Hada T2: Programación Dirigida por Eventos

Un nuevo paradigma de programación.

Departamento de Lenguajes y Sistemas Informáticos Universidad de Alicante

Objetivos del tema

- Conocer el paradigma de la *Programación Dirigida por Eventos*.
- Aprender a programar bajo este nuevo paradigma.
- Saber y entender cómo se puede realizar en **C#**.
- Conocer, distinguir y saber emplear los conceptos de *evento* y *callback*.

Preliminares

- En términos de la estructura y la ejecución de una aplicación representa lo opuesto a lo que hemos hecho hasta ahora: ***programación secuencial*.**
- La manera en la que escribimos el código y la forma en la que se ejecuta éste está determinada por los sucesos (***eventos***) que ocurren como consecuencia de la interacción con el mundo exterior.
- Podemos afirmar que representa un nuevo ***paradigma de programación*,** en el que todo gira alrededor de los eventos o cambios significativos en el estado de un programa.

Programación secuencial vs. dirigida por eventos I

- En la ***programación secuencial*** le decimos al usuario lo que puede hacer a continuación, desde el principio al final del programa.
- El tipo de código que escribimos es como éste:

```
repetir
   presentar_menu ();
   opc = leer_opcion ();
   ...
   si (opc == 1) entonces accion1 ();
   si (opc == 2) entonces accion2 ();
   ...
 hasta terminar
```

Programación secuencial vs. dirigida por eventos II

- En la ***programación dirigida por eventos*** indicamos:
  1. ¿Qué cosas -*eventos*- pueden ocurrir?
  2. Lo que hay que hacer cuando estos ocurran
- El tipo de código que escribimos es como éste:

```
son_eventos (ev1, ev2, ev3...);
...
cuando_ocurra ( ev1, accion1 );
cuando_ocurra ( ev2, accion2 );
repetir
  ...
hasta terminar
```

Programación secuencial vs. dirigida por eventos III

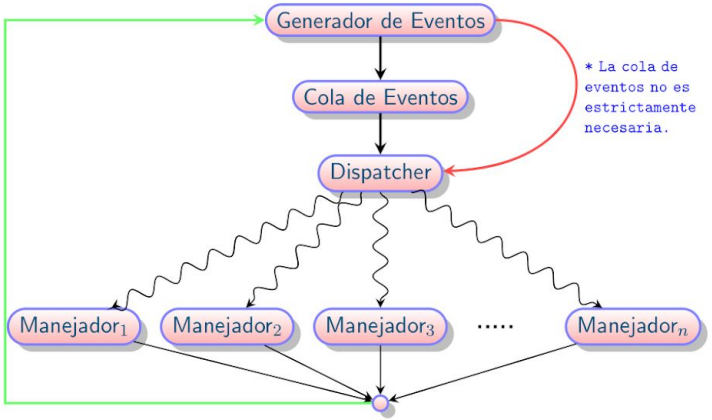
- A partir de este punto **los eventos pueden ocurrir en cualquier momento** y marcan la ejecución del programa.
- Aunque no lo parezca plantean un problema serio: **el flujo de la ejecución del programa escapa al programador.**
- El usuario (como fuente generadora de eventos) toma el control sobre la aplicación.
- Esto implica tener que llevar cuidado con el diseño de la aplicación teniendo en cuenta que **el orden de ejecución** del código no lo marca el programador y, además, **puede ser distinto cada vez**.

Esqueleto de una aplicación dirigida por eventos I

- Al principio de la misma llevamos a cabo una iniciación de todo el sistema de eventos.
- **Se definen todos los eventos que pueden ocurrir.**
- Se prepara el generador o generadores de estos eventos.
- **Se indica qué código se ejecutará en respuesta a un evento producido** *-ejecución diferida de código-*.
- Se espera a que se vayan produciendo los eventos.
- **Una vez producidos son detectados por el "*dispatcher*" o planificador de eventos**, el cual se encarga de invocar el código que previamente hemos dicho que debía ejecutarse.

