C# - Reflection

(An overview of the Reflection capabilities of .Net)
Originally released in 2013, updated in 2017.

Marc Stan
marc@marcstan.net

ABSTRACT

In computer science, reflection is the ability of a computer program to examine and modify the structure and behavior of itself at runtime. In .Net the name is also used for the ability of the framework to reflect itself. The following work will provide an overview over the Reflection capabilities of the .Net framework. Any sample code is in C#.

1. REQUIREMENTS

A set of code samples accompanies this document. They can be downloaded from my site or gitlab. In order to run them, Visual Studio 2015 (or later) as well as the .Net Framework 4.5.2 (or later) are required. One sample will also use PostSharp (an 'aspect oriented framework'). To run the sample, PostSharp is downloaded from nuget. Upon first run it will require you to aquire a license. There is a free license that is more than adequate for the sample. In order to remove PostSharp afterwards, kill the PostSharp.Compiler Service in taskmanager and delete the "C:\ProgramData\PostSharp" directory.

2. INTRODUCTION

The document is split into multiple parts:

- A general overview over Reflection in .Net with information about the metadata stored by .Net, as well as an overview over attributes and their use cases
- A comparison with the keyword `dynamic` which came with .Net 4.0 and
- A deep dive into the Reflection.Emit namespace and dynamic code generation in general

3. REFLECTION

In .Net every class automatically derives from `System.Object`. `System.Object` contains a set of methods that is thus accessible on each instance of each class. One of these methods is

```
Type GetType();
```

Listing 1: Type-Class

which returns information about the given type. This information is **instance independent**. It contains all sorts of information including: whether the class is abstract, public, protected, .. as well as the list of Interfaces it implements and the set of members it has.

3.1 Metadata

In the first sample '1-Metadata' we take a closer look at the metadata exposed by the Type class. The solution contains two projects: A library ('CarFactory') and a console application ('1-Metadata'). The console application references the library for the **sole purpose** of copying the dll during build time, so that is accessible during runtime for the '1-Metadata' application. The same could be achieved, by manually building 'CarFactory', then copying the dll to the '1-Metadata' output folder. Again: No type of 'CarFactory' is directly referenced.

The code shows, that by using the static `LoadFrom` method on the assembly class, the user is able to get an instance of the assembly class that contains detailed information about a specific assembly (in this case CarFactory.dll).

The code then iterates over all types and outputs information for each type, as well as information for all members for each type. As is visible in the code, `GetMembers` returns all members of a class. This includes Properties, Methods, Fields and Events. For the simplicity of this sample, detailed information is emitted only for Fields, Methods and Properties. The information is also limited to the name of each member, as well as its Type (return type of methods and types of Properties/Fields).

We can already see two things here:

- First of all there is a very detailed level of information accessible, thanks to the metadata stored by .Net. This allows all sorts of programs that otherwise couldn't be written at or (or at least not without manually writting lots of redundant code).
- Secondly, as seen in the output of the sample, there seem to be some additional methods defined that one would not expect at first. The methods `ToString`, `GetHashCode`, `GetType` and `Equals` are part of the `object` class, which we now know is the base class for each .Net entity. Additionally there are get_* and set_* methods which we did not define. However they look similar to the properties we have defined in our sample class.

The reason why .Net generates get_* and set_* methods is simple: For each property that has been defined using the short syntax

```
public int MyProperty { get; set; }
```

Listing 2: Auto. Property

.Net actually creates a backing field and two **Methods** that access the field:

```
private int _myProperty;
public int get_MyProperty()
{
        return _myProperty;
}
public void set_MyProperty(int m)
{
        _myProperty = m;
}
```

Listing 3: Compiler generated code

Each programmer that tries to use the property will still see it as a property, however when it is compiled, the property access is actually converted to method calls. The private backing field that is compiler generated will also have a name that is not conficting with any member of the class (and won't be accessible to the developer at all), however it is not possible to add get/set Methods with the signature described above, as the compiler reserves these names as soon as a property is added to the class (see the next listing for an example).

```
public int MyProperty { get; set; }
public void set_MyProperty(int m)
{
        // this method cannot be named
           that way,
        // as the signature void
           set_MyProperty(int) is
           reserved by the compiler due
           to int MyProperty existing in
           the class
        // the same counts for the int
           get_MyProperty signature
}
```

Listing 4: Compiler generated Code reserves get & set names

As demonstrated, it is very simple to access the metadata in .Net. The `System.Type` class provides a rich set of properties and methods that allow developers to query almost any aspect of a given class at runtime. This set of information is static: It cannot be edited during runtime for an existing type and it cannot be directly extended either.
In the next section we will take a look at extending this information set.

