Jorge Luis De Armas
Miguel Katrib

Diseño por Contratos en .NET 4.0

Un deseo hecho realidad

Como continuidad de nuestra promesa de hablar sobre lo que vendrá con .NET 4.0 y Visual Studio 2010, el presente artículo está dedicado a una de estas novedades, cuyas ideas tal vez aún no sean muy conocidas por algunos desarrolladores: Code Contracts.

Aunque en .NET 4.0 solo estará disponible como biblioteca de ejecución (es decir, no integrado en la sintaxis de los lenguajes de programación, como sí lo fue desde el primer momento LINQ), **Code Contracts** [1] es un mecanismo para especificar requerimientos y garantías (**contratos**) que no se pueden representar solamente con metadatos establecidos por las signaturas de los métodos y los tipos.

Estos contratos se expresan en forma de pre-condiciones, pos-condiciones e invariantes asociados a los objetos, y permitirán mejorar las capacidades de programación en tres aspectos fundamentales:

- Análisis estático del código, o sea, la posibilidad de detectar errores de programación sin ejecutar el código.
- Chequeo de inconsistencias en tiempo de ejecución.
- Documentación del código.

Un poco de historia

La verificación automática de programas ha sido un viejo anhelo de la Ciencia de la Computación; como demostración fehaciente de ello tenemos los trabajos pioneros de **Dijkstra** [2] y **Hoare** [3], que sentaron las bases para que ya desde hace tiempo en un lenguaje como CLU [4][1] aparecieran las **aserciones**.

Con el desarrollo de la programación orientada a objetos, y posteriormente la programación orientada a componentes, en donde la reusabilidad se convierte en la piedra angular, requerimos de mecanismos que nos den garantías no solo acerca de nuestro propio código, sino del código desarrollado por otros. No fue hasta la propuesta de **Bertrand Meyer** de **Diseño por Contratos** (*Design by Contracts*) en su lenguaje **Eiffel** [5] que el uso de las aserciones se integró verdaderamente a un lenguaje de programación[2].

La idea de Meyer, aunque genial, se basa en una metáfora muy simple. Entre el que diseña una clase y el cliente de la misma se establece un "contrato" que define los deberes y derechos de ambos. Estos deberes y derechos se van a representar, en la terminología de Eiffel, en forma de pre-condiciones, pos-condiciones e invariantes, tal y como se muestra en la tabla 1.

Una **pre-condición** es una cláusula lógica que debe estar libre de efectos colaterales, y que debe cumplirse como requisito para ejecutar el método al que está asociada. Un uso frecuente de las pre-condiciones es validar los parámetros de entrada de los métodos (algo que no puede satisfacerse solo con la declaración y tipado estático de los parámetros).

Una **pos-condición** es una cláusula lógica que debe cumplirse luego de la ejecución del método, y funge como garantía de lo que éste hace. Por tanto, las pos-condiciones son evaluadas justo después de ejecutado el método y antes de retornar al código llamante.

[1] Honor a Barbara Liskov, su creadora, que recientemente ha recibido el premio Turing por todas sus contribuciones.

[2] Esta fue la causa principal por la que en su momento nos enamoramos a primera vista de Eiffel (ver nuestros trabajos [6] y [7]).

Miguel Katrib es doctor y profesor jefe de programación del departamento de Ciencia de la Computación de la **Universidad de La Habana.** Miguel es líder del grupo WEBOO, dedicado a la orientación a objetos y la programación en la Web (*www.weboomania.com*). Es entusiasta de .NET y redactor de **dotNetManía**. Colaborador de DATYS Tecnología y Sistemas.

Jorge Luis de Armas (Jochy) es Arquitecto Principal y desarrollador de la empresa de software DATYS Tecnología y Sistemas. Es además colaborador y profesor invitado del grupo WEBOO de la Universidad de la Habana. (*jldearmas.blogspot.com*)

	Deberes u obligaciones	Derechos o beneficios
Cliente de la clase	Pre-condiciones	Pos-condiciones e invariantes
Diseñador de la clase	Pos-condiciones e invariantes	Pre-condiciones

Tabla 1. Relación de deberes y derechos entre clientes y diseñadores de clases.

Finalmente, un **invariante** es una cláusula que establece las condiciones bajo las cuales el estado de un objeto es "correcto". Por tanto, los invariantes se validan luego de creados los objetos y después de ejecutar cualquier método[3].

```
indexing
    description: "Clase Monedero"
class
    MONEDERO
create
    Crea
feature – Se exportan todos
    Saldo: DOUBLE;
    Depositar(cant: DOUBLE) is
        require
            cant >= 0;
        do
            Saldo := Saldo + cant;
        ensure
            Saldo = old Saldo + cant;
        end
    Extraer(cant: DOUBLE) is
        require
            cant >= 0;  cant <= Saldo;
        do
            Saldo := Saldo - cant;
        ensure
            Saldo = old Saldo - cant;
        end
    Crea(saldoInicial: DOUBLE) is
        require
            saldoInicial >= 0;
        do
            Saldo := saldoInicial;
        ensure
            Saldo = saldoInicial;
        end
invariant
    Saldo >= 0;
end
```

Listado 1. Clase Monedero en Eiffel.

Un ejemplo: monedero en Eiffel

El código Eiffel del listado 1 nos ilustra los tres tipos de aserciones. Aún cuando el lector no conozca la sintaxis de Eiffel, creemos que pueda captar la esencia de este ejemplo. En él se presenta una clase `Monedero` que cumple tres funciones básicas: ver el saldo que contiene el monedero y depositar y extraer cantidades de dinero en el mismo.

