Reflection

•••

Not Always Evil?

Bruce Dunwiddie

shriop@hotmail.com (Yes, I still use Hotmail. At least it’s not Yahoo.)

linkedin.com/in/bdunwiddie (sorry, no twitter, tumblr, instagram, pinterest, tinder, ashley madison, etc, etc, etc) #theycallmebruce #mynameisbruce #theystillcallmebruce

csvreader.com - DataStreams .Net ETL Framework

sqldatadictionary.com - SQL Data Dictionary

Chaotic Neutral

So I can do whatever the hell I want.

I didn't sleep well last night so I made my coffee this morning with Red Bull instead of water. I got half way to work before I realized I forgot my car.

Martial Arts

What I think I look like

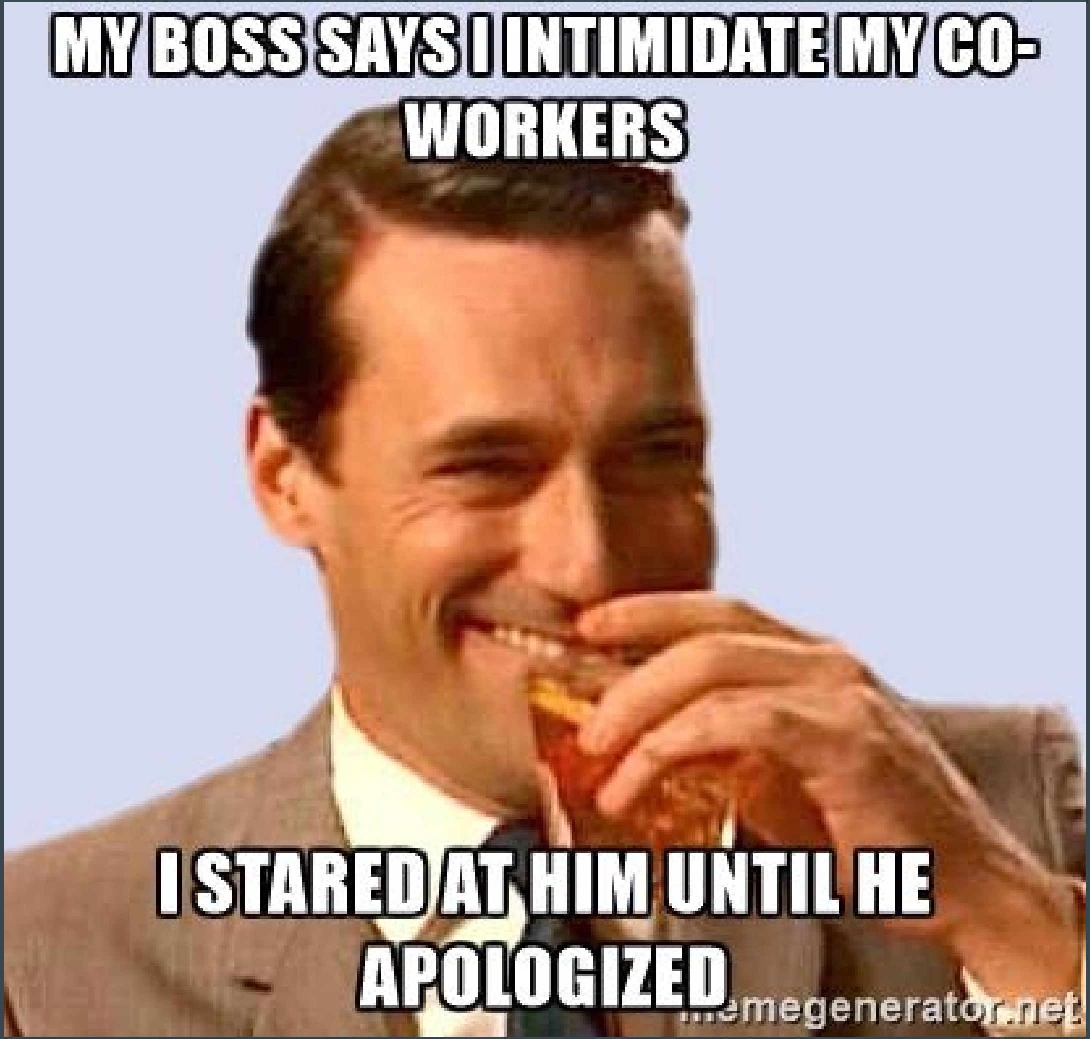
THAT DRUNKEN AWKWARD MOMENT...
WHEN YOUR PANTS RIP IN THE CROTCH WHILE DOING A HALF-NAKED FLYING HIGH KICK THROUGH YOUR FRIENDS LIVING ROOM.
drunkwithstyle.com

What I actually look like

Running

What I think I look like

What I actually look like


MY BOSS SAYS I INTIMIDATE MY CO-WORKERS
I STARED AT HIM UNTIL HE APOLOGIZED
memegenerator.net

Reflection Categories

Introspection

Manipulation

Instantiation

Invocation

Type Loading

Type Introspection

Reflection allows you to examine a class's definition at runtime using properties available through a System.Type object.