3.2 Attributes

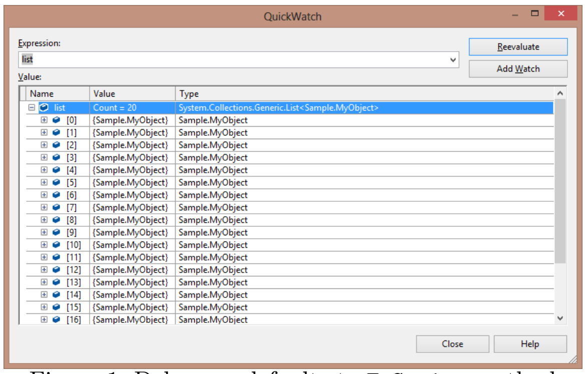
In order to extend the information for each object, .Net introduces attributes. Attributes are merely annotations that allow the user to extend elements with custom data. Attributes make it very convenient to append data to existing elements.
In .Net, attributes can be defined on a all sorts of different elements: classes, members, parameters and even the assembly itself. The attribute data is stored alongside the other metadata and can be easily retrieved using reflection.
There are many use cases where attributes are helpful. Examples include:

- Unit testing: appending attributes to classes/methods to mark them as 'runnable' by the unit test framework (e.g. `TestAttribute` on each method and `TestFixture` on each class, as done by NUnit. When starting the Unit testing tool, it will automatically determine, which tests to run)

- Serialization: appending attributes to classes/properties to mark them as serializable, so that e.g. `System.Xml.XmlSerializer` will automatically serialize the specific properties to an XML file. Without these attributes, the user would have to write the entire serialization code by himself, easily adding hundreds of lines of code and unnecessary complexity.

- Enriching the debugger: While the .Net debugger itself is already very powerful, .Net developers will often face the debugger showing unhelpful information. Notably the displaying of lists, which often causes the elements to be formated as their classnames.

  The next figure shows the issue:



Name	Value	Type
list	Count = 20	System.Collections.Generic.List<Sample.MyObject>
[0]	{Sample.MyObject}	Sample.MyObject
[1]	{Sample.MyObject}	Sample.MyObject
[2]	{Sample.MyObject}	Sample.MyObject
[3]	{Sample.MyObject}	Sample.MyObject
[4]	{Sample.MyObject}	Sample.MyObject
[5]	{Sample.MyObject}	Sample.MyObject
[6]	{Sample.MyObject}	Sample.MyObject
[7]	{Sample.MyObject}	Sample.MyObject
[8]	{Sample.MyObject}	Sample.MyObject
[9]	{Sample.MyObject}	Sample.MyObject
[10]	{Sample.MyObject}	Sample.MyObject
[11]	{Sample.MyObject}	Sample.MyObject
[12]	{Sample.MyObject}	Sample.MyObject
[13]	{Sample.MyObject}	Sample.MyObject
[14]	{Sample.MyObject}	Sample.MyObject
[15]	{Sample.MyObject}	Sample.MyObject
[16]	{Sample.MyObject}	Sample.MyObject

Figure 1: Debugger defaults to `ToString` method

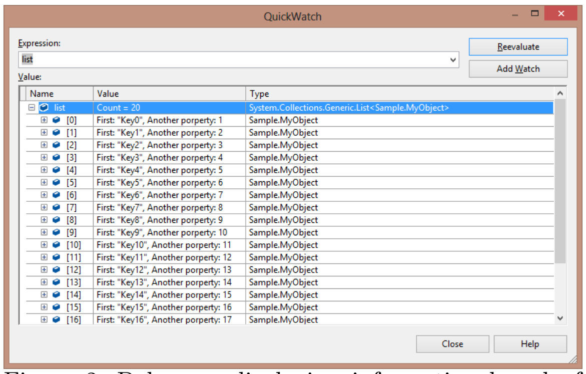
When the debugger displays an object, it defaults to the `ToString` method of the given object. If the `ToString` method was not overriden, it defaults to the full name (including namespace) of the object. As can be seen in figure 1, this is very unhelpful, when dealing with lists as each element has to be manually expanded to see all properties.

A workaround would be to manually override the `ToString` method and formatting its return value so that it displays all helpful information. This is often problematic, as the `ToString` method may already be used in production code to display data to the user.

The .Net framework provides an alternative in the form of the `System.Diagnostics.DebuggerDisplayAttribute`.

By applying the attribute to a class, the programmer is able to alter the display behaviour of the given class, without having to modify the `ToString` method because the debugger favors the display option selected by the attribute over the default `ToString` method.

