieve within your time constraints or budgets?

Mock objects can help you overcome these types of obstacles and, as an added benefit, can help enforce good design practices. In this article, I will describe how to use a publicly available dynamic mock object library called NMock, and will provide you with some of the design guidelines that I have found to be of practical use in my own work.

NMock is a great tool, but unfortunately it doesn't currently come with documentation or samples, so it's difficult to figure out what to do with it, or even for what use it is intended. I will try to illustrate the purpose and use of NMock by showing you how to write unit tests for a business logic library which is dependent on a data access layer (DAL).

**Understanding the Unit Testing Problem**

Before we embark on our journey through the sample code I've prepared for this article, it's necessary to understand the problem: unit testing libraries that have a complex set of dependencies. It is best to design libraries to have a minimal set of dependencies, but sometimes this is not practical, especially when the library has not been designed with testability in mind. When unit testing libraries with a complex set of dependencies, these dependencies may require difficult or time-consuming setup procedures which are detrimental to the test process. Consider the case of a business logic library which uses a data access component to communicate with a database server, as illustrated on the left side of **Figure 1**.

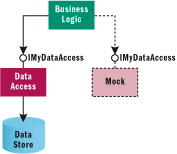


Figure 1 **Mock Testing Concept**

Although the binding between the business logic library and the data access component is interface-based, the business logic library still needs to be able to access properties and methods on the data access component during unit testing, and that's where the trouble begins.

The data access component needs a connection to a database to work. This means that you need to configure the component to access the right database, and you almost certainly need to set up a test instance of the database so that you don't risk corrupting a production database during testing. If you are using SQL Server™ 2000, this may be a fairly manageable task, but what if the data access component provides access to a mainframe-based database? In the latter case, setting up a test instance of the database may be impractical at best. And even when using SQL Server, time may be a prohibitive factor.

However, configuring the DAL for test is not the only problem in this scenario. If your data access component is part of your development project, it may not even exist yet, and even if it does, it most likely will still be under development with a number of bugs, both known and unknown. In this case, you risk possible unit test failure due to bugs in the DAL even if the code under test has no defects. In other words, in such a case the library under test would not be properly isolated from other modules, so at that point the test is actually no longer a unit test.

**Mock Objects to the Rescue**

Traditionally, unit testing terminology has included the concepts of drivers and stubs. During your career as a software developer, you have probably created quite a few of these drivers and stubs for each module you've needed to test.

A driver is a piece of software that is written with the sole purpose of accessing properties and methods on a library to test the functionality of that library. A stub is a piece of software whose only purpose is to provide the library under test with an implementation of any modules it may need to communicate with to perform its work.

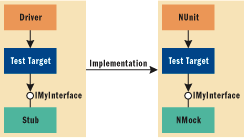


Figure 2 **Unit Testing Concepts and Implementation**

As illustrated in **Figure 2**, this concept translates well into practical terms. With the tools available today, you have a framework for implementing both drivers and stubs in a dynamic and flexible way. As an implementation of a driver, I use NUnit (currently at version 2.2), and for stubs I use NMock (currently at version 1.1) to create dynamic mock objects at run time. If you refer back to the right side of the diagram in **Figure 1**, a mock object implementing the IMyDataAccess interface has replaced the intended production DAL. NMock can create such a mock object at run time.

It is worth noting that NMock uses reflection to create mock implementations of interfaces at run time, so to use NMock it is necessary to code against interfaces instead of implementations. As this is already a well-regarded best practice, NMock helps to enforce this important design technique.

**The Shopping Sample Library**

The download for NMock consists of an assembly that can be used right away; no installation or configuration is necessary (see <http://www.nmock.org>). To illustrate the use of NMock, I'll show you how to write unit tests for a very simplified shopping basket library which uses a DAL. The library uses the DAL by accessing a data access object that implements the IShoppingDataAccess interface, which is shown here:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl05_code');" \o "Copy Code)

using System;

namespace NMockExample.Shopping

{

public interface IShoppingDataAccess

{

string GetProductName(int productID);

int GetUnitPrice(int productID);

BasketItem[] LoadBasketItems(Guid basketID);

void SaveBasketItems(Guid basketID, BasketItem[] basketItems);

}

}

The Basket class illustrated in **Figure 3** contains functionality to perform operations such as calculating the subtotal, saving the basket to the data store, and adding a new item to the basket. You may have noticed that the constructor takes as a parameter an object that implements the IShoppingDataAccess interface. The class will use this object to perform data access operations. For more information on this pattern, which is known as "Inversion of Control," see Martin Fowler's article on the subject at [Inversion of Control Containers and the Dependency Injection pattern](http://www.martinfowler.com/articles/injection.html).