Esqueleto de una aplicación dirigida por eventos II

- Todo esto se realiza de forma ininterrumpida hasta que finaliza la aplicación.
- A esta ejecución ininterrumpida es a lo que se conoce como el **bucle de espera de eventos.**
- Las aplicaciones con un interfaz gráfico de usuario siguen este esquema de programación que acabamos de comentar. *Gráficamente sería algo así*:



Analogía

- Voy a un restaurante de comida rápida *vs* saco dinero de un cajero

La diferencia clave entre estos escenarios es que en el primero el siguiente cliente puede ser servido mientras espero mi comida (me avisarán). En el segundo, quien quiera usar el cajero deberá esperar a que yo acabe (secuencial).

El principio de HollyWood I

- Las pautas de uso de la *programación dirigida por eventos* se pueden resumir con el llamado *principio de Hollywood*.
- Este es muy sencillo de entender: **No nos llame...ya le llamaremos**
- Se emplea sobre todo cuando se trabaja con *frameworks*.
- El flujo de trabajo se parece a esto:
  1. En el caso de trabajar con un *framework* implementamos un interfaz y en el caso más sencillo escribimos el código a ejecutar más adelante.
  2. Nos registramos...es decir, indicamos de algún modo cuál es el código a ejecutar posteriormente.
  3. Esperamos a que se llame -al código registrado previamente- cuando le corresponda, recuerda: ***No nos llame...ya le llamaremos***.

El principio de HollyWood II

- El programador ya no *dicta* el flujo de control de la aplicación, sino que son los eventos producidos los que lo hacen.
- Puedes consultar más información sobre él aquí.
- Si quieres ampliar más tus conocimientos sobre él debes saber que a este principio también se le conoce por otros nombres:
  1. Inversión de control: IoC.
  2. Inyección de dependencias DI.
- Puedes ampliar más información sobre estas técnicas de programación en asignaturas como Programación-3.

¿Necesitamos un lenguaje de programación especial? I

- No, La ejecución diferida de código tiene sus orígenes en el concepto de Callback.
- En Lenguaje **C** un *callback* no es más que un puntero a una función.
- En la propia biblioteca estándar de **C** hay varios ejemplos de ello.
  - Hagamos un pequeño ejercicio: en la biblioteca estándar de **C** (`#include <stdlib.h>`)

    disponemos de la función "*qsort - ordena un vector*". El prototipo de la misma es:

```
void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *));
```

- Identifica cada uno de sus parámetros. Trata de entender qué es el parámetro "*compar*". **Propón un posible valor para ese parámetro**.

Parámetros

- **base** -- Es el puntero al primer elemento del array a ordenar.
- **nmemb** -- Este es el número de elementos en el array al que apunta *base*.
- **size** -- Es el tamaño en bytes de cada elemento del array.
- **compar** -- Esta es la función que compara dos elementos.
- **callback** es un trozo de código ejecutable que se le pasa como argumento a otro código, del cual se espera que llame *call back* (execute) a la función pasada como argumento cuando sea conveniente.

¿Necesitamos un lenguaje de programación especial? II

- Cuando tengas hecho todo este estudio trata de crear un programa ejecutable mínimo que te sirva para comprobar cómo funciona `qsort`.

```
int int_cmp (const void* pe1, const void* pe2) {
      int e1 = *((const int*) pe1);
      int e2 = *((const int*) pe2);
      return e1-e2;
}

int main () {
   int v[4] = {4,0,-1,7};

   for (int i = 0; i < 4; i++)
     printf("v[%d]=%d\n", i, v[i]);

   qsort (v, 4, sizeof(int), int_cmp);
   printf("\n");
   for (int i = 0; i < 4; i++)
     printf("v[%d]=%d\n", i, v[i]);
   return 0;
}
```

Ejecución diferida

Ventaja: Podría llamar a cualquier método con la misma signatura

Delegados en C# I

- Se pueden usar delegados p.e. para pasar métodos como argumentos de otros métodos (repasa el ejemplo de *qsort* en **C** visto antes).

- En lugar de simplemente pasar el método como parámetro, primero declaramos una variable del tipo ***delegado.***

- Un delegado es un ***tipo de dato*** que representa una referencia a un método con una serie de parámetros y un tipo de resultado.