The next figure shows the objects formatted based of a rule applied by the `DebuggerDisplayAttribute`:



Expression: list

Value:

Name	Value	Type
list	Count = 20	System.Collections.Generic.List<Sample.MyObject>
[0]	First: "Key0", Another porperty: 1	Sample.MyObject
[1]	First: "Key1", Another porperty: 2	Sample.MyObject
[2]	First: "Key2", Another porperty: 3	Sample.MyObject
[3]	First: "Key3", Another porperty: 4	Sample.MyObject
[4]	First: "Key4", Another porperty: 5	Sample.MyObject
[5]	First: "Key5", Another porperty: 6	Sample.MyObject
[6]	First: "Key6", Another porperty: 7	Sample.MyObject
[7]	First: "Key7", Another porperty: 8	Sample.MyObject
[8]	First: "Key8", Another porperty: 9	Sample.MyObject
[9]	First: "Key9", Another porperty: 10	Sample.MyObject
[10]	First: "Key10", Another porperty: 11	Sample.MyObject
[11]	First: "Key11", Another porperty: 12	Sample.MyObject
[12]	First: "Key12", Another porperty: 13	Sample.MyObject
[13]	First: "Key13", Another porperty: 14	Sample.MyObject
[14]	First: "Key14", Another porperty: 15	Sample.MyObject
[15]	First: "Key15", Another porperty: 16	Sample.MyObject
[16]	First: "Key16", Another porperty: 17	Sample.MyObject

Figure 2: Debugger displaying information based of `DebuggerDisplayAttribute`

- File importer: As can be seen in the next code sample ('2-Attributes') further down.

Most programmers tend to use (framework) attributes more often than they create their own. While serializing a XML file manually takes many lines of code, even with the advanced `System.Xml.XDocument` class, using the `Serializable` attribute on a class to tell the default serializer to save and reload all properties of the given class is almost no work. The saving & loading is done by helper classes that are already in the .Net Framework (notably `System.Xml.XmlSerializer`).

If a special requirement arises which requires the use of custom attributes, the user can define his own attributes very easily. In order to create an attribute, the user must create a new class and derive it from the `System.Attribute` class. Additionally the newly created class name must end in Attribute (by convention). If the attribute was to be named `ImporterType`, name it `ImporterTypeAttribute`. The compiler will substitute one for the other, allowing you to omit the `Attribute` suffix when applying the attribute to a member.

The next code listing shows an attribute as could be used by a unit testing framework:

```
[AttributeUsage(AttributeTargets.Method,
    AllowMultiple = false]
public class TestMethodAttribute :
    Attribute
{

}
```

Listing 5: Attribute class

The newly generated attribute class already has a framework attribute applied to itself: `System.AttributeUsage`, which dicates several things: First of all, on which targets the new user created attribute is valid: This can be anything from assembly level over member level to parameter level (even combinations by using logical ORs). In this case, the attribute is only valid on methods. Secondly it tells the framework, whether multiple attributes can be applied to the same target (which in this sample is obviously not desired, since each method can only be one: a **test method** or **not a test method**). The newly generated attribute class could now be filled with properties/constructors, which allow the user to add custom information into the attribute.

Now, a unit testing tool could iterate over all types in a assembly and check for the specific attribute and run the methods. In order to work, the method must not take any arguments (this is a convention held by most unit test frameworks, as it is very difficult to determine the correct values for arguments in an automated system). We will however not build our own Unit testing framework.

We rather take a look at a potential file importer plugin mechanism. Importing different file types is a requirement in many applications these days (Microsoft Office, Open Office and Autodesk 3DS Max to name a few). They all need to import dozens of different file formats in order to satisfy the wide range of users, which may often have files in formats used in different software products (consider Word **.doc** vs. Word **.docx** vs. Open Office **.odt**). In many cases, the programmers of the software may not know all file formats they need to import ahead of time (e.g. due to new standards being developed) or simply too many file formats as that the developers could write importers for every one of them. In that case, the developers may decide to build a plugin system that can be extended by the customer or other programmers.

The code sample '2-Attributes' contains such an importer. The sample is split into three libraries: '2-Attributes' (console application), 'ImporterBase' (contains the API for the plugin mechanism) and 'ImporterLib', which already contains a few implementations to 'support' a few basic file formats (these classes are actually only stubs to show the concept).

Again, the sample application '2-Attributes' only holds a reference to the ImporterLib in order to load it automatically but doesn't directly reference anything from it (same as in the first sample). Upon running, the demo application loads the ImporterLib into the current AppDomain and then iterates over all types in it to find every type that has the `ImporterType` attribute applied to it and derives from our `Importer` class.

In the end, each importer will be queried for information about the file type it supports. As can be seen in the code: There are four "dummy" importers provided by default: **.db**, **.xml**, **.temp** and **.dummy**.

While this sample is very basic and lacks safety checks, it shows the basic concept behind the loading mechanism of a plugin system based of information stored in attributes.

