Implementing Dynamic Interfaces

By Bill Wagner

June 2010

One great advantage of dynamic programming is the ability to build types whose public interfaces change at runtime based on how you use these types. C# provides that ability through dynamic, the System.Dynamic.DynamicObject class, and the System.Dynamic.IDynamicMetaObjectProvider interface. Using these tools, you can create your own types that have dynamic capabilities.

The simplest way to create a type with dynamic capabilities is to derive from System.Dynamic.DynamicObject. That type implements the IDynamicMetaObjectProvider interface using a private nested class. This private nested class does the hard work of parsing expressions and forwarding those to one of a number of virtual methods in the DynamicObject class. That makes it a relatively simple exercise to create a dynamic class, if you can derive from DynamicObject. There are a number of examples on MSDN that discuss how to use this class. A great place to start is this article: Walkthrough: Creating and Using Dynamic Objects.

Using DynamicObject makes it much easier to implement a type that behaves dynamically. DynamicObject hides much of the complexity of creating dynamic types. It has quite a bit of implementation to handle dynamic dispatch for you. But sometimes you want to create a dynamic type and you can't use DynamicObject because you need a different base class. Or, you may want to have additional control over the mechanism involved implementing the dynamic behavior.

For that reason, I'm going to show you how to create the dynamic dictionary by implementing IDynamicMetaObjectProvider yourself, instead of relying on DynamicObject to do the heavy lifting for you. Using IDynamicMetaObjectProvider ties into the DLR infrastructure at a lower level. The DLR extensively uses expression trees to implement dynamic behavior. Fully covering the expression trees is beyond the scope of this article, but you can read the MSDN article on expression trees and documentation on the DLR Web site.

Implementing IDynamicMetaObjectProvider means implementing one method: GetMetaObject. Here's a version of DynamicDictionary that implements IDynamicMetaObjectProvider, instead of deriving from DynamicObject:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Dynamic;
using System.IO;
using System.Linq.Expressions;

class DynamicDictionary : IDynamicMetaObjectProvider
{
```

```
#region IDynamicMetaObjectProvider Members
DynamicMetaObject IDynamicMetaObjectProvider.GetMetaObject(
    System.Linq.Expressions.Expression parameter)
{
    return new DynamicDictionaryMetaObject(parameter, this);
}
#endregion

private class DynamicDictionaryMetaObject : DynamicMetaObject
{
    internal DynamicDictionaryMetaObject(
        System.Linq.Expressions.Expression parameter,
        DynamicDictionary value)
        : base(parameter, BindingRestrictions.Empty, value)
    {
    }

    public override DynamicMetaObject BindSetMember(SetMemberBinder binder,
        DynamicMetaObject value)
    {
        // Method to call in the containing class:
        string methodName = "SetDictionaryEntry";

        // setup the binding restrictions.
        BindingRestrictions restrictions =
            BindingRestrictions.GetTypeRestriction(Expression, LimitType);

        // setup the parameters:
        Expression[] args = new Expression[2];
        // First parameter is the name of the property to Set
        args[0] = Expression.Constant(binder.Name);
        // Second parameter is the value
        args[1] = Expression.Convert(value.Expression, typeof(object));

        // Setup the 'this' reference
        Expression self = Expression.Convert(Expression, LimitType);

        // Setup the method call expression
        Expression methodCall = Expression.Call(self,
                typeof(DynamicDictionary).GetMethod(methodName),
                args);

        // Create a meta object to invoke Set later:
        DynamicMetaObject setDictionaryEntry = new DynamicMetaObject(
            methodCall,
            restrictions);
        // return that dynamic object
        return setDictionaryEntry;
    }
```

```
    public override DynamicMetaObject BindGetMember(GetMemberBinder binder)
    {
        // Method call in the containing class:
        string methodName = "GetDictionaryEntry";

        // One parameter
        Expression[] parameters = new Expression[]
        {
            Expression.Constant(binder.Name)
        };

        DynamicMetaObject getDictionaryEntry = new DynamicMetaObject(
            Expression.Call(
                Expression.Convert(Expression, LimitType),
                typeof(DynamicDictionary).GetMethod(methodName),
                parameters),
            BindingRestrictions.GetTypeRestriction(Expression, LimitType));
        return getDictionaryEntry;
    }

    public override DynamicMetaObject BindInvokeMember(
        InvokeMemberBinder binder, DynamicMetaObject[] args)
    {
        StringBuilder paramInfo = new StringBuilder();
        paramInfo.AppendFormat("Calling {0}(", binder.Name);
        foreach (var item in args)
            paramInfo.AppendFormat("{0}, ", item.Value);
        paramInfo.Append(")");

