

Capítulo

1

DIPLOMATURA EN PROGRAMACION EN .NET
Proyecto Ventas

El Patrón de Diseño MVC (Model-View-Controller)

Patrones de Diseño

A lo largo del taller, como consultor asignado al proyecto, usted se encuentra en libertad de elegir y aplicar los patrones de diseño que le sean conveniente para el desarrollo de la aplicación. Para ayudarlo en esta tarea, se expondrán algunos de los patrones importantes para aplicarlos al diseño y puesta en marcha de la aplicación.

Definición de patrón de diseño

Tomando la definición de Gamma, un patrón de diseño es “la descripción de la comunicación de objetos y clases que son personalizadas para resolver un problema de diseño general en un contexto particular”.

En otras palabras, cuando una solución particular aparece una y otra vez en un mismo sistema o en varios, dicha solución se puede abstraer más allá del contexto para generalizarla a través de su composición y relaciones. Con esto se logra un modelo genérico que luego se pueda aplicar como una solución particular dentro del contexto de un problema específico.

La formalización del uso de patrones define tres elementos esenciales en un patrón:

- *El nombre del patrón*, el cual se utiliza para manejar y describir el patrón
- *El problema*, el cual indica cuando el patrón es aplicable
- *La solución*, la cual describe los elementos que conforman el diseño

El patrón MVC, por ejemplo, es uno de los más importantes y en él se fundamentan las aplicaciones distribuidas. También puede ser aplicado a aplicaciones de otro tipo. En nuestro caso en particular, este patrón se aplicará a la aplicación Info ya que es particularmente útil para aplicaciones con interfaces gráficas que manejan modelos de datos y tienen las siguientes características:

- La información se comparte entre muchos usuarios
- Cualquier usuario puede ver o actualizar dicha información
- La información observada por los usuarios debe estar siempre actualizada

El patrón de diseño MVC

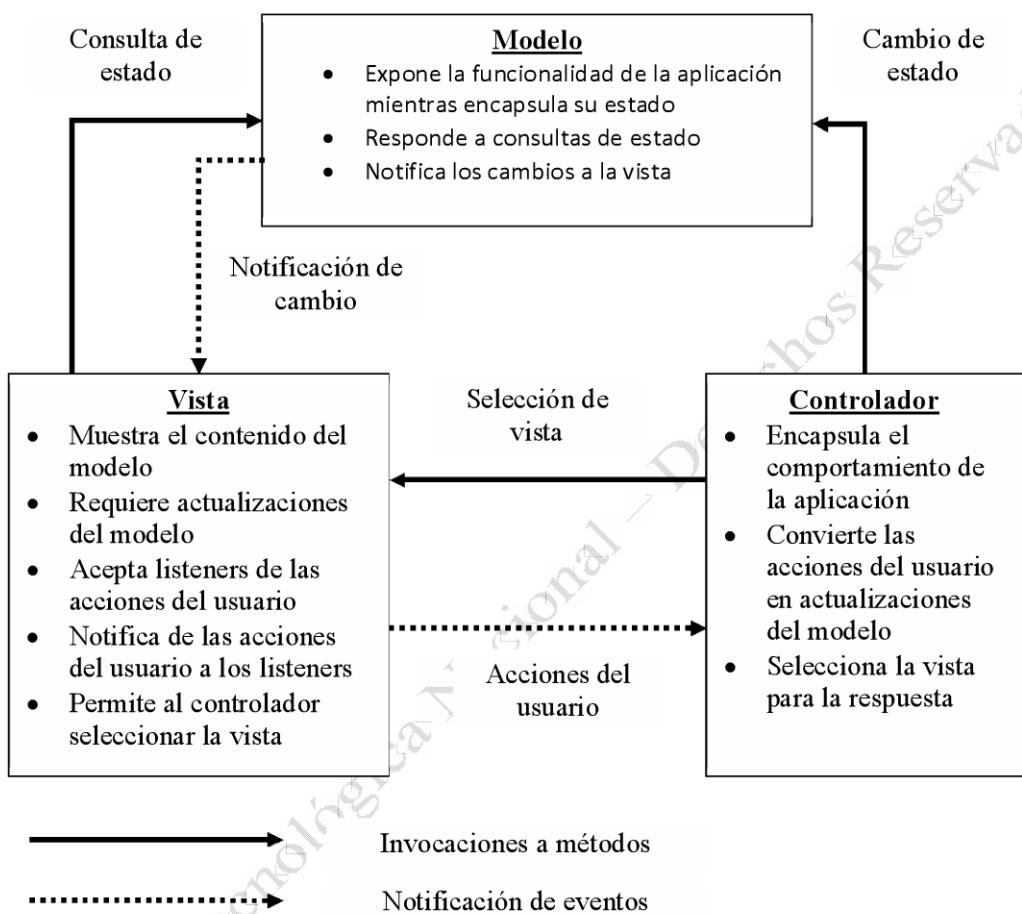
Se puede utilizar el patrón de diseño MVC para dividir la funcionalidad entre los objetos que están involucrados en mantener y presentar los datos. Esto permite minimizar el grado de acoplamiento entre los objetos.