Note que los métodos `Depositar` y `Extraer` tienen como pre-condición, a cumplir por el código cliente que invoca al método, que el parámetro que se le pase tenga un valor positivo. Esta misma pre-condición le garantiza al desarrollador del método de la clase no tener que preocuparse de controlar esto dentro del código del método. El método `Extraer` tiene, además, otra pre-condición que indica que la cantidad a extraer debe ser menor o igual que el saldo disponible.

Las pos-condiciones de ambos métodos establecen el compromiso del diseñador de la clase con el código cliente de la misma, de que luego de realizar la operación de depósito o extracción el `Saldo` quedará establecido correctamente.

Por último, el invariante garantiza que el saldo del monedero siempre se mantenga mayor o igual que cero.

Contratos en otros lenguajes

Lamentablemente, en .NET originalmente no se incluyó el Diseño por Contratos[4], y aunque hay implementaciones de Eiffel para .NET, lo cierto es que Eiffel no se ha popularizado entre los desarrolladores de .NET. Ha habido varias propuestas para implementar contratos en diferentes lenguajes (comerciales y de investigación) que incluso los ofrecen directamente como parte de su sintaxis. Entre estos se pueden destacar **Delphi Prism** [8] (anteriormente llamado **Oxygene**) y **Spec#** [9].

Delphi Prism es un lenguaje comercial de la familia Pascal, inspirado en Delphi, donde el Diseño por Contratos se incorpora utilizando una sintaxis similar a la de Eiffel, aunque con algunas limitaciones. Su mayor insuficiencia es que no permite definir aserciones para las interfaces. Tampoco la relación con la herencia es la deseada, y la posibilidad de hacer referencia a los valores originales (**old**) en las pos-condiciones se reduce solo a los tipos por valor.

Los autores de este trabajo con miembros del equipo de Pex en el reciente Tech-Ed Berlín 2009

Por otra parte, Spec# es un lenguaje experimental desarrollado por el propio Microsoft Research a partir de C# (2.0) y para el cual además se implementó una herramienta para verificación estática del código conocida como **Boogie**. El listado 2 muestra la misma clase `Monedero` implementada en Spec#. Nótese el uso de nuevos elementos, como son las palabras claves `requires`, `ensures` e `invariant`, así como los atributos `[SpecPublic]` y `[Pure]`. Es a partir de la experiencia del equipo de Spec# que Microsoft ha desarrollado **Code Contracts**, para generalizar los resultados de Spec# de modo que puedan ser utilizados por cualquier lenguaje cuyo compilador genere código IL.

[3] Usted puede leer en la propuesta de Meyer [5] sobre la conveniencia de evaluar los invariantes también antes de ejecutar cada método; discutir sobre esto se sale del espacio de este artículo.

[4] Tenemos que confesar que, en los orígenes de .NET, cuando preguntamos a miembros del equipo de Microsoft sobre el por qué de la no inclusión de Diseño por Contratos, nunca recibimos una respuesta convincente. Luego nos entusiasmamos con la aparición activa de Bertrand Meyer en algunos fórums de Microsoft, e incluso por la foto conjunta en la que apareció con Bill Gates. Pero después esas expectativas se desvanecieron.

```
using System;
using Microsoft.Contracts;

namespace Weboo
{
 public class Monedero
 {
  [SpecPublic]
  private float m_Saldo;

  public Monedero(float saldoInicial)
    requires saldoInicial >= 0;
    ensures m_Saldo == saldoInicial;
  {
    m_Saldo = saldoInicial;
  }

  public float Saldo
  {
    [Pure]
    get
        ensures result == m_Saldo;
    {
      return m_Saldo;
    }
  }

  public void Depositar(float cant)
    requires cant >= 0;
    ensures m_Saldo == old(m_Saldo) + cant;
  {
    m_Saldo += cant;
  }

  public void Extraer(float cant)
    requires cant >= 0;
    requires cant <= m_Saldo;
    ensures m_Saldo == old(m_Saldo)-cant;
  {
    m_Saldo -= cant;
  }

  invariant m_Saldo >= 0;
 }
}
```

Listado 2. Clase Monedero en Spec#.

Code Contracts (contratos en .NET)

Code Contracts ofrece cuatro recursos fundamentales:

- La clase estática **Contract**, que suministra un grupo de métodos estáticos que permiten especificar las precondiciones, pos-condiciones e invariantes.
- Nuevos atributos que permiten colocar meta-información en el código de los diferentes lenguajes, de modo de poder ofrecer información adicional sobre los contratos, para que luego pueda ser utilizada por las herramientas.
- Herramientas que trabajan sobre el código IL:
  - **ccrewrite.exe** modifica el código IL de un ensamblado para garantizar el chequeo de los contratos en tiempo de ejecución.
  - **cccheck.exe** realiza la verificación estática de los contratos (hasta donde es posible, claro, porque hay código que dependerá de valores de tiempo de ejecución).
  - **ccdocgen.exe** extiende el archivo de documentación (XML) que generan los compiladores a partir de los comentarios de documentación (**///** en C#), incluyendo la información relativa a los contratos.
  - **asmmeta.exe** genera ensamblados de referencia auxiliares, que se utilizan durante la verificación estática o la conversión del código IL con **cccheck.exe** y **ccrewrite.exe**, respectivamente.
- Ensamblados de referencia con información de contratos para los ensamblados estándar de .NET 4.0.

