



Mario del Valle,
Miguel Katrib

Clonación total

de objetos .NET en C# 3.0

Este artículo es un divertimento que nos muestra cómo se pueden combinar atractivos recursos de .NET y C# 3.0 como los atributos, la reflexión, la genericidad, los métodos extensores y la capacidad de consulta de LINQ para implementar un método que pueda hacer la clonación completa de objetos .NET.

Qué es clonar en el mundo de la programación y los objetos

Pudiéramos decir que clonar es hacer un duplicado de un objeto de tal modo que el duplicado o clon tenga los mismos valores y que, al menos inmediatamente después de la clonación, reaccione a su uso de la misma forma que el original.

Pero esta definición no es precisa. ¿Hasta qué nivel se lleva la tal duplicación? Si de humanos se tratase, ¿hacerle un clon a un humano significa hacerle también un clon al cónyuge, a los hijos, a la casa y a la cuenta bancaria? En un mundo de datos por valor y datos por referencia, como lo es el mundo de los objetos y como lo es en especial .NET, ¿qué es un clon? Cuando un objeto a su vez tiene una referencia a otro objeto, ¿clonar la referencia es también clonar el objeto referenciado? No hay una respuesta única; diferentes lenguajes de programación han asumido posturas diferentes y otros no se han pronunciado sobre esto.

Tal vez quizás sea Eiffel [1] el lenguaje que haya llegado más lejos en esto de la clonación. En Eiffel, la clase raíz **ANY** de la jerarquía de tipos (lo que sería el **System.Object** de .NET) tiene dos métodos: **clone** y **deepclone**. El método **clone** hace una copia superficial de todas las variables de instancia (lo que se conoce en inglés como *shallow copy*), mientras que el método **deepclone** hace una copia en profundidad (*deep copy*); es decir, si un componente del objeto original es una referencia, hace a su vez recur-

sivamente una copia en profundidad del objeto referenciado. Usted puede perspicazmente darse cuenta de que esto deja abiertas algunas interrogantes ¿Por qué siempre ir a los extremos? Es decir, ¿no puede suceder que algunos campos queramos duplicarlos en profundidad y otros no? ¿Qué pasa si en las referencias hay ciclos? Por ejemplo, si un mismo objeto **x** se refiere en más de un lugar a un segundo objeto **y** entonces ¿el clon de **x** tendrá uno o más de un clon de **y**?

En este artículo vamos a ver lo que hay actualmente en .NET sobre clonación y lo que podemos hacer para ampliar y mejorar estas capacidades.

Tipos por valor y tipos por referencia en .NET

La capacidad de definir tipos por valor o por referencia que nos ofrece .NET nos da de partida una posibilidad de lograr efectos de clonación. Considere las clases **Fecha** y **FechaPorValor** del listado 1.

Si trabajamos con **Fecha**, que ha sido definida por referencia, y hacemos:

```
Fecha nuevoAño = new Fecha(1,1,2007);  
Fecha nuevoAño1 = nuevoAño;
```

Entonces **nuevoAño** y **nuevoAño1** se estarán refiriendo al mismo objeto, como nos muestra la figura 1.

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. Es entusiasta de .NET y redactor de dotNetManía.

Mario del Valle es instructor de Programación en C# de la cátedra de Programación e Ingeniería de Software del departamento de Ciencia de la Computación de la Universidad de La Habana. Es desarrollador del grupo WEBOO dedicado a la tecnología .NET



Aprende con los expertos

Tenemos programados más de 100 cursos para todo el año 2007 sobre las siguientes tecnologías:

- SQL Server 2005
- Business intelligence en plataformas Microsoft
- ASP .NET 2.0
- .NET Framework 2.0
- .NET Framework 3.0
- Biztalk Server 2006
- Team System

Para más información visita:
www.solidqualitylearning.com

No te pierdas el III SQLU Summit Madrid

Del 11 al 15 de junio del 2007

- Una **profunda e increíble experiencia educativa** en la cual compartirás con un equipo docente compuesto por algunos de **los más reconocidos mentores, MVP y regional directors**.
- Solamente contenido de **alto nivel técnico** con una clara aplicación a sistemas reales, sin sesiones de relleno, la mejor experiencia educativa que se puede esperar de Solid Quality Learning.
- Con las **últimas tecnologías** mostradas en la conferencia **TechEd 2007 North America**.

