Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Ambientes Virtuais de Execução – Semestre de Verão – 2014/2015

Segunda série de exercícios

Objectivos: Prática com a API de reflexão, *Custom Attributes*, Delegates e Genéricos
Data limite de entrega: 25 de Maio de 2015

NOTA 1: resolva esta a série de exercícios completando a base de código disponível neste repositório Github
NOTA 2: implemente todos os testes unitários que entender necessários para validar o correcto funcionamento das funcionalidades pedidas.

1. Implemente a função utilitária `AndThen` com a seguinte descrição:

```
Func<T, Rafter> AndThen<T, Rself, Rafter>(this Func<T, Rself> self, Func<Rself, Rafter> after)
```

Na Figura 1 é apresentado um exemplo de utilização da função AndThen

```
        static int[] measurer(String[] words) {
            return words.Select(w => w.Length).ToArray();
        }
        static int sum(int[] nrs) {
            return nrs.Aggregate((prev, curr) => prev + curr);
        }
        static void Main(string[] args) {
            Func<String, String[]> splitter = line => line.Split(' ');
            Func<String, int> nrOfChars = splitter.AndThen(measurer).AndThen(sum);
            String src = "Phasellus quam turpis feugiat sit amet ornare in";
            Console.WriteLine(nrOfChars(src)); // 41
        }
```

Figura 1

2. Implemente a função utilitária `ChainMethods` com a seguinte descrição:

```
Func<T, T> ChainMethods<T>(string path)
```

que retorna uma nova instância de `Func<T, T>` cuja execução é igual à chamada encadeada de todos os métodos estáticos compatíveis com `Func<T, T>` do *assembly* localizado em path.
NOTA: deve usar a função `AndThen` da alíena 1.

Por exemplo, dado um *assembly* `ArraysUtils.dll` com uma classe com a definição igual à da Figura 2, então a Figura 3 apresenta um caso de utilização da função `ChainMethods`.

```
static class DoubleArraysUtils{
  static double[] sqrt(double[] a) {
    return a.Select(n => Math.Sqrt(n)).ToArray();
  }
  static double[] duplicate(double[] a) {
    return a.Select(n => n * 2).ToArray();
  }
  static double[] square(double[] a) {
    return a.Select(n => n * n).ToArray();
  }
}
```

Figura 2

```
double[] a1 = { 16, 25 };
Func<double[], double[]> f =
   ChainMethods<double[]>("ArraysUtils.dll");
double[] res = f(a1); // res = {64, 100}
```

Figura 3

3. Pretende-se adicionar suporte para a especificação de *validadores* (instâncias de `IValidator<T>`) que devem ser executados conjuntamente com os métodos encadeados por `ChainMethods`.
   A interface `IValidator<T>` tem a seguinte definição: `interface IValidator<T>{ bool validate(T arg); }`

   Implemente uma nova função utilitária `ChainMethodsValidators` com comportamento semelhante ao da função `ChainMethods`, mas que executa o validador especificado em cada método caso exista, conforme especificado no exemplo das Figuras 4 e 5.
   Caso o resultado da validação seja false, então lança a excepção `ValidationException`.
   Caso os parâmetros do validador não sejam compatíveis com os parâmetros do método anotado, lança a excepção `IllegalMethodForValidator`, detalhando o nome do método

```
class NonNullArgument : IValidator<Object>
{
  public bool validate(object arg) {
    return arg != null;
  }
}

class NonNegativeDoubleArray : IValidator<double[]>
{
  public bool validate(double[] arg)
  {
    return arg.Where(a => a < 0).Count() == 0;
  }
}
```

Figura 4

```
static class ArraysUtils{

  [Validator(typeof(NonNullArgument))]
  static double[] sqrt(double[] a) {
    return a.Select(n => Math.Sqrt(n)).ToArray();
  }

  [Validator(typeof(NonNullArgument))]
  static double[] duplicate(double[] a) {
    return a.Select(n => n * 2).ToArray();
  }

  [Validator(typeof(NonNullArgument))]
  [Validator(typeof(NonNegativeDoubleArray))]
  static double[] square(double[] a) {
    return a.Select(n => n * n).ToArray();
  }
}
```

Figura 5

14 de Maio de 2015