Los dos primeros elementos forman parte del nuevo espacio de nombres **System.Diagnostics.Contracts** y se implementan dentro de la versión de **mscorlib.dll** para .NET 4.0, mientras que los dos últimos se incluyen con el *plug-in* para integrar Code Contracts con Visual Studio [10]. Cuando este *plug-in* se descarga e instala, las herramientas mencionadas anteriormente son utilizadas automáticamente durante el proceso de generación de los proyectos. Es posible configurar para cada proyecto la forma en la que se desea utilizar la verificación de los contratos (figura 1).

Es de destacar que el uso de Code Contracts no es exclusivo de .NET Framework 4.0 y Visual Studio 2010. El *plug-in* que Microsoft distribuye para instalar las herramientas antes mencionadas también permite utilizar Code Contracts con versiones anteriores de .NET y las diferentes ediciones de Visual Studio 2008. Para utilizar Code Contracts con versiones del CLR anteriores a la 4.0, hay que incluir explícitamente en nuestros proyectos la referencia al ensamblado **Microsoft.Contracts.dll**.

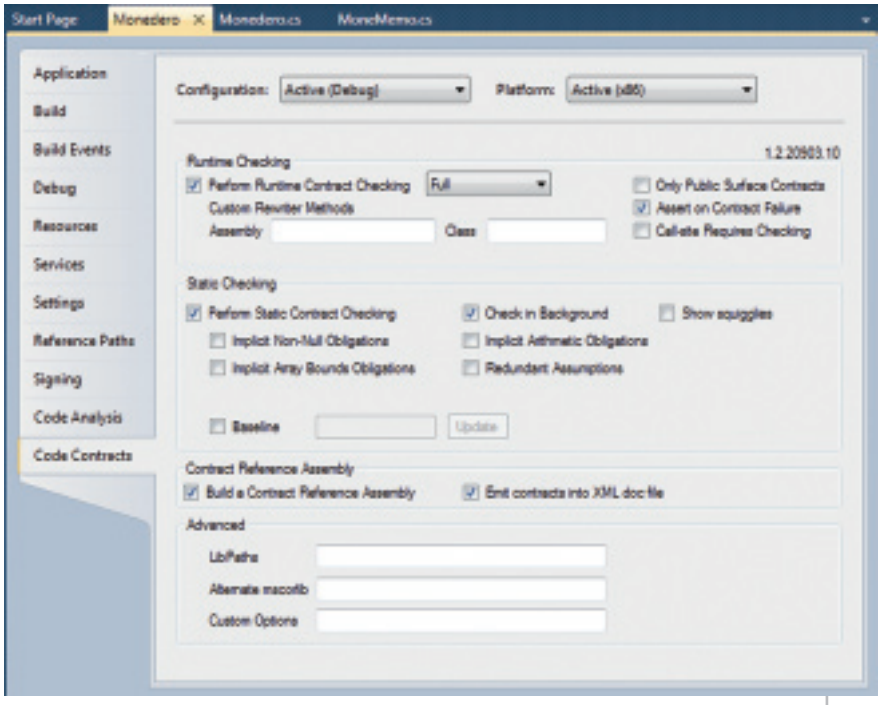

Figura 1. Ventana de configuración de Code Contracts en Visual Studio.

El incluir Code Contracts en .NET 4.0 en forma de una librería, en vez de como una extensión sintáctica a los lenguajes que se distribuyen con .NET, es ventajoso desde el punto de vista de que hace posible utilizarla desde cualquiera de los lenguajes disponibles para .NET. Sin embargo, esto le resta elegancia a la propuesta y exige una mayor disciplina de uso de parte de los programadores[5]. Al trabajar sobre código IL, las herramientas son más generales y simples, pues no dependen de la sintaxis de un lenguaje particular ni de un compilador específico. En particular, el verificador estático utiliza interpretación abstracta, que es más rápida y predecible que la verificación estática de Spec#[6] e incluso puede inferir invariantes de bucles y contratos de métodos.

Como parte de la integración con Visual Studio, están disponibles plantillas de código (*code snippets*) para C# y VB.NET (figura 2) que nos ayudan al tecleo de los contratos, algo que siempre se agradece.

En el listado 3 puede verse una implementación de **Monedero** utilizando Code Contracts. Note que las pre-condiciones y las pos-condiciones no se escriben con una sintaxis extensión de la de C#, sino invocando a los métodos **Contract.Requires** y **Contract.Ensures** dentro del código de los métodos.

También se dispone de otros mecanismos complementarios para usar en las

```
using System;
using System.Diagnostics.Contracts;

namespace Weboo
{
  public class Monedero
  {
    public Monedero(float saldoInicial)
    {
      Contract.Requires(saldoInicial >= 0);
      Contract.Ensures(Saldo == saldoInicial);
      Saldo = saldoInicial;
    }

    public float Saldo
    {
      get; private set;
    }

    public virtual void Depositar(float cant)
    {
      Contract.Requires(cant >= 0, "Depositar siempre una cantidad positiva.");
      Contract.Ensures(Saldo == Contract.OldValue(Saldo) + cant,
        "Después de depositar, el saldo debe ser igual al saldo anterior " +
        "más la cantidad depositada.");
      Saldo += cant;
    }

   public virtual void Extraer(float cant)
    {
      Contract.Requires(cant >= 0, "Extraer siempre una cantidad positiva.");
      Contract.Requires(cant <= Saldo, "No extraer nunca más de lo que hay.");
      Contract.Ensures(Saldo == Contract.OldValue(Saldo) - cant,
        "Después de extraer, el saldo debe ser igual al saldo anterior " +
        "menos la cantidad extraída.");
      Saldo -= cant;
    }

    [ContractInvariantMethod]
    protected void Invariante()
    {
      Contract.Invariant(Saldo >= 0);
    }
  }
}
```

Listado 3. Clase Monedero en C# con Code Contracts.