Como el nombre lo indica, participan principalmente tres tipos de clases en el patrón de diseño:

- **Model (modelo)**: representa los datos de la aplicación y las reglas de negocio que gobiernan el acceso y modificación de esos datos.
- **View (vista)**: realiza las salidas por la interfaz gráfica de los datos que se encuentran en el modelo
- **Controller (controlador)**: define el comportamiento de la aplicación a través de interpretar y convertir las acciones del usuario en acciones realizadas por el modelo

Diplomatura en Programación .Net

El diagrama muestra las responsabilidades de cada uno de los participantes en el patrón

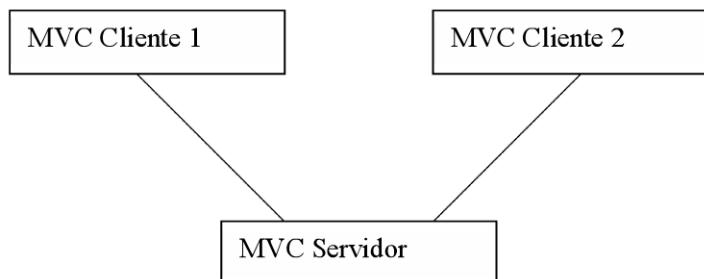


Introducción a la aplicación Info

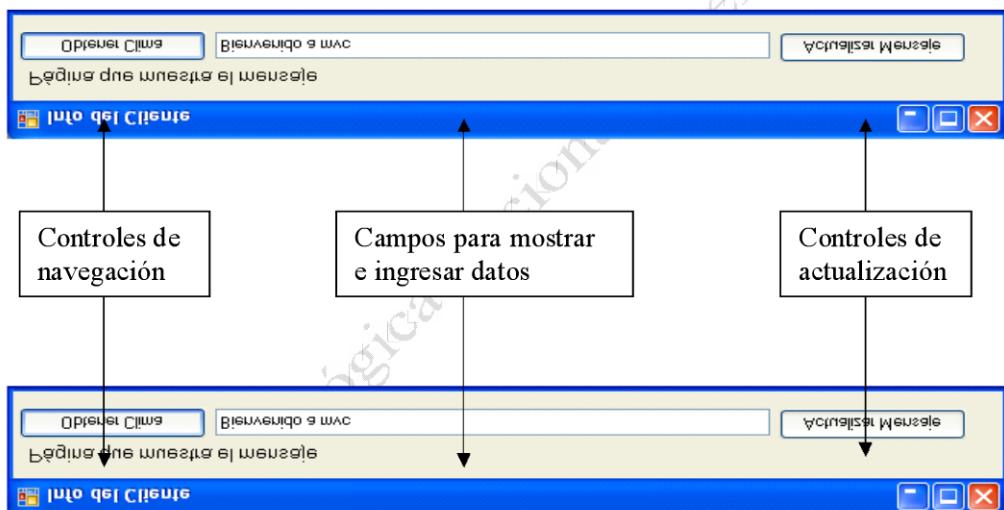
En esta sección se introduce una aplicación sencilla que sirve como ejemplo de la implementación del patrón MVC.

La aplicación Info sirve para mostrar información y actualizarla. Consiste en un único servidor con múltiples clientes conectados a él. El siguiente diagrama esquematiza el diseño

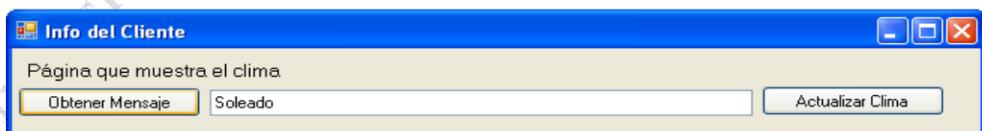
Diplomatura en Programación .Net



Cada cliente está habilitado para ver y cambiar la información presentada en varias salidas gráficas. Cada salida gráfica (a la que también se lo llama página de salida) contiene campos para mostrar datos, campos de ingreso de datos y controles de navegación de la página. En la siguiente figura se muestra como arranca la aplicación y los elementos que contiene



Cuando se presiona el botón “Obtener clima” en la salida “INFO del Cliente”, el formulario cambia al siguiente estado



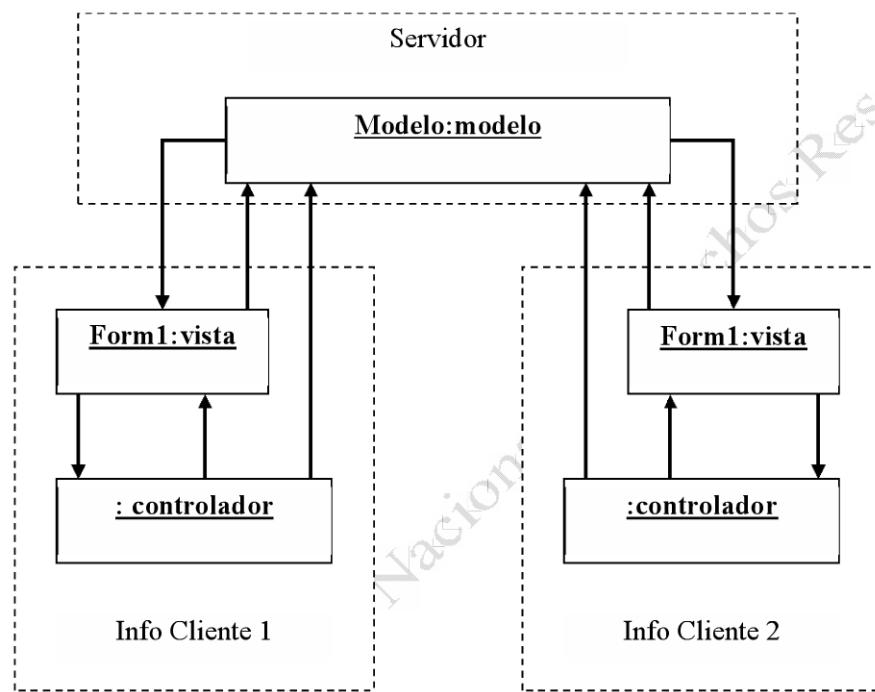
Si se presiona “Obtener mensaje”, la salida queda de la siguiente manera