Para más información nos puedes escribir a:
ibinfo@solidqualitylearning.com



SQL

Solid Quality Learning
Tel +34.902.999.832
www.solidqualitylearning.com

Microsoft
GOLD CERTIFIED
Partner

Advanced Infrastructure Solutions
Data Management Solutions
Learning Solutions

```
class Fecha
{
    int d,m,a;
    public Fecha(int dia, int mes, int año)
    {
        d=dia; m=mes; a=año;
    }
    public int Dia
    {
        get { return d; }
        set { d = value; }
    }
    //...
}

struct FechaPorValor
{
    int d,m,a;
    public FechaPorValor(int dia, int mes, int año)
    {
        d=dia; m=mes; a=año;
    }
    public int Dia
    {
        get { return d; }
        set { d = value; }
    }
    //...
}
```

Listado 1 Definición de Fecha por valor y por referencia

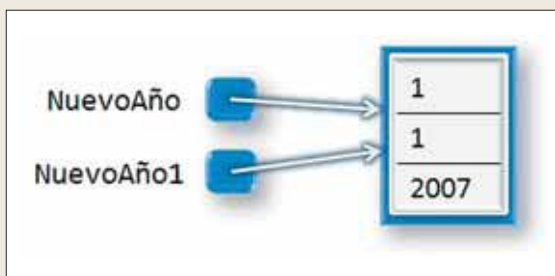


Figura 1

Pero si trabajamos con **FechaPorValor**, definida como **struct**, y hacemos:

```
FechaPorValor nuevoAño = new FechaPorValor(1,1,2007);
FechaPorValor nuevoAño1 = nuevoAño;
```

Entonces lo que estamos haciendo es copiar los valores de los componentes del objeto **nuevoAño** en los componentes respectivos del objeto **nuevoAño1**. A tal efecto, es como si en **nuevoAño1** tuviésemos un duplicado de **nuevoAño** (figura 2).



Figura 2

La clonación en .NET

Consideremos ahora el tipo **Persona** definido en el listado 2. Si hacemos:

```
Persona clint = new Persona("Clint Eastwood",
                             new FechaPorValor(31,5,1930));
Persona clintActor = clint;
```

```
class Persona
{
    string nombre;
    Fecha fechaNac;
    public Persona(string nombre, Fecha fechaNac)
    {
        this.nombre = nombre; this.fechaNac = fechaNac;
    }

    public string Nombre
    {
        get { return nombre; }
        set { nombre = value; }
    }

    public Fecha Cumpleaños
    {
        get { return fechaNacimiento; }
        set { fechaNacimiento = value; }
    }
}
```

Listado 2 Definición de clase **Persona**

Tenemos que ambas variables **clint** y **clintActor** se refieren a la misma persona.

Pero, ¿y si queremos tener un clon del mismo objeto en cada variable? .NET ofrece en la clase raíz **System.Object** el método **MemberwiseClone**, que nos hace una copia de primer nivel (superficial) de un objeto. Es decir, **MemberwiseClone** crea una nueva instancia del mismo tipo que el original y copia cada uno de los campos del objeto original en el corres-

pondiente campo de la nueva instancia. Si un campo es de un tipo por valor, copia el valor bit a bit; si es de un tipo por referencia, copia la referencia, es decir, no hace a su vez una copia del objeto referenciado; en este caso el objeto original y la copia quedarán refiriéndose al mismo objeto.

Si se hiciera por ejemplo:

```
Persona doble = clint.MemberwiseClone();
```

Suponemos que debería obtenerse el resultado que se muestra en la figura 3.

Sin embargo, en ambos casos hacer `clint.MemberwiseClone()` provoca un error de compilación, porque el método `MemberwiseClone` es `protected`.

Parece ser que en términos genéticos .NET es cuidadoso para que no hagamos clonaciones a la ligera. Es decir, .NET no nos lo pone fácil para que un código cliente haga clonaciones de un objeto. Al ser `protected`, solo desde una clase heredera podemos usar `MemberwiseClone`; de modo que si queremos aprovecharnos de la existencia de este método es quien define un tipo quien debe definir su propio método de clonación, el cual entonces es quien podrá auxiliarse de `MemberwiseClone`.