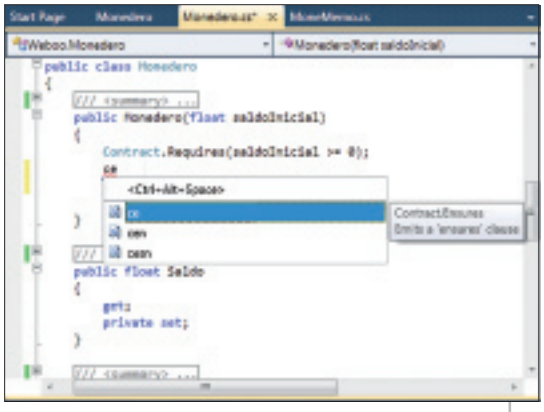

Figura 2. Plantillas de código C# para insertar pre-condiciones, pos-condiciones e invariantes.

pos-condiciones; tal es el caso del método **Contract.OldValue**, que permite referirse al valor que computó una expresión antes de ejecutar el método. Note que en el listado 3 se usa **Contract.OldValue(Saldo)** para referirse al valor de **Saldo** antes de ejecutar los métodos **Depositar** y **Extraer**.

Otro recurso para usar en las pos-condiciones es referirnos al resultado devuelto por el método; esto se logra invocando a **Contract.Result<T>** (ver listado 5), y mediante **Contract.ValueAtReturn<T>** podemos también hacer referencia a los valores de los parámetros de salida del método (si los tuviere).

Para convertir de forma sencilla a esta notación de contratos un código de verificación del tipo **if-then-throw** que ya existiese dentro de algún método legado, se dispone del método **Contract.EndContractBlock**. Colocando una invocación a este método se marca el fin del segmento de código que hace de "pre-condición" (listado 4).

[5] Los creadores de Code Contracts (que en buen número coinciden con los de Spec#) argumentan esto diciendo que lo diseñaron sobre la base de la experiencia con Spec# y lo que aprendieron sobre qué funciona y qué no (ver [11]).

[6] ¡La verificación estática de Spec# incluye un demostrador de teoremas!

```
public Empleado(Departamento dpto)
{
  if (dpto == null)
    throw new ArgumentNullException();
  Contract.EndContractBlock();
  Dpto = dpto;
}
```

Listado 4. Uso de EndContractBlock para definir un bloque de contrato a partir de código legado.

Cuantificadores en los contratos

En Code Contracts existe la posibilidad de usar, dentro de los contratos, el cuantificador universal **Contract.ForAll** y el cuantificador existencial **Contract.Exists**.

Contract.ForAll tiene dos argumentos. El primero es una colección expresada por el tipo genérico **IEnumerable<T>**, mientras que el segundo es un predicado (método de un argumento que retorna un **bool**) expresado mediante el tipo **Predicate<T>**. El método retorna **true** si todos los elementos de la colección evalúan el predicado a **true**, y **false** en caso contrario. Mientras tanto, **Contract.Exists** tiene exactamente los mismos parámetros, pero retorna **true** tan pronto encuentra un elemento de la colección que evalúe el predicado a **true**; el resultado será **false** si el predicado no se cumple para ningún elemento.[7]

Note, por ejemplo que el resultado de:

```
Contract.Ensures(Contract.ForAll(
  <colección>, <predicado>));
```

Es el mismo que el de:

```
Contract.Ensures(<colección>.All(
  <predicado>));
```

Del mismo modo, las dos construcciones siguientes producen el mismo resultado:

```
Contract.Ensures(Contract.Exists(
  <colección>, <predicado>));
Contract.Ensures(<colección>.Any(
  <predicado>));
```

La posibilidad de usar cuantificadores aumenta la capacidad expresiva de los contratos (vea en [6] otros ejemplos que hace un tiempo desarrollamos para el lenguaje Eiffel y adáptelos a Code Contracts). Muchos tipos relacionados con colecciones, y en especial la "maquinaria" de LINQ, se verían enriquecidos en su definición y documentación si incluyesen ahora contratos con cuantificadores.

En el listado 5 mostramos cómo podrían añadírsele dos pos-condiciones a un método como **GroupBy**. Si, por ejemplo, cambiamos la sentencia: **return Enumerable.GroupBy(source, selector);** por la instrucción: **return Enumerable.GroupBy(source, selector).Skip(1);** (note que estamos forzando que el resultado sea incorrecto al quitar uno de los grupos), violaríamos la segunda pos-condición. En ese caso, si el chequeo de las pos-condiciones está activado, la ejecución de **GroupBy** provocaría una violación de la segunda pos-condición, produciendo el mensaje que se muestra en la figura 3.

```
static class ContractedLINQMethods
{
  public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
    this IEnumerable<T> source, Func<T, K> selector)
  {
    Contract.Ensures(
      Contract.ForAll(Contract.Result<IEnumerable<IGrouping<K,T>>>(),
        x => x.All(y => selector(y).Equals(x.Key) && source.Contains(y))),
      "Todos los elementos de un grupo tienen la misma llave del grupo y " +
      "están en la colección original.");

    Contract.Ensures(Contract.Result<IEnumerable<IGrouping<K, T>>>().
      Select(x => x.Count()).Sum() == source.Count(),
      "La suma de las cantidades de cada grupo es igual a la cantidad de la " +
      "colección original");

    //...otras pos-condiciones para GroupBy

    return Enumerable.GroupBy(source, selector);
  }

  //...otros métodos de LINQ
}
```