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 3 Basket Class

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl30_ctl00_ctl00_code');" \o "Copy Code)

using System;

using System.Collections;

using System.Configuration;

namespace NMockExample.Shopping

{

public class Basket

{

private ArrayList basketItems\_;

private Guid basketID\_;

private IShoppingDataAccess dataAccess\_;

public Basket(IShoppingDataAccess dataAccess)

{

Initialize(dataAccess);

}

public void AddItem(BasketItem item)

{

basketItems\_.Add(item);

}

public void Save()

{

dataAccess\_.SaveBasketItems(basketID\_,

(BasketItem[])basketItems\_.ToArray

(typeof(BasketItem)));

}

public decimal CalculateSubTotal()

{

decimal subTotal = 0;

foreach(BasketItem item in basketItems\_)

{

subTotal += item.GetPrice();

}

return subTotal;

}

private void Initialize(IShoppingDataAccess dataAccess)

{

dataAccess\_ = dataAccess;

basketItems\_ = new ArrayList();

basketID\_ = Guid.NewGuid();

}

}

}

The Basket class contains an internal list of BasketItem objects, which are shown in **Figure 4**. Given a product ID, a BasketItem object uses its internal data access object to provide product information such as unit price and product name.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 4 BasketItem Class

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl31_ctl00_ctl00_code');" \o "Copy Code)

using System;

namespace NMockExample.Shopping

{

public class BasketItem

{

private decimal unitPrice\_;

private int productID\_;

private int quantity\_;

private IShoppingDataAccess dataAccess\_;

private string productName\_;

public BasketItem(int productID,

int quantity, IShoppingDataAccess dataAccess)

{

Initialize(productID, quantity, dataAccess);

}

public decimal UnitPrice { get{ return unitPrice\_; } }

public int ProductID

{

get { return productID\_; }

set

{

productID\_ = value;

unitPrice\_ = dataAccess\_.GetUnitPrice(productID\_);

productName\_ = dataAccess\_.GetProductName(productID\_);

}

}

public int Quantity

{

get { return quantity\_; }

set { quantity\_ = value; }

}

public string ProductName { get { return productName\_; } }

public decimal GetPrice()

{

return unitPrice\_ \* quantity\_;

}

private void Initialize(int productID,

int quantity, IShoppingDataAccess dataAccess)

{

dataAccess\_ = dataAccess;

ProductID = productID;

Quantity = quantity;

}

}

}

As you can see, most operations in this Shopping sample library make use of data access methods, which means that I need a way to isolate the Shopping library from the DAL if I want to write meaningful unit tests—tests that truly reflect the correctness of the method under test and that are not affected by one of its dependencies and skewing the results.

**NMock to the Rescue**

All classes in the Shopping sample library expose a way to externally define the data access implementation they use, so we can use NMock to provide a dynamic mock implementation of IShoppingDataAccess to the Shopping classes.

Let's start by looking at a very simple unit test which demonstrates how to create a mock instance of IShoppingDataAccess using NMock. Before writing any code, it is necessary to add a reference to NMock.dll and add a using statement for the NMock namespace. The following code is a very simple unit test of the Basket class's Save method, though at the moment no assertions are being performed:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl10_code');" \o "Copy Code)

DynamicMock dataAccess = new DynamicMock(typeof(IShoppingDataAccess));

Basket b = new Basket((IShoppingDataAccess)dataAccess.MockInstance);

b.Save();

To create a mock object with NMock, instantiate a new DynamicMock object, passing to its constructor the type you want it to implement. The mock object itself doesn't implement the type but is an object which contains information about the behavior of the mock object and can emit an implementation of the specified type. This is exactly what happens in the next line of code, where the Basket constructor is passed a reference to the DynamicMock object's MockInstance property.