Diplomatura en Programación .Net



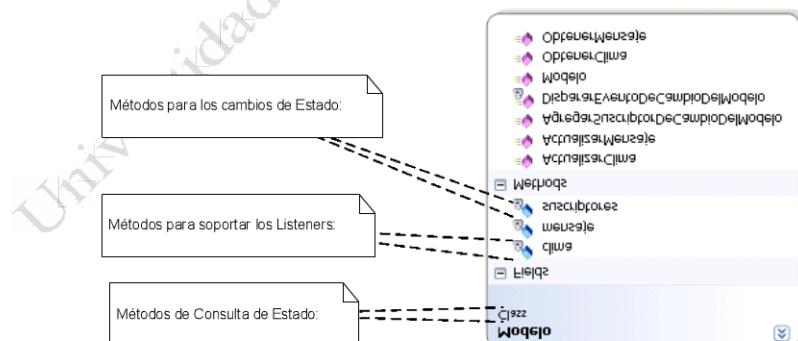
Cómo funciona el patrón MVC en la aplicación Info

Gráfico de diseño e la aplicación:



En las próximas secciones se verá el funcionamiento de cada uno de los objetos que participan en el patrón de diseño MVC.

El modelo de la aplicación Info



El modelo es responsable de:

Diplomatura en Programación .Net

- Encapsular los datos:

Esto involucra prevenir el acceso directo a los datos

C#

```
private String mensaje = "Bienvenido a mvc";
private String clima = "Soleado";
```

VB

```
Private mensaje As String = "Bienvenido a mvc"
Private clima As String = "Soleado"
```

- Proveer los métodos para consultar el estado:

Los métodos de cambio de estado retornan el estado (valores almacenados) encapsulados por el modelo. En el patrón de diseño MVC, la vista invoca estos métodos.

C#

```
public String ObtenerMensaje()
public String ObtenerClima()
```

VB

```
Public Function ObtenerMensaje() As String
Public Function ObtenerClima() As String
```

Estos métodos cambian el estado del modelo y notifican a la vista mediante el llamado al método

DispararEventoDeCambioDelModelo(Me.clima).

Estos métodos son los que se representan con la flecha que va desde el controlador al modelo en el gráfico de diseño.

- Proveer la notificación del cambio de estado a los suscriptores registrados:

Se pueden usar los métodos de cambio de estado para alterar el estado (valores de datos almacenados) encapsulado en el modelo. En el patrón de diseño MVC, la vista subscribe y recibe las notificaciones del cambio de estado.

Para implementar esta característica, el modelo provee un método público, el cual es uno de los suscriptores, que registra el cambio del estado del modelo porque “escucha” (listen) dichos cambios. El modelo puede tener uno o más métodos de notificación de cambios de estados, ya sean privado o protegidos.

El siguiente método de registro de uno de los suscriptores es invocado por la vista (en el constructor del formulario):

C#

```
modelo.AgregarSuscriptorDeCambioDelModelo(this);
```

Diplomatura en Programación .Net

VB

```
_modelo.AgregarSuscriptorDeCambioDelModelo(Me)
```

En la clase Modelo

C#

```
public void AgregarSuscriptorDeCambioDelModelo(Form1 vista)
```

VB

```
Public Sub AgregarSuscriptorDeCambioDelModelo(ById vista As Form1)
```

El siguiente método de notificación de evento de cambio en el modelo es invocado por los métodos de cambio de estado `ActualizarMensaje()` y `ActualizarClima(String clima)` (en la clase Modelo)

C#

```
private void DispararEventoDeCambioDelModelo(Object o)
```

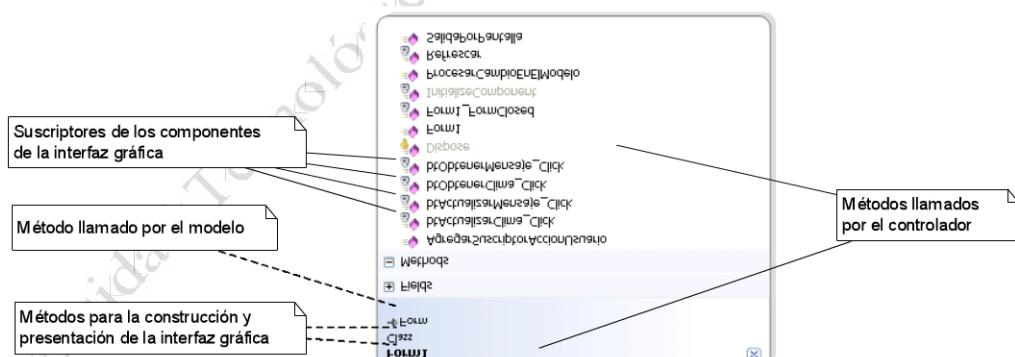
VB

```
Private Sub DispararEventoDeCambioDelModelo(ById o As Object)
```

Este método a la vez, informa a todos los objetos del tipo Form registrados en él del cambio en el modelo.

Este método es representado por la flecha punteada que va desde el modelo a la vista en el gráfico de diseño.

La vista de la aplicación Info



La vista es responsable de:

- Proveer y mantener la representación visual del estado contenido en el modelo, el cual es un proceso en dos partes que consiste en la actividades de inicio (lo que se muestra cuando arranca) y el continuo mantenimiento:

Diplomatura en Programación .Net

En el arranque el constructor invoca al método

AgregarSuscriptorDeCambioDelModelo() para registrarse para los eventos de cambio del modelo. Este obtiene el estado del modelo a través de invocar los métodos de consulta de estado del modelo (ObtenerMensaje() y ObtenerClima()) y mostrando por pantalla la página inicial.

