Licenciatura em Engenharia Informática e de Computadores

Trabalho
Época Especial

Ambientes Virtuais de Execução
2018 / 2019 ver.

Autores:
André Martins - 44858

Índice

Introdução

Pretende-se implementar uma *framework* que oferece um método que retorna uma nova instância de uma classe cujas propriedades e métodos virtuais são acrescentados com funcionalidades adicionais.

As funcionalidades adicionais são descritas através de *custom attributes* anotados sobre as propriedades e os métodos virtuais.

Os *custom attributes* têm o comportamento que se pode observar na figura seguinte.

```
class Stock
{
    public Stock(string name, string index) { ... }

    [NonNull]
    public virtual string Market { get; set; } // set dará excepção para valores null
    [Min(73)]
    public virtual long Quote { get; set; } // set dará excepção para valores < 73
    [Min(0.325)]
    public virtual double Rate { get; set; } // set dará excepção para valores < 0,325
    [Accept("Jenny", "Lily", "Valery")]
    public virtual string Trader{get; set; } // set só aceita valores Jenny, Lily e Valery
    [Max(58)]
    public virtual int Price { get; set; } // set dará excepção para valores > 58

    // dará excepção se o estado de this ou algum dos parâmetros tiver sido alterado
    // pela execução do método anotado -- BuildInterest
    [NoEffects]
    public double BuildInterest(Portfolio port, Store st) { ... }
}
```

Enhancer

Enhancer Attributes

Todos os atributos usados derivam do atributo base, *EnhancerAttribute*, este que deriva da classe base de todos os atributos, *Attribute*. É uma classe abstrata e tem apenas um único método virtual, *Check*

```
namespace Enhancer
{
    public abstract class EnhancerAttribute : Attribute
    {
        public abstract void Check(object[] args);
    }
}
```

O método recebe no *array args* todos os argumentos do método anotado, mas também recebe no índice zero, a referência(*this*) da instância do objeto que chama o método.

Os *custom attributes* criados neste trabalho foram os seguintes:

NonNull

Este atributo lança exceção quando algum valor recebido é *null*. Para tal é percorrido os argumentos do método anotado e verificado se algum é *null*.

```
public class NonNull : EnhancerAttribute
{
    public override void Check(object[] args)
    {
        for (int i = 1; i < args.Length; i++)
        {
            if (args[i] == null)
            {
                throw new ArgumentException("Invalid argument");
            }
        }
    }
}
```

Min

O atributo *Min* lança exceção quando o valor passado nos argumentos é menor do que o valor indicado no atributo anotado.

```
public class Min : EnhancerAttribute
{
    private readonly double value;

    public Min(double value)
    {
        this.value = value;
    }

    public override void Check(object[] args)
    {
        IConvertible convert = args[1] as IConvertible;

        if (convert.ToDouble(null) < value)
            throw new ArgumentException("Minimum value is " + value);
    }
}
```

Max

Este atributo funciona da mesma maneira que o atributo *Min* mas para valores maiores em vez de menores. Sendo o código destes idêntico.

```
public class Max : EnhancerAttribute
{
    private readonly double value;

    public Max(double value)
    {
        this.value = value;
    }

    public override void Check(object[] args)
    {
        IConvertible convert = args[1] as IConvertible;

        if (convert.ToDouble(null) > value)
            throw new ArgumentException("Maximum value is " + value);
    }
}
```

Accept

O atributo *Accept* indica quais os valores que são aceites nos argumentos do método anotado. É percorrido todos os argumentos e comparado com os valores passados aquando da anotação do atributo. Caso algum argumento seja diferente dos indicados no atributo, é lançada exceção.

```
public class Accept : EnhancerAttribute
{
    private readonly object[] valid;

    public Accept(params object[] valid)
    {
        this.valid = valid;
    }

    public override void Check(object[] args)
    {
        bool equal = false;
        for (int i = 1; i < args.Length; i++)
        {
            foreach (string s in valid)
            {
                if (s.Equals(args[1]))
                {
                    equal = true;
                    break;
                }
            }
            if (!equal) throw new ArgumentException("Not accepted parameter");
            equal = false;
        }
    }
}
```