Listado 5. Poniendo contratos a métodos de LINQ.

```
static class ContractedLINQMethods
{
  public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
    this IEnumerable<T> source, Func<T, K> selector)
  {
    //...
    Contract.Ensures(Contract.ForAll(source,
      (x => Contract.Result<IEnumerable<IGrouping<K, T>>>().
       Any(y => y.Contains(x)))),
    "Todo elemento de la colección original está en algún grupo.");
    //...
  }
  //...
}
```

Listado 6. Añadiendo una nueva pos-condición a GroupBy.

[7] Realmente el efecto de estos dos métodos puede lograrse usando los recursos de LINQ. No hay una razón clara del porqué de esta redundancia, que la propia documentación de Code Contracts menciona. Podemos especular que para crear cultura en los programadores o facilitar la implementación de alguna de las herramientas.

Sin embargo, la pos-condición del método `GroupBy` que se muestra en el listado 6 provoca el inesperado error de la figura 4, que se produce incluso antes de ejecutar nuestro método `ContractedLINQMethods.GroupBy`[8].

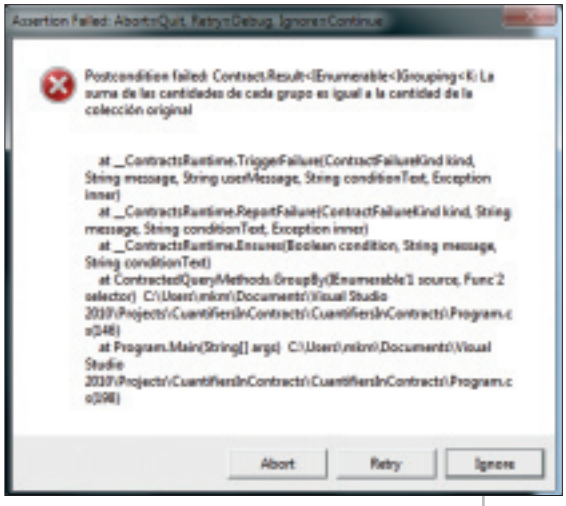

Figura 3. Mensaje por violación de pos-condición.

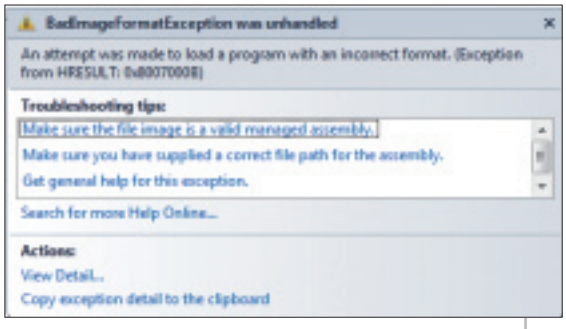

Figura 4. Bug al incluir pos-condición a GroupBy.

Contratos y la herencia

En Code Contracts los contratos se heredan. Es decir, las pre-condiciones y pos-condiciones se aplican a las respectivas redefiniciones que pueda haber de los métodos en clases derivadas. Sin embargo, para evitar conflictos asociados con el hecho de que un código no cumpla con una pre-condición, porque se esté usando un objeto de un tipo derivado en lugar de un objeto del tipo base, y la clase derivada pueda haber añadido pre-condiciones "más fuertes", Code Contracts no permite adicionar nuevas pre-condiciones a un método en una clase derivada[9].

En el caso de las pos-condiciones e invariantes, no es problema adicionarlos al heredar, pues crear pos-condiciones o invariantes "más fuertes" da más garantías al cliente de la clase.

El listado 7 muestra un ejemplo de clase `MonederoConMemoria`, que hereda de `Monedero`.

```
public class MonederoConMemoria : Monedero
{
  private List<float> trans;

  public MonederoConMemoria(float saldoMinimo)
    : base(saldoMinimo)
  {
    Contract.Requires(saldoMinimo >= 0);
    Contract.Ensures(Transacciones != null);
    Contract.Ensures(Transacciones.Count() == 1);
    Contract.Ensures(Transacciones.First() ==
                     saldoMinimo);

    trans = new List<float> { saldoMinimo };
  }

  public override void Depositar(float cant)
  {
    Contract.Ensures(Transacciones.Count() ==
      Contract.OldValue(Transacciones.Count()) + 1);
    Contract.Ensures(Transacciones.Last() == cant);

    base.Depositar(cant);
    trans.Add(cant);
  }

  public override void Extraer(float cant)
  {
    Contract.Ensures(Transacciones.Count() ==
      Contract.OldValue(Transacciones.Count()) + 1);
    Contract.Ensures(Transacciones.Last() == -cant);

    base.Extraer(cant);
    trans.Add(-cant);
  }

  public IEnumerable<float> Transacciones
  {
    get { return trans; }
  }

  [ContractInvariantMethod]
  protected void Invariante()
  {
    Contract.Invariant(Transacciones.Sum() == Saldo);
  }
}
```

Listado 7. Contratos y herencia. Clase MonederoConMemoria.