El mantenimiento continuo de la página que se muestra en pantalla se consigue por las respuestas a las notificaciones acerca de los cambios de estado enviados por el modelo. Para recibir cambios de estado del modelo, la vista provee el siguiente método:

C#

```
public void ProcesarCambioEnElModelo(Object evento)
```

VB

```
Public Sub ProcesarCambioEnElModelo(ByVal evento As Object)
```

Este método es el que se dibuja con una flecha punteada en el gráfico de diseño que recibe el evento disparado por el modelo.

La vista obtiene de esta manera el nuevo estado del modelo por invocar los métodos de consulta de estado (ObtenerMensaje() y ObtenerClima()) actualizando por este medio la salida en pantalla.

- Hacer funcionar los requerimientos de cambio en la página que se muestra en pantalla:

En MVC, el controlador tiene la responsabilidad de la navegación de la página que se muestra por pantalla. La vista provee el siguiente método para que el controlador invoque:

C#

```
public void SalidaPorPantalla(String display)
```

VB

```
Public Sub SalidaPorPantalla(ByVal display As String)
```

Este método se representa con la flecha que va del controlador a la vista en el gráfico de diseño.

- Capturar y proveer notificaciones acerca de las acciones del usuarios con eventos:

La vista es responsable de capturar las acciones del usuario como ser el ingreso de texto o la presión de los botones. El controlador es el responsable de interpretar estas acciones. Como consecuencia, el controlador necesita ser informado de todas las acciones del usuario.

Diplomatura en Programación .Net

Para implementar esta característica, la vista provee un método para registrar los suscriptores de las acciones del usuario. La vista también provee uno o más métodos privados de notificación de acciones del usuario.

El siguiente método para registrar un suscriptor es invocado por el controlador:

C#
public void AgregarSuscriptorAccionUsuario(Controlador con)

VB
Public Sub AgregarSuscriptorAccionUsuario(ByVal con As Controlador)

Utilizando los eventos de la interfaz gráfica, la vista captura la presión de los botones e informa al controlador invocando el método apropiado provisto por el controlador.

Los eventos se agregaron en el constructor de la vista:

C#
btActualizarClima.Click += new
System.EventHandler(this.btActualizarClima_Click);

btActualizarMensaje.Click += new
System.EventHandler(this.btActualizarMensaje_Click);

btObtenerClima.Click += new
System.EventHandler(this.btObtenerClima_Click);

btObtenerMensaje.Click += new
System.EventHandler(this.btObtenerMensaje_Click);

VB
AddHandler Me.btActualizarClima.Click, AddressOf _
Me.btActualizarClima_Click
AddHandler Me.btActualizarMensaje.Click, AddressOf _
Me.btActualizarMensaje_Click
AddHandler Me.btObtenerClima.Click, AddressOf _
Me.btObtenerClima_Click
AddHandler Me.btObtenerMensaje.Click, AddressOf _
Me.btObtenerMensaje_Click

Los manejadores de eventos son los métodos:

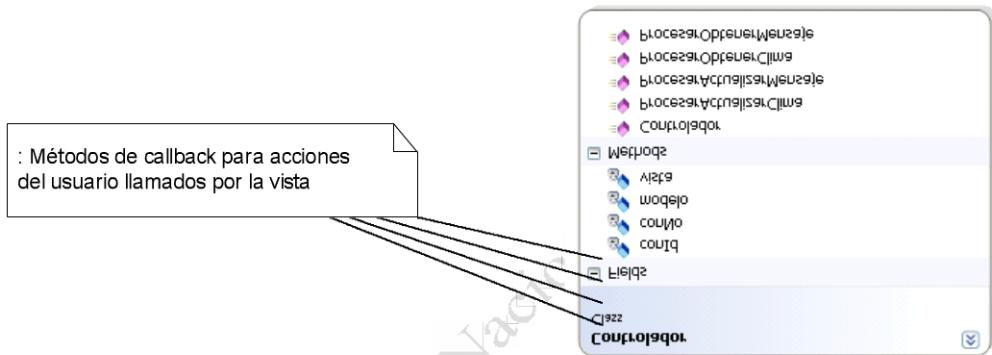
C#
private void btActualizarClima_Click(object sender,
EventArgs e)
private void btActualizarMensaje_Click(object sender,
EventArgs e)
private void btObtenerMensaje_Click(object sender,

Diplomatura en Programación .Net

```
EventArgs e)
private void btObtenerClima_Click(object sender,
EventArgs e)

VB
Private Sub btActualizarClima_Click(ByVal sender As _
Object, ByVal e As EventArgs)
Private Sub btActualizarMensaje_Click(ByVal sender As _
Object, ByVal e As EventArgs)
Private Sub btObtenerMensaje_Click(ByVal sender As _
Object, ByVal e As EventArgs)
Private Sub btObtenerClima_Click(ByVal sender As _
Object, ByVal e As EventArgs)
```

El controlador de la aplicación Info



El controlador de la aplicación Info es la responsable de interpretar las acciones del usuario capturadas por la vista. Este es un proceso en dos partes, que consiste en las actividades de inicio y el monitoreo continuo de actividades.