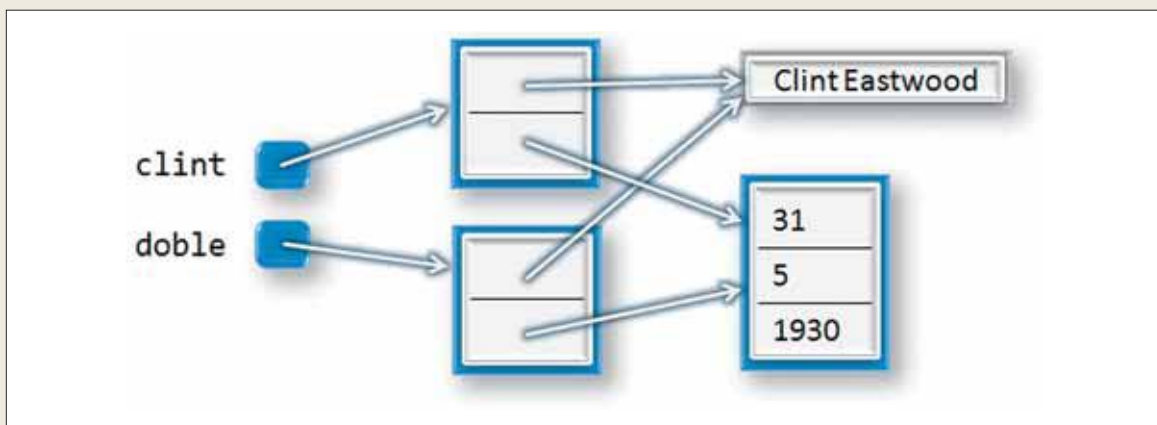


Figura 3

Pero si `Persona` estuviese definido usando el tipo `FechaPorValor` en lugar de `Fecha`, entonces al hacer:

```
Persona doble = clint.MemberwiseClone();
```

Suponemos que debería obtenerse el resultado que se muestra en la figura 4.

De este modo, debería ser la clase `Persona` la encargada de ofertar un método para hacer clones de personas, siguiendo por ejemplo un patrón como el siguiente:

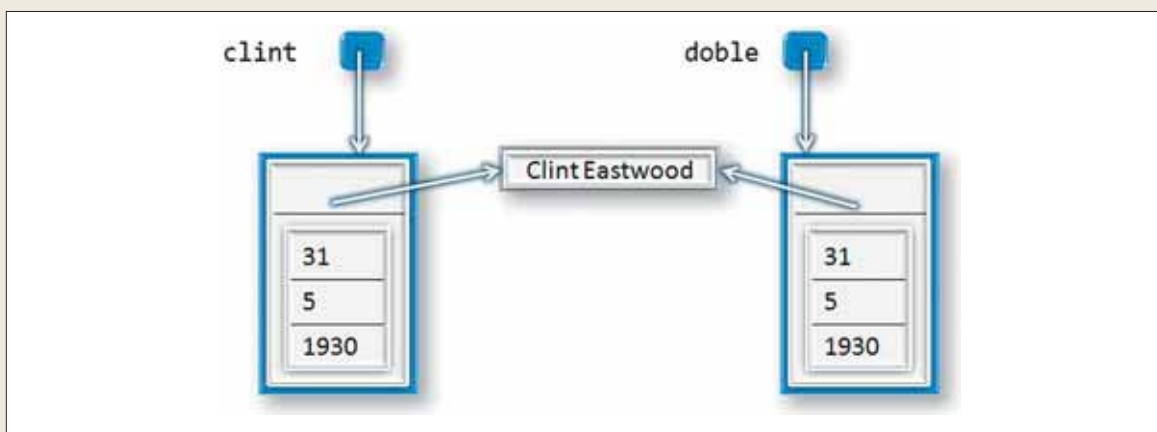


Figura 4

```
class Persona
{
    //...
    public Persona Clone()
    {
        return (Persona) MemberwiseClone();
    }
    //...
}
```

De este modo, ahora sí podemos obtener un doble haciendo:

```
Persona doble = clint.Clone();
```

La interfaz ICloneable

Con **ICloneable**, .NET nos ofrece un protocolo común que deben implementar todos aquellos tipos que deseen indicar que de ellos se puede hacer un clon. Esta interfaz tiene un único miembro, el método:

```
object Clone()
```

La clase **Persona** puede entonces definirse en la forma:

```
class Persona: ICloneable
{
    //...
    public object Clone()
    {
        //...implementación de Clone...
        //Por ejemplo return MemberwiseClone();
    }
    //...
}
```