It is very easy for third party programmers to conform to the API and write their own importer.

The only problem the sample faces is that it does not allow arbitrary assemblies to be loaded (you can define the PRODUCTION preprocessing directive to allow loading of any dlls in this sample, but the next sample will go into more detail).

Sample '3-PluginSystem' will solve this issue, by loading all dll's from a given directory (so clients can place their plugins in a specific directory and will be loaded, too).

It is a common thing to simply load all assemblies from the 'plugins' subdirectory. Then a load routine evaluates each type in the loaded assemblies and checks whether it is a valid plugin. If so, the plugin gets instanciated and initialized and can be used just as any other `class`.

This sample demonstrated the flexibility of a pluginsystem. Each plugin (which in this case needs to implement the `IPlugin` interface) will be loaded and can specify for which `EntryPoint` it should be loaded. In a real world application, plugins would also have a priority for running and could be user configured (activated/deactivated). For the simplicity of this sample, these features are not included.

In this section examples where the usage of `Attributes` is useful or even required have been discussed and the extensibility of the .Net metadata system has been displayed.

4. DYNAMIC

The `dynamic` keyword was introduced in .Net 4.0. Scott Hanselman [3] describes the `dynamic` keyword very well: "Oh, well it's statically-typed as a dynamic type."

To understand that, we have to back up a bit. In .Net there exists the CLR (Common Language Runtime), which is essentially the broker between high level C# code (or really any statically typed language that can be used with .Net such as C#, F# or VB.Net) and the low level machine code (native code).

When compiling a .Net executable or library, the output is an assembly that contains not directly the low level machine code, but rather an assembler-like abstraction that is **platform independent** and named IL or MSIL (short for "Microsoft Intermediate Language"). When the program executes on a given platform, the CLR is responsible for managing the code.

This includes: managing the memory, garbage collection and threading as well as converting the IL code to the actual low level code (x86/x64/..) on a method per method basis (also know as JIT or "Just in time" compilation).

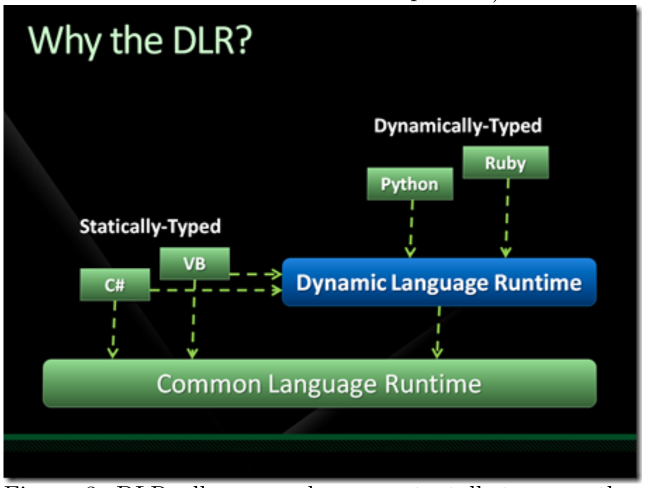


Figure 3: DLR allows any language to talk to any other language

As can be seen in the figure, the DLR (Dynamic Language Runtime) sits ontop of the CLR and essentially allows any language to talk to any other language. This means that - as can be seen in [3] - you can call C# code from a (Iron)Python script and vica versa very easily.

With the DLR Microsoft also introduced the `dynamic` keyword, which is not to be confused with the `var` keyword in C#.

`var` simply implies that the developer is "too lazy" to write out the type:

```
var list = new List<int>();
List<int> list = new List<int>();
```

Listing 6: var keyword

For the compiler these two previous lines of code are identical, as the type of the `list` instance can be directly inferred from the statement to the right (and will be inferred during build time - allowing the developer to see the type and its members in IntelliSense). This is also possible for method calls and pretty much everything else in C#, since it is a **statically typed** language.

An example where var is very helpful is `Linq` code. In this paper the workings of `Linq` will not be described further. Simply consider the following code:

```
var list = new List<int>();
var result = (from x in list
              where x > 0
              select x);
```

Listing 7: Linq statement

The type of result in this case would be `IEnumerable<int>`. More advanced `Linq` statements will have even more complicated return types. As a develper you may not know the type ahead of the query, so you would need to write the query, look at the return value and then define it for the variable. With `var` you don't need to.

Also anonymous types [4] exist since C# 4.0, which require `var`, however they will not be described in this work.

The `dynamic` keyword is another matter.

