Manejo de errores en C#

Índice

1 Introducción

En esta sección se trata de la gestión de errores. Como postura ante los errores en el código, debe asumirse que los errores siempre van a estar presentes. La actitud correcta no es esperar que no ocurran, sino todo lo contrario: de hecho, es necesario estar preparado para tratar de capturar cuantos errores sean posibles, lo antes posible. Cuanto antes se localice un error, mejor, pues se evitará que casos como que, por ejemplo, esté latente cuando el módulo sea integrado a la aplicación final, y que este error provoque otros errores en ésta muy difíciles de rastrear; o que se manifiesten cuando se añada funcionalidad al código que antes no estaba presente.

Específicamente, las ideas del Diseño o programación por Contrato, tienen sus raíces en los métodos formales para la construcción de software, pero mantienen una visión más pragmática. Requieren muy poco esfuerzo extra pero generan software mucho más confiable. La idea fue introducida en una fecha tan temprana como 1992 por Bertrand Meyer, y su lenguaje de programación Eiffel.

La programación por contrato puede ser imaginada como la aplicación a la construcción de software de los contratos que rigen los asuntos de las personas. Cuando dos personas establecen un contrato se desprenden de éste, las obligaciones y beneficios de cada una.

Si ahora trasladamos estos conceptos al diseño del software, lo que se busca obtener es que este tipo de contratos en software especifican, en forma no ambigua, las relaciones entre las rutinas y los llamadores de las mismas. Así, un sistema sería como un conjunto de elementos de software interrelacionados, donde cada uno de los elementos tiene un objetivo con vistas a satisfacer las necesidades de los otros. Dichos objetivos son los contratos. Los contratos deben cumplir, por lo menos, con dos propiedades: ser explícitos y formar parte del elemento de software en sí mismo.

El Diseño por Contrato da una visión de la construcción de sistemas como un conjunto de elementos de software cooperando entre sí. Los elementos juegan en determinados momentos alguno de los dos roles principales proveedores o clientes. La cooperación establece claramente obligaciones y beneficios, siendo la especificación de estas obligaciones y beneficios los contratos.

Un contrato entre dos partes protege a ambas. Por un lado protege al cliente por especificar cuanto debe ser hecho y por el otro al proveedor por especificar el mínimo servicio aceptable.

2 Capacidades de C# para depuración

Antes de pasar a la programación por contrato, es necesario establecer varios recursos disponibles en C# para crear escribir logs y crear aserciones, de manera que sean sencillos de utilizar más tarde.

2.1 Creación de logs

Un recurso muy conocido en el desarrollo es el de la creación de *logs*. Un *log* es básicamente un archivo de texto en el que se escribe una línea por cada evento importante llevado a cabo con éxito. Normalmente, también se muestran allí los errores, por lo que es un recurso importantísimo para conocer cuál fue el error y qué estaba haciendo la aplicación cuando este se produjo.

Este recurso se usa sobre todo en el perfil de depuración (DEBUG) del desarrollo, pero no quita que se distribuyan aplicaciones que hacen un uso importante del *logging* ante la previsión de que se puedan producir errores no detectados durante el desarrollo.

En Opciones del proyecto >> Compilador, en el editor de símbolos, debe añadirse “TRACE” para que se pueda activar el *logging*. Normalmente, los símbolos del perfil de depuración incluyen por defecto DEBUG, y ninguno en el perfil de liberación (RELEASE). Así, en el caso de que se quiera activar el logging, el contenido de los símbolos del perfil de depuración debería ser “DEBUG;TRACE”, y en el perfil de RELEASE (si se quiere activar esta posibilidad una vez liberada la aplicación), “TRACE”.

Un ejemplo puede verse a continuación.

```
class Ppal {
            [Conditional("DEBUG")]
            private static void CreateConsoleTracing()
            {
                  Trace.Listeners.Add( new ConsoleTraceListener( true ) );
            }

            private static void CreateFileTracing()
            {
                  var tracerFile = new TextWriterTraceListener( "events.log" );
                  Trace.Listeners.Add( tracerFile );
            }

            public static void Main(string[] args)
            {
                  // Creando un listener para la consola
                  CreateConsoleTracing();

                  // Creando un listener para un archivo
                  CreateFileTracing();

                  Trace.WriteLine(  “Ppal.Main(): “,
                                    “Creados los listeners para los logs” );
                  Trace.Indent();
                  Trace.WriteLine( “Ppal.Main(): “, “Performing main task” );
                  System.Console.WriteLine( “Hello, world!” );
                  Trace.Unindent();
                  Trace.WriteLine( “Ppal.Main(): “, “Finished” );
            }
}
```

El atributo *Conditional(“DEBUG”)* hace que el método afectado no se compile si no está presente el símbolo DEBUG. Es decir, solo se compila (así como las llamadas a dicho método), en el perfil de desarrollo. En el caso de la creación del *listener* de la consola, es seguro que no se quiere que este exista en el perfil de RELEASE de la aplicación, por lo que se ha añadido dicho atributo.

La forma de hacer que los mensajes generados para los *logs* realmente se guarden en algún sitio es crear *listeners*. Los *listeners* se registran en la clase Trace, de manera que podemos duplicar o triplicar…, sin límite, los mensajes en la consola, en un archivo, en cualquier medio. En el caso del ejemplo, tenemos dos *listeners*: uno que es la propia consola (clase **ConsoleTraceListener**), y otro que es un archivo (clase **TextWriterTraceListener**). Durante la creación de los *listener*, se pasa, respectivamente, **true** si se desea que se use la salida estándar de errores, o **false** si se quiere utilizar la salida estándar en cuanto al primero; y un nombre de archivo en el caso del segundo.