Manipulation

Reflection allows you to get and set values dynamically on fields and properties of an object at runtime.

Instantiation

Reflection allows you to create an instance of a class at runtime.

Invocation

Reflection allows you to dynamically execute a method on an object at runtime.

Type Loading

Reflection allows for locating and loading new class definitions into the current application domain for use.

Example Reflection in Frameworks

NUnit

Introspection
Instantiation
Invocation
Type Loading

JSON.Net

Introspection
Manipulation
Instantiation

Ninject

Introspection
Manipulation
Instantiation
Invocation
Type Loading

Type

System.Type gives access to every part of a class’s definition at runtime, including fields, properties, methods, and constructors.

Getting a Type Instance

Object GetType()

Assembly GetTypes()

typeof(...)

Object GetType()

```
// get a type at runtime from an instance of an object
Type exampleClass = new TypesExample().GetType();
```

Assembly GetTypes()

```
// look at all types defined in an assembly and filter them
Type exampleClass =
    Assembly.GetExecutingAssembly()
        .GetTypes()
        .Single(t => t.Name == "TypesExample");
```

typeof()

```
// get a type definition at compile time
Type exampleClass = typeof(TypesExample);
```

Useful Type Properties

Name
Namespace
FullName
BaseType

Attributes
Implements

IsClass
IsInterface
IsEnum
IsPrimitive
IsArray
IsPublic
IsAbstract
IsGenericType

Useful Type Methods

GetFields(...)

GetProperties(...)

GetMethods(...)

GetConstructors(...)

Important BindingFlags Filters

Public

NonPublic

Static

Instance

DeclaredOnly

Methods for Object Manipulation

FieldInfo GetValue(…)

FieldInfo SetValue(…)

PropertyInfo GetValue(…)

PropertyInfo SetValue(…)

Using PropertyInfo

```
public class User
{
      public User()
      {
            ID = Guid.NewGuid();
      }

      public Guid ID
      {
            get;

            private set;
      }

      public string Name { get; set; }
}
```

PropertyInfo GetValue(...)

```
Type userType = typeof(User);

PropertyInfo nameProperty = userType.GetProperty("Name");

string name = (string)nameProperty.GetValue(
    user);
```

PropertyInfo SetValue(...)

```
// we can't set the ID because of the private access modifier

// this line won't compile
// user.ID = Guid.NewGuid();

// or can we?
Type userType = typeof(User);

PropertyInfo idSetter = userType.GetProperty("ID");

idSetter.SetValue(
     user,
     Guid.NewGuid());
```

Instantiation Techniques

ConstructorInfo Invoke(…)

Activator CreateInstance(…)

ConstructorInfo Invoke

```
public static object Create(Type typeToConstruct)
{
    // find the default parameterless constructor
    ConstructorInfo constructor = typeToConstruct
        .GetConstructors()
        .Single(c => c.GetParameters().Length == 0);

    return constructor.Invoke(null);
}
```

Activator CreateInstance

```
public static object Create(Type typeToConstruct)
{
    return Activator.CreateInstance(typeToConstruct);

    // there's also Activator.CreateInstance<T>()
}
```

Invocation Techniques

Use an Interface

MethodInfo Invoke

Define a Delegate

Create a Typed Func

Create an Expression Tree

Use a dynamic Type

Compile a Wrapper Class (Advanced)

Use an Interface

```
IValidator validator =
    (IValidator)ValidatorFactory.GetValidator();

bool isValid = validator.IsValid(password);
```

Interface Pros and Cons

Pros

Almost as fast as direct invocation.

Simple.

Not limited to a single object instance.

Cons

Signature must already be defined.

Object must implement interface.

MethodInfo Invoke

```
object validator = ValidatorFactory.GetValidator();

MethodInfo isValidMethod = validator
    .GetType()
    .GetMethod("IsValid");

bool isValid = (bool)isValidMethod
    .Invoke(validator, new object[] { password });
```

MethodInfo Invoke Pros and Cons

Pros	Cons
Signature not required ahead of time.	Slow.
Not limited to a single object instance.	
Code is only moderately complex.	