Instead of checking the type at compile type, the evaluation is postponed until runtime. To better understand that process, we will consider the following class:

```
public class WebCalc
{
        public int Add(int x, int y)
        {
                return x + y;
        }
}
```

Listing 8: Basic calculator

A simple calculator that adds numbers. It is written in C#, but if it was written in Python or Ruby, the following code would not work, since we could not directly instanciate it in C#:

```
WebCalc calc = GetCalc(); // returns an
    instance of the WebCalc that was
    defined in python
// does not work, since the type of calc
    is not known
int result = calc.Add(1, 2);
```

Listing 9: Basic calculator

`dynamic` provides help in these scenarios.

```
dynamic calc = GetCalc();
int result = calc.Add(1, 2);
```

Listing 10: Dynamic

The `dynamic` keyword tells the compiler to ignore the return type of `GetCalc` and also all further typesafety of `calc` at compile time.

At runtime, the `dynamic` keyword will cause the framework to dynamically resolve the type. Only then will be checked, whether the `Add` method actually exists.

The same could be achieved using reflection:

```
object calc = GetCalc();
Type t = calc.GetType();
int result = calcType.InvokeMember("Add",
    BindingFlags.InvokeMethod, null, calc,
     new object[] { 1, 2 });
int actualValue = Convert.ToInt32(result);
```

Listing 11: Dynamic

But as is clearly visible, using reflection requires a lot more code.

Reflection by itself is also slower than the `dynamic` keyword. The `dynamic` keyword can be described as reflection with cached/compiled code, which will be generated once and then executed faster than using the indirections reflection produces.

`dynamic` makes it very convenient to access other languages like Ruby or Python.

When developing an application in only C# there is **almost** no need for the `dynamic` keyword and its use is almost always a code smell as its usage defeats the purpose of C#'s static typing.

5. DYNAMIC CODE GENERATION

So far we have learned how to query basic metadata in .Net, how to append custom metadata and have seen a few examples where the metadata is very useful. The next section will take a look at dynamic code generation.

5.1 Use cases

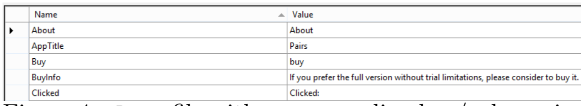
There are quite a few use cases where emitting code at runtime is helpful or even required. Since they are more complicated and often not intuitive at first, we will look over a few examples, before actually looking at code generation.

- The first example are similar classes. Take the `Vector2`, `Vector3` and `Vector4` classes for example (as they exist in e.g. the XNA Framework). They all have similar code concerning `Add`, `Subtract`, `Multiply` and probably a few more methods. The code for these methods is obviously very similar as it takes all the fields (`X`, `Y`, `Z` and even `W` depending on the dimension of the vector) on the given class and performs operations on them.

  While writting the same code trice may not seem like much effort, that can quickly change if the requirement changes and forces you to write `Vector5` through `Vector16` classes.

- Another example is the generation of code based of a xml file (or its corresponding xsd file). The Microsoft `.resx` format does this for example. The user can use the `.resx` file to store translations for many different languages by assigning each translation a unique key. (For information about the `.resx` file see section 7).

  A programmer can then access they key in his program by one of two ways. Consider the following table which I have stored as `Translations.resx` in my project (not included with this paper):

Name	Value
About	About
AppTitle	Pairs
Buy	buy
BuyInfo	If you prefer the full version without trial limitations, please consider to buy it.
Clicked	Clicked:

  Figure 4: `.Resx` file with corresponding key/value pairs

  The translations are accessible either by using the following code:

```
// create once and store globally
ResourceManager mgr = new
    ResourceManager("Namespace.of.your
    .app.NameOfResxFile", Assembly.
    GetExecutingAssembly());
var resourceCulture = Thread.
    CurrentThread.CurrentUICulture;

// access in code to return specific
    translation
string translated = mgr.GetString("
    AppTitle", resourceCulture);
```

  Listing 12: Resource manager

  Or by simply accessing `Translations.AppTitle` (where **Translations** is the name of the `.resx` file).

  The second method is possible, because Visual Studio has a tool that can automatically generate code for .resx files. You can take a look at that code, by looking at the `Translations.resx.cs` class (which is the 'code behind' class for the resx file). You will see that it contains a static string property for each line in the table, making it very convinient for programmers to access the translations. (In order to generate a `.resx` file, create a new project and choose add -> 'new file'. The `.resx` filetype will be selectable).

```
/// <summary>
///   Looks up a localized string
    similar to Pairs.
/// </summary>
public static string AppTitle {
    get {
        return ResourceManager.
            GetString("AppTitle",
            resourceCulture);
    }
}
```

  Listing 13: Localized string

- Another example are proxy classes. Proxy classes are often used with object relation mappers (ORM).