La clase **Trace** tiene varios métodos de conveniencia: *Indent()* que incrementa el indentado de la salida del *log*, *Unindent()* que hace lo contrario. Por su parte, *WriteLine(s1, s2)* permite mostrar mensajes en el log, mientras que *WriteLine(s, sformat, obj1, obj2,…, objn)* permite especificar mensajes con formato (*sformat*), además de los datos a formatear (*obj1*, *obj2*,…, *objn*).

2.2 Asserts

El mecanismo de aserciones se basa en proveer métodos que, tras comprobar una condición, muestran un mensaje y opcionalmente, paran la ejecución. En el caso concreto de C#, se trata de los métodos de la clase **Assert**, que se incluyen o no en la compilación según se trate del perfil de depuración o no.

Así, el mecanismo principal es la clase **Debug**, en el espacio de nombres *System.Diagnostics*, que soporta varios métodos útiles en depuración. El más importante es *Assert()*, que acepta una condición, una cadena de texto, una segunda cadena de texto de formato, y un número variable de datos a imprimir. Por ejemplo:

```
Debug.Assert( op2 != 0, “op2 es 0”, “{0}”, op2 );
```

En la anterior línea se comprueba la condición de que la variable *op2* sea distinta de 0. En caso de que no se cumpla, se muestra el mensaje, por la consola de errores, “op2 es 0”, y a continuación, el valor de *op2* (0) mediante la cadena de formato.

Más abajo aparece un ejemplo completo de una clase **Div**, que utiliza *Assert()* para comprobar que el segundo operador no sea 0, lo cual produciría como resultado NaN.

```
namespace AssertExample.Core {
	public class Div: Operator {
		public Div(double a, double b): base( 2 )
		{
			this.Op1 = a;
			this.Op2 = b;
		}

		public Div(double[] ops): base( ops )
		{
			Debug.Assert(
				this.ops.Length == 2,
				"Wrong arguments for Divider",
				"Args={0}", this.ops.Length
			);
		}

		public Div(string[] ops): base( ops )
		{
			Debug.Assert(
				this.ops.Length == 2,
				"Wrong arguments for Divider",
				"Args={0}", this.ops.Length
			);
		}

		public double Op1 {
			get {
				return this.ops[0];
			}
			set {
				this.ops[0] = value;
			}
		}

		public double Op2 {
			get {
				return this.ops[1];
			}
			set {
				this.ops[1] = value;
			}
		}
```

```
            public override double Result {
                  get {
                        Log( this );

                        Debug.Assert(
                              this.Op2 != 0,
                              "op2 == 0!!",
                              "{0} / {1}",
                              this.Op1, this.Op2 );
                        return this.Op1 / this.Op2;
                  }
            }

            public override string ToString()
            {
                  return string.Format(
                        "[Divider: Op1={0}, Op2={1}]",
                        this.Op1, this.Op2 );
            }
      }
}
```

La clase base de **Div** aparece a continuación. Nótese que el método estático *Log()* está marcado con el atributo condicional **System.Diagnostics.Conditional**, de tal forma que las llamadas a él no serán compiladas en caso de que se compile en el perfil de depuración.

```
namespace AssertExample.Core {
      public abstract class Operator {
            public Operator(int num)
            {
                  this.ops = new double[num];
            }

            public Operator(double[] ops)
            {
                  this.ops = (double[]) ops.Clone();
            }

            public Operator(string[] ops)
            {
                  this.ops = new double[ ops.Length ];

                  for(int i = 0; i < ops.Length; ++i) {
                        double op;

                        if ( !double.TryParse(
                              ops[ i ], out op ) )
                        {
                              op = 0;
                        }

                        this.ops[ i ] = op;
                  }

                  return;
            }

            public abstract double Result {
                  get;
            }
```

```
            [Conditional("DEBUG")]
            protected static void Log(Operator op)
            {
                  Console.WriteLine( op.ToString() );
            }

            protected double[] ops;
      }
}
```

Si se desea también se puede utilizar el método "más tradicional" (aunque más sucio, y mucho menos legible) de la compilación condicional a la hora de llamar a *Log()*:

```
public class Div {
      // ...más cosas...
      public override double Result {
            get {
      #if DEBUG
                  Log( this );
      #endif
                  Debug.Assert(
                        this.Op2 != 0,
                        "op2 == 0!!",
                        "{0} / {1}",
                        this.Op1, this.Op2 );
                  return this.Op1 / this.Op2;
            }
      }
}
```

La única ventaja es que no es necesario marcar el método *Log()* con el atributo *Conditional("DEBUG")*, como se veía en el listado más arriba.

A la hora de compilar, será necesario indicarle al compilador el perfil de depuración. También es necesario indicarle por dónde se va a producir la salida de los mensajes derivados de *Debug.Assert()*, antes de la ejecución del programa.

```
      $ mcs -debug -D:DEBUG -out:AssertExample.exe
            -r:System.Windows.Forms.dll,System.Drawing.dll
            Core/Operator.cs
            Core/Div.cs
            Gui/Ppal.cs
            Gui/MainWindow.cs
      $ export MONO_TRACE_LISTENER=Console.Error
      $ ./AssertExample.exe
```