NoEffects

Por fim o atributo *NoEffects* verifica se existe alguma alteração no estado da instância corrente(*this*) feita pelo método anotado. Para verificar se existe alterações, é guardado todos os valores das propriedades e dos campos, e comparado com as propriedades e campos do estado após a execução do método anotado.

```
public class NoEffects : EnhancerAttribute
{
    Object[] state;

    public override void Check(object[] args)
    {
        if (state == null)//Before base method
        {
            PropertyInfo[] pi = args[0].GetType().GetProperties();
            FieldInfo[] fi = args[0].GetType().GetFields();
            state = new object[pi.Length + fi.Length];
            int i = 0;
            foreach(var p in pi)
            {
                state[i] = p.GetValue(args[0]);
                i++;
            }
            foreach (var f in fi)
            {
                state[i] = f.GetValue(args[0]);
                i++;
            }
        }
        else//After base method
        {
            PropertyInfo[] pi = args[0].GetType().GetProperties();
            FieldInfo[] fi = args[0].GetType().GetFields();
            int i = 0;
            foreach (var p in pi)
            {
                if (state[i] == null)
                {
                    if (p.GetValue(args[0]) != null) throw new Exception("Changed state of curr object");
                }
                else if(!state[i].Equals(p.GetValue(args[0]))) throw new Exception("Changed state of curr object");
                i++;
            }
            foreach (var f in fi)
            {
                if (state[i] == null)
                {
                    if (f.GetValue(args[0]) != null) throw new Exception("Changed state of curr object");
                }
                else if (!state[i].Equals(f.GetValue(args[0]))) throw new Exception("Changed state of curr object");
                i++;
            }

        }
    }
}
```

Build

O método *Build* é a base de todo o trabalho, é o método responsável por toda a criação do novo tipo.

O método foi implementado de maneira a ser o mais genérico possível.

```
public static T Build<T>(params object[] args)
```

Este método começa por criar um *assembly* dinâmico, um módulo, e o novo tipo. O novo tipo tem o nome da classe base, acrescido de "Enhanced". É também definido o novo tipo como derivado da classe base.

```
Type klass = typeof(T);
string NAME = klass.Name + "Enhanced";

AssemblyBuilder asmBuilder = AssemblyBuilder.DefineDynamicAssembly(
    new AssemblyName(NAME),
    AssemblyBuilderAccess.RunAndSave);

ModuleBuilder modBuilder = asmBuilder.DefineDynamicModule(NAME, NAME + ".dll");

TypeBuilder tb = modBuilder.DefineType(NAME);

tb.SetParent(klass);
```

De seguida é implementado os construtores. São implementados todas os diferentes construtores disponíveis da classe base. Nestes construtores apenas é chamado o construtor correspondente da base.

```
//build constructors
ConstructorInfo[] ci = klass.GetConstructors(BindingFlags.Public);

foreach (ConstructorInfo c in ci)
{
    ParameterInfo[] parameters = c.GetParameters();
    Type[] argTypes = new Type[parameters.Length];
    int i = 0;
    foreach (ParameterInfo p in parameters)
    {
        argTypes[i] = p.ParameterType;
        i++;
    }

    ConstructorBuilder cb = tb.DefineConstructor(MethodAttributes.Public,
                    CallingConventions.Standard, argTypes);

    ImplementConstructor(c, cb, parameters.Length);
}
```

As propriedades e métodos implementados são aqueles que têm atributos, que derivam de *EnhancedAttribute*, anotados e que sejam virtuais ou abstratos. Assim aquando da implementação apenas são selecionados aquelas que cumprem os requisitos acima indicados.

Para as propriedades, foi necessário criar um campo e os métodos *get* e *set* correspondentes. O método *get* foi implementado da maneira normal, retornando apenas o valor do campo.

```
//build properties
PropertyInfo[] pi = GetProperties(klass);

foreach (PropertyInfo p in pi)
{
    FieldBuilder fb = tb.DefineField(p.Name, p.PropertyType, FieldAttributes.Private);

    PropertyBuilder pb = tb.DefineProperty(p.Name, PropertyAttributes.HasDefault, p.PropertyType, null);

    //get method

    MethodBuilder mbget = tb.DefineMethod("get_" + p.Name,
        MethodAttributes.Public |
        MethodAttributes.Virtual |
        MethodAttributes.SpecialName |
        MethodAttributes.HideBySig, p.PropertyType, Type.EmptyTypes);

    ILGenerator il = mbget.GetILGenerator();
    il.Emit(OpCodes.Ldarg_0);
    il.Emit(OpCodes.Ldfld, fb);
    il.Emit(OpCodes.Ret);

    //set method
    MethodBuilder mbset = tb.DefineMethod("set_" + p.Name,
        MethodAttributes.Public |
        MethodAttributes.Virtual |
        MethodAttributes.SpecialName |
        MethodAttributes.HideBySig, null, new Type[] { p.PropertyType });

    ImplementSetMethod(mbset, p.GetSetMethod(), p.Name, fb);

    pb.SetSetMethod(mbset);
    pb.SetGetMethod(mbget);
}
```