  An object relation mapper essentially takes a set of .Net classes and maps them to a database, so that each class represents one table (or rather one line in a specific table).

```
public class Author
{
  public string Name { get; set; }
  public bool Admin { get; set; }
  public DateTime Birthday { get; set;
     }
  public int PostCount { get; set; }
}
```

  Listing 14: Author class

  The `Author` class can be very easily used in the code.

  Using an ORM (like NHibernate) allows to automatically create database table with columns for the `Author` class. The developer then has a `List<Author>` object (or similar) that represents his table, without having to write any SQL statements.

  ORM will often create proxy objects (e.g. `AuthorProxy`) that wrapps the original `Author` class and derives from it, so the programmer doesn't see any difference between accessing a `AuthorProxy` and an `Author` object. The proxy object is then used to lazy load data from the database and to tell the database when to edit/reload rows and tables. The proxy class essentially wraps each member of the original class and introduces additional logic to each property/field (e.g. a flag indicating whether any property has been edited and needs to be saved back to the database).

Since database access needs to be fast, using reflection during runtime would be the wrong approach. Instead proxy classes are created at startup - since the database schema is already known - allowing for fast access of data during runtime without the performance penalty of reflection.

5.2 Overview

There are three possible ways of emitting code at runtime:

- Reflection.Emit namespace with the `AssemblyBuilder` and `TypeBuilder` classes allows to directly emit MSIL (Microsoft Intermediate Language) code
- Syste.CodeDom namespace allows to build a code tree
- System.Linq.Expressions allows to dynamically compile expressions at runtime to speed them up.

In this paper, System.CodeDom will not be described, only `Reflection.Emit` and `System.Linq.Expressions`.

5.3 Reflection.Emit

Reflection.Emit is the low-level approch to generating code. The programmer is required to define instruction, that should be executed, directly using IL code. IL code is the **platform independent** code that every .Net application emits when it is being built. The IL code is then - at runtime - converted to the actual x86/x64 (or some other) code that runs on the specific platform.

This process happens for every single method that has not yet been executed. The JIT (Just in Time) Compiler generates the platform specific code and caches it for further use.

By directly writing IL code, the programmer is able to generate code segments that are not possible in regular C# syntax.

However the programmer is **not** forced to write every single IL code statement by himself, but rather has a set of .Net classes that enable him to efficiently write IL code.

This is demonstrated in the next sample ('4-ILEmit'). The sample includes wrapper classes that make it easier to access the actual API's. In the `Program.cs` file, a dynamic assembly will be defined using the `Builder` helper class. Furthermore the newly generated assembly will create a type dynamically and will define properties and methods on the given type.

Since the actual process is rather complicated (and would create 200+ lines of code easily), I introduced the Generic classes in the `ILEmit` library which wrap the logic into little packages.

These types wrap the actual calls and make it a lot easier to create a new dll. The actual process includes:

- Creating an `AssemblyName` instance
- Creating an `AssemblyBuilder` using the `AppDomain.CurrentDomain.DefineDynamicModule` method
- Defining a module in the `AssemblyBuilder` (in .Net every dll has one module - using the `AssemblyBuilder` class, a programmer could potentially introduce multiple modules into a single assembly, however this feature is never used in .Net)
- (Optional) Defining the entry point of the module (ultimately making the assembly being created **executable**)
- Generating multiple types in the assembly using the `AssemblyBuilder.DefineType` instance method
- For each type:
  - Creating properties, fields and methods using the corresponding `TypeBuilder.DefineProperty`), `TypeBuilder.DefineField` or `TypeBuilder.DefineMethod` instance methods
  - Creating the type using the instance method `TypeBuilder.CreateType`.

    Note that calling this method **closes** the dynamic type creation for the specific type, meaning once you call `CreateType` no more properties/fields/-methods can be created/edited on the specific type.

    In order to finish the generation of the assembly, you need to call `CreateType` on **each** newly generated type, meaning that you need to keep a reference to each `TypeBuilder class` up until no more properties/methods/fields need to be introduced into the type
- Finally use the `AssemblyBuilder.Save` instance method to save the newly generated assembly.

The assembly is then saved to the disk and can be used by any other .Net program (i.e. there is no difference between a manually generated assembly and an assembly created by Reflection.Emit).

With `Reflection.Emit` it is also possible to generate a dll on the fly, to keep it directly in the RAM and directly execute it from there (useful for long running application that need to do reflection intensive work only once).

The '4-ILEmit' sample will create a dynamic type inside a dynamic assembly and write it to disk.

The dynamic type can be inspected with any Decompiler such as .Net Spy, Just Decompile, dotPeek, ILSpy, etc.

It has exactly one method (AddValues) that takes two integer arguments and returns their sum.

At the end the functionality will be tested by dynamically loading the assembly and executing the code with two random values and displaying the result, combining everything (reflection, attributes and code generation) in one package.