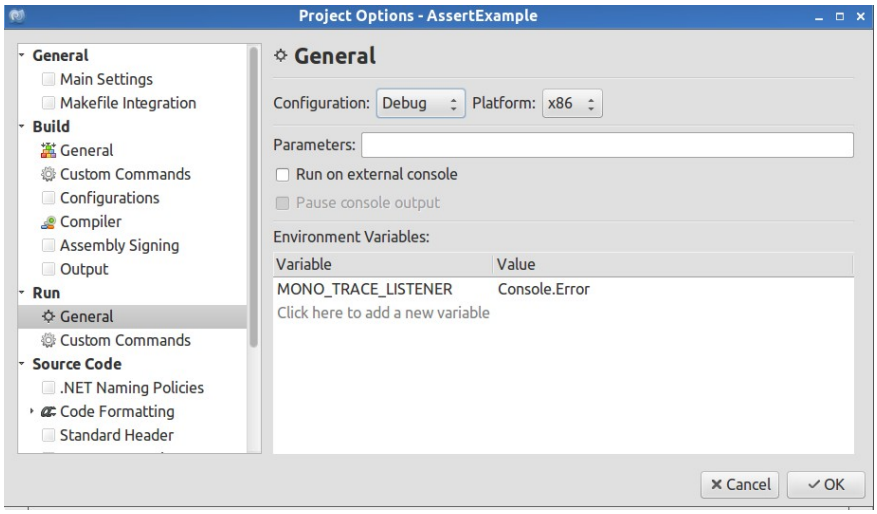
Al compilar, *-debug* hace que el compilador genere la información de depuración. *-D:DEBUG*, hace que se compile con la constante DEBUG activada, mientras con *-r* se incluyen las referencias necesarias para el proyecto. Al definir la variable de entorno `MONO_TRACE_LISTENER`, se consigue que la salida de los métodos de la clase **Debug** se muestren por la consola de errores. Es muy importante recalcar que si no se define la constante DEBUG, o no se le da valor a la variable de entorno, no se mostrará la salida de *Assert()*.

Así, la compilación para el caso en el que se desee la aplicación final (perfil *release*), en el que el rendimiento no se ve afectado por los procesos de depuración, pues para las llamadas a estos no se ha generado código, se muestra a continuación.

```
$ mcs -out:AssertExample.exe
            -r:System.Windows.Forms.dll,System.Drawing.dll
      Core/Operator.cs
      Core/Div.cs
      Gui/Ppal.cs
      Gui/MainWindow.cs
$ ./AssertExample.exe
```

De esta forma es indiferente que se defina o no la variable de entorno.

En el caso de que se esté utilizando un entorno, como MonoDevelop, este ya crea de forma automática la constante DEBUG en el perfil de depuración, si bien no define la variable de entorno requerida. Para ello, es necesario abrir las opciones del proyecto (no de la solución), y en la sección *Run >> General*, añadir la variable de entorno para el perfil *debug*. El símbolo DEBUG aparece ya creado en *Build >> Compiler*.



La salida de *Assert()* se mostrará entonces en la pestaña *Application Output*.

Hay que notar que en cuanto se añade un *listener* a **Trace**, también se está haciendo de manera automática, a **Debug**. Es decir, haber habilitado *listeners* para hacer *logging* hará que automáticamente se muestren los mensajes de error en caso de que algún *assert* se evalúe a falso. Eso sí, solo con la variable de entorno se mostrará la pila de llamadas y se interrumpirá la ejecución.

2.3 Excepciones

La gestión de excepciones en C# es muy parecida a la de C++, o Java, pioneros en este aspecto.

```
public class Ppal {
    public static int Divide(int a, int b)
    {
      if ( b == 0 ) {
        throw new ArgumentException(
                  "el divisor no puede ser cero" );
      }

      return a / b;
    }

    public static int Main()
    {
        int a;
        int b;


        try {
            Console.WriteLine( "Introduzca un número: " );
            a = Convert.ToInt32( Console.ReadLine() );
            Console.WriteLine( "Introduzca un número: " );
            b = Convert.ToInt32( Console.ReadLine() );

            Console.WriteLine(
                  "Resultado: {0}", Divide( a, b ) );
        } catch(FormatException e) {
            Console.WriteLine(
                  "\nERROR número inválido: '{0}'",
                  e.Message );
            Environment.Exit( -1 );
        } catch(ArgumentException e) {
            Console.WriteLine(
                  "\nERROR en argumento: '{0}'",
                  e.Message );
            Environment.Exit( -2 );
        } catch(Exception e) {
            Console.WriteLine( "\nERROR inesperado: '{0}'",
                        e.Message );
            Environment.Exit( -3 );
        }

        return 0;
    }
}
```

Es obligatorio en C# que los objetos lanzados pertenezcan a clases derivadas de **Exception**, siendo posible crear nuevas clases de excepciones solo haciéndolas derivar de la clase **Exception**.

C#, al igual que Java, acepta cláusulas *finally*. Estas cláusulas se colocan al final de los bloques *catch*, y se ejecutan siempre, independientemente de que se haya lanzado una excepción o no. Son muy útiles para liberar recursos que se hayan reservado previamente, como por ejemplo, un fichero.

```
public class Ppal {
    public static int Divide(int a, int b)
    {
      if ( b == 0 ) {
        throw new ArgumentException(
                  "el divisor no puede ser cero" );
      }

      return a/b;
    }

    public static int Main()
    {
        int a;
        int b;
        StreamReader reader;

        try {
            reader = new StreamReader( "input.txt" );

            Console.WriteLine(
                  "Leyendo primer argumento..." );
            a = Convert.ToInt32( reader.ReadLine() );
            Console.WriteLine(
                  "Leyendo segundo argumento..." );
            b = Convert.ToInt32( reader.ReadLine() );

            Console.WriteLine( "Resultado: {0}",
                        Divide( a, b ) );
        } catch(FormatException e) {
            Console.WriteLine(
                  "\nERROR número inválido: '{0}'",
                  e.Message );
            Environment.Exit( -1 );
        } catch(ArgumentException e) {
            Console.WriteLine(
                  "\nERROR en argumento: '{0}'",
                  e.Message );
            Environment.Exit( -2 );
        } catch(Exception e) {
            Console.WriteLine(
                  "\nERROR inesperado: '{0}'",
                  e.Message );
            Environment.Exit( -3 );
        }
        finally {
            reader.Close();
        }

        return 0;
    }
}
```

