Being Meta

Jevgenij Nekrasov
[Meta]

• From *wikipedia*

  • **Meta** ("after", or "beyond") is a prefix to indicate a concept which is an abstraction from another concept, used to complete or add to the latter
• Yet another language to learn syndrome
  • Learn if there are no hidden ideological contradictions
  • Mostly vertically
• Protect you “living” area
IT Technologies box

Hello, I am .NET developer
• Seeing the opportunities for simplicity in the system we develop

• Archived by our “favorite” language features
  • Encapsulation, abstraction, inheritance and polymorphism
SOLID box

"I want interface segregation, where client-specific interfaces are much better than one general-purpose interface."

The I in SOLID
[Meta] programming

• Computer program that writes new computer program
  • Not a compiler
• Computer program that manipulates other programs at runtime
• Writing programs at runtime and manipulating programs at runtime aren’t mutually exclusive concepts
  • In real metaprogramming world we’re doing both
We trust Infoof

```csharp
var randomType = Type.GetType("System.Random");
var nextMethod = randomType.GetMethod("Next", Type.EmptyTypes);
var random = nextMethod.Invoke(
    Activator.CreateInstance(randomType), null);

var result = from prop in @this.GetType().GetProperties(
    BindingFlags.Instance | BindingFlags.Public)
    where prop.CanRead
    select string.Format("{0}: {1}", prop.Name, prop.GetValue(@this, null));
```

- Ability to read contents of a program (metadata) and execute its code
  - Meta object protocol
- Create extensible apps and manipulating code members at runtime
  - Eric Lippert - principal developer on the C# compiler team
Impracticality

- "Friday" syndrome when you start using reflection
- No possibility to define new members
- Performance concerns
  - In general, execution time is always slower using reflection
- Britteness

```csharp
private static void TimeDirectCall()
{
    var stopwatch = Stopwatch.StartNew();
    for (var x = 0; x < 500000; x++)
    {
        var random = new Random().Next();
    }
    stopwatch.Stop();
    Console.WriteLine(stopwatch.Elapsed.ToString());
}

private static void TimeReflectionCall()
{
    var stopwatch = Stopwatch.StartNew();
    for (var x = 0; x < 500000; x++)
    {
        var randomType = Type.GetType("System.Random");
        var nextMethod = randomType.GetMethod("Next", Type.EmptyTypes);
        var random = nextMethod.Invoke((Activator.CreateInstance(randomType), null));
    }
    stopwatch.Stop();
    Console.WriteLine(stopwatch.Elapsed.ToString());
}
```
Override .ToString() for debugging purposes for each entity

- Performance issue
  - Reflection doesn’t like loneliness
  - Use caching strategies per T-type
- Small portion of reflection is good, but not everywhere
- The key form of metaprogramming in .NET
Function that returns larger of two values

```
public static int Max(int left, int right)
{
    return left < right ? right : left;
}
```

```
public class Info : IComparable
{
    public int CompareTo(object obj)...
}
```

```
public static T Max<T>(T left, T right)
{
    return left < right ? right : left;
}
```

```
public static T Max<T>(T left, T right)
where T : IComparable<T>, IComparable
{
    return (left.CompareTo(right) < 0)
        ? right : left;
}
```

```
let max left right = if left < right then right else left;
max 1.2;
mmax 3.0 4.0;
mmax "hello" "world";
'a -> 'a -> 'a
```
public static T Max<T>(T left, T right)
{
    if (left is IComparable<T>)
    {
        return ((left as IComparable<T>).CompareTo(right) < 0)
                   ? right : left;
    }

    if (left is IComparable)
    {
        return ((left as IComparable).CompareTo(right) < 0)
                   ? right : left;
    }

    throw new ApplicationException(string.Format(  
            "Type \{0\} must implement one of IComparable or IComparable\{\{0\}\} interface",
            typeof(T).Name));
}
• Do we need the `max` function to be completely `dynamic` at run-time?
  • The data types we’re working with are well-known at compile time

• What if there is a tool to capture the comparison strategy for all data types?
  • Welcome to T4 world (`*.tt`)
Text Template Transformation Toolkit (T4)