DynamicMock uses reflection to emit an implementation of the desired type and expose it through the MockInstance property—in this case an implementation of IShoppingDataAccess. As MockInstance returns an object of type System.Object, it is necessary to cast the returned reference to the desired type.

In this case, the dynamically created implementation of IShoppingDataAccess is passed to the Basket constructor, and when the Save method is called, the Basket object can call the SaveBasketItems method on its internal IShoppingDataAccess member. Nothing was specified in the setup code, so the DynamicMock object doesn't know what to do with the call to SaveBasketItems. In this case, it doesn't matter because the method returns void. As such, this particular unit test only really tests that the Basket constructor and the Save method can be executed without exceptions.

Although this is not the most exciting unit test, it already demonstrates how to dynamically create a mock implementation of an interface. Had I passed a null reference to the Basket constructor instead of the mock object, the code would have thrown an exception when the Save method of the Basket class was called.

**Setting Up Results**

The default behavior of a mock instance is to return null from every method call that returns a reference type and to return the default value from every method call that returns a value type, so using it in the fashion I just illustrated provides very limited usefulness. It may be adequate for void methods, but falls short when you need to test a method with a return value.

Happily, the DynamicMock class allows me to modify the behavior of its MockInstance property. Before I pass the mock instance to my test target, I can call some methods on the DynamicMock class, which will instruct the mock instance to behave as I need it to.

To illustrate this, I wrote a simple unit test to test the BasketItem constructor. If you examine the code for the constructor in **Figure 4**, you will notice that during construction, the BasketItem class's ProductID property is being set, which results in two calls to the DAL. Both calls return a value, so if I had repeated my previous action, a NullReferenceException would have been thrown by the call to GetUnitPrice. When dealing with methods with return values, the default behavior of the mock instance isn't sufficient anymore. It is necessary to instruct the mock object about what to do, as shown in **Figure 5**.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 5 Setting Up Results with NMock

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl33_ctl00_ctl00_code');" \o "Copy Code)

DynamicMock dataAccess = new DynamicMock(typeof(IShoppingDataAccess));

dataAccess.SetupResult("GetUnitPrice", 99, typeof(int));

dataAccess.SetupResult("GetProductName", "The Moon", typeof(int));

BasketItem item = new BasketItem(1, 2,

(IShoppingDataAccess)dataAccess.MockInstance);

Assert.AreEqual(99, item.UnitPrice);

Assert.AreEqual("The Moon", item.ProductName);

Assert.AreEqual(198, item.GetPrice());

Note that after I've created the DynamicMock object, but before I begin using it in my target library, I use the SetupResult method to modify the behavior of the mock instance. I configure it so that every time it receives a call to the GetUnitPrice method of IShoppingDataAccess, it should return 99, and every time it receives a call to GetProductName, it should return "The Moon." The SetupResult method is defined like this:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl13_code');" \o "Copy Code)

void SetupResult(System.String methodName,

System.Object returnVal, params System.Type[] argTypes)

The first two parameters supply the method name and return value, respectively. The next parameter is an array of types that will be used to identify which method was called. Why is this necessary? Imagine for a moment that there are overloaded versions of the GetUnitPrice method, one taking as a parameter an integer value, and the other taking a System.Guid value. If the parameter list wasn't supplied to the DynamicMock object, it wouldn't know which one of the two GetUnitPrice methods I meant. To uniquely identify the method, I need to supply DynamicMock with enough information to allow it to infer the method's signature.

By calling SetupResult as I do here

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl14_code');" \o "Copy Code)

SetupResult("GetUnitPrice", 99, typeof(int));

I indicate that when the mock instance receives a call to the GetUnitPrice method that takes a single integer argument, it should return a value of 99. This allows DynamicMock to uniquely identify the method in question.

The last three lines of the unit test are just standard NUnit assertions to test that the newly created BasketItem object contains the expected data, and that it calculates the correct price.

**Setting Expectations**

The SetupResult method is still a rather blunt instrument. Any call to the configured method is answered with the same return value, no matter what parameters are used in the method call. If I want to test that the Basket class calculates correct subtotals when containing different items, I can't easily use the SetupResult method. If I tried to add two different items to the Basket object, both items would get the same unit price, as the call to GetUnitPrice would return whatever I defined with SetupResult.