Es posible crear nuestras propias excepciones, como se ve en el ejemplo, derivadas siempre de **Exception** o de una subclase de **Exception**. En este caso, es necesario crear tres constructores, como se puede apreciar más abajo. Uno sin argumentos, otro con una cadena como mensaje, y otro con una cadena y una excepción interna.

```
public class DivisionByZero: ArgumentException {
    public DivisionByZero()
      : base() {}

    public DivisionByZero(string msg)
      : base(msg) {}

    public DivisionByZero(string msg, Exception inner)
            : base(msg, inner) {}
}

public class Ppal {
    public static int Divide(int a, int b)
    {
      if ( b == 0 ) {
        throw new DivisionByZero(
                  "el divisor no puede ser cero" );
      }

      return a/b;
    }

    public static int Main()
    {
        int a;
        int b;
        StreamReader reader;

        try {
            reader = new StreamReader( "input.txt" );

            Console.WriteLine(
                  "Leyendo primer argumento..." );
            a = Convert.ToInt32( reader.ReadLine() );
            Console.WriteLine(
                  "Leyendo segundo argumento..." );
            b = Convert.ToInt32( reader.ReadLine() );

            Console.WriteLine(
                  "Resultado: {0}", Divide( a, b ) );
        } catch(FormatException e) {
            Console.WriteLine(
                  "\nERROR número inválido: '{0}'",
                  e.Message );
            Environment.Exit( -1 );
        } catch(DivisionByZero e) {
            Console.WriteLine(
                  "\nERROR división por cero: '{0}'",
                  e.Message );
            Environment.Exit( -2 );
        } catch(Exception e) {
            Console.WriteLine(
                        "\nERROR inesperado: '{0}'",
                         e.Message );
            Environment.Exit( -3 );
        }
        finally {
            reader.Close();
        }

        return 0;
    }
}
```

2.4 Resumen

Aserciones y excepciones son técnicas válidas para validar el código en cualquier proyecto. Sin embargo, es necesario tener en cuenta que la técnica de aserciones es demasiado basta como para poder aplicarla con éxito a la validación de la entrada del usuario. También es necesario tener en cuenta que la gestión de excepciones parece ideal para todas las situaciones, pero evidentemente consume mucho más recursos que la primera, que puede ser "eliminada" cuando el proyecto se envía a producción.

Tipo de error	assert()	try... catch
Validar entrada del usuario	**No**	Sí
Errores de la lógica de negocio de la aplicación	Sí	Sí (aunque con pérdida de rendimiento)

3 Programación por contrato

Los contratos de software se especifican mediante la utilización de expresiones lógicas denominadas aserciones. En el Diseño por Contratos se utilizan tres tipos de aserciones:

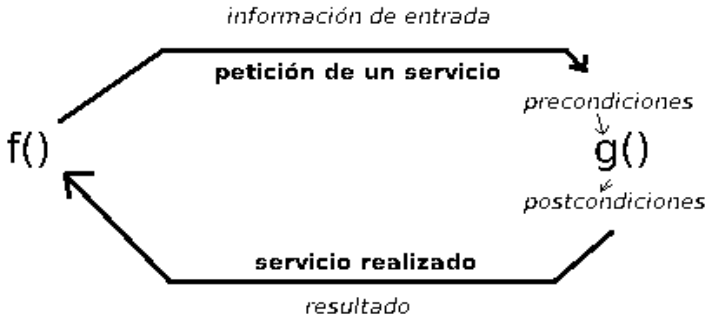


Figura 1: Esquema básico de funcionamiento: f llama a g, por lo que g realiza un servicio para f.

- Precondiciones
- Poscondiciones
- Invariantes de clase

Se denominan aserciones porque son condiciones que deben cumplirse. Su incumplimiento invalida totalmente el software, hace que éste deje de trabajar, pues el incumplimiento de un contrato, como se vé en la figura 1, indica que existe un error en el programa.

De una manera algo formal, se puede explicar también mediante el concepto de tripleta de Hoare, la cual es una notación matemática que viene de la validación formal de programas. Sea A alguna computación y P, Q aserciones, entonces la siguiente expresión:

```
{ P } A { Q }
```

representa lo que se llama fórmula de corrección. La semántica de dicha fórmula es la siguiente: cualquier ejecución de A que comience en un estado en el cual se cumple P dará como resultado un estado en el cual se cumple Q. Por ejemplo

```
{ Prec.: longitud( str1 ) > 4 }
subcadena( str1, 0, 4, str2 );
{ Postc.: longitud( str2 ) == 4 }
```

En este ejemplo, se parte de un estado en el que la cadena str1 debe tener más de cuatro caracteres, puesto que se van a extraer para guardarlos en la cadena str2. La precondición (P, como se la mencionaba antes), precisamente, trata de asegurar que ese estado sea exactamente el estado de partida, comprobando si la cadena tiene al menos cuatro caracteres. Entonces se ejecuta la tarea, (extraer una subcadena de otra), y la postcondición (Q, tal y como se mencionaba antes) trata de verificar que el trabajo se ha cumplido correctamente (si se han extraído cuatro caracteres de str1, debería haber cuatro caracteres en str2).

Por supuesto, este ejemplo es trivial. Ningún programador haría precondiciones y postcondiciones para una tarea tan sencilla como ésta (aunque la precondición nunca estaría de más).

3.1 Precondiciones

Las *precondiciones* son aquellas condiciones que se deben dar para asegurar que una tarea puede llevarse a cabo.

```
public class Mates {
      public static int Divide(int a, int b)
      {
              Debug.Assert( b != 0 );

              return a / b;
      }
}
```

3.2 Postcondiciones

Las *postcondiciones* son complementarias a las *precondiciones*. Comprueban que el trabajo realizado se ha podido llevar a buen término.