El inconveniente aquí es que para poder ser general, el método **Clone** devuelve un objeto de tipo **object**, cuando lo que queremos es que devuelva un objeto del tipo de la clase que lo está implementando. Es por eso que es el código cliente el que debe hacer la correspondiente conversión explícita:

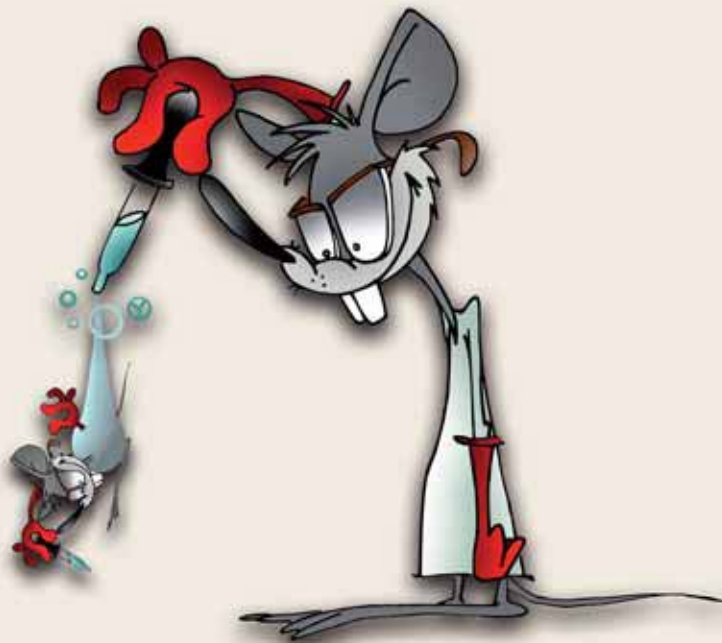
```
Persona doble = (Persona)clint.Clone();
```

Esto fuerza al código cliente a llenarse de *casts* e implica pasar a detectar en tiempo de ejecución errores que podrían detectarse tempranamente. Si se hiciese

```
Fecha f = (Fecha) clint.Clone();
```

Se produciría una excepción en ejecución, pero no error en compilación.

Veremos en la sección siguiente cómo haciendo uso de la genericidad de .NET 2.0 podemos obviar este inconveniente.



Clone con genericidad

Podemos obviar la necesidad de hacer conversiones si usamos un recurso como la **genericidad**, ya existente desde .NET 2.0 (para información sobre genericidad, ver otros trabajos de **dotNetManía** [2,3]). Podemos hacer:

```
interface ICloneable<T>
{
    T Clone();
}
```

Y entonces definir **Persona** siguiendo el patrón:

```
class Persona: ICloneable<Persona>
{
    //...
    public Persona Clone()
    {
        //...
    }
}
```

Ahora puede hacerse:

```
Persona doble = clint.Clone();
```

Sin necesidad de *cast*, y ahora entonces el intento de hacer algo como:

```
Fecha f = (Fecha) clint.Clone();
```

Darí error de compilación.

Una clonación en profundidad

¿Cómo implementar una clonación en profundidad para la clase **Persona**? Una solución podría basarse en hacer primero una copia de primer nivel usando **MemberwiseClone** y luego para cada campo que sea de tipo **ICloneable** sustituir el valor del campo por el resultado de hacerle **Clone**. Este proceso se aplicaría recursivamente para todos los campos que a su vez implementen **ICloneable**. Así por ejemplo, el **Clone** de **Persona** haría un **MemberwiseClone** y luego a su vez aplicaría **Clone** para el campo **fechaNacimiento** (listado 3).

```
class Persona: ICloneable<Persona>
{
    string nombre;
    Fecha fechaNac;
    public Persona(string nombre, Fecha fechaNac)
    {
        this.nombre = nombre; this.fechaNac = fechaNac;
    }
    //...
    public Persona Clone()
    {
        Persona clon = (Persona) MemberwiseClone();
        clon.fechaNac = fechaNac.Clone();
    }
}
```

Listado 3 Definición de **Clone** en **Persona**

Generalizar una solución como ésta tiene el inconveniente de que habría que programar un algoritmo similar por cada tipo que queramos implementar **ICloneable**.