To solve this problem, there are other methods in the DynamicMock class I can use, as illustrated in **Figure 6**. In this test, I add two different BasketItem objects to the Basket object. During object instantiation, each BasketItem object will call the GetUnitPrice and GetProductName methods to retrieve information from the DAL.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 6 Setting Up Expectation with NMock

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl36_ctl00_ctl00_code');" \o "Copy Code)

DynamicMock dataAccess = new DynamicMock(typeof(IShoppingDataAccess));

dataAccess.ExpectAndReturn("GetUnitPrice", 99, 1);

dataAccess.ExpectAndReturn("GetProductName", "The Moon", 1);

dataAccess.ExpectAndReturn("GetUnitPrice", 47, 5);

dataAccess.ExpectAndReturn("GetProductName", "Love", 5);

Basket b = new Basket((IShoppingDataAccess)dataAccess.MockInstance);

b.AddItem(new BasketItem(1, 2,

(IShoppingDataAccess)dataAccess.MockInstance);

b.AddItem(new BasketItem(5, 1,

(IShoppingDataAccess)dataAccess.MockInstance);

b.Save();

decimal subTotal = b.CalculateSubTotal();

Assert.AreEqual(245, subTotal);

To configure the mock instance to return appropriate values for different parameter values, I use the ExpectAndReturn method. This method looks similar to the SetupResult method, but has a few important differences. The ExpectAndReturn method configures the mock instance to expect one call to the method, with the supplied parameter values. Instead of using parameter types to identify the target method, the method signature is inferred from the supplied parameter values.

When I use the following parameters

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl17_code');" \o "Copy Code)

ExpectAndReturn("GetUnitPrice", 99, 1);

the mock instance will return 99 when the GetUnitPrice method is called with a parameter value of 1. If the parameter value is different from 1, a VerifyException is thrown by NMock. Additionally, since the code in **Figure 6** defines the next expected call to GetUnitPrice to have a parameter value of 5, a VerifyException will be thrown if the method isn't called with that parameter. Since the mock instance is told to expect a total of two calls to GetUnitPrice, any third call would also result in an exception, even in cases where the parameter is 1 or 5.

**Let the Mock Participate in the Test**

As you've probably already noticed, the ExpectAndReturn method is a lot less forgiving than the SetupResult method. In my opinion, this is a good thing because it allows you to add a whole new level of control to your tests.

Consider a test where you use SetupResult. Aside from the obvious disadvantage that all calls to the same method return the same value, you don't get much validation out of the mock object. Regardless of whether your target library called a method zero, once, or many times, you wouldn't notice the difference. Conversely, you can use ExpectAndReturn to validate that a method was called the expected number of times, with the expected parameters, and in the expected order. With this approach, a unit test not only has the opportunity to test that a module returns the correct values, but also that it communicates with any external dependencies as designed. In fact, with some modules where most of the implementation is internal, this may very well be the only way you can perform tests in any meaningful way.

Thus, letting the mock object itself participate in the test is a very good idea, so I'll point out a few ways you can make NMock even less forgiving of the unexpected than its default settings allow.

If you take a good look at the unit test code in **Figure 6**, you may notice something odd: I didn't set an expectation for a call to SaveBasketItems, and even though the Save method is called on the Basket class, the test succeeds. This happens because SaveBasketItems returns void, so NMock can more or less just ignore the call. You can (and should) change this behavior by turning on the Strict option of the DynamicMock class, like this:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl18_code');" \o "Copy Code)

dataAccess.Strict = true;

The Strict property defaults to false, but if you change it to true, any call which does not have an expectation set will cause a VerifyException to be thrown. In the case of the unit test in **Figure 6**, turning on the Strict option will suddenly cause the unit test to fail, as it should. However, in this case the unit test fails because the test code is faulty, not because there is a bug in the test target, so the test code should be corrected.

It may seem obvious to use ExpectAndReturn to set up an expectation for the call to SaveBasketItems, and in fact you can do that by specifying null as the return value. However, for setting up expectations for void methods, the Expect method is a more intuitive alternative. It is used and behaves just like ExpectAndReturn, except that it doesn't take a parameter that defines a return value.