Supóngase la siguiente función, que divide un número entre dos.

```
public class Mates {
      public static int DividePorDos(int a)
      {
              int resultado = a >> 1;

              Debug.Assert( resultado == ( a / 2 ) );

              return resultado;
      }
}
```

Para obtener un mejor rendimiento, en lugar de utilizar la división normal, se empleará el desplazamiento de bits. El *assert* utilizado asegura que la división siempre va a funcionar correctamente. Este es un tipo de *postcondición* muy utilizado, en el que se comprueba el correcto resultado de una optimización, aunque por supuesto hay muchos usos aparte de este.

3.3 Invariantes de clase

Las invariantes de clase son básicamente condiciones que deben cumplirse siempre en sus objetos. Por supuesto, “Siempre” puede ser reducido a que sean ciertas después de la ejecución de cualquiera de sus funciones miembro públicas. Una forma sencilla de emplear invariantes de clase es reunirlas en una función y llamarla en las *precondiciones* y *postcondiciones* de cada método. Por supuesto, existen muchas otras aproximaciones y también varios lenguajes de programación, como *Eiffel* o *D*, que soportan invariantes de manera nativa, con lo que no es necesario hacer esto.

En el siguiente ejemplo, se crea la clase *Natural*, usando un entero para representar a los números naturales.

```
public class Natural: IComparable, IComparable<Natural> {
    public Natural(int x = 0)
    {
        this.Value = x;
    }

    public int Value {
        get {
            return this.num;
        }
        set {
            this.num = value;
            this.ChkInvariant();
        }
    }
```

```
    public override string ToString()
    {
        return this.Value.ToString();
    }

    // más cosas...

    [Conditional("DEBUG")]
    private void ChkInvariant()
    {
        Debug.Assert( this.num >= 0 );
    }

    private int num;
}
```

Se utiliza una propiedad para asegurar que cualquier modificación al atributo num suponga la comprobación de la invariante. Al utilizar la propiedad incluso desde dentro de la propia clase, se asegura que siempre se compruebe que el valor siga siendo un número natural. A continuación, se muestran las operaciones básicas soportadas.

```
public class Natural: IComparable, IComparable<Natural> {

    // más cosas...

    public Natural Substract(int x)
    {
      this.Value -= x;
      return this;
    }

    public Natural Add(int x)
    {
      this.Value += x;
      return this;
    }
}
```

Para una clase como **Natural**, es importante sobreescribir los métodos *Equals()* y *GetHashCode()*. Además, también es interesante implementar las interfaces *IComparable()*. Todo esto facilitará la sobrecarga de los operadores aritméticos + y -, así como los operadores lógicos >, <, ==, y !=.

```
public class Natural: IComparable, IComparable<Natural> {

    // más cosas...

    public override bool Equals(object o)
    {
      bool toret = false;

      if ( o != null
        && o is Natural )
      {
            toret = ( this.Value == ( (Natural) o ).Value );
      }

      return toret;
    }
```

```
    public override int GetHashCode()
    {
      return this.Value.GetHashCode();
    }

    public int CompareTo(object o)
    {
      int toret = 1;

      if ( o != null
        && o is Natural )
      {
            toret = this.Value.CompareTo( ( (Natural) o ).Value );
      }

      return toret;
    }

    public int CompareTo(Natural n2)
    {
      return this.Value.CompareTo( n2.Value );
    }
}
```

A continuación, los operadores de conversión explícita (casting), y los aritméticos +, -, ++ y –.

```
public class Natural: IComparable, IComparable<Natural> {

    // más cosas…

    public static explicit operator int(Natural n)
    {
        return n.Value;
    }

    public static explicit operator Natural(int x)
    {
        return new Natural( x );
    }

    public static Natural operator+(Natural n1, Natural n2)
    {
      return new Natural( n1.Value ).Add( n2.Value );
    }

    public static Natural operator-(Natural n1, Natural n2)
    {
      return new Natural( n1.Value ).Substract( n2.Value );
    }

    public static Natural operator++(Natural n1)
    {
      ++n1.Value;
      return n1;
    }

    public static Natural operator--(Natural n1)
    {
      --n1.Value;
      return n1;
    }
}
```

Y finalmente, los operadores lógicos.

```
public class Natural: IComparable, IComparable<Natural> {

    // más cosas…

      public static bool operator==(Natural n1, Natural n2)
      {
            return ( (object) n1) != null && n1.Equals( n2 );
      }

      public static bool operator!=(Natural n1, Natural n2)
      {
            return !( n1 == n2 );
      }

      public static bool operator<(Natural n1, Natural n2)
      {
            return ( (object) n1) != null && n1.CompareTo( n2 ) < 0;
      }

      public static bool operator>(Natural n1, Natural n2)
      {
            return ( (object) n1) != null && n1.CompareTo( n2 ) > 0;
      }

      public static bool operator<=(Natural n1, Natural n2)
      {
            return ( n1 == n2 || n1 < n2 );
      }

      public static bool operator>=(Natural n1, Natural n2)
      {
            return ( n1 == n2 || n1 > n2 );
      }
}
```

En este caso, la invariante de clase se comprueba después de la ejecución de aquellos métodos que suponen una variación en el contenido del objeto *Natural*, por el simple hecho de utilizar la propiedad *Value*.. Como es obvio, la variable nunca puede ser menor que cero, pues en ese caso no se estaría hablando de números naturales. Nótese que se usa la estructura de la clase en tres capas, de manera que cada capa se asienta sobre la anterior (atributo *num* y propiedad *Value*, operaciones *Add()* y *Substract()*, métodos *Equals()* y *CompareTo()*, y finalmente, los operadores sobrecargados.