        Expression[] parameters = new Expression[]
        {
            Expression.Constant(paramInfo.ToString())
        };
        DynamicMetaObject methodInfo = new DynamicMetaObject(
            Expression.Call(
            Expression.Convert(Expression, LimitType),
            typeof(DynamicDictionary).GetMethod("WriteMethodInfo"),
            parameters),
            BindingRestrictions.GetTypeRestriction(Expression, LimitType));
        return methodInfo;
    }
}

private Dictionary<string, object> storage = new
    Dictionary<string, object>();

public object SetDictionaryEntry(string key, object value)
{
    if (storage.ContainsKey(key))
        storage[key] = value;
```

```
            else
                storage.Add(key, value);
            return value;
        }

        public object GetDictionaryEntry(string key)
        {
            object result = null;
            if (storage.ContainsKey(key))
            {
                result = storage[key];
            }
            return result;
        }

        public object WriteMethodInfo(string methodInfo)
        {
            Console.WriteLine(methodInfo);
            return 42; // because it is the answer to everything
        }

        public override string ToString()
        {
            StringWriter message = new StringWriter();
            foreach (var item in storage)
                message.WriteLine("{0}:\t{1}", item.Key, item.Value);
            return message.ToString();
        }
    }
```

GetMetaObject() returns a new DynamicDictionaryMetaObject whenever it is called. Here's where the first complexity enters the picture. GetMetaObject() is called every time any member of the DynamicDictionary is invoked. Call the same member 10 times, GetMetaObject() gets called 10 times. Even if your methods are statically defined in DynamicDictionary, GetMetaObject() will be called, and will intercept those methods to invoke possible dynamic behavior. Remember that dynamic objects are statically typed as dynamic, therefore have no compile time behavior defined. Every member access is dynamically dispatched.

The DynamicMetaObject is responsible for building an Expression Tree that executes whatever code is necessary to handle the dynamic invocation. Its constructor takes the expression and the dynamic object as parameters. After the dynamic meta object is constructed, one of its Bind methods is called. The method's responsibility is to construct another DynamicMetaObject that contains the expression to execute the dynamic invocation. The first DynamicMetaObject is responsible for creating expression trees whenever an operation must be bound to a target. (That operation could be anything: an operator, property accessor, or method call). The second DynamicMetaObject is responsible for executing an expression tree for a single operation on a single object. That may seem a bit cumbersome, but it helps

the DLR's caching mechanism with providing more efficient dispatch. Let's walk through the two Bind methods necessary to implement the DynamicDictionary: BindSetMember and BindGetMember.

BindSetMember constructs an expression tree that will call DynamicDictionary.SetDictionaryEntry() to set a value in the dictionary. Here's its implementation:

```
public override DynamicMetaObject BindSetMember(SetMemberBinder binder,
    DynamicMetaObject value)
{
    // Method to call in the containing class:
    string methodName = "SetDictionaryEntry";

    // setup the binding restrictions.
    BindingRestrictions restrictions =
        BindingRestrictions.GetTypeRestriction(Expression, LimitType);

    // setup the parameters:
    Expression[] args = new Expression[2];
    // First parameter is the name of the property to Set
    args[0] = Expression.Constant(binder.Name);
    // Second parameter is the value
    args[1] = Expression.Convert(value.Expression, typeof(object));

    // Setup the 'this' reference
    Expression self = Expression.Convert(Expression, LimitType);

    // Setup the method call expression
    Expression methodCall = Expression.Call(self,
            typeof(DynamicDictionary).GetMethod(methodName),
            args);

    // Create a meta object to invoke Set later:
    DynamicMetaObject setDictionaryEntry = new DynamicMetaObject(
        methodCall,
        restrictions);
    // return that dynamic object
    return setDictionaryEntry;
}
```

Metaprogramming quickly gets confusing, so let's walk through this slowly. The first line sets the name of the method called in the DynamicDictionary, "SetDictionaryEntry". Notice that SetDictionary returns the right hand side of the property assignment. That's important because this construct must work:

DateTime current = propertyBag2.Date = DateTime.Now;

Without setting the return value correctly, that construct won't work.

Next, this method initializes a set of BindingRestrictions. Most of the time, you'll use restrictions like this one, restrictions given in the source expression, and for the type used as the target of the dynamic invocation.

The rest of the method constructs the method call expression that will invoke SetDictionaryEntry() with the property name, and the value used. The property name is a constant expression, but the value is a Conversion expression that will be evaluated lazily. Remember that the right hand side of the setter may be a method call or expression with side effects. Those must be evaluated at the proper time. Otherwise, setting properties using the return value of methods won't work:

propertyBag2.MagicNumber = GetMagicNumber();

Of course, to implement the dictionary, you have to implement BindGetMember as well. BindGetMember works almost exactly the same way. It constructs an expression to retrieve the value of a property from the dictionary.