- En el inicio, el constructor invoca el método de la vista AgregarSuscriptorAccionUsuario() para registrar los eventos derivados de las acciones del usuario
- El monitoreo continuo de las acciones del usuario se consigue a través de las notificaciones de estas acciones enviadas por la vista. Para recibir las notificaciones de las acciones del usuario, el controlador provee los siguientes métodos:

C#

```
public void ProcesarObtenerMensaje()
public void ProcesarObtenerClima()
public void ProcesarActualizarMensaje(String mensaje)
public void ProcesarActualizarClima(String clima)
```

VB:

```
Public Sub ProcesarObtenerMensaje()
Public Sub ProcesarObtenerClima()
Public Sub ProcesarActualizarMensaje(ByVal mensaje As String)
```

Diplomatura en Programación .Net

```
Public Sub ProcesarActualizarClima(ByVal clima As String)
```

Estos métodos se representan por la flecha punteada que va desde la vista al controlador (recepción de eventos enviados por la vista) en el gráfico de diseño.

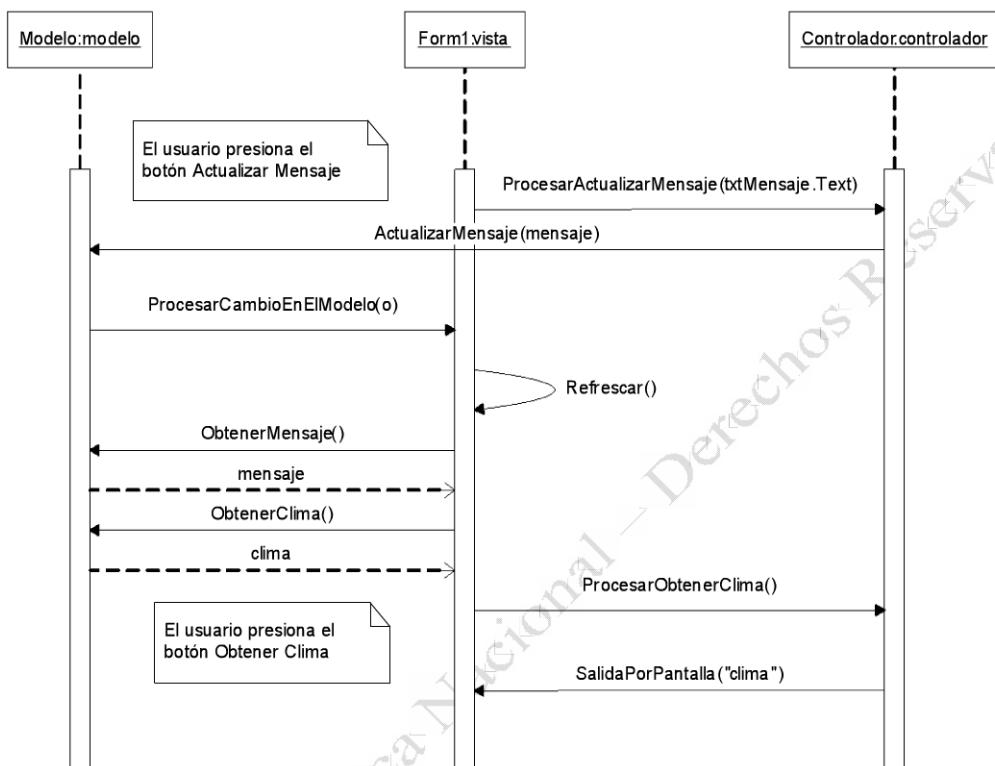
El controlador responde a las acciones del usuario con:

- La invocación del método de la vista en VB
`Public Sub SalidaPorPantalla(ByVal mostrar As String)` o el
método `public void SalidaPorPantalla(String mostrar)` para informar
a la vista que página de salida debe mostrar
- Invocando, en caso de ser aplicable, el método apropiado de cambio de estado del
modelo (`public void ActualizarMensaje(String mensaje)` o `public
void ActualizarClima(String clima)` en C#,
`ActualizarMensaje(ByVal mensaje As String)` o
`Public Sub ActualizarClima(ByVal clima As String)` en VB) para
modificar los datos encapsulados en el modelo.

Diagrama de secuencia inicial de los participantes del patrón MVC en Info

Diplomatura en Programación .Net

Diagrama de Secuencia de la interacción de los participantes del patrón MVC en la aplicación Info



Laboratorio 0 – Aplicación Info



Reconstrucción del código de la aplicación Info

Manejando Delegados y Eventos

Delegados

Un delegado en C# es similar a un puntero a función en C o C++. El uso de un delegado le permite al programador para encapsular una referencia a un método dentro de un objeto delegado. El objeto delegado puede pasarse luego al código que puede llamar al método que esté referenciando, sin tener que saber en tiempo de compilación cual será dicho método. A diferencia de los punteros a funciones en C o C++, los delegados están orientados a objetos y son tipos seguros.

Una declaración de delegado define un tipo que encapsula un método con un conjunto de argumentos y tipo retornado. Para los métodos estáticos, un objeto delegado encapsula el método a ser llamado. Para los métodos de instancia, un objeto delegado encapsula una instancia y el método de dicha instancia. Si se tiene un objeto delegado y un conjunto adecuado de argumentos, puede invocar el delegado con esos argumentos.

Una propiedad interesante y útil de un delegado es que no sabe o no se preocupa por la clase del objeto al que hace referencia. Puede ser cualquier objeto, lo único que importa es que los tipos de argumento y el tipo de valor retornado del método coincidan con el del delegado. Esto hace que los delegados ideales para realizar una invocación "anónima".

Otra propiedad importante de los delegados es la capacidad de "multicasting", es decir, componer varias referencias a manejadores en un mismo delegado. Esto se puede hacer con los operadores += o suma y asignación en C# y con el llamado `[Delegate].Combine` en VB para agregar un manejador o -= o resta y asignación en C# para quitarlo o el llamado `[Delegate].Remove` en VB.