- It is **TEXT** transformation toolkit
  - No dependency on any programming language
    - But only C# and VB.NET supported as ctrl languages
- Created by Gareth Jones (from **DSL** team)
  - Based on ASP.NET page-processing engine
- Has clear separation between boilerplate text and ctrl code
- Supported by MS
using System;

public static class TemplatedMathConverter
{
    #pragma warning disable 1998
    public static #= type.Name # Max(#= type.Name # left, #= type.Name # right)
    {
        Type[] typesToGenerate = new[]
        {
            typeof(int),
            typeof(double),
            typeof(object),
            typeof(uint),
            typeof(float)
        };
        var comparable =
            (from intf in type.GetInterfaces()
             where typeof(IComparable<>).MakeGenericType(type).IsAssignableFrom(intf)
             select intf).FirstOrDefault();
        if (comparable != null)
        {
            return left.CompareTo(right) < 0 ? right : left;
        }
    }
    #pragma warning restore 1998
}
public static class TemplatedMathConverter
{
    public static Int32 Max(Int32 left, Int32 right)
    {
        return left.CompareTo(right) < 0 ? right : left;
    }
    public static Double Max(Double left, Double right)
    {
        return left.CompareTo(right) < 0 ? right : left;
    }
    public static Object Max(Object left, Object right)
    {
        throw new ApplicationException(
            "Type Object must implement one of IComparable or IComparable<Object> interfaces");
    }
    public static UInt32 Max(UInt32 left, UInt32 right)
    {
        return left.CompareTo(right) < 0 ? right : left;
    }
    public static Single Max(Single left, Single right)
    {
        return left.CompareTo(right) < 0 ? right : left;
    }
}

public static int Max(int left, int right)
{
    return left < right ? right : left;
}

public static string ToStringReflection<T>(this T @this)
{
    return string.Join(
        Constants.Separator,
        (new List<string>())
            .From(prop in @this.GetProperties(BindingFlags.Instance | BindingFlags.Public)
            .GetProperties(BindingFlags.Instance | BindingFlags.Public)
            .Where(prop.CanRead)
            .Select(string.Format("{0}: {1}",
                prop.Name, prop.GetValue(@this, null))).ToArray()));
}
Try real world T4 [Meta]

- Generate Knockout view-models from data-models, keep them in sync, mark properties you want to include with special attribute []
- Create `Comparer` class for all data-model in your DataModel assembly
- Oleg Sych blog
CodeDOM

• System.CodeDom
  • First level namespace, in .NET framework since the beginning (1.0)

• Uses code graph – hierarchical data structure
  • Can be parsed from string or created step by step (~alia expression trees)
  • Describe code in mostly language-independent data structure

• Generates code in variety of languages!
• Compile code in run-time into .NET assemblies
• Force you to think like a programming language
static CodeNamespace BuildProgram()
{
    var ns = new CodeNamespace("BeingMeta.PlayGround");
    var systemImport = new CodeNamespaceImport("System");
    ns.Imports.Add(systemImport);
    var programClass = new CodeTypeDeclaration("Program");
    ns.Types.Add(programClass);
    var methodMain = new CodeMemberMethod
    {
        Attributes = MemberAttributes.Static,
        Name = "Main"
    };
    methodMain.Statements.Add(
        new CodeMethodInvokeExpression(
            new CodeSnippetExpression("Console"),
            "WriteLine",
            new CodePrimitiveExpression("Hello, world!")
        ));
    programClass.Members.Add(methodMain);
    return ns;
}
```c#
namespace BeingMeta.PlayGround
{
    using System;
    public class Program
    {
        static void Main()
        {
            Console.WriteLine("Hello, world!");
        }
    }
}
```

```cpp
namespace BeingMeta
{
    namespace PlayGround
    {
        using namespace System;
        using namespace System;
        ref class Program
        {
            static System::Void Main();
        }
    }
}
```

```vb
Imports System
Namespace BeingMeta.PlayGround
    Public Class Program
        Shared Sub Main()
            Console.WriteLine("Hello, world!")
        End Sub
    End Class
End Namespace
```
Generating code with Reflection.Emit

• Provides substantial performance boost by emitting IL directly
• Support for DSLs
  • Regex(“\d{3}-(\d{3})?”) and etc
• Ability to persist dynamic logic for future use
  • XmlSerializer
• Using .NET functionality not supported in your language
  • Like try-catch-fault/filter (not just matching on type)-finally blocks
  • Throw exceptions that don’t inherit from Exception
  • Create methods calls known as “tail calls” (eliminate stack before calling in recursive call)
Microsoft **Intermediate** Language