Delegados en C# II

- Podemos declarar variables de este tipo y asignarles un valor, el cual es un método cuya signatura se corresponde con la del delegado -se permite el uso de varianza en el tipo del resultado-.
- Posteriormente podemos invocar estos métodos a través del *delegado*.
- A un delegado le podemos asignar métodos de clase (static) y de instancia.
    - *Un delegado también puede contener una referencia a un objeto, así que podemos apuntar a una función/método en una instancia concreta de la clase. Un puntero a función no permite esto.*

Delegados en C#: **Declaración**

- La sintaxis para declarar un delegado es la siguiente:

```
public delegate int PerformCalculation(int x, int y);
```

- Necesitamos el espacio de nombres System.Delegate

Delegados en C# ejemplo

- Un ejemplo de uso:

1.declaración

```
delegate Persona menor (Persona p1, Persona p2);
class Persona {
  …
  //métodos que serán parámetros de una función
  public static Persona nombreMenor (Persona p1, Persona p2){...}
  public static Persona edadMenor   (Persona p1, Persona p2) {...}

  public static void muestraPrecede (Persona p1, Persona p2,
                                     menor cf) {
    Person p = cf(p1, p2);
    if (p != null) {
     Console.WriteLine ("{0} with age {1}.", p.name, p.age);
    }
  }
  // DATOS // PROPIEDADES C#
  public string nombre {get; set;}
  public int edad {get; set;}
}
```

2. invocación

```
public static void Main () {
  Persona juan = new Persona ("Juan", 20);
  Persona andres = new Persona ("Andres", 30);
  Persona.muestraPrecede  (juan, andres,
                           Persona.edadMenor );
  Persona.muestraPrecede  (juan, andres,
                           Persona.nombreMenor );
}
```

3. instanciación

Propiedades C#

- Una propiedad es un miembro que proporciona un mecanismo flexible para **leer, escribir o calcular el valor de un campo privado**.
- Las propiedades se pueden usar como si fueran miembros de datos públicos, pero en realidad son métodos especiales denominados *descriptores de acceso*.
- Esto permite acceder fácilmente a los datos a la vez que proporciona la seguridad y la flexibilidad de los métodos.

Propiedades con campos de respaldo

- Para devolver el valor de la propiedad se usa un descriptor de acceso de propiedad get, mientras que para asignar un nuevo valor se emplea un descriptor de acceso de propiedad set.
- Estos descriptores de acceso pueden tener diferentes niveles de acceso.
- La palabra clave value se usa para definir el valor que va a asignar el descriptor de acceso set.

```
using System;
class TimePeriod
{
   private double _seconds;
   public double Hours
   {
       get { return _seconds / 3600; }
       set {
          if (value < 0 || value > 24)
             throw new ArgumentOutOfRangeException(
                   $"{nameof(value)} must be between 0 and 24.");

          _seconds = value * 3600;
       }
   }}
```

```
class Program
{
   static void Main()
   {
       TimePeriod t = new TimePeriod();
       // The property assignment causes the 'set' accessor to be
called.
       t.Hours = 24;

       // Retrieving the property causes the 'get' accessor to be
called.
       Console.WriteLine($"Time in hours: {t.Hours}");
   }
}
// The example displays the following output:
//    Time in hours: 24
```

Propiedades implementadas automáticamente

- En algunos casos, los **descriptores de acceso de propiedad get y set** simplemente asignan un valor a un campo de respaldo o recuperan un valor de él sin incluir ninguna lógica adicional. Mediante las propiedades implementadas automáticamente, puede simplificar el código y conseguir que el compilador de C# le proporcione el campo de respaldo de forma transparente.
- Una propiedad implementada automáticamente se define mediante las palabras clave get y set sin proporcionar ninguna implementación.

```
using System;

public class SaleItem
{
   public string Name
   { get; set; }

   public decimal Price
   { get; set; }
}
```

```
class Program
{
   static void Main(string[] args)
   {
       var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
      Console.WriteLine($"{item.Name}: sells for
{item.Price:C2}")   }
}
// The example displays the following output:
//     Shoes: sells for $19.95
```

Funciones lambda en C#

- C # permite crear funciones anónimas (**sin nombre**) también llamadas funciones lambda.
- ***Se definen en el momento que se van a usar.***
- Podemos hacerlo de diferentes maneras:

```
// Create a delegate.
delegate void Tdel(int x);

// Instantiate the delegate using an anonymous method.
Tdel d = delegate(int k) { /* ... */ };
```

Especificamos los parámetros de entrada (si hay) a la izquierda del operador lambda => y situamos la expresión o bloques de instrucciones al otro lado.

```
delegate int Tdel(int i);
static void Main(string[] args) {
    Tdel myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}

// Num. of param. ≠ 1 → use of parenthesis
 1.   (x, y) => x == y
 2.   (int x, string s) => s.Length > x
 3.   () => SomeMethod()
```

cero parámetros de entrada

A veces, es difícil o imposible para el compilador deducir los tipos de datos. Cuando esto pasa, se pueden especificar explícitamente.