La declaración de un delegado es la siguiente

C#
`delegate void ElDelegado(string s);`

VB
`Delegate Sub ElDelegado(ByVal s As String)`

La declaración de la variable de referencia a un delegado es la siguiente:

C#
`ElDelegado a, b, c, d;`

VB
`Dim a, b, c, d As ElDelegado`

Por otra parte, la asignación de valores a un delegado se corresponde con la declaración del mismo y la declaración del prototipo de los métodos que se le asignarán. El siguiente código ejemplifica lo enunciado

Diplomatura en Programación .Net

```
C#
namespace DelegatesComposicionCS
{
    delegate void ElDelegado(string s);

    class Program
    {
        public static void Hola(string s)
        {
            Console.WriteLine(" Hola, {0}!", s);
        }

        public static void Adios(string s)
        {
            Console.WriteLine(" Adiós, {0}!", s);
        }

        public static void Main()
        {
            ElDelegado a, b, c, d;

            // Crear el objeto a del tipo delegate que referencia
            // al método Hola:
            a = new ElDelegado(Hola);
            // Crear el objeto b del tipo delegate que referencia
            // al método Adios:
            b = new ElDelegado(Adios);
            // Los dos delegates, a y b, se componen para formar c,
            // el cual llama a ambos métodos en orden:
            c = a + b;
            // Remover a del delegate compuesto, dejando sólo b
            // en d, el cual llama al método Adios
            d = c - a;

            Console.WriteLine("Invocando al delegado a:");
            a("A");
            Console.WriteLine("Invocando al delegado b:");
            b("B");
            Console.WriteLine("Invocando al delegado c:");
            c("C");
            Console.WriteLine("Invocando al delegado d:");
            d("D");

            System.Console.ReadKey();
        }
    }
}
```

VB

```
Delegate Sub ElDelegado(ByVal s As String)

Public Sub Hola(ByVal s As String)
    Console.WriteLine(" Hola, {0}!", s)
```

Diplomatura en Programación .Net

```
End Sub

Public Sub Adios(ByVal s As String)
    Console.WriteLine(" Adiós, {0}!", s)
End Sub

Sub Main()
    Dim a, b, c, d As ElDelegado

    ' Crear el objeto a del tipo delegate que referencia
    ' al método Hola:
    a = New ElDelegado(AddressOf Hola)
    ' Crear el objeto b del tipo delegate que referencia
    ' al método Adios:
    b = New ElDelegado(AddressOf Adios)
    ' Los dos delegates, a y b, se componen para formar c,
    ' el cual llama a ambos métodos en orden:
    c = [Delegate].Combine(a, b)
    ' Remover a del delegate compuesto, dejando sólo b
    ' en d, el cual llama al método Adios

    d = [Delegate].Remove(c, a)

    Console.WriteLine("Invocando al delegado a:")
    a("A")
    Console.WriteLine("Invocando al delegado b:")
    b("B")
    Console.WriteLine("Invocando al delegado c:")
    c("C")
    Console.WriteLine("Invocando al delegado d:")
    d("D")
    System.Console.ReadKey()
End Sub
```

Eventos

Un evento en C# es el modo que tiene una clase de proporcionar notificaciones a los clientes (suscriptores) de la clase cuando ocurre un suceso que debe ser manejado en un objeto. El uso más habitual para los eventos se produce en las interfaces gráficas; normalmente, las clases que representan controles de la interfaz disponen de eventos que se notifican cuando el usuario hace algo con el control (por ejemplo, hacer clic en un botón).

Los eventos, sin embargo, no sólo se utilizan para interfaces gráficas. Los eventos proporcionan un medio apropiado para que los objetos puedan señalizar cambios de estado que pueden resultar útiles para los clientes de ese objeto. Los eventos constituyen unidades importantes para crear clases que se pueden reutilizar en diferentes programas. Sobre todo, pueden convertirse en una herramienta muy útil para señalar sucesos asíncronos. De esta manera, si una determinada operación, por ejemplo, señala la activación de determinado código que gestiona una o varias

reglas de negocio, disparar un evento puede ser una técnica eficaz para que los manejadores de dichas reglas sean invocados.

Los eventos se declaran mediante un tipo especial de delegados que no retornan valor y suelen tener, por norma, los parámetros prefijados (por ejemplo, el primer argumento es una referencia al objeto que disparó el evento y el segundo una referencia a un objeto que posee los argumentos necesarios en forma de propiedades para que el evento los opere en caso de ser necesario).

Recordar que un objeto delegado encapsula un método de modo que se pueda llamar de forma anónima. Un evento es el modo que tiene una clase de permitir a los clientes proporcionar delegados a los métodos para llamarlos cuando se produce el evento. Cuando ocurre el evento, se llama a los delegados que proporcionan los clientes para el evento.

Otra manera de declarar eventos es mediante la clase `EventHandler`. Este es un delegado predefinido que representa específicamente un método manejador de eventos para uno que no retorna datos. Si el evento genera datos, se debe proporcionar un tipo de evento propio personalizado para los datos necesarios y crear un delegado en donde el tipo del segundo parámetro es el personalizado, o utilizar la clase genérica del delegado `EventHandler<TEventArgs>` y sustituir el tipo personalizado por el parámetro de tipo genérico.