Define a Delegate

```
delegate bool IsValidDelegate(string input);

...

object validator = ValidatorFactory.GetValidator();

MethodInfo isValidMethod = validator
      .GetType()
      .GetMethod("IsValid");

IsValidDelegate callIsValid =
      (IsValidDelegate)isValidMethod.CreateDelegate(
            typeof(IsValidDelegate),
            validator);

bool isValid = callIsValid(password);
```

Delegate Pros and Cons

Pros

Fast.

Object does not need to implement interface.

Cons

Signature must already be defined.

Limited to a single object instance.

Must define a new delegate for each signature.

Create a Typed Func

```csharp
object validator = ValidatorFactory.GetValidator();

MethodInfo isValidMethod = validator
      .GetType()
      .GetMethod("IsValid");

Func<string, bool> callIsValid = (Func<string, bool>)Delegate.CreateDelegate(
      typeof(Func<string, bool>),
      validator,
      isValidMethod);

bool isValid = callIsValid(password);
```

Typed Func Pros and Cons

Pros

Fast.

Object does not need to implement interface.

Don't have to define a new delegate for each signature.

Cons

Signature must already be defined.

Limited to a single object instance.

Create an Expression Tree

```
object validator = ValidatorFactory.GetValidator();

MethodInfo isValidMethod = validator
     .GetType()
     .GetMethod("IsValid");

Func<object, object[], object> callIsValid = GetFuncFromExpression(
     validator.GetType(),
     "IsValid");

bool isValid = (bool)callIsValid(
     validator,
     new object[] { password });
```

GetFuncFromExpression

```
private static Func<object, object[], object> GetFuncFromExpression(
       Type instanceType,
       string methodName)
{
       MethodInfo method = instanceType.GetMethod(methodName);

       ParameterExpression instance = Expression.Parameter(typeof(object), "i");

       ParameterExpression allParameters = Expression.Parameter(typeof(object[]), "params");

       ParameterInfo[] methodParameters = method.GetParameters();

       List<Expression> parameters = new List<Expression>();

       ...
```

GetFuncFromExpression (Cont.)

```
...

for (int i = 0; i < methodParameters.Length; i++)
{
        ParameterInfo parameter = methodParameters[i];

        ConstantExpression indexExpr = Expression.Constant(i);

        BinaryExpression item = Expression.ArrayIndex(
                allParameters,
                indexExpr);

        UnaryExpression converted = Expression.Convert(
                item,
                parameter.ParameterType);

        parameters.Add(converted);
}

...
```

GetFuncFromExpression (Cont.)

```
        ...

        Expression methodExp = Expression.Call(
                Expression.Convert(instance, method.DeclaringType),
                        method,
                        parameters.ToArray());

        if (methodExp.Type.IsValueType)
        {
                methodExp = Expression.Convert(methodExp, typeof(object));
        }

        Expression<Func<object, object[], object>> methodCall =
                Expression.Lambda<Func<object, object[], object>>(
                        methodExp,
                        instance,
                        allParameters);

        Func<object, object[], object> func = methodCall.Compile();

        return func;
}
```

Expression Tree Pros and Cons

Pros

Can be 25x faster than MethodInfo Invoke.

Signature not required ahead of time.

Not limited to a single object instance.

Cons

Still 3-5x slower than direct method call.

Code is very complex.

Use a dynamic Type

```
dynamic validator = ValidatorFactory.GetValidator();

bool isValid = validator.IsValid(password);
```

dynamic Pros and Cons

Pros

20x faster than MethodInfo Invoke.