Nos interesa obtener una implementación general del método **Clone** que sirva para cualquier tipo. Para ello nos basaremos en la genericidad, la reflexión y las nuevas capacidades de C# 3.0 y LINQ. Por otra parte, queremos que sea quien define un tipo el que decida cuáles de sus campos desea que a su vez sean clonados en profundidad. Para ello, como veremos en la sección siguiente, nos basaremos en el uso de **atributos**.

Cómo indicar qué campos de un objeto deben ser clonados

Para indicar que queremos que un campo de un objeto sea a su vez duplicado cuando se haga un duplicado de dicho objeto definimos el atributo¹ **DeepClone**, como se muestra a continuación.

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple=false)]
class DeepCloneAttribute : System.Attribute { }
```

Note que a la propia definición del atributo se le ha indicado a su vez un atributo **AttributeUsage**, con el cual especificamos que el atributo **DeepClone** solo se podrá asociar a campos (variables de instancia). De este modo, si queremos indicar que el tipo **Persona** pueda ser duplicado en profundidad duplicando a su vez el campo **fechaNacimiento**, se podría haber definido entonces en la forma que se muestra en el listado 4.

```
class Persona
{
    string nombre;
    [DeepClone]
    Fecha fechaNac;
    public Persona(string nombre, Fecha fechaNac)
    {
        this.nombre = nombre; this.fechaNac = fechaNac;
    }
    //...
}
```

Listado 4 Definición de **Persona** usando atributo **DeepClone**

Falta ahora definir un método **Clone** que tenga en cuenta este atributo.

Un método extensor como método general de clonación

Para colocar un método que sirva de clonación general, nos valdremos del recurso de los **métodos extensores** que introduce C#3.0 [5].

Definimos entonces:

```
static class DesignPatterns
{
    public static T Clone<T>(this T x)
    {
        //...
    }
}
```

La existencia de tal clase en nuestro proyecto nos permite entonces escribir:

```
Persona doble = clint.Clone();
```

Esta llamada al método **Clone** será interpretada por el compilador de C# 3.0 como:

```
Persona doble = DesignPatterns.Clone<Persona>(clint)
```

¹ Los atributos son un importante aporte de .NET del que ya se ha hablado en otros trabajos de **dotNetManía** (ver [4]).

Note que no ha habido necesidad de usar ningún *cast* ni indicar en cada tipo que implementa **ICloneable**. La definición genérica del método **Clone** nos previene de cometer errores como:

```
Fecha f = clint.Clone();
```

Porque la definición genérica del método **Clone** nos dice que devuelve un objeto del mismo tipo que el de su parámetro (**clint** en este caso).

Para hacer el clon en profundidad, lo primero que queremos hacer es una copia de primer nivel (superficial) del objeto; para ello tenemos el método **MemberwiseClone** en la clase **object**, pero no podemos aplicar éste porque no estamos dentro de un método de instancia, y no podemos hacer **x.MemberwiseClone()** porque este método es **protected**. Usando **reflexión** vamos a lograr invocar al método:

```
Type t = x.GetType();
MethodInfo method = t.GetMethod("MemberwiseClone",
    BindingFlags.NonPublic | BindingFlags.Instance);
T result = (T) method.Invoke(x, null);
```

Note cómo utilizamos el parámetro genérico **T** para hacer la conversión explícita sin que esto trascienda al cliente del método **Clone**. Cuando se llame **clint.Clone()**, el compilador resolverá que el parámetro de tipo genérico es en este caso **Persona**.

Ahora queremos clonar a su vez a aquellos campos que hemos copiado y que tienen especificado el atributo **DeepClone**. Estos campos los extraemos por reflexión usando la siguiente **sentencia LINQ**²:

```
var fields = from field in t.GetFields(
    BindingFlags.NonPublic |
    BindingFlags.Public |
    BindingFlags.Instance ),
    attrs in field.GetCustomAttributes(
        typeof(DeepCloneAttribute), false)
    where !field.FieldType.IsValueType
    select field;
```

Esto nos hace la selección de aquellos campos que son de instancia y están “marcados” con el atributo **DeepClone** y que no son de tipo por valor. A estos campos se les aplica ahora a su vez recursivamente el método **Clone**:

```
foreach (FieldInfo field in fields)
{
    field.SetValue(result, field.GetValue(x).Clone());
    //...
}
```

Para obtener una implementación general del método Clone nos basaremos en la genericidad, la reflexión y las nuevas capacidades de C# 3.0 y LINQ

Un campo puede a su vez ser un *array*. En tal caso, el **Clone** ejecutado por el segmento de código anterior aplicará el **MemberwiseClone** del *array*, el cual bastaría en el caso de que los elementos del *array* sean de un tipo por valor. Pero cuando los elementos del *array* son de un tipo por referencia, hay que aplicarle a su vez el método **Clone** a cada elemento:

```
if (field.FieldType.IsArray)
{
    if (!field.FieldType.GetElementType().IsValueType)
    {
        object[] items = (object[])field.GetValue(result);
        items = (object[]) items.Clone();
        for (int k=0; k<items.Length; k++)
            items[k] = items[k].Clone();
        field.SetValue(result, items);
    }
}
```

El código completo del método se muestra en el listado 5. Solo hemos tenido en cuenta aquí el caso de *arrays* unidimensionales; dejamos al lector interesado que complete esta implementación con toda la casuística asociada a los tipos de *arrays*.

Cómo clonar un objeto que tiene varias referencias a un mismo objeto

Cuando hacemos un clon de un objeto que tiene varias referencias a un mismo objeto, queremos que en el clon el objeto referido aparezca duplicado una sola vez; es decir, queremos que el clon mantenga la misma estructura de enlaces que el original.

² LINQ es una de las nuevas maravillas que ofrecerán .NET y C#. Puede ver más sobre las consultas de LINQ en la serie que ha venido publicando en dotNetManía el colega Octavio Hernández (ver [6]).

```
static class DesignPatterns
{
    public static T Clone<T>(this T x)
    {
        Type t = x.GetType();
        MethodInfo method = t.GetMethod("MemberwiseClone",
                                         BindingFlags.NonPublic |
                                         BindingFlags.Instance);
        T result = (T) method.Invoke(x, null);
        //Clonando en profundidad
        var fields =
            from field in t.GetFields(BindingFlags.NonPublic |
                                     BindingFlags.Public |
                                     BindingFlags.Instance),
            attrs in field.GetCustomAttributes(
                typeof(DeepCloneAttribute), false)
            where !field.FieldType.IsValueType
            select field;
        foreach (FieldInfo field in fields)
        {
            if (field.FieldType.IsArray)
            {
                object[] items = (object[])field.GetValue(result);
                items = (object[])items.Clone();
                if (!field.FieldType.GetElementType().IsValueType)
                {
                    for (int k = 0; k < items.Length; k++)
                        items[k] = Clone(items[k]);
                }
                field.SetValue(result, items);
            }
            else
            {
                field.SetValue(result, Clone(field.GetValue(x)));
            }
        }
        return result;
    }
}
```

Listado 5 Definición del método `Clone`

```
class Film
{
    string titulo;
    [DeepClone]
    Persona director;
    int duracion;
    [DeepClone]
    Persona[] actores;
    public Film(string titulo, Persona director,
               int duración, params Persona[] actores)
    {
        this.director=director;
        this.titulo=titulo;
        this.actores=actores;
    }
    //...
}
```

Listado 6 Definición de una clase `Film`

Consideremos el tipo `Film` del listado 6. Note que a las variables `director` y `actores` le hemos puesto el atributo `DeepClone` para indicar que al clonar un film los valores de estos campos deben ser a su vez clonados.

Si construimos ahora el film “Los Puentes de Madison”, en el que **Clint Eastwood** es director y a su vez actor:

```
Persona clint = new Persona("Clint Eastwood",
                           new Fecha(31, 5, 1930));
Persona meryl = new Persona("Meryl Streep",
                           new Fecha(22, 6, 1949));
Film puentesMadison =
    new Film("Los Puentes de Madison",
            clint, 200, clint, meryl);
```

Observe que el mismo objeto `clint` de tipo `Persona` aparece dos veces referido (figura 5). Si ahora hiciéramos:

```
copiapuentesMadison = puentesMadison.Clone();
```


Y aplicáramos el código del listado 4, se repetiría el clon del objeto `clint` y nos quedaría como resultado lo que nos muestra la figura 6. Esto no es exactamente una réplica del objeto `puentesMadison` que nos muestra la figura 5.

Peor aún, si un objeto incluyese una referencia a sí mismo (figura 7), ¡el intento de hacer un clon nos llevaría a un proceso infinito!

La solución para esto se muestra en el listado 7. Lo que hemos hecho es registrar cada original y su clon, para no repetir la clonación si el original apareciese repetido.

Conclusiones

Claro que lo deseable para algunos tal vez sería que .NET hubiese incluido también un método `DeepClone` en la raíz `object`. De todos modos, el objetivo fundamental de este artículo no ha sido decir la última palabra sobre cómo debe entenderse la clonación de un objeto y cuándo hacerla. Posiblemente sobre esto siempre habrá puntos de vista diferentes. Lo interesante ha sido mostrar el uso combinado de atributos, reflexión, genericidad, métodos extensores y expresiones de consulta LINQ para la solución simple de un mismo problema, lo que nos confirma una vez más el poder expresivo de .NET y C#.

Puede descargarse el código completo de la clonación en el site de [dotNetMania](http://dotNetMania.com). 

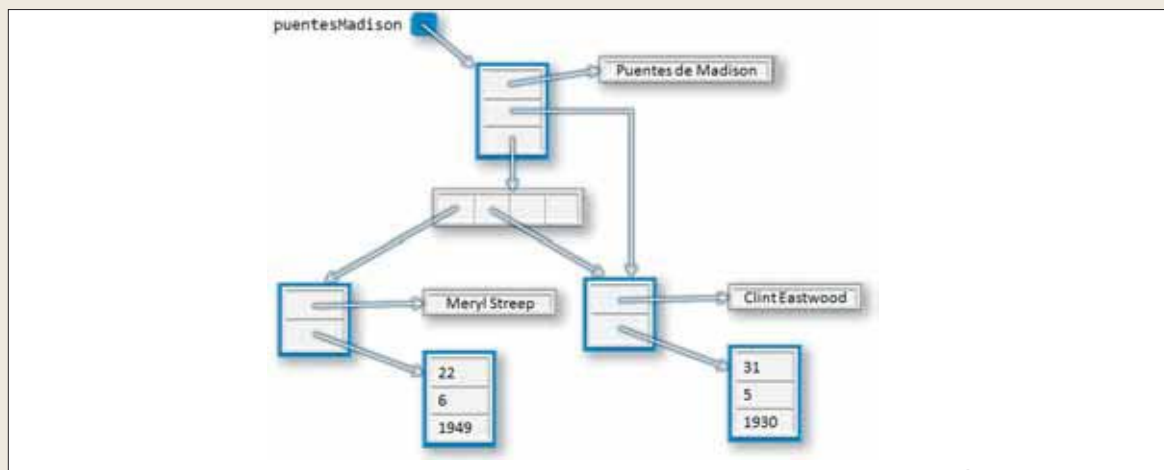


Figura 5

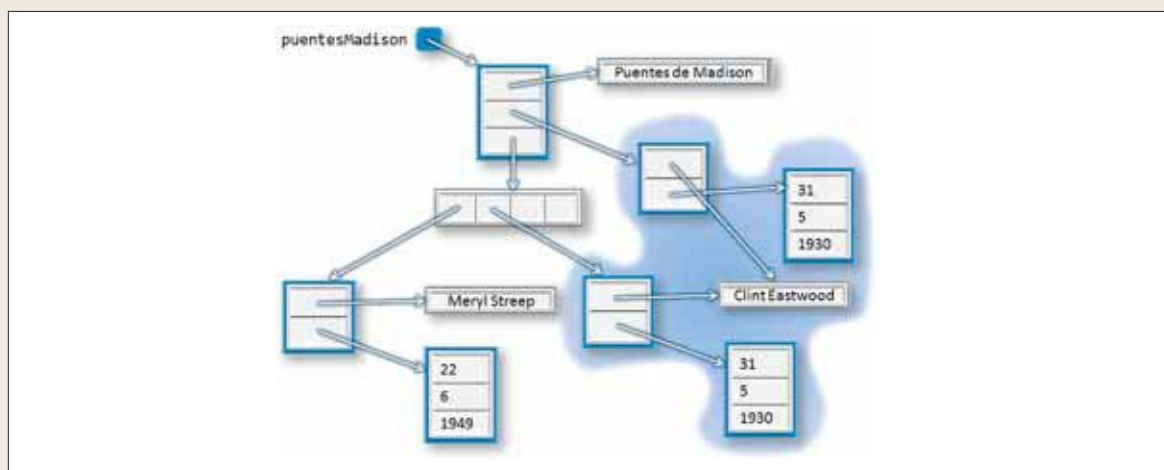


Figura 6

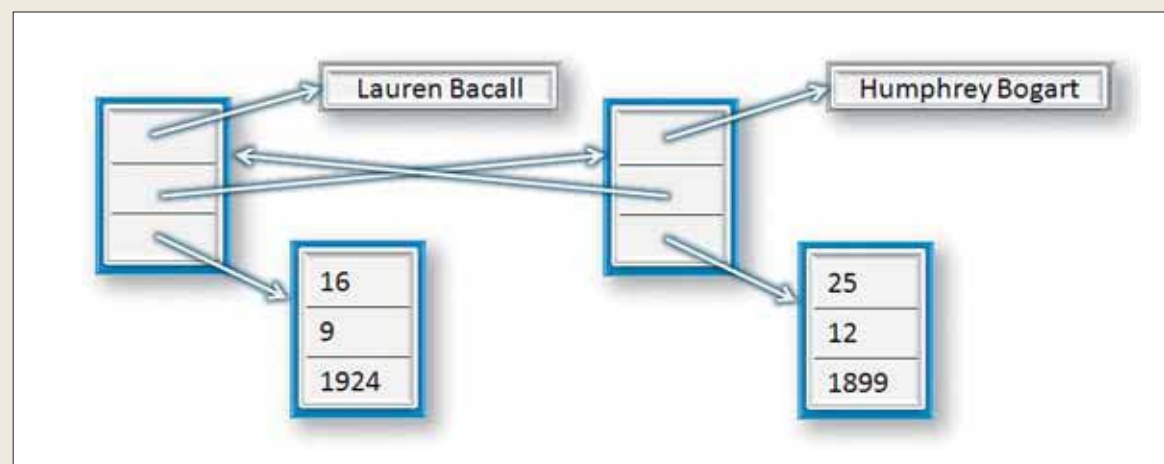


Figura 7

