
Introducción a Visual Studio .NET 2010

Introducción a la Plataforma .NET

Simplificando mucho las cosas para poder dar una definición corta y comprensible, se podría decir que la **plataforma .NET** es un *amplio conjunto de bibliotecas de desarrollo que pueden ser utilizadas por otras aplicaciones para acelerar enormemente el desarrollo y obtener de manera automática características avanzadas de seguridad, rendimiento, etc...*

En realidad .NET es mucho más que eso ya que ofrece un entorno gestionado de ejecución de aplicaciones, nuevos lenguajes de programación y compiladores, y permite el desarrollo de todo tipo de funcionalidades: desde programas de consola o servicios Windows hasta aplicaciones para dispositivos móviles, pasando por desarrollos de escritorio o para Internet.

El entorno de ejecución CLR

.NET ofrece un entorno de ejecución para sus aplicaciones conocido como **Common Language Runtime** o **CLR**. La **CLR** es la implementación de Microsoft de un estándar llamado **Common Language Infrastructure** o **CLI**. Éste fue creado y promovido por la propia Microsoft pero desde hace años es un estándar reconocido mundialmente por el ECMA.

El **CLR/CLI** esencialmente define un entorno de ejecución virtual independiente en el que trabajan las aplicaciones escritas con cualquier lenguaje .NET. Este entorno virtual se ocupa de multitud de cosas importantes para una aplicación: desde la gestión de la memoria y la vida de los objetos hasta la seguridad y la gestión de subprocesos.

Todos estos servicios unidos a su independencia respecto a arquitecturas computacionales convierten la **CLR** en una herramienta extraordinariamente útil puesto que, en teoría, cualquier aplicación escrita para funcionar según la **CLI** puede ejecutarse en cualquier tipo de arquitectura de hardware. Por ejemplo Microsoft dispone de implementación de .NET para Windows de 32 bits, Windows de 64 bits e incluso para Windows Mobile, cuyo hardware no tiene nada que ver con la arquitectura de un ordenador común.

El lenguaje intermedio y el CLS

Al contrario que otros entornos, la plataforma .NET no está atada a un determinado lenguaje de programación ni favorece a uno determinado frente a otros. En la actualidad existen implementaciones para varias decenas de lenguajes que permiten escribir aplicaciones para la plataforma .NET. Los más conocidos son **Visual Basic .NET**, **C#** o **J#**, pero existen implementaciones de todo tipo, incluso de ¡COBOL!.

Lo mejor de todo es que cualquier componente creado con uno de estos lenguajes puede ser utilizado de forma transparente desde cualquier otro lenguaje .NET. Además, como ya se ha comentado, es posible ejecutar el código .NET en diferentes plataformas y sistemas operativos.

¿Cómo se consigue esta potente capacidad?

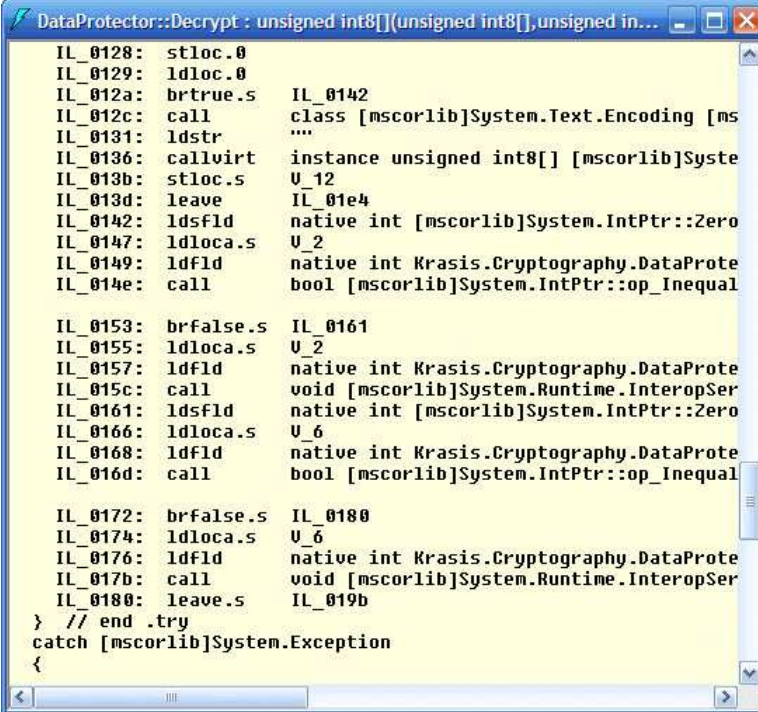
Dentro de la CLI, existe un lenguaje llamado *IL* (*Intermediate Language* o *Lenguaje Intermedio*) que está pensado de forma independiente al procesador en el que se vaya a ejecutar. Es algo parecido al código ensamblador pero de más alto nivel y creado para un hipotético procesador virtual que no está atado a una arquitectura determinada.

