Problema 3 de Programación
Curso 2012-2013
Calculadora Simple

NOTA: Si usted está leyendo este documento sin haber extraído el compactado que descargó del sitio, ciérrelo ahora, extraiga todos los archivos en el escritorio, y siga trabajando desde ahí. Es un error común trabajar en la solución dentro del compactado, lo cual provoca que los cambios **no se guarden**. Si usted comete este error y entrega una solución vacía, no tendrá oportunidad de reclamar.

La siguiente clase ha sido diseñada para describir una operación:

```
/// <summary>
/// Describe una operación especificando el operador y el valor con el que operar
/// </summary>
public class Operacion
{
    /// <summary>
    /// Constructor, crea una instancia a partir del operador y el valor
    /// especificados
    /// </summary>
    /// <param name="operador">Operador que caracteriza la operación, solamente
    /// son aceptados uno de los 4 valores siguientes: ('+','-','*','/')
    /// </param>
    /// <param name="operando">Valor con el que se realizará la operación</param>
    public Operacion(char operador, int operando)
    {
        Operador = operador;
        Operando = operando;
    }

    /// <summary>
    /// Propiedad de solo lectura para acceder al Operador que caracteriza la
    /// operación
    /// </summary>
    public char Operador { get; private set; }

    /// <summary>
    /// Propiedad de solo lectura para acceder al Operando con que se realizará la
    /// operación
    /// </summary>
    public int Operando { get; private set; }

    /// <summary>
    /// Redefinición del método ToString() para facilitar la visualización de
    /// objetos de este tipo
    /// </summary>
    /// <returns>Retorna una cadena que representa la operacion</returns>
    public override string ToString()
    {
        return string.Format("({0} {1})", Operador, Operando);
    }
}
```

NOTA: La clase descrita anteriormente ya está implementada como parte de los ensamblados que se proveen para la realización del examen. Esta clase ha sido implementada para usarse tal y como se ha descrito, usted no necesita realizar su propia implementación de la misma.

La siguiente `interface` ha sido diseñada para encapsular las operaciones básicas de una Calculadora acumulativa.

```
public interface ICalculadora
{
    /// <summary>
    /// Procesa la instrucción especificada. Si la operación especificada
    /// es '/' y el valor es 0 el método debe lanzar una excepción del
    /// tipo DivideByZeroException.
    /// </summary>
    /// <param name="operacion">Instancia de la clase Operacion que describe
    /// la operacion a realizar.
    /// </param>
    void Ejecuta(Operacion operacion);

    /// <summary>
    /// Deshace la última instrucción, si no hay ninguna instrucción el
    /// método no tiene ningún efecto.
    /// </summary>
    void Undo();

    /// <summary>
    /// Deja sin efecto la última instrucción "Undo". El método solamente
    /// tiene efecto cuando es invocado después de una instrucción de tipo
    /// "Undo" o "Redo".
    /// </summary>
    void Redo();

    /// <summary>
    /// El método calcula el resultado de realizar todas las operaciones
    /// efectivas. Una operación efectiva es toda aquella que no ha sido
    /// "des-hecha" a través de la instrucción "Undo", o que después
    /// de "des-hecha" ha sido "re-hecha" a través de la instrucción "Redo"
    /// </summary>
    /// <returns>Retorna el resultado calculado</returns>
    int ResultadoActual();

    /// <summary>
    /// Devuelve una secuencia de todas las operaciones efectivas realizadas
    /// hasta el momento
    /// </summary>
    /// <returns>La secuencia calculada</returns>
    IEnumerable<Operacion> OperacionesEfectivas();
}
```

NOTA: La interface descrita anteriormente ya está definida como parte de los ensamblados que se proveen para la realización del examen. Esta interface ha sido definida para usarse tal y como se ha descrito, usted no necesita realizar su propia definición de la misma.

Problema

Usted debe proveer una implementación de la interface descrita anteriormente. El objetivo de esta clase es realizar operaciones aritméticas en forma acumulativa al estilo calculadora. Esta clase deberá ofrecer también un mecanismo para “des-hacer” o “re-hacer” operaciones mediante los métodos Undo y Redo.

```
public class CalculadoraSimple: ICalculadora
{
    /// <summary>
    /// Constructor sin parámetros
    /// </summary>
    public CalculadoraSimple()
    {
        //TODO: Implementar las inicializaciones necesarias
    }

    /// <summary>
    /// Procesa la instrucción especificada. Si la operación especificada
    /// es '/' y el valor es 0 el método debe lanzar una excepción del
    /// tipo DivideByZeroException.
    /// </summary>
    /// <param name="operacion">Instancia de la clase Operacion que describe
    /// la operacion a realizar.
    /// </param>
    public void Ejecuta(Operacion operacion)
    {

    }

    /// <summary>
    /// Deshace la última instrucción, si no hay ninguna instrucción el
    /// método no tiene ningún efecto.
    /// </summary>
    public void Undo()
    {

    }

    /// <summary>
    /// Deja sin efecto la última instrucción "Undo". El método solamente
    /// tiene efecto cuando es invocado después de una instrucción de tipo
    /// "Undo" o "Redo".
    /// </summary>
    public void Redo()
    {

    }

    /// <summary>
    /// El método calcula el resultado de realizar todas las operaciones
    /// efectivas. Una operación efectiva es toda aquella que no ha sido
    /// "des-hecha" a través de la instrucción "Undo", o que después
    /// de "des-hecha" ha sido "re-hecha" a través de la instrucción "Redo"
    /// </summary>
    /// <returns>Retorna el resultado calculado</returns>
    public int ResultadoActual()
    {

    }
```

```
    /// <summary>
    /// Devuelve una secuencia de todas las operaciones efectivas realizadas hasta el
    /// momento.
    /// </summary>
    /// <returns>La secuencia calculada</returns>
    public IEnumerable<Operacion> OperacionesEfectivas()
    {

    }
}
```