```

static class DesignPatterns
{
    public static T Clone<T>(this T x)
    {
        Hashtable references = new Hashtable();
        T result = TrueClone(x, references);
        references.Clear();
        return result;
    }
    private static T TrueClone<T>(this T x, Hashtable references)
    {
        if (references.Contains(x)) return (T)references[x];
        Type t = x.GetType();
        MethodInfo method = t.GetMethod("MemberwiseClone",
                                         BindingFlags.NonPublic |
                                         BindingFlags.Instance);

        T result = (T) method.Invoke(x, null);
        references.Add(x, result);
        //Clonando en profundidad
        var fields =
            from field in t.GetFields(BindingFlags.NonPublic |
                                     BindingFlags.Public |
                                     BindingFlags.Instance),
            attrs in field.GetCustomAttributes(
                typeof(DeepCloneAttribute), false)
            where !field.FieldType.IsValueType
            select field;
        foreach (FieldInfo field in fields)
        {
            if (field.FieldType.IsArray)
            {
                object[] items = (object[])field.GetValue(result);
                items = (object[])items.Clone();
                if (!field.FieldType.GetElementType().IsValueType)
                {
                    for (int k = 0; k < items.Length; k++)
                        items[k] = TrueClone(items[k], references);
                }
                field.SetValue(result, items);
            }
            else
            {
                field.SetValue(result, TrueClone(field.GetValue(x), references));
            }
        }
        return result;
    }
}

```

 Listado 7 Definición del método `Clone` para no clonar dos veces a un mismo objeto

Referencias

- [1] Meyer, Bertrand “Eiffel: The Language”, Prentice-Hall, 1992.
- [2] Som, Guillermo “Generics y Visual Basic .NET”, en **dotNetManía** N° 8, octubre de 2004
- [3] Del Valle, Mario y Katrib, Miguel “Reflexionando y haciendo reflexión sobre la genericidad en C#2.0”, en **dotNetManía** N° 20, diciembre de 2005
- [4] Hernández, Yamil y Katrib, Miguel “Aspectos e intercepción de métodos en .NET”, en **dotNetManía** N° 10, diciembre de 2004
- [5] C#3.0 Specification, <http://msdn2.microsoft.com/en-us/vcsharp/aa336745.aspx>
- [6] Hernández, Octavio “Lo que nos traerá ORCAS: la tecnología LINQ”, en **dotNetManía** N° 25, abril de 2006