- Some good tools
  - ILDasm/ILSpy, pverify.exe
- OPCodes understanding
  - ld, st, ldc, call and etc.
- Evaluation stack understanding
- A lot of patience
Back to `.ToString()` implementation
private AssemblyBuilder Assembly { get; set; }
private ModuleBuilder Module { get; set; }
private AssemblyName Name { get; set; }

public Reflection.Emit.MethodGenerator()
{
    this.Name = new AssemblyName() { Name = Guid.NewGuid().ToString("N") };  
    this.Assembly = AppDomain.CurrentDomain.DefineDynamicAssembly(
        this.Name, AssemblyBuilderAccess.Run);
    this.Module = this.Assembly.DefineDynamicModule(this.Name.Name);
}

public Func<T, string> Generate<T>()
{
    var target = typeof(T);
    var type = this.Module.DefineType(target.Namespace + "." + target.Name);
    var methodName = "ToString" + target.GetHashCode().ToString();  
    var method = type.DefineMethod(methodName,
        MethodAttributes.Static | MethodAttributes.Public,
        typeof(string), new Type[]{typeof(T)});

    method.GetILGenerator().Generate(target);
    var createdType = type.CreateType();

    var createdMethod = createdType.GetMethod(methodName);
    return (Func<T, string>)Delegate.CreateDelegate(
        typeof(Func<T, string>), createdMethod);
}
internal static void Generate(this ILGenerator @this, Type target)
{
    var properties = target.GetProperties(BindingFlags.Public | BindingFlags.Instance);

    if (properties.Length > 0)
    {
        var stringBuilderType = typeof(StringBuilder);

        var toStringLocal = @this.DeclareLocal(typeof(StringBuilder));

        @this.Emit(OpCodes.Newobj, stringBuilderType.GetConstructor(Type.EmptyTypes));
        @this.Emit(OpCodes.Stloc_0);
        @this.Emit(OpCodes.Ldloc_0);

        var appendMethod = stringBuilderType.GetMethod("Append", new Type[] { typeof(string) });
        var toStringMethod = typeof(StringBuilder).GetMethod("ToString", Type.EmptyTypes);

        for (var i = 0; i < properties.Length; i++)
        {
            CreatePropertyForToString(
                @this.properties[i], appendMethod, 
                i < properties.Length - 1);
        }

        @this.Emit(OpCodes.Pop);
        @this.Emit(OpCodes.Ldloc_0);
        @this.Emit(OpCodes.Callvirt, toStringMethod);
    }
}
```csharp
private static void CreatePropertyForToString(ILGenerator generator, PropertyInfo property,
   MethodInfo appendStringMethod, bool needsSeparator)
{
    if (property.CanRead)
    {
        generator.Emit(OpCodes.Ldstr, property.Name + " : ");
        generator.Emit(OpCodes.Callvirt, appendStringMethod);
        generator.Emit(OpCodes.Ldarg_0);

        var propertyGet = property.GetGetMethod();

                       propertyGet);

        var appendStringTyped = typeof(StringBuilder).GetMethod("Append",
                              new Type[] { propertyGet.ReturnType });

        if (appendStringTyped.GetParameters()[0].ParameterType != propertyGet.ReturnType)
        {
            if (propertyGet.ReturnType.IsValueType)
            {
                generator.Emit(OpCodes.Box, propertyGet.ReturnType);
            }
        }

        generator.Emit(OpCodes.Callvirt, appendStringTyped);

        if (needsSeparator)
        {
            generator.Emit(OpCodes.Ldstr, " || ");
            generator.Emit(OpCodes.Callvirt, appendStringMethod);
        }
    }
}
```
Using ILDasm/ILSpy to cheat

• Compile code that you can write
• Use tools to see generated IL
When creating assembly is too much

• Emit small amount of code, why do we need dynamic assembly at all?
• Not garbage collectable :/ (only in 4.0)
• DynamicMethod() can help!
private static Func<T, string> CreateToStringViaDynamicMethod<T>()
{
    var target = typeof(T);

    var toString = new DynamicMethod("ToString" + target.GetHashCode().ToString(),
        typeof(string), new Type[] { target });

    toString.GetILGenerator().Generate(target);
    return (Func<T, string>)toString.CreateDelegate(typeof(Func<T, string>));
}
Performs better? But why?

```csharp
.try {
    IL_0010: br.s IL_001b
    // loop start (head: IL_001b)
    IL_0012: ldloc.1
    IL_0013: callvirt instance !0 class [mscorlib]System.Collections.Generic.IEnumerator`1<int32>::get_Current()
    IL_0018: stloc.0
    IL_0019: nop
    IL_001a: nop

    IL_001b: ldloc.1
    IL_001c: callvirt instance bool [mscorlib]System.Collections.IEnumerator::MoveNext()
    IL_0021: stloc.2
    IL_0022: ldloc.2
    IL_0023: brtrue.s IL_0012
    // end loop

    IL_0025: leave.s IL_0037
} // end .try
```
Main pitfalls