ACLARACIONES:

- Su implementación debe contener un único constructor, sin parámetros, el cual será usado en la evaluación automática para crear instancias de esta clase. En caso de no contener este constructor todos los casos de prueba se considerarán fallidos.
- Se considerará que la calculadora se inicia con el valor 0 (cero) como valor acumulado. Luego las operaciones serán ejecutándose acumulativamente según se vayan aplicando. De modo que la aplicación sucesiva del método Ejecuta con los parámetros "+ 2", "+ 3", "* 5" daría como resultado 25.
- Cada operación se aplica al resultado acumulado
- La operación de división realizará la división entera. Es decir el resultado de hacer 7 / 3 es 2.
- Aplicar el método `Undo` significa des-hacer la última instrucción efectiva. En el caso en que no existan instrucciones efectivas, ya sea porque la calculadora está recién construida o porque ya se han "des-hecho" todas las operaciones realizadas con anterioridad entonces la aplicación de este método NO TIENE EFECTO. Las operaciones "des-hechas" no intervienen en el cálculo del resultado, excepto en el caso que son "re-hechas" mediante la aplicación del método `Redo`.
- La aplicación de un `Redo` solamente tiene efecto después de la si se ha aplicado con anterioridad un `Undo`. De manera general se podrán realizar tantos `Redo` efectivos como `Undo` se hayan realizado inmediatamente antes. De igual modo un `Redo` que no corresponda a un `Undo` no tendrá efecto.
- El iterador `OperacionesEfectivas` deberá retornar todas las operaciones efectivas. Es decir, aquellas que intervendrían en el cálculo del resultado si se aplicara el método `ResultadoActual`.

Ejemplos

A continuación se muestran algunos ejemplos del uso de esta clase:

Ejemplo 1:

```
CalculadoraSimple cs = new CalculadoraSimple();

cs.Ejecuta(new Operacion('+', 2));
cs.Ejecuta(new Operacion('+', 3));
cs.Ejecuta(new Operacion('*', 10));

Console.WriteLine(cs.ResultadoActual()); //imprime 50

cs.Undo();

Console.WriteLine(cs.ResultadoActual()); //imprime 5

cs.Undo();
```

```
Console.WriteLine(cs.ResultadoActual()); //imprime 2

cs.Redo();

Console.WriteLine(cs.ResultadoActual()); //imprime 5
```

Ejemplo 2:

```
CalculadoraSimple cs = new CalculadoraSimple();

cs.Ejecuta(new Operacion('*', 0)); //operacion 1

Console.WriteLine(cs.ResultadoActual()); // imprime 0

cs.Ejecuta(new Operacion('/', 5)); //operacion 2

Console.WriteLine(cs.ResultadoActual()); // imprime 0

cs.Ejecuta(new Operacion('+', 10)); //operacion 3
cs.Ejecuta(new Operacion('+', 10)); //operacion 4
cs.Ejecuta(new Operacion('+', 10)); //operacion 5
cs.Ejecuta(new Operacion('+', 10)); //operacion 6

Console.WriteLine(cs.ResultadoActual()); // imprime 40

cs.Undo(); //des-hace la opereación 6

Console.WriteLine(cs.ResultadoActual()); // imprime 30

cs.Ejecuta(new Operacion('+', 5)); //operacion 7

cs.Redo(); //sin efecto pq la instrucción inmediata anterior no fue "Undo"

Console.WriteLine(cs.ResultadoActual()); // imprime 35

cs.Undo();  //des-hace la opereación 7
cs.Undo();  //des-hace la opereación 5 (la operacion 6 ya había sido des-hecha)

Console.WriteLine(cs.ResultadoActual()); // imprime 20

cs.Redo(); //re-hace la operacion 5
cs.Redo(); //re-hace la operacion 7
cs.Redo(); //sin efecto porque solo se habían hecho dos Undo inmediatamente antes

Console.WriteLine(cs.ResultadoActual()); // imprime 35
//Note que en este caso la operacion 6 fue des-hecha y no interviene en el resultado

foreach (var operacion in cs.OperacionesEfectivas())
{
    Console.WriteLine(operacion);
}

//Note que la operación 6 no se considera efectiva (fue deshecha y no se rehízo)

//Esto imprime las operaciones 1, 2, 3, 4, 5 y 7, es decir se escribiría
// (* 0)
// (/ 5)
// (+ 10)
// (+ 10)
// (+ 10)
```

```
// (+,5)
```

Ejemplo 3:

```
CalculadoraSimple cs = new CalculadoraSimple();

cs.Ejecuta(new Operacion('+', 5)); //operacion 1

cs.Undo(); //des-hace la operacion 1
cs.Undo(); //sin efecto, no existen instrucciones efectivas

Console.WriteLine(cs.ResultadoActual()); //imprime 0

cs.Ejecuta(new Operacion('/', 0)); //lanza DivideByZeroException
```