Se usa esta clase para detectar un error de programación bastante típico en el siguiente trozo de código:

```
class Test {
      static void Main()
      {
            for(var i = (Natural) 5; i >= (Natural) 0; --i) {
                  Console.WriteLine( i );
            }
      }
}
```

3.4 Pasos a realizar para aplicar programación por contrato

Para obtener los beneficios de este tipo de diseño, se pueden recordar una serie de pasos a seguir:

1. Separar consultas de comandos. Este principio también fue inicialmente explicado detalladamente en Meyer. La idea es que las rutinas de una clase deben ser (en lo posible) o comandos o consultas pero no ambas cosas. Las consultas devuelven un valor (ej. funciones) y los comandos pueden cambiar el estado interno del objeto.
2. Separar consultas básicas de consultas derivadas. La intención es conseguir un conjunto de especificación formado por consultas que denominamos básicas, de tal forma que el resto de las consultas puedan derivarse de las básicas.
3. Para cada consulta derivada escribir una postcondición especificando su resultado en términos de una o más consultas básicas. Esto permite conocer el valor de las consultas derivadas conociendo el valor de las consultas básicas. Idealmente, sólo un conjunto mínimo de especificación tiene la obligación de ser exportado públicamente.
4. Para cada comando escribir una precondición que especifique el valor de cada consulta básica. Dado que el resultado de todas las consultas puede visualizarse a través de las consultas básicas, con este principio se garantiza el total conocimiento de los efectos visibles de cada comando.
5. Para toda consulta o comando decidir una precondición adecuada. Este principio se auto explica ya que permite definir claramente el contrato de cada rutina.

4 Pruebas de unidad

Las pruebas de unidad son muy parecidas a la programación por contrato, aunque a otro nivel (mayor) de abstracción, normalmente una clase o un módulo. En cuanto al contrato, se debe interpretar que las *postcondiciones* e invariantes de clase de una clase deben probarse utilizándolas como clientes. Se trata de escribir juegos de comprobaciones que validen si una clase funciona correctamente o no. Muchas veces se habla de hecho de *Test-driven development* o Desarrollo Basado en Pruebas, de manera que primero se escriben los tests que tiene que pasar una clase o un módulo, y posteriormente se implementa esta, depurándola hasta que pasa las pruebas de unidad.

En realidad, el Desarrollo Basado en Pruebas es mucho más radical, llevando las pruebas de unidad al extremo de ser el centro de la funcionalidad, de tal forma que es aceptable escribir código claramente incorrecto o incompleto, siempre y cuando cumpla con las pruebas. Entonces se mejoran las pruebas para que el código falle, se refactoriza el código si es necesario, y se crea nuevo código que cumpla con las pruebas. Esta aproximación es excesivamente radical, mientras las Pruebas de Unidad son verdaderamente útiles en cualquier proyecto.

La gran ventaja de crear estas pruebas (muchas veces llamadas directamente *tests*) es que ante cualquier modificación se pueden volver a pasar dichas pruebas de unidad, de forma que se comprueba si la modificación realizada introduce o no nuevos errores en el código. De esta manera, funcionan como pruebas de regresión.

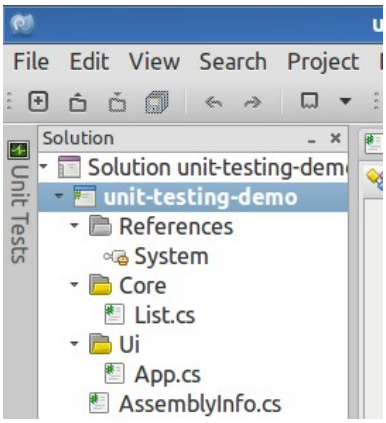
Lo ideal sería apartar totalmente las pruebas del código fuente realmente útil, de manera que en el mismo tan solo queden aquellos verificaciones relacionadas con diseño por contrato. Los *frameworks* existentes que propocionan la posibilidad de incorporar fácilmente pruebas de unidad al desarrollo del proyecto funcionan exactamente de esta forma, separando el código de pruebas del propio código del proyecto.

4.1 Aplicando pruebas de unidad en C#

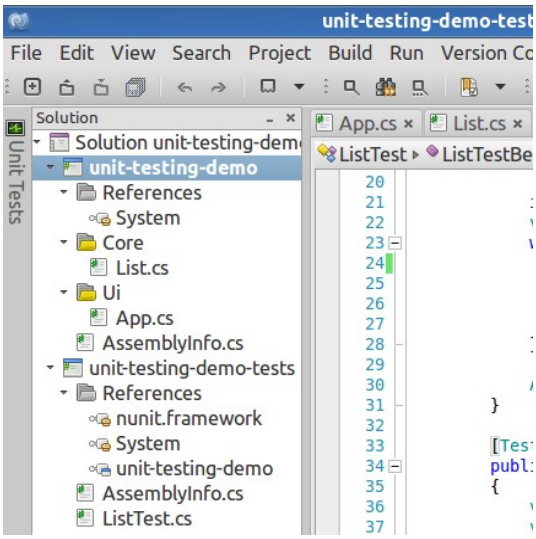
Las pruebas de unidad en C# se estructuran alrededor del *framework* NUnit[1]. Si bien existen otros muchos *frameworks* para pruebas de unidad, incluyendo la solución propia de Microsoft, publicada con Visual Studio, NUnit es ampliamente utilizada debido a estar basada en un *framework* como