Marco de *Prog. dirigida por eventos* en C# y .Net

- El modelo empleado en C# para hacer uso de esta técnica es el siguiente:
  - Introduce un elemento nuevo, el *evento* (**event**).
  - **Los eventos representan los cambios de estado de un objeto y a ellos les podemos "*conectar*" el código que queramos que se ejecute cuando se generen (*handlers*).**

Un evento es un mecanismo de comunicación entre objetos

Marco de *Prog. dirigida por eventos* en C# y .Net

- Cada vez que se produzca un **evento** para un objeto concreto de una clase, **se ejecutarán todos los *handlers* que le hayamos asociado**, secuencialmente uno tras otro y no tiene porqué ser en el orden en que se conectaron.

Ejemplo simple

```
public class VideoEncoder

{

     public void EncodeVideo(Video video)

     {

     //encode logic

     _mailService.send(new Mail())

}}
```

si queremos añadir un nuevo método pq queremos enviar ahora un SMS, hay que recompilar todo

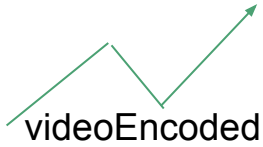
podemos usar eventos para solucionarlo

Emisor - Receptor

EMISOR DEL EVENTO

Publisher

videoEncoder class



RECEPTOR DEL EVENTO

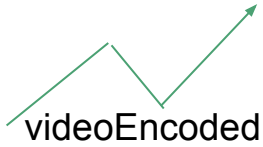
Subscriber

mailService class

Emisor - Receptor

EMISOR DEL EVENTO

Publisher



videoEncoder class

RECEPTOR DEL EVENTO

Subscriber

mailService class

RECEPTOR DEL EVENTO

Subscriber

MessageService class

Ejemplo simple

```
public class VideoEncoder

{

    public void EncodeVideo(Video video)

    {

    //encode logic

    OnVideoEncoded();

}}
```

añadimos un nuevo método que deberá **NOTIFICAR A LOS SUSCRIPTORES**

↓

enviándoles un mensaje

↓

Esto en C#: *invocando a un método de los suscriptores*

Pero cómo sabe OnVideoEncoded a qué método llamar?

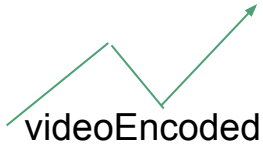
- Necesitamos un **acuerdo o contrato** entre los *publishers* (emisor) y los *subscribers* (receptor)
- ***un método con una signatura específica***
- ***Event handler***

```
public void OnVideoEncoded (object sender, EventArgs a)
```

Emisor - Receptor

EMISOR DEL EVENTO

Publisher



videoEncoder class

RECEPTOR DEL EVENTO

Subscriber

mailService class

RECEPTOR DEL EVENTO

Subscriber

MessageService class

- ***Event handler***

```
public void OnVideoEncoded
(object sender, EventArgs a)
```

Delegados

- Con un delegado le indicamos al Publisher a qué método puede llamar cuando se invoca el evento.
- Acuerdo/contrato entre el publisher (emisor) y el subscriber (receptor).

- **Determina la signatura del método manejador** (event handler) en el subscriber.