• Hard to learn
  • Learning IL isn’t a skill set that most .net developers have
• Hard to debug
Generating code with expressions

- Represents code as data
  - Expressions are data structures
  - Eventually turns into something that the runtime can execute – namely, IL
  - Supports exception handling, flow control and etc.
- No IL knowledge! No unintended results!
  - Still expressions are slower

```csharp
Expression<Func<int, int, int>> add = (x, y) => x + y;
var result = add.Compile()(2, 3);
```
// f(x) = ((3 * x) / 2) + 4

```csharp
var parameter = Expression.Parameter(typeof(double));
var method = Expression.Lambda(
    Expression.Add(
        Expression.Divide(
            Expression.Multiply(
                Expression.Constant(3d), parameter),
            Expression.Constant(2d)),
        Expression.Constant(4d)),
    parameter).Compile();
```

```csharp
var method = new DynamicMethod("m",
    typeof(double), new Type[] { typeof(double) });
var parameter = method.DefineParameter(1, ParameterAttributes.In, "x");
var generator = method.GetILGenerator();
generator.Emit(OpCodes.Ldc_R8, 3d);
generator.Emit(OpCodes.Ldarg_0);
generator.Emit(OpCodes.Mul);
generator.Emit(OpCodes.Ldc_R8, 2d);
generator.Emit(OpCodes.Div);
generator.Emit(OpCodes.Ldc_R8, 4d);
generator.Emit(OpCodes.Add);
generator.Emit(OpCodes.Ret);
var compiledMethod = method.CreateDelegate(
    typeof(Func<double, double>)) as Func<double, double>;
```
Something interesting

• Expressions are immutable objects
  • In .NET 4 we have ExpressionVisitor (before that it was marked as internal)
• Using DynamicQueryable (System.Linq.Dynamic)
  • Now it is in nuget

```csharp
bricks.Select(p => p.Year).Distinct().OrderBy(i => i);

bricks.Select("Year").Distinct().OrderBy("it");
brick.Where("Value.Contains("a")");
```
Generating code with IL rewriting (IL weaving)

- Used in AOP (PostSharp)
- After the compilation the assembly is not frozen – it can be changed
- Allows you to run business logic and not boiler plated code, like logging, .ToString(), null checks and etc
- Unfortunately .NET Framework doesn’t support IL rewriting natively
Cecil

- Weaving code

1. Load assembly with Cecil
2. Execute all discovered injectors
3. Save changes to the assembly

- Implement your attribute (injector) with custom logic
- Create MSBuild task to bind and execute injectors
```csharp
public void Foo(string value)
{
    if (value == null)
    {
        throw new ArgumentNullException("value");
    }
}

public sealed class NotNullAttribute : InjectorAttribute<ParameterDefinition>
{
    protected override void OnInject(ParameterDefinition target)
    {
        if (!target.ParameterType.IsValueType)
        {
            var method = (target.Method as MethodDefinition);
            // .. more code
            var processor = method.Body.GetILProcessor();
            var first = processor.Body.Instructions[0];
            processor.InsertBefore(first, processor.Create(OpCodes.Ldarg, target));
            processor.InsertBefore(first, processor.Create(OpCodes.Brtrue_S, first));
            // and so on
        }
    }
}
```
Good metaprogramming sub-system

- Represent code as data
  - You can manipulate it easily
- Caching strategy per type or per assembly
DLR or DSLDD?