1 http://www.nunit.org/

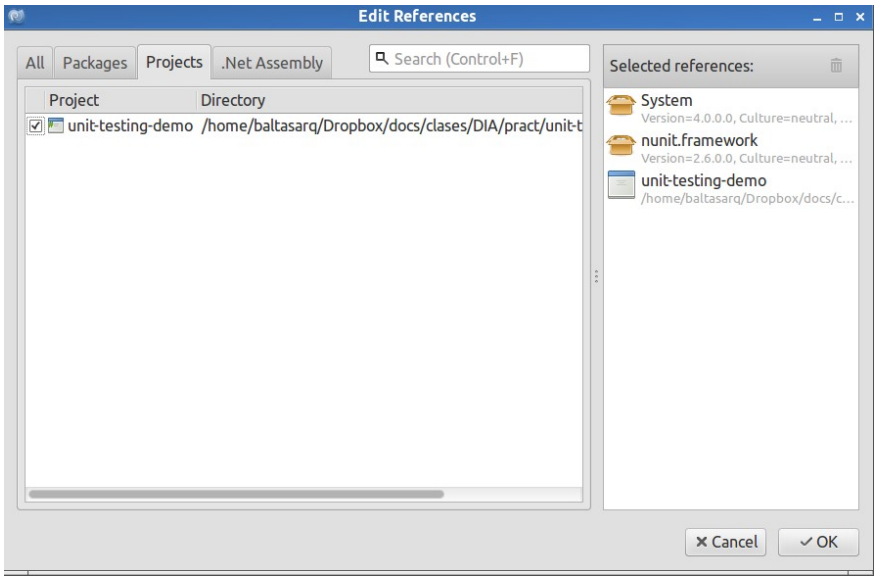
JUnit, que cuenta con una amplia trayectoria. Además, NUnit se puede usar de forma indiferente en Visual Studio (con integración, si se desea) o en MonoDevelop (donde está también integrado, si se desea), o de forma totalmente independiente mediante **nunit-gui**, descargable desde su página.



El primer paso para realizar pruebas de unidad en un proyecto (en este caso, *unit-testing-demo*), es crear bajo la solución de ese proyecto una nueva librería (*dll*).



Una vez hecho esto, es necesario, para esa nueva librería, añadir una referencia al proyecto del que se quiere realizar pruebas de unidad, a la vez que a *NUnit.framework*, la dll que realmente proporciona la funcionalidad requerida.



El código utilizado es el de una lista, *List.cs*, que proporciona la funcionalidad de una lista enlazada de forma simple. En primer lugar, se crea el nodo del que va a estar compuesto la lista en sí.

```
/// <summary>
/// Single-linked list.
/// </summary>
public class List<T> {
    /// <summary>
    /// A node of the list.
    /// </summary>
    internal class Node {
        public Node(T data, Node next)
        {
            this.Data = data;
            this.Next = next;
        }

        /// <summary>
        /// The data held in the node.
        /// </summary>
        public T Data {
            get; set;
        }

        /// <summary>
        /// Next node. The end is marked with null.
        /// </summary>
        public Node Next {
            get; set;
        }
    }

    // más cosas…
}
```

A continuación, un iterador que encapsula el nodo cuando lo utilice el usuario para recorrer la lista.

```
public class List<T> {
    /// <summary>
    /// An iterator, used to traverse the list.
    /// </summary>
    public class Iterator {
        internal Iterator(List<T> l, Node n)
        {
            this.list = l;
            this.node = n;
        }

        public Iterator(List<T> l)
                  : this( l, l.begin ) {}

        /// <summary>
        /// Sets the iterator at the beginning of the list.
        /// </summary>
        public void SetBegin()
        {
            this.node = this.list.begin;
        }
```

```
        /// <summary>
        /// Goes to the next node in the list.
        /// </summary>
        /// <returns>
        /// The node that is going to be returned
        ///  by the Current property.
        /// </returns>
        public void Next()
        {
            if ( this.Current != null ) {
                this.node = this.node.Next;
            }

            return;
        }

        /// <summary>
        /// Determines whether there is a next node.
        /// </summary>
        /// <returns>
        /// true if Next() can be called, false otherwise.
        /// </returns>
        public bool HasNext()
        {
            return ( this.node != null );
        }

        /// <summary>
        /// The list being traversed.
        /// </summary>
        public List<T> List {
            get {
                return this.list;
            }
        }

        /// <summary>
        /// The current data pointed by the iterator.
        /// </summary>
        public T Current {
            get {
                return this.node.Data;
            }
        }

        private List<T> list;
        private Node node;
    }
```

Finalmente, la funcionalidad de la lista en sí, soportando adiciones, un iterador al nodo inicial y otro al final.

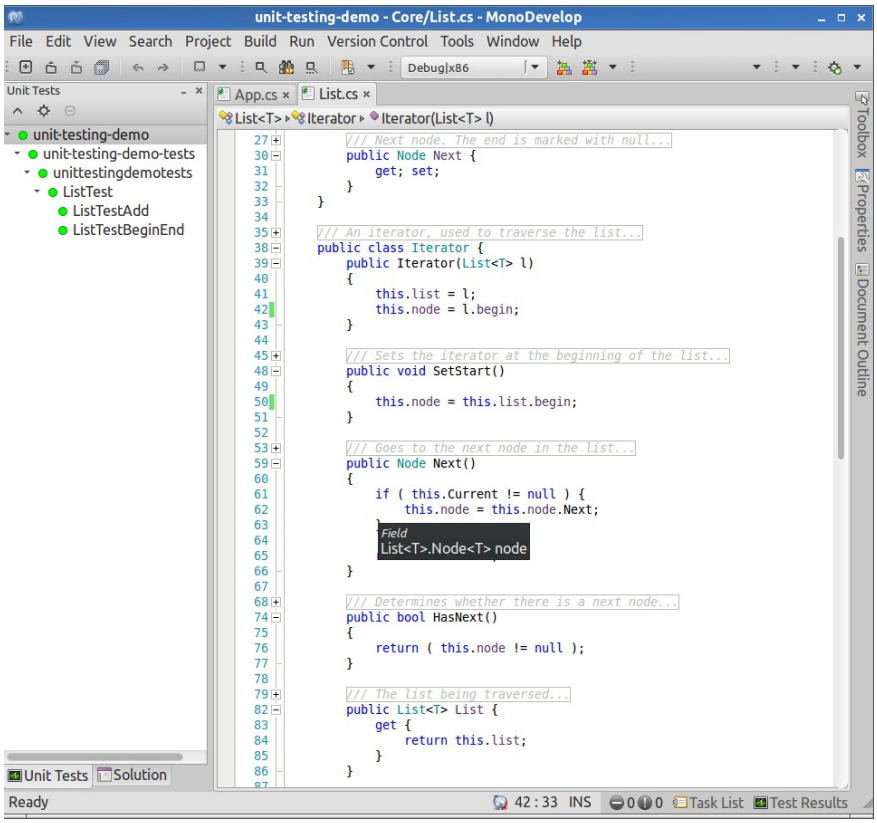
```
public class List<T> {
    public List()
    {
        this.begin = this.end = null;
    }

    /// <summary>
    /// Adds a new element to the end of the list.
    /// </summary>
    /// <param name="x">
    /// The new data element to append.
    /// </param>
    public void Add(T x)
    {
        var newNode = new Node( x, null );

        // Append at the end
        if ( this.end == null ) {
            this.end = newNode;
        } else {
            this.end.Next = newNode;
            this.end = this.end.Next;
        }

        // Set start, if needed
        if ( this.begin == null ) {
            this.begin = this.end;
        }
    }

    /// <summary>
    /// Returns the first node of the list.
    /// </summary>
    public Iterator Begin {
        get{
            return new Iterator( this, this.begin );
        }
    }

    /// <summary>
    /// Returns the last node of the list.
    /// </summary>
    public Iterator End {
        get{
            return new Iterator( this, this.end );
        }
    }

    private Node end;
    private Node begin;
}
```