Pasos

- Definir el delegado

public delegate void VideoEncodedEventHandler(object sender, EventArgs args);

- Definir el evento basándonos en el delegado

public event VideoEncodedEventHandler videoEncoded;

- Lanzar el evento

videoEncoded(this, EventArgs.Empty);

Ejemplo simple

```
public class VideoEncoder{

public delegate void VideoEncodedEventHandler(object sender, EventArgs args);

public event VideoEncodedEventHandler videoEncoded;

     public void EncodeVideo(Video video) {//encode logic

     OnVideoEncoded();

}}

protected virtual void OnVideoEncoded()  //notify subscribers

{if videoEncoded!= null //if we have any subscriber

videoEncoded(this, EventArgs.Empty);

}
```

Vamos a crear subscribers (handlers)

```
public class MailService

{

public void OnVideoEncoded(object source,
EventArgs e)

{

Console.WriteLine("MailService: enviando email");

}
```

```
class Program{

static void Main(string[] args){

Video video= new Video();

VideoEncoder v=new VideoEncoder(); //publisher

MailService ms=new MailService(); //subscriber

v.videoEncoded+=ms.OnVideoEncoded;

//pointer to that method, we do the subscription

v.Encode(video);
```

Vamos a crear subscribers (handlers)

```
public class MessageService

{

public void OnVideoEncoded(object source,
EventArgs e)

{

Console.WriteLine("MessageService: enviando
sms");

}
```

```
class Program{

static void Main(string[] args){

Video video= new Video();

VideoEncoder v=new VideoEncoder(); //publisher

MailService ms=new MailService(); //subscriber

MessageService mg = new MessageService();

v.videoEncoded+=ms.OnVideoEncoded;

v.videoEncoded+=mg.OnVideoEncoded;

//pointer to that method, we do the subscription

v.Encode(video); }
```

Parámetros en el evento

```
public class VideoEventArgs: EventArgs

{

public Video v { get; set;}



}
```

Ejemplo simple

```
public class VideoEncoder{

public delegate void VideoEncodedEventHandler(object sender, VideoEventArgs args);

public event VideoEncodedEventHandler videoEncoded;

     public void EncodeVideo(Video video) {//encode logic

     OnVideoEncoded(video);

}}

protected virtual void OnVideoEncoded(Video video)  //notify subscribers

{if videoEncoded!= null //if we have any subscriber

videoEncoded(this, new VideoEventArgs(){Video=video;});

}
```

Modificamos los subscribers (handlers)

```
public class MessageService

{

public void OnVideoEncoded(object source,
VideoEventArgs e)

{

Console.WriteLine("MessageService: enviando
sms");

}
```

```
public class MailService

{

public void OnVideoEncoded(object source,
VideoEventArgs e)

{

Console.WriteLine("MailService: enviando email");

}
```

Otro ejemplo de eventos en C# I

- Vamos a crear una clase que representa una **persona** y otra que representa un **vehículo**.
- En la clase vehículo se crearán un número concreto de personas que viajarán en él.
- Los objetos de la clase persona generarán el *evento* **cinturonQuitado** cada vez que uno de los pasajeros del vehículo se quite el cinturón de seguridad.

Resumen clases del ejemplo

- Program.cs ➔ método main
  - Llama al constructor de *vehiculo* que genera la lista de personas y aleatoriamente “quita el cinturon” de una persona poniendo a false la propiedad cinturonQuitado.
- Vehiculo
  - “construye” la lista de personas. Conecta el manejador al evento y lo invoca. El manejador también se define aquí. (Nunca se invoca directamente al manejador, se llamará cuando se lance el evento).
- Persona
  - Aquí se define el evento.
- cinturonQuitadoArgs : EventArgs
  - *Clase especial EventArgs para guardar info sobre cambios en el cinturón.*

Ejemplo de eventos en C# II

```
public class Persona {
        public Persona (string nombre, bool cinturon)
        {
            this.nombre = nombre;
            this.cinturon = cinturon;
        }

        public string nombre { get; set; }

        private bool _cinturon;
        public bool cinturon
        {
            get { return _cinturon; }
            set {
                _cinturon = value;

                if (!_cinturon && cinturonQuitado!=null) {
                    //Lanzar evento velocidad
                    cinturonQuitado(this, new CinturonQuitadoArgs(value));
                }
            }
        }

↓
```

```
↓

        public void quitarCinturon ()
        {
            this.cinturon = false;
        }
//definición evento
 public event EventHandler
<CinturonQuitadoArgs> cinturonQuitado;


      // Fin de la clase Persona
    }

⬇
```

declaración evento

Declaración evento en C#

```
public event EventHandler <CinturonQuitadoArgs> cinturonQuitado;
```

- visibilidad + palabra clave *event*
- Delegado EventHandler + EventArgs (o tipo derivado)
- nombre del evento

Qué representa EventHandler ???
La **signatura de métodos/funciones** que pueden conectarse con ese evento + **parámetros del evento**

Delegados y eventos

- No necesitamos declarar un delegado, podemos utilizar el delegado EventHandler definido en la biblioteca de clases de C#

  ```
  delegate void EventHandler(object sender, EventArgs e);
  ```

Delegados y eventos II

- Por tanto… el prototipo de los métodos manejadores de eventos deben coincidir con el prototipo del *delegado EventHandler*.