[8] Este bug persiste en el beta 2 de VS 2010. En el momento de entregar este artículo a imprenta, ya lo habíamos reportado al fórum de Code Contracts para su solución. Pero parece que ésta quedará postergada para una próxima versión. Para más información, vea *social.msdn.microsoft.com/Forums/en-IE/codecontracts*.

[9] Una propuesta más elaborada es la del lenguaje Eiffel, que permite que la clase derivada pueda poner pre-condiciones "más débiles" (que exijan menos del código cliente). Pero controlar lo que significa "más débil" implicaría una implementación más compleja e impondría una práctica de uso más disciplinada, algo que al parecer los autores de Code Contracts consideraron no se compensaba con el valor práctico real que añadir pre-condiciones pudiese aportar.

En esta nueva clase se redefinen los métodos **Depositar** y **Extraer** para además mantener una traza de todas las transacciones hechas. En ambos se hereda la pre-condición, pero se agrega una pos-condición que garantiza al cliente de la clase que la transacción se ha almacenado. Además, se ha agregado un nuevo invariante para garantizar que la suma de las transacciones realizadas es igual al saldo del monedero. Note el uso que se hace del operador **Sum** de LINQ en la especificación del contrato del invariante:

```
Contract.Invariant(
  Transacciones.Sum() == Saldo);
```

Contratos para tipos de interfaz

Una interfaz es la definición más abstracta que puede haber de un tipo. Como es sabido, lamentablemente en las interfaces de C# (y de los lenguajes .NET) solo se describen las signaturas de los métodos y propiedades y no se puede escribir código[10], por lo que el patrón que Code Contracts ha seguido hasta ahora no se puede aplicar a una interfaz, porque no se puede poner dentro de la misma código como **Contract.Requires(...)** o **Contract.Ensures(...)**, ni tampoco añadir un método para los invariantes.

La solución que ofrece Code Contracts a este problema exige una disciplina de uso. La definición de la interfaz hay que precederla del atributo:

```
[ContractClass(typeof(
 <nombre de la clase con los contratos>))]
```

Con esto se indica cuál es la clase que implementa los contratos de la interfaz. A su vez, para indicar a qué interfaz corresponde, la definición de la clase que incluye los contratos de la interfaz debe estar precedida por el atributo:

```
[ContractClassFor(typeof(
 <nombre de la interfaz>))]
```

En el listado 8 se muestra cómo poner contratos a una interfaz como **IMyList<T>**. Note que hay que replicar cada uno de los métodos o propiedades de la interfaz a los que se quiera asociar contratos. Invitamos al lector a poner contratos a toda la interfaz **IList<T>**.

El tipo definido por una clase que implemente una interfaz es un subtipo del tipo definido por la interfaz, de modo que todo lo dicho anteriormente con relación a los contratos y la herencia se aplica también para las interfaces. Es decir, la implementación de una interfaz mantiene las mismas pre-condiciones definidas para los métodos de la interfaz y no permite añadir nuevas, pero sí pueden añadirse más pos-condiciones e invariantes.

```
using System;
using System.Diagnostics.Contracts;

namespace Weboo
{
  [ContractClass(typeof(IListContracts))]
  interface IMyList<T>
  {
    void Add(T item);
    int Count {get;}
    T this[int index] { get; set; }
     Contains(T item);
    ...
  }
  [ContractClassFor(typeof(IMyList))]
  sealed class IListContracts: IMyList
  {
    public void IMyList.Add(T item)
    {
      Contract.Ensures(Contract.OldValue(((IMyList)this).Count) ==
        ((IMyList)this).Count + 1);
      Contract.Ensures(((IMyList)this).Contains(item));
    }
    public int IMyList.Count
    {
      get
      {
        Contract.Ensures(0 <= Contract.Result<int>());
      }
    }
    public T this[int index]
    {
      get
      {
        Contract.Requires(index >= 0 && index < ((IMyList)this).Count);
        Contract.Ensures(((IMyList)this).Contains(Contract.Result<T>()));
        Contract.Ensures(Contract.OldValue(((IMyList)this).Count) ==
          ((IMyList)this).Count);
      }
      set
      {
        Contract.Requires(index >= 0 && index < ((IMyList)this).Count);
        Contract.Ensures(((IMyList)this)[index] == value);
        Contract.Ensures(Contract.OldValue(((IMyList)this).Count) ==
          ((IMyList)this).Count);
      }
    }
    //...
    [ContractInvariantMethod]
    public void IMyIListInvariant()
    {
      Contract.Invariant((IMyList)this).Count >= 0);
    }
  }
}
```

Listado 8. Contratos e interfaces.

[10] Desde siempre nos hemos lamentado de la limitación de que no pueda ponerse código en un tipo de interfaz, como si hacerlo introdujese "impurezas". El verdadero dolor de cabeza para implementar la capacidad de poner código en la definición de un tipo de interfaz es que en la interfaz se definan variables de instancia. Pero debería poder ponerse código siempre que dicho código solo utilice parámetros y otros métodos y propiedades de la interfaz. Esto propiciaría la reusabilidad y evitaría lo que frecuentemente debe haberle ocurrido al lector: tener que replicar un mismo código en las diferentes implementaciones de una interfaz. De hecho, algo similar es posible en otros lenguajes de programación a través del concepto de *mixins*.

¿Chequeo estático o en tiempo de ejecución?

Code Contracts permite la verificación de los contratos tanto estáticamente como en tiempo de ejecución, lo cual podemos configurar en la pestaña correspondiente de las propiedades del proyecto (figura 1).