```
public override DynamicMetaObject BindGetMember(
    GetMemberBinder binder)
{
    // Method call in the containing class:
    string methodName = "GetDictionaryEntry";

    // One parameter
    Expression[] parameters = new Expression[]
    {
        Expression.Constant(binder.Name)
    };

    DynamicMetaObject getDictionaryEntry = new DynamicMetaObject(
        Expression.Call(
            Expression.Convert(Expression, LimitType),
            typeof(DynamicDictionary).GetMethod(methodName),
            parameters),
        BindingRestrictions.GetTypeRestriction(Expression, LimitType));
    return getDictionaryEntry;
}
```

I add one more function so that you can see a bit more about how IDynamicMetaObjectProvider does its work. I create an implementation of BindInvokeMember() that prints the method name and all parameters to the console for any method that you call. This shows some of the extra work that you need to do to implement more dynamic behavior in an object. When your type involves dynamic methods, suddenly you need to determine more than just the property name to get or set. You need to look at each parameter to a method, including its type information. Remember that you also need to know the target of the method invocation. You already saw how to use the Expression property of the DynamicMetaObject to find the correct target.

To add runtime dispatch of methods, you override BindInvokeMember.The first parameter to BindInvokeMember gives you information about the method: its name, and return type. The second parameter contains the list of parameters. Notice that the parameter list is not a System.Object array. Instead, it's an array of DynamicMetaObjects. You're working with dynamic objects, so it makes sense that the parameters may themselves be dynamic, right? My override takes a rather simple way out, and just looks at the value of each DynamicMetaObject in the array of parameters, and creates a string that displays that information for you. Notice that you need to build up an expression tree containing the

code you want executed in order to correctly override BindInvokeMember. The underlying framework will execute the expression to provide the value for you:

```
public override DynamicMetaObject BindInvokeMember(
    InvokeMemberBinder binder, DynamicMetaObject[] args)
{
    StringBuilder paramInfo = new StringBuilder();
    paramInfo.AppendFormat("Calling {0}(", binder.Name);
    foreach (var item in args)
        paramInfo.AppendFormat("{0}, ", item.Value);
    paramInfo.Append(")");

    Expression[] parameters = new Expression[]
    {
        Expression.Constant(paramInfo.ToString())
    };
    DynamicMetaObject methodInfo = new DynamicMetaObject(
        Expression.Call(
        Expression.Convert(Expression, LimitType),
        typeof(DynamicDictionary).GetMethod("WriteMethodInfo"),
        parameters),
        BindingRestrictions.GetTypeRestriction(Expression, LimitType));
    return methodInfo;
}
```

Before you go off and think this isn't that hard, let me leave you with some thoughts from the experience writing this code. This is about as simple as a dynamic object can get. You have two APIs: property get, property set. The semantics are very easy to implement. Even with this very simple behavior, it was rather difficult to get right. Expression trees are hard to debug. They are hard to get right. More sophisticated dynamic types would have much more code. That would mean much more difficulty getting the expressions correct.

Furthermore, keep your mind on one of the opening remarks I made: every invocation on your dynamic object will create a new DynamicMetaObject, and invoke one of the Bind members. You'll need to write these methods with an eye toward efficiency and performance. They will be called a lot, and they have much amount of work to do.

Implementing dynamic behavior can be a great way to approach some of your programming challenges. When you look at creating dynamic types, your first choice should be to derive from System.Dynamic.DynamicObject. On those occasions where you must use a different base class, you can implement IDynamicMetaObjectProvider yourself, but remember that this is a complicated problem to take on. Furthermore, any dynamic types involve some performance costs, and implementing them yourself may make those costs greater.

The ExpandoObject is the most common use case for creating a dynamic object. An Expando object can easily implement the properties that come across the network in JSON or XML format. You can implement more dynamic behavior for different JSON types. Leveraging dynamic capabilities means that you can create fewer concrete types, and yet have data driven types implement behavior based on the data they contain. For example, your JSON object may know how to transform itself into HTML display.

You may create object that know how to control navigation in your web application. In general, you should consider creating your own dynamic implementations when you see that the behavior of a type is driven by the data it contains. Phil Haack wrote about this concept in his article: Fun With Method Missing and C# 4, where he implemented method_missing (from Ruby) in C# 4.0. As you look around you can find several examples of creating dynamic objects, as long as you look outside of the familiar C# background. Look at the dynamic programming models from the Ruby and Python communities. You'll see many different techniques that you can now leverage in C# 4.0.