- Por tanto, por defecto los manejadores de evento devuelven *void* y tienen dos parámetros:
  - ***Objeto que lanza el evento***
  - ***Argumentos del evento:*** *información específica del evento (EventArgs o tipo derivado)*

Ejemplo de eventos en C# III

```
public class CinturonQuitadoArgs : EventArgs
{
    public bool cinturon { get; set; }
    public CinturonQuitadoArgs (bool cinturon)
    {
        this.cinturon = cinturon;
    }
}
```

Ejemplo de eventos en C# IV

```
public class Vehiculo
{
    List<Persona> personas { get; set; }

    public Vehiculo (int numeroPersonas)
    {
        personas = new List<Persona>();
        for (int i = 0; i < numeroPersonas; i++) //se crea un cjto de personas
        {
            Persona p = new Persona("Persona" + i.ToString(), true);
            p.cinturonQuitado += cuandoCinturonQuitado; // Conexión del callback al evento!
            personas.Add(p);
        }
    }
```

conexión del evento y manejador

⬇

Ejemplo de eventos en C# V

```
    public void quitarCinturonAleatorio ()
    {
        Random rand = new Random();
        int posicion = rand.Next(0, personas.Count - 1);
        personas[posicion].quitarCinturon(); // Forzamos a que se emita el evento!
    }

    private void cuandoCinturonQuitado (object sender, CinturonQuitadoArgs args)
    {
        Persona p = (Hada.Persona) sender;
        Console.WriteLine("¡¡Alguien se ha quitado el cinturón!!");
        Console.WriteLine("Persona: " + p.nombre);
        Console.WriteLine("Cinturon: " + args.cinturon);
    }

    // Fin de la clase Vehiculo
}
```

invoca evento (llamada a quitarCinturon)

handler

Ejemplo de eventos en C# VI

```
// Main para probar el código de eventos

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Main");
        Vehiculo v = new Vehiculo(4);
        v.quitarCinturonAleatorio();  //Lanza el evento
    }
}
```

Ejemplo de eventos en C# VII

```
public class Persona {
        public Persona (string nombre, bool cinturon)
        {
            this.nombre = nombre;
            this.cinturon = cinturon;
        }

        public string nombre { get; set; }

        private bool _cinturon;
        public bool cinturon
        {
            get { return _cinturon; }
            set {
                _cinturon = value;

                if (!_cinturon) {
                    //Lanzar evento velocidad
                    cinturonQuitado(this, new CinturonQuitadoArgs(value));
                }
            }
        }
```

invoca evento (llamada a quitarCinturon)

↓

```
        public void quitarCinturon ()
        {
            this.cinturon = false;
        }
//definición evento
 public event EventHandler
<CinturonQuitadoArgs> cinturonQuitado;


      // Fin de la clase Persona
    }
```

⬇

Ejemplo de eventos en C# VIII

Recapitulando:

- **Definición evento:** Para cada evento que pueda generar una clase debemos definir una variable en la clase de tipo **EventHandler<T>**:

```
public event EventHandler<...> cinturonQuitado;
```

- **Parámetros del evento:** El tipo genérico **T** se instanciará al de la clase **EventArgs** o derivada de ella:

```
public event EventHandler<CinturonQuitadoArgs> cinturonQuitado;
```

Ejemplo de eventos en C# IX

Recapitulando:

- **Definición y asociación manejadores:**

A esta variable de tipo **evento** le conectaremos todos aquellos métodos que necesitemos, p.e.:

```
private void cuandoCinturonQuitado(object sender, CinturonQuitadoArgs e)...
p.cinturonQuitado += cuandoCinturonQuitado
```

- Lo invocaremos así: `cinturonQuitado(this, new CinturonQuitadoArgs(value));`