5.3.1 IL code

Writing IL code is very complicated and error prone. The provided sample was very simple as it only added two arguments. A single operation in C# (add two values) required 4 IL codes (load first argument, load second argument, add and finally return value).

In this work I will not explain the `OpCodes` in detail, they are - for the most part - very similar to the actual assembler opcodes. IL and assembler are similar enough to understand what the single statements do.

Furthermore the easiest way to generate these IL statements is to create the high level code, compile it and take a closer look at the resulting code using the ÏL Disassembler" that comes with Visual Studio.

`Reflection.Emit` is obviously the most demanding way of generating code on the fly. The next section will provide simpler code generation means.

5.4 System.Linq.Expressions

System.Linq.Expressions provide a more fluent way of generating code. There are two methods that can be used:

- Writting an expression in code and compiling it to a static method (compiling expressions is limited to being compiled to static methods, no type definition is possible)
- Defining an expression with fluent syntax

First we take a look at compiling existing expressions:

```
Expression<Func<int, bool>> oddNumbers = i
     => i % 2 == 1;
Func<int, bool> result = oddNumbers.
    Compile();
```

Listing 15: Expressions

In the first line, a Functor was wrapped into an `Expression` object. As a programmer, one can now access the expression as a tree of (sub)expressions. The next line compiles the expression. This will create a functor at runtime, that evaluates whether a number is odd or not.

The next listing decomposes an existing expression like i => i > 5.

```
var param = (ParameterExpression)exprTree.
    Parameters[0];
var operation = (BinaryExpression)exprTree
    .Body;
var left =(ParameterExpression)operation.
    Left;
var right =(ConstantExpression)operation.
    Right;

Console.WriteLine("Decomposed␣expression:␣
    {0}␣=>␣{1}␣{2}␣{3}", param.Name, left.
    Name, operation.NodeType, right.Value)
    ;
```

Listing 16: Decomposed Expression

As can be seen in the listing above, which is a sample from [5], it is possible to access all elements of the expression during runtime.

The reverse is also possible: An expression can be built at runtime dynamically.

This is possible via the static methods on the `Expression` class and the fact, that the expression tree API was moved into the DLR with C# 4.0.

On the MSDN Blog, [9], Alexandra Rusina shows how a function that determines the factorial of a given number can be written with expressions. While the code using expression is harder to read than the original function, she also shows the corresponding IL code that is a lot harder to read.

Expressions therefore present an easier way of creating dynamic code than `Reflection.Emit`.

As stated previously, Expressions are only helpful when single methods are required, as **expressions** cannot represent **types**. Expressions are useful, when dealing with queries that are expensive to build but occur often. The queries can be compiled to methods using expression trees to speed up their execution time.

6. SUMMARY

In this paper, we have learned about the basic principles of Refletion, the metadate concept, a way to extend the metadata (via attributes).

We have seen use cases where Reflection is necessary and helps to write productive code. Furthermore we have taken a look at different code generation techniques that allow dynamic code generation at runtime to speed up code execution. All in all, Reflection is a very powerful toolset that can help any developer write more efficient and shorter code.

7. OUTLOOK

This paper has discussed reflection in .Net and the upsides and downsides where illustrated. As a prospect, the last sample '5-INotifyPropertyChangedSample' illustrates the use of reflection in the case of the `INotifyPropertyChanged` interface that is a core feature of Databinding. Databinding itself will not be explained in this paper.

The sample shows four different implementations of `INotifyPropertyChanged`

- The first implementation is the `FirstProperty` on the `SampleViewModel`. It implements the basic `INotifyPropertyChanged` pattern. Since the event takes a string, this faces potential refactoring issues (property gets renamed, but refactoring doesn't detect that the string must also be changed). This can lead to inconsistencies (e.g. value gets updated but UI does not reflect the new value because NotifyChanged was not called for the correct property name).

- The second implementation is the `SecondProperty` which takes an expression as an argument. As we have seen before, expressions are direct representations of source code. Thus it is possible to extract the name of the property that was passed into the function. This is now safe for refactoring, however the call to reflection slows the process down.

- The third implementation is the `ThirdProperty` which uses a C# 5 feature: CallerMemberName. The `CallerMemberNameAttribute` is a compile-time directive that replaces the property it is attached to with the string name of the callee. In our case the callee is the property, so the property name is inserted. In order to use it, the property must be applied to a string parameter and the parameter must be optional. It can return callee names for properties, methods and events. For Constructors/Deconstructors it returns a fixed string (such as ".ctor"for any constructor).