When setting up the expectation for SaveBasketItems, a problem manifests itself. The SaveBasketItems method takes as its first parameter a Guid, which is the ID of the Basket object. If you examine the definition of the Basket class in **Figure 3**, you will see that this ID is generated internally during object construction, but is not externally accessible (you may argue that this constitutes one of many design flaws of the Basket class, but it illustrates my point). Since I have no way of knowing the value of the basketID parameter in advance, how can I define the expectation? The answer lies in the NMock Constraint classes:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl19_code');" \o "Copy Code)

dataAccess.Expect("SaveBasketItems", new IsTypeOf(typeof(Guid)),

new BasketItem[]{item1, item2});

Instead of a Guid value, I use a new Constraint instance that instructs the mock object that any Guid value is acceptable. For the other parameter (the array of BasketItems), I still indicate the expected value, so although the basketID parameter is allowed to take any Guid value, the basketItems parameter is still as restricted as I can make it. In fact, NMock expectations always work with Constraint classes when setting parameter expectations. The IsEqual constraint is the default, which is why I haven't demonstrated this syntax until now.

The explicit syntax

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl20_code');" \o "Copy Code)

dataAccess.ExpectAndReturn("GetUnitPrice", 47, new IsEqual(5));

is equivalent to

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl21_code');" \o "Copy Code)

dataAccess.ExpectAndReturn("GetUnitPrice", 47, 5);

Even when using the Strict option, you are not guaranteed that the mock object validates that all methods are called the expected number of times. In the case where a method is called too many times, NMock throws an exception the first time this happens, but it doesn't automatically discover that a method was called fewer times than expected. Although it has no way of automatically knowing when the unit test is over, you can inform it manually:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl22_code');" \o "Copy Code)

dataAccess.Verify();

Calling the Verify method at the end of your unit test tells the mock object that the test is over and that it should verify that all members were accessed the expected number of times. If a member happened to be accessed fewer times than expected, a VerifyException would be thrown.

To summarize the steps thus far, you set the Strict option to true at the beginning, stick with the expectation-setting methods throughout, and call the Verify method at the end of the unit test. With very little effort, you'll have added a whole new level of validation to your unit tests.

**Designing for Testability**

As I mentioned previously, I have designed my classes so that I can pass a reference to the data access object in the constructor. When using dynamic mocks, the mock is only defined at run time so it is necessary to provide some way for the dependent module to be assigned from outside the consuming module itself. Of course, this does not mean this is the only way. Usually, I overload the constructor so that I have one constructor in which dependencies can be assigned at run time and one in which these parameters are not included. The second type of constructor will usually become the default. For example, the Basket class could be instantiated in the following way:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl23_code');" \o "Copy Code)

Basket b = new Basket();

The constructors which explicitly assign dependent objects at run time would typically be used only for tests. The constructor where the assignment is hidden would be used in production code. Why this distinction?

With the decoupled approach used throughout this article, modules communicate with other classes via interfaces. An interface defines methods, properties, indexers, and events, but it doesn't define constructors. This means that when a class needs to instantiate an interface implementation at run time, it has to make assumptions about the constructor.

In my opinion, the most reasonable assumption is to use the default parameterless constructor. Let's imagine that I want to use a discount calculation object with my shopping basket and I have two different types to choose from: a data-based discount class and an algorithm-based class, as illustrated in **Figure 7**. The data-based class uses a database to look up discounts defined there, whereas the algorithm-based class uses some internal algorithm to calculate discounts. When instantiating an implementation of the IMyDiscount interface, it doesn't make any sense to pass a reference to a data access module. The algorithm-based class doesn't need it, and at design time I don't know which of the two is going to be used. Using a constructor that passes a reference to a data access object would be making an unreasonable assumption about the implementation.