Gestión de eventos

Quizás éste sea el principal uso de los delegados. Notificar a uno o varios componentes el acontecimiento de un determinado evento con el fin de que dichos componentes tomen alguna acción al respecto. Este es el fundamento de la programación orientada a eventos (uno de los pilares de la programación visual). Utilizando eventos, los componentes de la interfaz avisan a la lógica de negocios que el usuario ha ejecutado alguna acción sobre los componentes de la misma (por ejemplo, presionar el botón del mouse o presionar una tecla).

Implementación de patrones de diseño

Existen muchos patrones de diseño basados en eventos (posiblemente el más conocido sea el patrón "Publicador-Suscriptor"). Utilizando eventos y delegados se puede establecer un mecanismo que permita que una clase interesada en los servicios de otra reciba una notificación instantánea en el momento en que acontezca un acontecimiento sobre el cual deba tomar alguna acción (por ejemplo, recibir una notificación cuando se realice una operación sobre una cuenta bancaria).

Llamadas asíncronas

La invocación a un componente relacionado puede tomar mucho tiempo en recibir respuesta. Esto sucede muchas veces cuando se manipulan grandes cantidades de datos, cuando se realiza una operación de E/S más lenta que el proceso que la maneja en memoria o cuando la lógica del servicio que se invoca es extremadamente compleja. Lo peor de esta situación es que muchas veces no se requiere la respuesta a dicha invocación para seguir trabajando, o bien, se pueden realizar otras tareas mientras se aguarda la finalización de esta. Son casos claros en los cuales no se necesita tener una acción sincronizada a su respuesta, sino más bien, cuando ocurre la respuesta se debe informar la situación al código encargado de "manejarl". Si se tiene

Diplomatura en Programación .Net

"guardada" la dirección del código a invocar, cuando ocurre situación lo único necesario es invocar al código en la dirección "guardada". En otros lenguajes como C o C++ a esto se lo denomina "puntero a función"

A estos punteros a funciones que se pasan a otro método para que éste indique su finalización, se les conoce como "funciones de retorno de llamado" (callback functions). Las funciones de retorno de llamados son el fundamento de la invocación asíncrona entre componentes (Web Services y .NET Remoting) en .NET.

Usando delegados

La sintaxis para la definición de delegados difiere según el lenguaje. No obstante, los conceptos fundamentales que se mostrarán a continuación son invariables independientemente del lenguaje que se utilice:

Lo primero que se debe hacer para definir un delegado es indicar cuál será la firma o prototipo de las funciones a la que ese delegado podrá apuntar (por firma se entiende el tipo de datos de los parámetros que se reciben, y el tipo de datos que se retorna):

C#

```
[Visibilidad] delegate [Tipo de retorno] NombreDelegado ( _  
    [Tipo] Param1, Param2 [Tipo], ...)
```

VB

```
[Visibilidad] Delegate [Sub|Function] NombreDelegado ( _  
    Param1 As [Tipo], Param2 As [Tipo], ...) {As [Tipo retornado]}
```

El segundo paso consiste en definir las variables que almacenarán los punteros de las funciones que se invocarán a través del delegado. Para mayor claridad se la llamará variable apuntador. Estas variables deben ser del tipo de datos del delegado (definido en el paso anterior):

C#

```
[Visibilidad] [Tipo Delegado] NombreApuntador
```

VB

```
[Visibilidad] NombreApuntador As [TipoDelegado]
```

Una vez definida la firma del delegado y declarado el apuntador (variable del tipo de delegado) para invocar las funciones, el tercer paso consiste en indicar cuáles serán las funciones a invocar. Para eso se utiliza el concepto de "sumar" o combinar métodos, dependiendo del lenguaje, con el apuntador (este concepto se explicó previamente):

C#

```
NombreApuntador += new TipoDelegado(NombreDeLaFuncionAAgregar)
```

VB

```
NombreApuntador = [Delegate].Combine(NombreApuntador, _  
    New TipoDelegado(AddressOf, NombreDeLaFuncionAAgregar))
```

VB ofrece una sintaxis un poco más simple en el caso del uso de delegados para manejo de eventos:

```
AddressOf NombreApuntador , NombreDeLaFuncionAAgregar
```

Ya se puede invocar al apuntador, lo cual provocará la ejecución de todas las funciones que le hayan sido agregadas. Para invocarlo simplemente basta con hacerlo por su nombre (pasándole los parámetros requeridos definidos en la firma del delegado).

Ejemplos

A continuación se mostrarán dos ejemplos del uso de delegados para la gestión de eventos e implementación del patrón "Publicador-Suscriptor".

Usando eventos

En este ejemplo sólo el paso 3 es necesario. La razón de esto es que la variable apuntador para el evento Click del botón (que de hecho recibe ese mismo nombre) ya se encuentra definida dentro de la clase System.Windows.Forms.Button provista por el framework de .NET. El delegado al que corresponde dicho apuntador (que es el mismo para los eventos de todos los controles gráficos) es de tipo System.EventHandler. Es decir, para el uso de este tipo de eventos, lo único que se debe hacer es agregar al apuntador que cada evento tiene a su manejador, o sea las funciones que serán invocadas cuando éste suceda. (Nota: algunas secciones del código han sido suprimidas para facilitar la comprensión del ejemplo, como por ejemplo el código generado por el diseñador en donde se creó un botón en el formulario con el nombre btnHolaMundo)

```
C#
public partial class frmDelegados : Form
{
    public frmDelegados()
    {
        InitializeComponent();
        //
        // btnHolaMundo
        //
        this.btnHolaMundo.Location = new System.Drawing.Point(104, 104);
        this.btnHolaMundo.Text = "Hola mundo";

        // inicio paso 3
        this.btnHolaMundo.Click += new System.EventHandler(ClickBotonHolaMundo);
        // fin paso 3
    }

    private void ClickBotonHolaMundo(object sender, EventArgs e)
    {
        MessageBox.Show(((Button)sender).Text);
    }
}
```