La verificación estática puede ser muy útil, pues nos permite detectar errores (por incumplimientos en los contratos) sin tener que ejecutar los programas. El código del listado 9 muestra el uso de un **MonederoConMemoria** al que se le hacen varios depósitos y extracciones. Note, sin embargo que el primer valor que se intenta depositar es negativo, lo que viola la pre-condición; vea en la figura 5 cómo al compilar el proyecto en Visual Studio 2010 se nos muestra dicho incumplimiento del contrato en la ventana de mensajes.

La verificación estática debe utilizarse con cuidado ya que requiere buen conocimiento de Code Contracts, y en ocasiones es necesaria nuestra intervención (utilizando algunos atributos) para que la herramienta pueda verificar cada contrato. La herramienta de verificación estática, de nombre **cccheck**, aún tiene algunas limitaciones que están descritas en la documentación y que deben tenerse en cuenta durante el desarrollo.

Durante la etapa de depuración, es recomendable la verificación de los contratos en tiempo de ejecución, de modo de poder desactivar los mismos cuando ya la aplicación esté en explotación y no afectar el rendimiento. Cuando utilizamos la verificación en tiempo de ejecución, si un contrato deja de cumplirse se disparará una excepción que muestra la condición que ha sido violada. El mismo ejemplo del listado 9, al ser ejecutado, dispara la excepción que se muestra en la figura 6.

```
using System;
namespace Weboo
{
  class Program
  {
    static void Main(string[] args)
    {
      MonederoConMemoria monedero =
        new MonederoConMemoria(200);

      float cant = 100;
      cant -= 150;
      monedero.Depositar(cant);

      monedero.Extraer(25);
      monedero.Extraer(100);
      monedero.Extraer(175);

      foreach (float v in
                monedero.Transacciones)
        Console.WriteLine(v);

      Console.ReadLine();
    }
  }
}
```

Listado 9. Uso de la clase MonederoConMemoria que viola la pre-condición de Depositar.

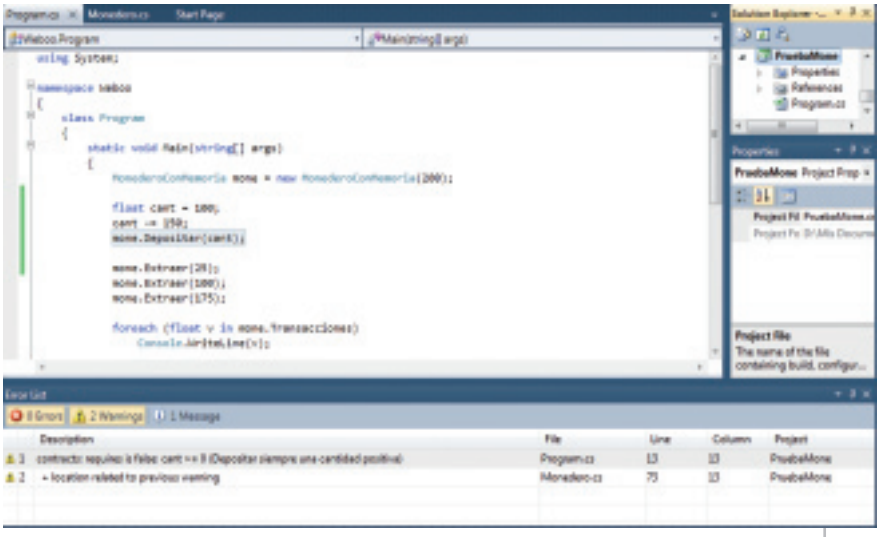

Figura 5. Chequeo estático de contratos.

En muchos casos, la verificación estática evitaría tener que crear pruebas unitarias (*unit tests*)[11]. En un próximo artículo mostraremos cómo ambas técnicas (contratos y pruebas unitarias) pueden combinarse de una manera muy efectiva. Una herramienta de nombre Pex ha sido desarrollada por parte del equipo de Code Contracts para generar pruebas unitarias a partir de contratos.

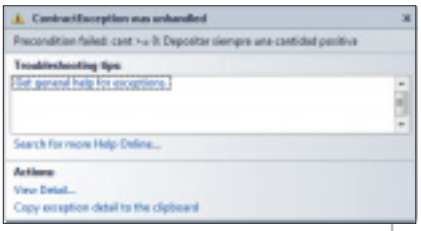

Figura 6. Mensaje con los detalles de la violación del contrato en el ejemplo del listado 9.

Documentación de los contratos de una clase

Como es conocido, C# permite la definición de comentarios de documentación con el prefijo **///** y utilizando un grupo de elementos XML predefinidos[12]. Estos comentarios son procesados por el compilador y exportados a un archivo **.xml** con el mismo nombre del ensamblado, si se incluye la opción correspondiente en las propiedades del proyecto. Este archivo es utilizado por el propio Visual Studio para mostrar ayuda contextual, y por otras herramientas como **SandCastle** [12] para generar documentación compatible con MSDN Library.

Code Contracts permite ampliar la capacidad de documentar el código incluyendo una herramienta (**ccdocgen.exe**) que extiende el archivo XML de documentación con la información relativa a los contratos. El uso de la misma está integrado en Visual Studio, y basta marcar la opción correspondiente en la pestaña de Code Contracts de las propiedades del proyecto (ver figura 1). Note que para que se incluya la documentación de los contratos en el archivo XML de documenta-

[11] Tan en boga en los últimos tiempos con el auge de las metodologías ágiles.