You may think that this is all very good, but then how can you define which interface implementation is going to be used at run time? The answer, of course, is by configuring the application. Consider a simple application configuration file for the Shopping library like the one shown here:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl24_code');" \o "Copy Code)

<?xml version="1.0" encoding="utf-8" ?>

<configuration>

<appSettings>

<add key="IShoppingDataAccessType"

value="NMockExample.DataAccess.ShoppingDataAccess, DataAccess" />

</appSettings>

</configuration>

In this configuration file, I define that for the implementation of the IShoppingDataAccess interface, my application should use the NMockExample.DataAccess.ShoppingDataAccess type, defined in the DataAccess assembly. The default constructor for the Basket class looks like this:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl25_code');" \o "Copy Code)

public Basket()

{

string typeName =

ConfigurationSettings.AppSettings["IShoppingDataAccessType"];

Type t = Type.GetType(typeName);

IShoppingDataAccess dataAccess =

(IShoppingDataAccess)Activator.CreateInstance(t);

Initialize(dataAccess);

}

As you can see, that's all there is to it. You get the string defining the type from the configuration file, create the type by calling the static GetType method of the Type class, and create an instance of the type by calling the static CreateInstance method of the Activator class. The CreateInstance method returns an object of type System.Object, so you need to cast it to the type you need to make further use of it.

One problem with this approach is that these three lines of code can't be unit tested with NMock. Either you need to create an empty stub assembly, or you need to test this code using your production implementation of the interface, which has all the problems I described at the beginning of the article. However, this is very simple code that always follows the same pattern: get the string from the configuration file, get the type using the string, and get an instance using the type. Chances are that if you have validated this code to work once, you shouldn't have any more problems with it. Errors might occur within this code, but that should only happen if the string happens to define a type or an assembly which couldn't be found, loaded, instantiated, or cast, or if there is a problem with the configuration file.

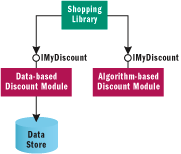


Figure 7 **Two Discount Classes**

If you are still not convinced that this is a good way to define the interface implementation in production code, consider the alternative. In the example from **Figure 7**, if you don't want to specify the implementation of IMyDiscount in the configuration file, you need to pass an instance of the implementation to the Shopping library at run time. In effect, this only defers the decision to the caller of the Shopping library. It is now the caller's responsibility to figure out which interface implementation to use.

If the decision is made at design time, it means that you end up hardcoding the implementation that should be used, and you lose most of the advantages of an interface-based design.

If the decision is made at run time, it needs to involve code similar to what I just described. As a result, not much has been gained, but you have now moved the decision to a place far from the module where it is needed.

Consider a shopping app that contains a shopping basket, which may need to calculate discounts. Imagine that the Basket class has been extended a bit to implement an IBasket interface and to consume the IMyDiscount interface. The shopping app will instantiate and manipulate an IBasket object, which may or may not be an instance of the Basket class. If you want to use the Basket class, you need to pass it an IMyDiscount object. One implementation of IMyDiscount (the data-based discount module) needs a data access object, so you need to pass this object to the Basket class as well. Now you have not only made the assumption that you are going to use the Basket class, but also that the Basket class is going to use the data-based discount module. This cannot be changed without modifying the code of both the shopping application itself and the Basket class (depending on what you want to change).

Given all of these potential problems, I find the most practical course to be to let any module itself determine and instantiate the interface implementations it consumes.

**Conclusion**

In real-world software development projects, it is a rare library that doesn't have some kind of external dependency with which it communicates frequently. When unit testing libraries, you need some kind of stub to simulate those external dependencies. NMock provides you with a valuable tool for achieving this with minimal effort. Although writing unit tests still involves work, NMock offers relief from writing, configuring, and deploying your own stub classes over and over again. Defining mock instances and their behaviors at run time also gives you a degree of flexibility and test validation you probably never would have been able to achieve if you wrote your own stubs for each external dependency.

Each time I found myself in a situation in which it seemed like NMock was holding me back, it turned out to be a good time to pause and rethink my design. Each time I discovered that by making my own design better, I could overcome my problem with unit testing and NMock as well. So NMock is not only a great tool for unit testing, it also forces you to make improvements to your own object-oriented code design.

**Mark Seemann**works for Microsoft Consulting Services in Copenhagen, Denmark, where he helps Microsoft customers and partners architect, design, and develop enterprise-level applications. Mark can be reached at [mseemann@microsoft.com](mailto:mseemann@microsoft.com).