Cardacci, Guillermo D.

Working Paper

Uso de Generics en Programación

Serie Documentos de Trabajo, No. 617

Provided in Cooperation with:
University of CEMA, Buenos Aires

Suggested Citation: Cardacci, Guillermo D. (2017) : Uso de Generics en Programación, Serie Documentos de Trabajo, No. 617, Universidad del Centro de Estudios Macroeconómicos de Argentina (UCEMA), Buenos Aires

This Version is available at:
http://hdl.handle.net/10419/176590

UNIVERSIDAD DEL CEMA
Buenos Aires
Argentina

Serie
DOCUMENTOS DE TRABAJO

Área: Ingeniería Informática

USO DE GENERICS EN PROGRAMACION

Darío G. Cardacci

Septiembre 2017
Nro. 617

www.cema.edu.ar/publicaciones/doc_trabajo.html
UCEMA: Av. Córdoba 374, C1054AAP Buenos Aires, Argentina
ISSN 1668-4575 (impreso), ISSN 1668-4583 (en línea)
Editor: Jorge M. Streb; asistente editorial: Valeria Dowding <jae@cema.edu.ar>

Uso de Generics en Programación

Guillermo D. Cardacci[*]

Septiembre 2017

Resumen: En la actualidad el uso de *Generics* está ampliamente difundido y le otorga a la programación un alto grado de reusabilidad y control. En particular cuando se menciona *Generics,* la reusabilidad y control se da sobre los tipos que deseamos que utilicen nuestras clases, estructuras e interfaces. El uso de genéricos y colecciones genéricas tienden a mejorar el rendimiento, debido a que suele evitar el proceso de boxing y unboxing tan costoso en términos de procesamiento. El concepto "genéricos o generics" es aplicable a clases, interfaces, métodos, delegados y parámetros. Esto implica que gran parte del código puede verse potenciado por esta característica.

Introducción.

Como se podrá observar a lo largo de este trabajo, la no utilización *Generics* implicaría tener que utilizar parámetros de un *tipo* lo suficientemente abstracto como para que se pueda enviar cualquier objeto de un *subtipo* deseado. En un caso extremo, deberíamos tomar el tipo más generalizado en la jerarquía de herencia, para lograr que cualquier subtipo pueda ser considerado y de esta forma ser enviado al parámetro definido.

Si lo antes dicho se extrapola al contexto de .Net, el parámetro debería ser de tipo *Object.* Hacer esto solucionaría el problema asociado a poder recibir cualquier objeto de cualquier tipo. No obstante, subyace otro problema al querer utilizar las funcionalidades de la interfaz, estas serán tan limitadas como las definiciones que posee el tipo más abstracto utilizado.

Como consecuencia, utilizar un parámetro del tipo *Object* llevará a que la interfaz que se observe sea la de *Object* y no la del objeto más especializado.

Por medio de algunos ejemplos se puede visualizar que utilizar *Generics* otorga una forma eficaz de mejorar la reusabilidad y la seguridad del código.

Caso A. Ejemplo con una clase no genérica.

Analicemos el siguiente ejemplo sencillo.

Tomemos una clase *Nodo* que representa los nodos de una lista *(figura 1)*.

[*] Los puntos de vista expresados en este trabajo son de exclusiva responsabilidad de su autor y no necesariamente expresan la posición de la Universidad del CEMA.

```
public class Nodo
{
    private Nodo siguentenodo;
    private string valor;
    1 referencia
    public Nodo(string pValor) { valor = pValor; }
    2 referencias
    public Nodo SiguienteNodo
    {
        get { return this.siguentenodo; }
        set { this.siguentenodo = value; }
    }
    1 referencia
    public string Valor
    {
        get { return this.valor; }
        set { this.valor = value; }
    }
}
2 referencias
```

Figura 1

También tenemos la clase *Lista,* que representa una lista simple enlazada tradicional, que se construye enlazando nodos que son instancias de la clase *Nodo (figura 2).*

```
public class Lista
{
    private Nodo primernodo = null;
    private Nodo ultimonodo = null;
    1 referencia
    public void Agregar(Nodo pNodo)
    {
        if (primernodo == null)
            { primernodo = pNodo; ultimonodo = pNodo; }
        else
            { ultimonodo.SiguienteNodo = pNodo; ultimonodo = pNodo; }
    }
    3 referencias
    public Nodo Primero() { return primernodo; }
}
```

Figura 2

Como podemos observar, en esta lista siempre se utilizarán objetos del tipo *Nodo* como elementos constitutivos. Si bien no tiene nada de malo, la realidad es que se presenta como un escenario algo restrictivo.

En términos de reusabilidad, sería muy interesante que los elementos de la lista puedan variar y no ser siempre objetos del tipo *Nodo*.