[12] La mayoría de los lenguajes para .NET incluyen esta facilidad, aunque la sintaxis puede variar.

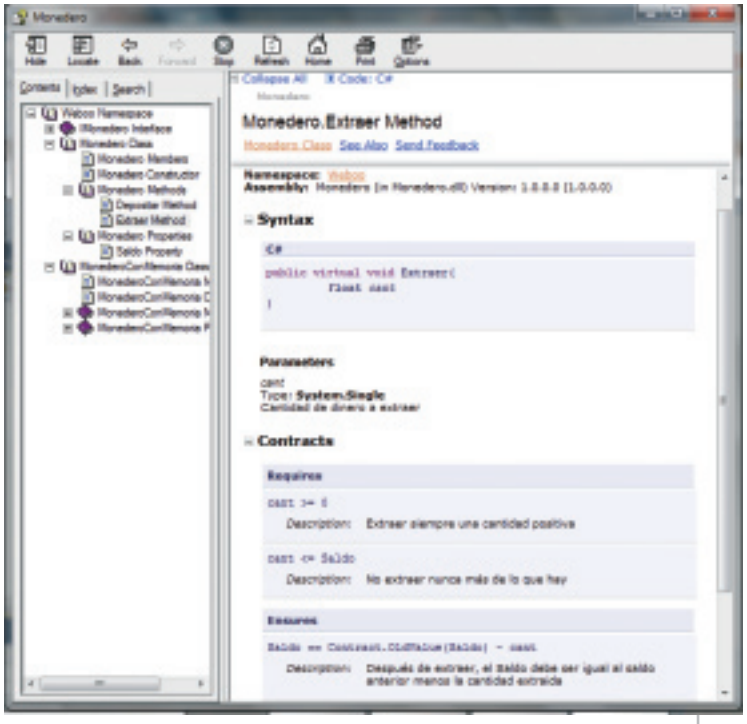

Figura 7. Archivo CHM con la documentación de las clases que utilizan en este artículo.

ción debe marcarse también la opción para generar el ensamblado de referencia de los contratos.

Lamentablemente, por ahora Visual Studio no utiliza la información de los contratos incluida en el archivo XML para mostrar ayuda contextual. Sin embargo, junto con la herramienta se incluye un parche para actualizar SandCastle, de modo que éste incluya la información de los contratos en los diferentes tipos de documentos que permite generar. En la figura 7 se muestra un archivo de ayuda CHM generado con SandCastle para las clases `Monedero` y `MonederoConMemoria` que muestra los contratos para el método `Extraer`[13].

Conclusión

Desde los inicios de .NET hemos defendido la inclusión del Diseño por Contratos en .NET (ver [13]); celebramos que finalmente Microsoft se haya decidido a incorporarlo a la plataforma.

Es posible que se requiera algún tiempo para que los desarrolladores de .NET se habitúen a las buenas prácticas de especificar su software usando contratos, pero una vez que se desarrolle este hábito se deberá apreciar un salto cualitativo considerable. Esperamos que muchas buenas prácticas y experiencias en el uso de contratos puedan trasmitirse a través de **dotNetManía**.

No obstante, la forma de utilización del Diseño por Contratos puede no ser aún la ideal. La propuesta actual requiere ser aplicada siguiendo un grupo de convenios que deben ser garantizados por los programadores, pero que no son controlados por los compiladores ni ayudados por Intellisense. Puede que futuras versiones de C#, VB.NET e incluso F#, tengan incorporados los contratos como parte de su sintaxis, facilitando con ello las buenas prácticas de su uso y la ayuda que nos puedan brindar las herramientas. Mientras tanto, en una próxima entrega de dotNetManía esperamos poder brindar una solución basada en un DSL para definir más fácilmente contratos, que luego solo haya que "asociar" a una clase .NET.

Referencias

[1] Microsoft Corporation. **Code Contracts User Manual,** septiembre de 2009.

[2] Dijkstra, E.W. **A Discipline of Programming**, Prentice Hall, 1976.

[3] Hoare, C.A.R. **An Axiomatic Basis for Computer Programming,** Communications of the ACM, Volume 12, Number 10, octubre de 1969.

[4] Liskov, B. **CLU Reference Manual**, Springer, 1983.

[5] Meyer, B. **Applying Design by Contracts,** Computer IEEE, octubre de 1992.

[6] Katrib M, Martínez I. **Collections and Iterators in Eiffel**, Journal of Object Oriented Programming, noviembre de 1993.

[7] Katrib M, Coira J. **Improving Eiffel assertions using quantified iterators**, Journal of Object Oriented Programming, noviembre de 1997.

[8] *http://prismrwiki.codegear.com/en/Main_Page*.

[9] *http://research.microsoft.com/en-us/projects/specsharp*.

[10] *http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx*.

[11] *http://blogs.msdn.com/somasegar/archive/2009/02/23/devlabs-code-contracts-for-net.aspx*.

[12] *http://sandcastle.codeplex.com*.

[13] Katrib M, Ledesma E, y Paneque L. **Including Assertions in .NET Assemblies**. .NET Developers Journal, septiembre de 2003.

[13] En el recién finalizado Tech-Ed Berlín 2009, ponentes del equipo de Pex hablaron de una futura integración de Code Contracts con Intellisense, y mostraron una posible notación dentro del editor para "visualizar" los contratos. Interrogados sobre cuándo sería realidad ese futuro, declararon sus aspiraciones de que fuese con la salida final de VS 2010.