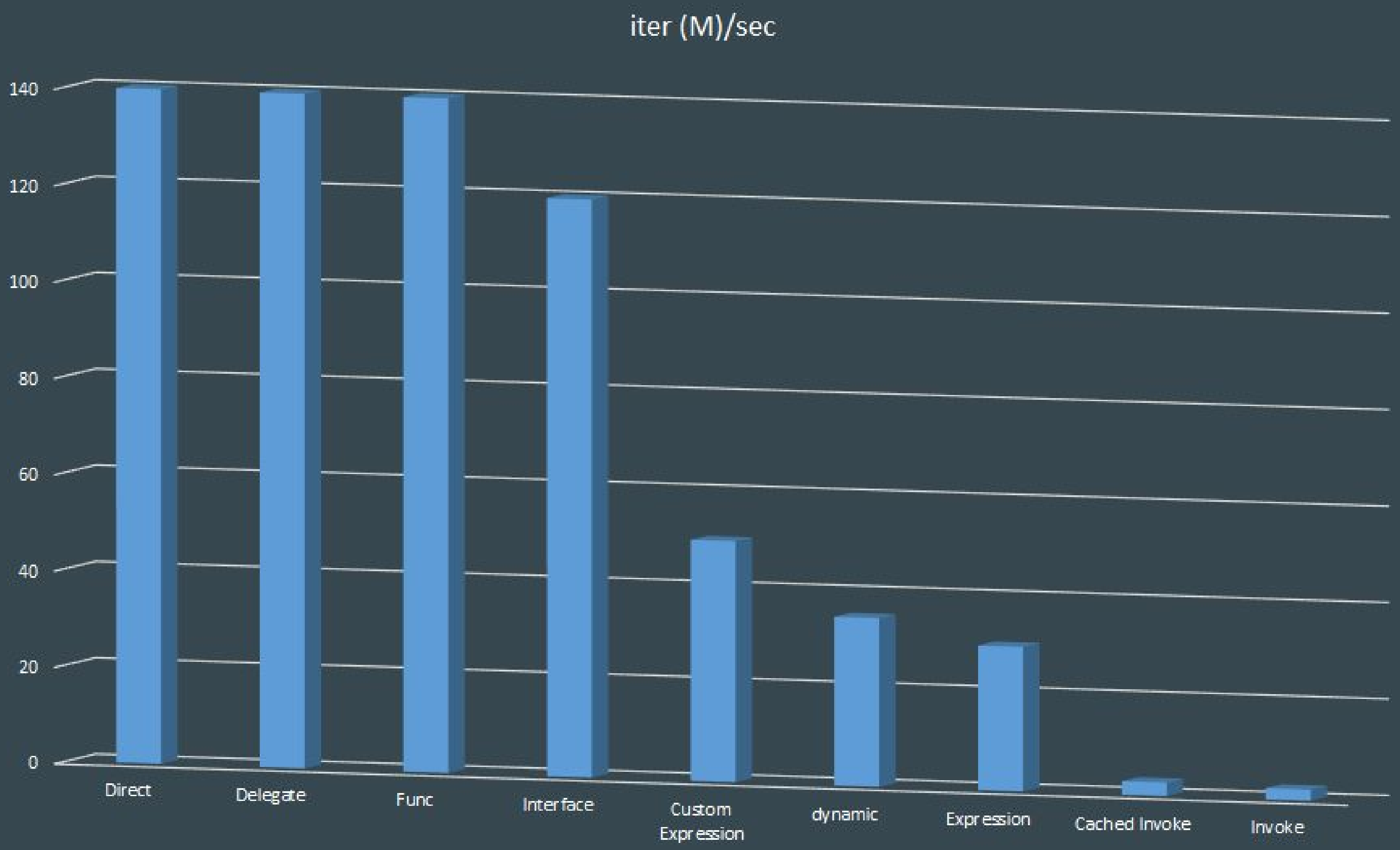
Simple.

Not limited to a single object instance.

Cons

Still 5x slower than direct method call.

Signature must already be defined.


iter (M)/sec
140
120
100
80
60
40
20
0
Direct
Delegate
Func
Interface
Custom Expression
dynamic
Expression
Cached Invoke
Invoke

Compile a Wrapper Class (Advanced)

1. Use type introspection to find method definition on class.
2. Write out a text file containing a new wrapper class that forwards method calls to introspected class instance through a previously compiled interface contract.
3. Use CodeDom or Roslyn to compile wrapper class.
4. Load wrapper class into current app domain.
5. Instantiate instance of wrapper class.
6. Call methods on wrapper class through interface to access introspected class instance.

Type Loading

```
Assembly assembly = Assembly.LoadFrom(
    pathToAssembly);

// since custom assembly is referencing our assembly for the IValidator definition
// we can check for equality on IValidator directly

// making the assumption that there is only a single implementation
// in the custom assembly

Type customValidatorType = assembly.GetTypes().Single(
    t => t.GetInterfaces().Where(
        i => i == typeof(IValidator)).Any());

IValidator validator =
     (IValidator)Activator.CreateInstance(customValidatorType);
```

Code Examples and Runner

github.com/bruce-dunwiddie/ReflectionPresentation

The GitHub solution has the runner application, a few more code examples, the PrettyName function for fixing the names of classes from introspection, both multi parameter and single parameter reusable expression tree generation functions, a cross app domain type loading example, along with programs simulating key logic from NUnit, JSON.Net, and Ninject.

Good or evil?

Thank You