• DSL?
  • Common vocabulary for problem domain and solution domain
DSLDD

- Why?
  - Readability – main criteria!
  - One+ more person in your team!
  - Create flexible system from scratch, customizable with DSL scripts
    - Expressive, higher level of abstraction, scalable API for you solution domain
- Pitfalls
  - Language design is hard, performance concerns, yet another language to learn syndrome, DSL language mess
Internal DSL

• All about mixing language paradigms
  • New skills, new knowledge, new possibilities
• Based on existing programming language (host language)
  • Builds domain specific semantics on top of it as a library
  • IronPython, IronRuby, Boo – decide, which language suits your needs or expertize
Internal **DSL** integration in .NET

- Specific script engines in DLR
  - Dino Viehland describes DLR as two layer subsystem
    - Inner layer contains .NET class like DynamicObject and etc – everyone, who installed .NET framework has them
    - Outer layer is specific hosting API, for example IronPython

```csharp
var scriptEngine = IronRuby.Ruby.CreateEngine();
var scriptScope = scriptEngine.ExecuteFile("Orders.dsl");
```
Internal DSL implementation patterns

```ruby
Order.create do
  store "Maxima"
  item "Milk"
  delivery_address "Kanarskio g."
  price "10"
end.save
end

Register.register(a)
Mailer.new
  .to(a.email_address)
  .subject("New order is accepted")
  .body("We accepted your order. It will be delivered tomorrow at 16")
  .send
end
```
Example of implicit context usage in Ruby with block to support closures (reflective metaprogramming)

Example of fluent API to improve readability, methods chaining

- Improve readability with fluent API and method chaining
- Implicit context makes DSLs less verbose and increase expressivity
Dynamic decorators

```csharp
public class Order
{
    public virtual decimal Price { get; }
}

public class OrderTaxFreeDecorator : Order
{
    private readonly Order _order;

    public OrderTaxFreeDecorator(Order order)
    {
        _order = order;
    }

    public override decimal Price
    {
        get
        {
            return _order.Price - (_order.Price * 0.12m);
        }
    }
}
```

```csharp
var price = new OrderTaxFreeDecorator(new Order());
price.Price;
```

- Hardwired relationship between Order and the decorator
- Expressivity and domain friendliness
- Put Order first and then decorators
- No runtime extensions

```csharp
o = Order.new('Senukai', 'Tv-set', '10', 'Kanarskio g').with TaxFree
o.price
```
• With() method dynamically extends Order with decorators
• In general decorators help you attach additional responsibilities to object ->
  • Make it dynamic improve the readability
Do we need dots? (.)

• Can we achieve something like this in internal DSL?

```ruby
new_order 'TV-set order' for client 'Jevgenij Nekrasov'
to buy 'Samsung MU756' at price = 1000
```

• Yes!
  • `method_missing` and `const_missing` in a rescue

```ruby
def parse(dsl_string)
  dsl = dsl_string.clone
  dsl.sub!(/at /, '')
  dsl.sub!(/for client /, ',')
  dsl.sub!(/for client /, ',')
  dsl.sub!(/to buy /, ':buy,')
end
```
External DSL

- Parser
  - Lexical analysis and parsing
- Semantic model
  - Independent structure of the DSL syntax, decoupled from the input, aligned and enhanced with domain knowledge
Designing external DSL with ANother Tool for Language Recognition (ANTLR)

```plaintext
please buy meat in maxima at limitprice = 4
please buy chesse in iki at limitprice = 3
please buy milk in maxima at 1
please buy chocolate butter in rimi at 10
```

```plaintext
orders
    : order+ EOF;

order
    : line NEWLINE {Orders.Add($line.value);};

line returns [Order value]
    : PLEASE BUY what where price
    {
        $value = new Order($what.value, $where.value, $price.value);
    };

what returns [string value]
    : ITEM {$value = $ITEM.text;};

where returns [string value]
    : IN ITEM {$value = $ITEM.text;};

limitprice returns [int value]
    : LPRICE EQ INT {$value = int.Parse($INT.text);};

price returns [int value]
    : AT (INT {$value = int.Parse($INT.text);} | limitprice {$value = $limitprice.value;});
```

```plaintext
lexer grammar OrderLexer;

EQ      : '=';
PLEASE  : 'please';
BUY     : 'buy';
IN      : 'in';
AT      : 'at';
LPRICE  : 'limitprice';
ITEM    : ('a'..'z'|'A'..'Z');
INT     : '0'..'9';
NEWLINE: ''? '
';
WS      : [ \r\t\n]+ -> skip ;
```
• Use parser generators, because any change of grammar means that you will have to make significant change in the code to implement it
• Use EBNF, instead of regular BNF
• Use custom actions that you want to execute on recognition of grammar rules
Thank you!
Questions