Para lograr este objetivo, debemos pensar la estructura de la lista de manera que los objetos que la componen sean genéricos. Esto implica que la lista dinámicamente se informe con que tipos de objetos va a trabajar.

Caso B. Ejemplo con una clase genérica.

Si la definición de la lista está dada por una clase, esto nos lleva a que la clase pueda recibir un tipo genérico.

Un tipo genérico es un parámetro de tipo que puede ser especificado por el usuario en el momento de generar una instancia de esa clase. A esto normalmente se lo denomina clase genérica.

```
public class Clase<T>
{

}
```

Figura 3

En la figura 3 se puede observar como la clase genérica *Clase* posee un parámetro de tipo *<T>.* Este parámetro será el encargado de recibir el tipo que se indique al momento de instanciarla. El argumento de tipo para esta clase puede ser cualquier tipo reconocido por el compilador. Se puede crear cualquier número de instancias del tipo construido, cada uno con un argumento distinto.

Trabajar de esta manera proporciona más flexibilidad y la posibilidad de reutilizar código.

Para ejemplificar lo tratado se analizará un escenario en el cual se desea construir una lista, similar la planteada en la *figura 2*. La diferencia radica en que los nodos podrán ser de cualquier tipo.

La funcionalidad básica de los nodos se establece en dos características: la primera es mostrar la información del nodo de manera personalizada y la segunda es que, estando situado en un nodo cualquiera, este posea la funcionalidad de retornar el siguiente nodo de la lista. En el caso singular que el nodo al que se le solicita el siguiente nodo sea el primero y la cantidad de nodos sea igual a uno o cuando el nodo sea el último, el valor retornado será *null (figura 4).*

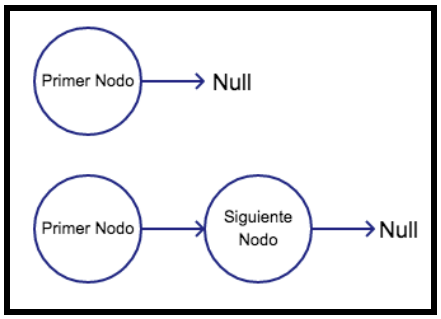

Figura 4

La manera de estandarizar estas dos funcionalidades es establecer un contrato. El mismo está dado en que cualquier clase que se desee transformar en un nodo de la lista tendrá que implementar la interfaz *Inodo*.

La interfaz *Inodo* garantizará que todo nodo tenga un método *Mostrar*, que satisface la primera característica mencionada y una propiedad *Siguiente*, que satisface la segunda. El método *Mostrar* retorna un *String* que contiene la información personalizada que el nodo es capaz de retornar. La propiedad *Siguiente,* de lectura y escritura, se utiliza para obtener el siguiente nodo de la lista o bien asignarle a un nodo su nodo siguiente o posterior.

En la figura 5 se puede observar la interfaz *Inodo*.

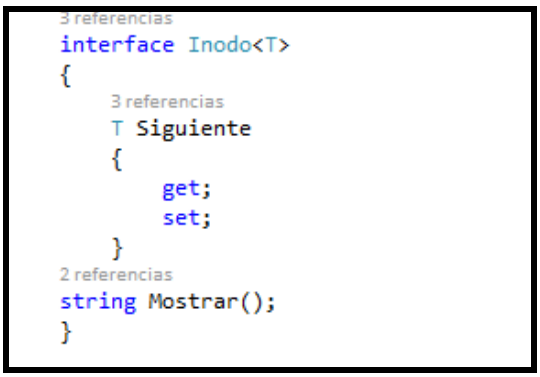

Figura 5

Como se puede apreciar la interfaz genérica *Inodo* recibe un tipo *<T>* que será indicado en el momento que se implemente en una clase o se herede a otra interfaz. Ese tipo *<T>* se utiliza como el tipo de la propiedad *Siguiente.*

Como se ha mencionado anteriormente si deseamos que una clase se comporte como nodo de una lista debe implementar *Inodo*.

En el siguiente ejemplo se desarrolla una lista de empleados. Para ello se necesita que los nodos sean empleados. Se asume que de un empleado interesa mantener su legajo, nombre y apellido.

El legajo será de tipo *int* y se administra a través de la propiedad *Legajo.* Nombre *y* apellido son de tipo *string* y las propiedades encargadas de leer y escribir estos valores son *Nombre* y *Apellido* respectivamente.