Cuando se compila una aplicación escrita en un lenguaje .NET cualquiera (da igual que sea VB, C# u otro de los soportados), el compilador lo que genera en realidad es un nuevo código escrito en este lenguaje intermedio. Así, todos los lenguajes .NET se usan como capa de más alto nivel para producir código IL.

Un elemento fundamental de la CLR es el compilador JIT (*just-in-time*). Su cometido es el de compilar bajo demanda y de manera transparente el código escrito en lenguaje intermedio a lenguaje nativo del procesador físico que va a ejecutar el código.

Al final, lo que se ejecuta es código nativo que ofrece un elevado rendimiento. Esto es cierto también para las aplicaciones Web escritas con ASP.NET y contrasta con las aplicaciones basadas en ASP clásico que eran interpretadas, no compiladas, y que jamás podrían llegar al nivel de desempeño que ofrece ASP.NET.

La siguiente figura muestra el aspecto que tiene el código intermedio de una aplicación sencilla y se puede obtener usando el desensamblador que viene con la plataforma .NET.



```
IL_0128: stloc.0
IL_0129: ldloc.0
IL_012a: brtrue.s IL_0142
IL_012c: call class [mscorlib]System.Text.Encoding [ms
IL_0131: ldstr
IL_0136: callvirt instance unsigned int8[] [mscorlib]Syste
IL_013b: stloc.s U_12
IL_013d: leave IL_01e4
IL_0142: ldsfld native int [mscorlib]System.IntPtr::Zero
IL_0147: ldloca.s U_2
IL_0149: ldfld native int Krasis.Cryptography.DataProte
IL_014e: call bool [mscorlib]System.IntPtr::op_Inequal

IL_0153: brfalse.s IL_0161
IL_0155: ldloca.s U_2
IL_0157: ldfld native int Krasis.Cryptography.DataProte
IL_015c: call void [mscorlib]System.Runtime.InteropServices
IL_0161: ldsfld native int [mscorlib]System.IntPtr::Zero
IL_0166: ldloca.s U_6
IL_0168: ldfld native int Krasis.Cryptography.DataProte
IL_016d: call bool [mscorlib]System.IntPtr::op_Inequal

IL_0172: brfalse.s IL_0180
IL_0174: ldloca.s U_6
IL_0176: ldfld native int Krasis.Cryptography.DataProte
IL_017b: call void [mscorlib]System.Runtime.InteropServices
IL_0180: leave.s IL_019b
} // end .try
catch [mscorlib]System.Exception
{
```

Figura 1.1. Código en lenguaje intermedio obtenido con ILDASM.exe

La especificación común de los lenguajes y el sistema de tipos comunes

Para conseguir la interoperabilidad entre lenguajes no sólo llega con el lenguaje intermedio, sino que es necesario disponer de unas "reglas del juego" que definan un conjunto de características que todos los lenguajes deben incorporar. A este conjunto regulador se le denomina *Common Language Specification (CLS)* o, en castellano, especificación común de los lenguajes.

Entre las cuestiones que regula la CLS se encuentran la nomenclatura, la forma de definir los miembros de los objetos, los metadatos de las aplicaciones, etc... Una de las partes más importantes de la CLS es la que se refiere a los tipos de datos.

Si alguna vez ha programado la API de Windows o ha tratado de llamar a una DLL escrita en C++ desde Visual Basic 6 habrá comprobado lo diferentes que son los tipos de datos de VB6 y de C++. Para evitar este tipo de problemas y poder gestionar de forma eficiente y segura el acceso a la memoria, la CLS define un conjunto de tipos de datos comunes (*Common Type System* o **CTS**) que indica qué tipos de datos se pueden manejar, cómo se declaran y se utilizan éstos y de qué manera se deben gestionar durante la ejecución.

Si nuestras bibliotecas de código utilizan en sus interfaces hacia el exterior datos definidos dentro de la CTS no existirán problemas a la hora de utilizarlos desde cualquier otro código escrito en la plataforma .NET.

Cada lenguaje .NET utiliza una sintaxis diferente para cada tipo de datos. Así, por ejemplo, el tipo común correspondiente a un número entero de 32 bits (***System.Int32***) se denomina ***Integer*** en Visual Basic .NET, pero se llama ***int*** en C#. En ambos casos representan el mismo tipo de datos que es lo que cuenta.

Nota:

En ASP 3.0 se suele usar VBScript como lenguaje de programación. En este lenguaje interpretado, al igual que en VB6, un *Integer* representaba un entero de 16 bits. Los enteros de 32 bits eran de tipo *Long*. Es un fallo muy común usar desde Visual Basic .NET el tipo *Integer* pensando que es de 16 bits cuando en realidad es capaz de albergar números mucho mayores. Téngalo en cuenta cuando empiece a programar.

Existen tipos por valor (como los enteros que hemos mencionado o las enumeraciones) y tipos por referencia (como las clases o los arrays). Más adelante veremos que significa esto.

La biblioteca de clases .NET

Todo lo que se ha comentado hasta ahora constituye la base de la plataforma .NET. Si bien es muy interesante y fundamental, por sí mismo no nos serviría de mucho para crear programas si debiésemos crear toda la funcionalidad desde cero.