En la nueva librería creada, es necesario escribir métodos que prueben cada uno de los aspectos de nuestra clase. Lo habitual es crear una clase de *testing* por cada clase a verificar.

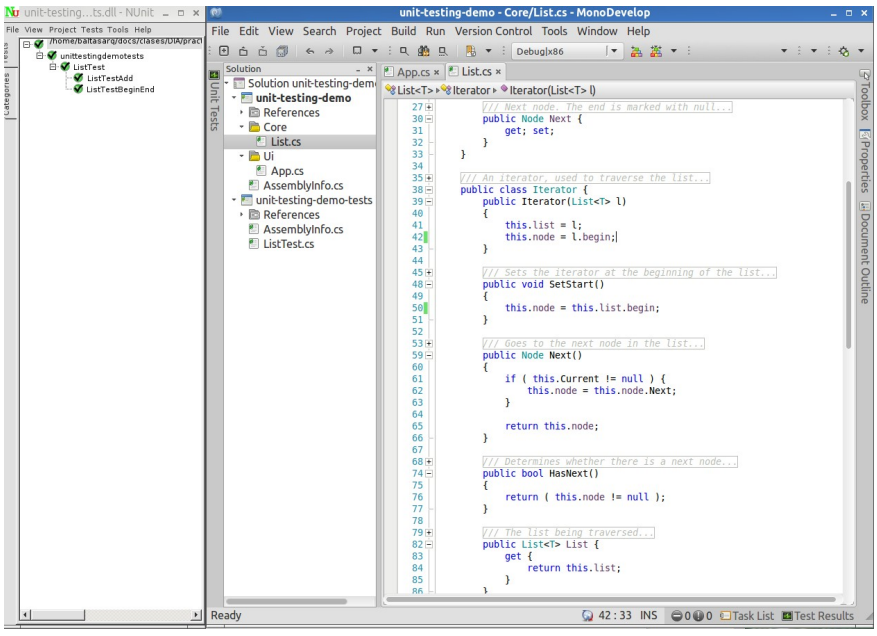
```
using NUnit.Framework;

using unittestingdemo.Core;

namespace unittestingdemotests {

	[TestFixture]
	public class ListTest {

		[Test]
		public static void ListTestAdd()
		{
			var l = new List<int>();
			var test = new int[] { 1, 2, 3 };

			foreach (int x in test) {
				l.Add( x );
			}

			int i = 1;
			var it = new List<int>.Iterator( l );
			while( it.HasNext() ) {
				Assert.AreEqual( i,
						it.Current );

				it.Next();
				++i;
			}

			Assert.AreEqual( test.Length + 1,
							i );
		}

		[Test]
		public static void ListTestBeginEnd()
		{
			var l = new List<int>();
			var test = new int[] { 1, 2, 3 };
			int beg = -1;

			foreach (int x in test) {
				l.Add( x );

				Assert.AreEqual( x,
					l.End.Current );

				if ( beg == -1 ) {
					beg = l.Begin.Current;
				}

				Assert.AreEqual( beg,
					l.Begin.Current );
			}
		}
	}
}
```

Como se puede observar, la clase que contiene los métodos se decora con el atributo **TestFixture**, mientras que cada uno de los métodos que realizan comprobaciones se decora con un **Test**.

Una vez creados los test, se pueden ejecutar dentro de MonoDevelop visualizando el panel de pruebas de unidad, y haciendo clic en *run* (ejecutar).



Alternativamente, es posible mediante el uso de **nunit-gui**, un programa que se puede descargar de *nunit.org*, ejecutar los tests independientemente, simplemente abriendo con la herramienta la librería (extensión *dll*) creada para los tests. En la siguiente imagen se puede ver **nunit-gui** a la izquierda de **MonoDevelop**.



5 Referencias

Libros

1. **Mithcell, R., McKim, J. (2001).** *Design by contract, by example*. Addison-Wesley Professional. ISBN 978-0201634600
2. **Osherove, R. (2009)**. *Art of Unit Testing: With Examples in .NET*. Manning Pubn. ISBN 978-1933988276

Referencias web

1. Diseño por contrato: http://es.wikipedia.org/wiki/Diseño_por_Contrato
2. Pruebas de unidad: http://es.wikipedia.org/wiki/Pruebas_de_unidad
3. NUnit: http://www.nunit.org/