La clase *Empleado* para poder ser nodo de una lista debe implementar la interfaz *Inodo*. Esto puede observarse en la figura 6.

```
public class Empleado : Inodo<Empleado>
{
    private Empleado siguenteempleado;
    private int legajo; private string nombre; private string apellido;
    1 referencia
    public Empleado(int pLegajo, string pNombre, string pApellido)
    { legajo = pLegajo; nombre = pNombre;apellido = pApellido; }
    3 referencias
    public Empleado Siguiente
    {
        get { return this.siguenteempleado; } set { this.siguenteempleado = value; }
    }
    1 referencia
    public int Legajo
    {
        get { return this.legajo; } set { this.legajo = value; }
    }
    1 referencia
    public string Nombre
    {
        get { return this.nombre; } set { this.nombre = value; }
    }
    1 referencia
    public string Apellido
    {
        get { return this.apellido; } set { this.apellido = value; }
    }

    2 referencias
    public string Mostrar()
    {
        return "Legajo: " + this.Legajo +
               "  Nombre: " + this.Nombre +
               "  Apellido: " + this.Apellido;
    }
}
7 referencias
```

Figura 6

Repasando la clase *Empleado* se puede observar la implementación de la interfaz *Inodo*.

```
public class Empleado : Inodo<Empleado>
```

Luego la declaración de las variables privadas

```
private Empleado siguenteempleado;
private int legajo;
private string nombre;
private string apellido;
```

Un constructor permite introducir: el legajo, el nombre y el apellido al instanciar un objeto del tipo *Empleado*.

```
public Empleado(int pLegajo, string pNombre, string pApellido)
```

```
{ legajo = pLegajo; nombre = pNombre;apellido = pApellido; }
```

Las propiedades *Legajo*, *Nombre* y *Apellido,* de lectura y escritura, permiten ejecutar estas acciones sobre las variables privadas respectivas preservando el encapsulamiento.

```
public int Legajo
{
  get { return this.legajo; } set { this.legajo = value; }
}
public string Nombre
{
  get { return this.nombre; } set { this.nombre = value; }
}
public string Apellido
{
  get { return this.apellido; } set { this.apellido = value; }
}
```

Finalmente, la propiedad *Siguiente* y el método *Mostrar,* los que necesariamente deben estar implementados pues la clase *Empleado* implementa la interfaz *Inodo<Empleado>*.
Obsérvese que el tipo pasado a la interfaz *Inodo* se utiliza en el tipo de la propiedad *Siguiente*.

```
public Empleado Siguiente
        {
         get { return this.siguenteempleado; }
                set { this.siguenteempleado = value; }
}

public string Mostrar()
    {
       return "Legajo: " + this.Legajo + "\r\n" +
            "Nombre: " + this.Nombre + "\r\n" +
            "Apellido: " + this.Apellido;
    }
```

En este momento tenemos la clase *Empleado* como una clase bien formada (por implementar *Inodo<T>*) para oficiar como nodo de nuestra lista.
La clase *Nodo* al igual que empleado al implementar la interfaz Inodo<T> califica para ser nodo de la lista. Su implementación se puede observar en la figura 7.

```
public class Nodo : Inodo<Nodo>

{
    private Nodo siguentenodo;
    private string valor;
    1 referencia
    public Nodo(string pValor) { valor = pValor; }
    4 referencias
    public Nodo Siguiente
    {
        get { return this.siguentenodo; }
        set { this.siguentenodo = value; }
    }
    1 referencia
    public string Valor
    {
        get { return this.valor; }
        set { this.valor = value; }
    }

    3 referencias
    public string Mostrar()
    {
        return this.Valor;
    }
}
```

Figura 7

Ahora se analizará la clase *Lista* de la figura 8.

```
public class Lista<T> where T : Inodo<T>
{
    private T primernodo;
    private T ultimonodo;
    2 referencias
    public void Agregar(T pNodo)
    {
        if (primernodo == null)
        { primernodo = pNodo; ultimonodo = pNodo; }
        else
        {
                ultimonodo.Siguiente = pNodo;
                ultimonodo = pNodo;
        }
    }
    2 referencias
    public T Primero() { return primernodo; }
}
```

Figura 8

La clase *Lista* recibe un tipo <T> como parámetro y se restringe para que lo que llega a *<T>* sea del tipo *Inodo<T>*. Todos los elementos susceptibles a ser considerados como nodos de la lista deben cumplir con condición de ser del tipo *Inodo<T>*.

Entre las características que posee la clase *Lista<T>*, se observan dos campos privados denominados “primernodo y ultimonodo” y dos métodos públicos llamados *Agregar* y *Primero.*

Agregar, se utiliza para colocar más nodos en la lista, los cuales ingresarán a continuación del último nodo. *Primero,* retorna el primer nodo de la lista.

El tipo *<T>* que recibe la clase se utilizará para tipar los dos campos privados, el parámetro *pNodo* del método *Agregar* y el valor de retorno del método *Primero*.