- The fourth implementation (`FourthProperty`) uses a C# 6 feature: nameof. Much like CallerMemberName it is a comnpile-time directive and inserts the string at runtime. Its advantage is that it goes well beyond what CallerMemberName can offer: nameof can be used anywhere and can use any property, method and parametername in its scope. This means a property such as `TotalCartSum` can also update another value `TotalTax` when it changes by using `nameof(TotalTax)` as shown below.

```
public int TotalCartValue
{
    get => _totalCartValue;
```

```
        set
        {
            if (_totalCartValue == value)
                return;
            _totalCartValue = value;
            NotifyPropertyChange(nameof(
                TotalCartValue));
            // not possible with
                CallerMemberAttribute, it
                would just insert "
                TotalCartValue" twice
            NotifyPropertyChange(nameof(
                TotalTax));
        }
    }

    public int TotalTax => ComputeTaxFrom(
        TotalCartValue);
```

Listing 17: nameof

- The final implementation is a simple auto. property inside `PostSharpViewModel` that by itself doesn't implement `INotifyPropertyChanged`. The attribute `NotifyPropertyChangedAttribute` that is applied to the `PostSharpViewModel` class is a PostSharp attribute that does two things:
  - First of all, it indicates that each class that has the `NotifyPropertyChangedAttribute` applied to itself has to implement the `INotifyPropertyChanged` interface. If it isn't already implemented PostSharp will insert the implementation automatically at compile time
  - Secondly each property in this class should call the `PropertyChanged` event, when its value is being set (more specifically: when its value changes).

  If you take a look at the code, you will see that nowhere is the `NotifyPropertyChangedAttribute` class used anywhere except that it is applied to the `PostSharpViewModel`. The way this attribute works is by being **compiled** into the dll. PostSharp will run after the normal build is finished, and will inject its own code and modify existing code. In this sample, PostSharp will introduce the `INotifyPropertyChanged` interface to the class (if it doesn't already derive from it) and will rewrite each property to invoke the OnPropertyChanged method.

  That way the `INotifyPropertyChanged` implementation is very simple to do for the programmer and doesn't have the runtime impacts reflection would have, since the code is **compiled**.

  Further implementation would allow for a `DoNotNotifyAttribute` that can be applied to specific properties which should not be modifed or allow the `NotifyPropertyChangedAttribute` to be applied to individual properties instead of the class.

APPENDIX

A. RESX FORMAT

Each `.resx` file stores a set of keys and **one** translation. Each translation must be in a seperate `.resx` file which has to be placed side by side with the original one and must be named with the pattern <ResxFileName>.<CountryCode>.resx. If a programmer would want to provide German, french and English translations (with english being the default language), he would create:

- Translations.resx
- Translations.de.resx
- Translations.fr.resx

files with identical keys and each file would contain the corresponding translation. The runtime will then (via the `ResourceManager`) detect the current language of the system and will fall back to the most appropriate one.

It is even possible to specify specific translations for countries i.e. en-GB for Great Britain or en-US for the United States.

B. REFERENCES

[1] Piyush S Bhatnagar. Dynamic code generation using codedom. http://www.codeproject.com/Articles/18676/Dynamic-Code-Generation-using-CodeDOM, 19.11.2012.

[2] Piyush S Bhatnagar. Dynamic types via reflection.emit. http://www.codeproject.com/Articles/18677/Dynamic-Assemblies-using-Reflection-Emit-Part-II-o, 19.11.2012.

[3] Scott Hanselman. C# 4.0 and dynamic. http://www.hanselman.com/blog/C4AndTheDynamicKeywordW 20.11.2012.

[4] MSDN Library. Anonymous types. http://msdn.microsoft.com/en-us/library/bb397696.aspx, 21.11.2012.

[5] MSDN Library. Expression trees. http://msdn.microsoft.com/en-us/library/bb397951.aspx, 21.11.2012.

[6] MSDN Library. Using reflection.emit. http://msdn.microsoft.com/en-us/library/3y322t50.aspx, 19.11.2012.

[7] Mike Snell & Lars Powers. Reflection in depth. http://www.code-magazine.com/article.aspx?quickid=0301051, 19.11.2012.

[8] Jeffrey Richter. *CLR via C#*. Microsoft Press, 2010.

[9] Alexandra Rusina. Generating dynamic methods with expression trees. http://blogs.msdn.com/b/csharpfaq/archive/2009/09/14/genera dynamic-methods-with-expression-trees-in-visual-studio-2010.aspx, 21.11.2012.

[10] Abhishek Sur. Dynamic types via reflection.emit. http://www.codeproject.com/Articles/121568/Dynamic-Type-Using-Reflection-Emit, 19.11.2012.

[11] Abhishek Sur. Reflection.emit in depth. http://www.abhisheksur.com/2010/10/dlr-using-reflectionemit-in-depth-part.html, 19.11.2012.