VB

```
Public Class frmDelegados
    Public Sub New()

        ' This call is required by the designer.
        InitializeComponent()

        ' Add any initialization after the InitializeComponent() call.
        Me.btnAdd.Location = New System.Drawing.Point(104, 104)
        Me.btnAdd.Text = "Hola mundo"

        'inicio paso 3
        AddHandler btnAdd.Click, AddressOf ClickButtonAdd
        'fin paso 3

    End Sub

    Private Sub ClickButtonAdd(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles MyBase.Click
        MessageBox.Show(CType(sender, Button).Text)
    End Sub
End Class
```

Patrón Publicador-Suscriptor

Modificando levemente el ejemplo anterior se puede mostrar este patrón de diseño antes mencionado. En el código se puede apreciar la capacidad de llamado a dos manejadores de eventos que responden al mismo suceso. A esta capacidad se la suele denominar multicasting

C#

Clase Publicador:

```
class Publicador
{
    //Paso 1: Definir la firma del evento a publicar
    public delegate void FirmaEventoAPublicar(string texto);

    //Paso 2: Definir el apuntador para guardar las direcciones
    //de las funciones a invocar
    public FirmaEventoAPublicar EventoAPublicar;

    public void OcurrioEvento(string texto)
    {
        //Paso 4: Invocar las funciones
        EventoAPublicar(texto);
    }
}
```

Clase Suscriptor:

```
class Suscriptor
{
    public void AvisemeAqui(string texto)
```

Diplomatura en Programación .Net

```
{  
    MessageBox.Show("Gracias por avisar. Este es el mensaje recibido:" +  
        texto);  
}  
}
```

Agregar e invocar los punteros a las funciones:

```
public partial class frmDelegados : Form  
{  
    private Publicador unPublicador = new Publicador();  
    private Suscriptor unSuscriptor = new Suscriptor();  
    private Suscriptor otroSuscriptor = new Suscriptor();  
  
    public frmDelegados()  
    {  
        InitializeComponent();  
        //  
        // btnHolaMundo  
        //  
        this.btnHolaMundo.Location = new System.Drawing.Point(104, 104);  
        this.btnHolaMundo.Text = "Hola mundo";  
  
        // inicio paso 3  
        this.btnHolaMundo.Click += new System.EventHandler(ClickBotonHolaMundo);  
        // fin paso 3  
    }  
    private void ClickBotonHolaMundo(object sender, EventArgs e)  
    {  
        //Paso 3: Agregar los punteros a las funciones a invocar  
        unPublicador.EventoAPublicar += new  
            Publicador.FirmaEventoAPublicar(unSuscriptor.AvisemeAqui);  
        unPublicador.EventoAPublicar += new  
            Publicador.FirmaEventoAPublicar(otroSuscriptor.AvisemeAqui);  
        // Llamar al método en el Publicador que invoca a las funciones  
        // agregadas al delegado  
        unPublicador.OcurrioEvento(((Button)sender).Text);  
    }  
}
```

VB

Clase Publicador:

```
Public Class Publicador  
    'Paso 1: Definir la firma del evento a publicar  
    Public Delegate Sub FirmaEventoAPublicar(ByVal texto As String)  
  
    'Paso 2: Definir el apuntador para guardar las funciones a invocar  
    Public EventoAPublicar As FirmaEventoAPublicar  
  
    Public Sub OcurrioEvento(ByVal texto As String)  
        'Paso 4: Invocar las funciones agregadas a través del invocador  
        EventoAPublicar(texto)
```

Diplomatura en Programación .Net

```
End Sub  
End Class
```

Clase Suscriptor:

```
Public Class Suscriptor  
    Public Sub AvisemeAqui(ByVal texto As String)  
        MessageBox.Show("Gracias por avisar. Este es el mensaje recibido:" & texto)  
    End Sub  
End Class
```

Agregar e invocar los punteros a las funciones:

```
Public Class frmDelegados  
    Private unPublicador As New Publicador  
    Private unSuscriptor As New Suscriptor  
    Private otroSuscriptor As New Suscriptor  
  
    Public Sub New()  
  
        ' This call is required by the designer.  
        InitializeComponent()  
  
        Me.btnAdd.Location = New System.Drawing.Point(104, 104)  
        Me.btnAdd.Text = "Hola mundo"  
  
        'inicio paso 3  
        AddHandler btnAdd.Click, AddressOf ClickButtonAdd  
        'fin paso 3  
  
    End Sub  
  
    Private Sub ClickButtonAdd(ByVal sender As Object, ByVal e As EventArgs)  
        'Paso 3: Agregar los punteros a las funciones a invocar  
        unPublicador.EventoAPublicar = [Delegate].Combine(unPublicador.EventoAPublicar,_  
            New Publicador.FirmaEventoAPublicar(AddressOf unSuscriptor.AvisemeAqui))  
        unPublicador.EventoAPublicar = [Delegate].Combine(unPublicador.EventoAPublicar,_  
            New Publicador.FirmaEventoAPublicar(AddressOf otroSuscriptor.AvisemeAqui))  
        'Llamar al método en el Publicador que invoca a las funciones agregadas al Delegado  
        unPublicador.OcurrioEvento(CType(sender, Button).Text)  
    End Sub  
End Class
```

Laboratorio 1 – Aplicación Info



En este laboratorio se utilizan los conceptos obtenidos en el laboratorio anterior para reemplazar los llamados a métodos por declaraciones y utilización de eventos.