Es fácilmente observable que toda la orquestación interna de la clase *Lista* se adapta en términos de tipo, al tipo *<T>* que recibe la Lista.

Un detalle a observar es cómo se maneja la implementación del método *Agregar*. Repasando su lógica se observa que recibe un objeto del tipo *<T>* en el parámetro *pNodo*.

```
public void Agregar(T pNodo)
```

La lista evalúa si su campo privado *primernodo* es *null.* En caso afirmativo el nodo que recibe *pNodo* es para esta lista el primer nodo y el último también.

```
if (primernodo == null)
{ primernodo = pNodo; ultimonodo = pNodo; }
```

En caso que la evaluación sea negativa el nodo recibido en *pNodo* deberá asignarse al *siguiente* de *ultimonodo.* Para poder utilizar la propiedad de solo lectura *Siguiente* se debe tipar *ultimonodo,* de manera que la propiedad sea accesible.

```
((Inodo<T>)ultimonodo).Siguiente = pNodo;
 ultimonodo = pNodo;
```

Luego se actualiza ultimonodo para que apunte al nodo recién ingresado.

Solo resta observar cómo se puede utilizar el objeto Lista y aprovechar su característica para recibir un parámetro *<T>*. Esto último es lo que la caracteriza como una lista genérica.

El siguiente código muestra cómo se declaran dos listas del tipo *Lista.* Una a la que se le pasa como parámetro el tipo *Empleado* y otra el tipo *Nodo*. En ambos casos es la misma lista genérica que recibe distintos tipos para adaptarse a lo solicitado.

También se instancian objetos de ambos tipos y se les asignan.

```
private Lista<Empleado> ListaEmpleado = new Lista<Empleado>();
```

O bien:

```
private Lista<Nodo> ListaPropia = new Lista<Nodo>();
```

Para agregar nodos a las listas se procede como indica el siguiente código.

```
ListaEmpleado.Agregar(
        New
        Empleado(Convert.ToInt16(textBox2.Text),textBox3.Text,textBox4.Text));
```

En el código anterior, se observa como se instancia un objeto *Empleado* y se le pasa como parámetro al método *Agregar* de la lista *ListaEmpleado*. El legajo, nombre y apellido del empleado se ingresan en las cajas de texto textBox2, textBox3 y textBox4 respectivamente, para que operen en el constructor de *Empleado*.

En el caso de la lista *ListaPropia* el método *Agregar* instancia un *Nodo* y al constructor se envía lo que el usuario ingresó en la caja de texto textBox1.

```
ListaPropia.Agregar(new Nodo(textBox1.Text));
```

Si deseamos mostrar el contenido de la lista solo resta seleccionar el destino, recorrer la lista e invocar al método mostrar para cada nodo de la lista.

Si el destino seleccionado para mostrar los elementos en un objeto *listBox,* el código de la figura 8 expone como quedaría.

```
private void Mostrar<T>(Lista<T> L) where T :Inodo<T>
{
    listBox1.Items.Clear();
    if (L.Primero() != null)
    {
        T N = L.Primero();

        do
        {
            listBox1.Items.Add(N.Mostrar());
            N = N.Siguiente;
        } while (N != null);
    }
}
```

Figura 9

Conclusión.

La utilización de clases genéricas permite que una clase pueda adaptar sus estructuras internas a un tipo particular. Esto redunda en que la misma sea flexible y que la cantidad de código resultante sea menor.

El concepto “genéricos o generics” es aplicable a clases, interfaces, métodos, delegados y parámetros. Esto implica que gran parte de nuestro código puede verse potenciado por esta característica.

El uso de genéricos y colecciones genéricas tienden a mejorar el rendimiento, debido a que suele evitar el proceso de boxing y unboxing tan costoso en términos de procesamiento.

Por otro lado, poder pasar un tipo en tiempo de ejecución permite estructuras menos rígidas. Esta posibilidad desalienta la idea de heredar un tipo para compartirlo como tipo de orden superior.

También se reduce la cantidad de código abocado al control de tipos y la seguridad. Este control se delega al compilador y no se realizará en tiempo de ejecución.

Cuando se usan parámetros de tipo genérico, estos admiten *covarianza* y *contravarianza.* Esta aptitud concede mayor flexibilidad al código desarrollado.

Si bien hay muchos puntos que alientan el uso de genéricos existen algunas limitaciones. Las enumeraciones no soportan genéricos, así como ciertos aspectos vinculados al contexto.

A pesar de ello, al desarrollar una aplicación se deberían tener en cuenta, ya que la balanza se inclina fuertemente hacia los aspectos que denotan sus ventajas.