O método *set* é implementado do mesmo modo que os restantes métodos. É chamado o método *Check* do atributo anotado, seguido do método base, e do método *Check* novamente. Isto porque para verificar as mudanças de estado é necessário correr o método antes e depois da chamada ao método base. Para que seja tudo o mais genérico possível, não havendo distinções nas implementações para atributos diferentes, é feito isto para quaisquer que seja o atributo.

```
//build methods
MethodInfo[] mi = GetMethods(klass);

foreach (MethodInfo method in mi)
{
    string name = method.Name;
    Type t = method.GetType();


    ParameterInfo[] parameters = method.GetParameters();
    Type[] array = new Type[parameters.Length];
    int idx = 0;
    foreach (ParameterInfo p in parameters)
    {
        array[idx] = p.ParameterType;
        idx++;
    }
    MethodBuilder metBuilder = tb.DefineMethod(method.Name,
        MethodAttributes.Public |
        MethodAttributes.Virtual |
        MethodAttributes.ReuseSlot,
        method.ReturnType,
        array);
    ImplementMethods(metBuilder, method, name);
}
```

Na implementação dos métodos, é criado um *array* contendo a referência para o objeto, e todos os argumentos do método, para ser passado ao método *Check*. Sendo necessário fazer *box* quando os argumentos são *value types*.

```
//Construção do array de parametros
il.Emit(OpCodes.Ldc_I4, nparam + 1);
il.Emit(OpCodes.Newarr, typeof(Object));

il.Emit(OpCodes.Dup);
il.Emit(OpCodes.Ldc_I4_0);
il.Emit(OpCodes.Ldarg_0);
il.Emit(OpCodes.Stelem_Ref);

for (int i = 1; i <= nparam; i++)
{
    il.Emit(OpCodes.Dup);
    il.Emit(OpCodes.Ldc_I4, i);
    il.Emit(OpCodes.Ldarg, i);
    if (pi[i - 1].ParameterType.IsValueType) il.Emit(OpCodes.Box, pi[i - 1].ParameterType);
    il.Emit(OpCodes.Stelem_Ref);
}
il.Emit(OpCodes.Stloc_S, arrayThis);
```

De seguida é necessário, por reflexão, encontrar o atributo anotado ao método. Sabendo o nome do membro corrente, é utilizado M*emberInfo* em vez de *MethodInfo* pois nas propriedades o método *set* não tem o atributo anotado, mas a respetiva propriedade sim. Após obter o membro, e posteriormente o atributo, é feito um *cast* para *EnhancedAttribute* e feito uma chamada virtual, *callvirt*, ao método *Check*.

```
//get curr member
il.Emit(OpCodes.Ldarg_0);
il.Emit(OpCodes.Call, typeof(Object).GetMethod("GetType"));
il.Emit(OpCodes.Callvirt, typeof(Type).GetMethod("get_BaseType"));
il.Emit(OpCodes.Ldstr, memberName);
il.Emit(OpCodes.Callvirt, typeof(Type).GetMethod("GetMember", new Type[] { typeof(String) }));
il.Emit(OpCodes.Ldc_I4_0);
il.Emit(OpCodes.Ldelem_Ref);
il.Emit(OpCodes.Stloc_S, member);

//attribute
il.Emit(OpCodes.Ldloc_S, member);
il.Emit(OpCodes.Ldtoken, typeof(EnhancerAttribute));
il.Emit(OpCodes.Call, typeof(Type).GetMethod("GetTypeFromHandle", new Type[] { typeof(RuntimeTypeHandle) }));
il.Emit(OpCodes.Call, typeof(CustomAttributeExtensions).GetMethod("GetCustomAttribute", new Type[] { typeof(MemberInfo), typeof(Type) }));
il.Emit(OpCodes.Castclass, typeof(EnhancerAttribute));
il.Emit(OpCodes.Stloc_S, a);

//check method(pre)
il.Emit(OpCodes.Ldloc_S, a);
il.Emit(OpCodes.Ldloc_S, arrayThis);
il.Emit(OpCodes.Callvirt, typeof(EnhancerAttribute).GetMethod("Check"));
```

Temos a seguir chamada ao método base, com todos os argumentos. Mas no caso do método *set* é aqui que este se difere, sendo apenas feito uma atribuição ao campo correspondente.

```
//base method
il.Emit(OpCodes.Ldarg_0);
for (int i = 1; i <= nparam; i++)
{
    il.Emit(OpCodes.Ldarg, i);
}

il.Emit(OpCodes.Call, m);
if (m.ReturnType != typeof(void)) il.Emit(OpCodes.Stloc_S, toret);
```

Por fim é feito uma última chamada ao método *Check* e retornado, caso exista o valor retornado pelo método base.

No final do método *Build* é criado o tipo, guardado o *assembly* e é usada a classe *Activator* para criar uma instância do novo tipo.