Obviamente esto no es así, y la plataforma .NET nos ofrece infinidad de funcionalidades "de fábrica" que se utilizan como punto de partida para crear las aplicaciones. Existen funcionalidades básicas (por ejemplo todo lo relacionado con la E/S de datos o la seguridad) y funcionalidades avanzadas en las que se fundamentan categorías enteras de aplicaciones (acceso a datos, creación de aplicaciones Web...).

Toda esta funcionalidad está implementada en forma de bibliotecas de funciones que físicamente se encuentran en diversas DLL (bibliotecas de enlazado dinámico). A su conjunto se le denomina *Base Classes Library* (*Biblioteca de clases base* o **BCL**) y forman parte integral de la plataforma .NET, es decir, no se trata de añadidos que se deban obtener o adquirir aparte.

La siguiente figura ilustra a vista de pájaro la arquitectura conceptual de la plataforma .NET. En ella se pueden observar los elementos que se han mencionado en apartados anteriores (lenguajes, CLR, CLS...) y en qué lugar de se ubican las bibliotecas de clases base:

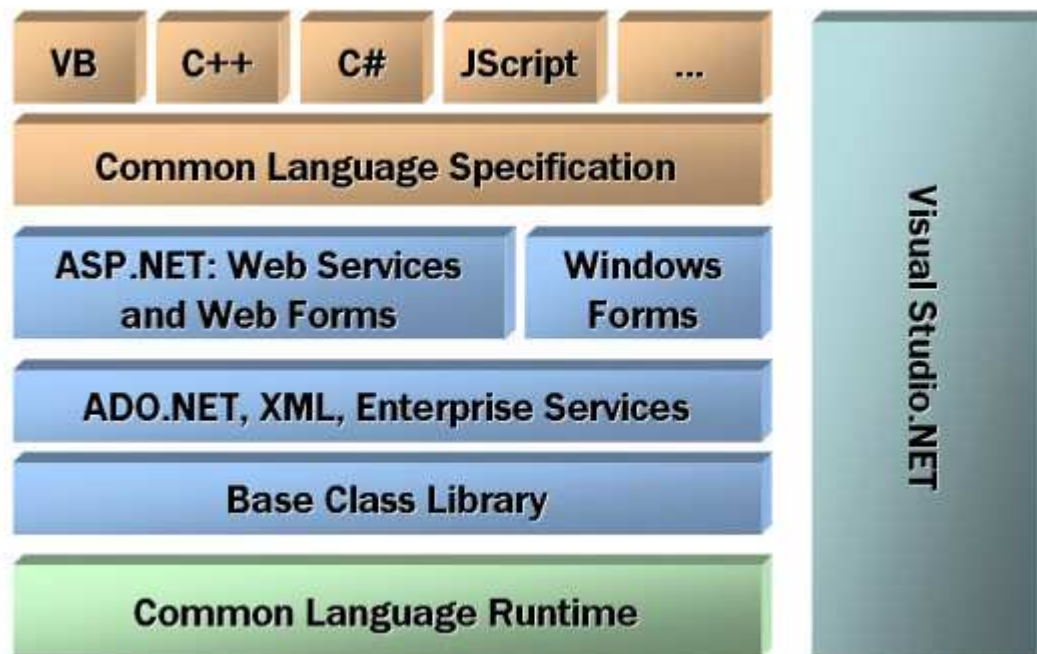


Figura 1.2. Distintos elementos de la plataforma .NET y cómo se relacionan entre sí.

Resulta muy útil para comprender lo explicado hasta ahora.

Todo lo que se encuentra en la BCL forma parte de la plataforma .NET. De hecho existe tal cantidad de funcionalidad integrada dentro de estas bibliotecas (hay decenas de miles de clases) que el mayor esfuerzo que todo programador que se inicia en .NET debe hacer es el aprendizaje de las más importantes. De todos modos Visual Studio ofrece mucha ayuda contextual (documentación, *Intellisense*...) y una vez que se aprenden los rudimentos resulta fácil ir avanzando en el conocimiento de la BCL a medida que lo vamos necesitando.

Los espacios de nombres

Dada la ingente cantidad de clases que existen debe existir algún modo de organizarlas de un modo coherente. Además hay que tener en cuenta que podemos adquirir más funcionalidades (que se traducen en clases) a otros fabricantes, además de poder crear nuevas clases propias.

Para solucionar este problema existen en todos los lenguajes .NET los **espacios de nombres** o **namespaces**.

Un espacio de nombres no es más que un identificador que permite organizar de modo estanco las clases que estén contenidas en él así como otros espacios de nombres.

Así, por ejemplo, todo lo que tiene que ver con el manejo de estructuras de datos XML en la plataforma .NET se encuentra bajo el espacio de nombres **System.Xml**. La funcionalidad fundamental para crear aplicaciones Web está en el espacio de nombres **System.Web**. Éste a su vez contiene otros espacios de nombres más especializados como **System.Web.Caching** para la persistencia temporal de datos, **System.Web.UI.WebControls**, que contiene toda la funcionalidad de controles Web para interfaz de usuario, etc...

Acceso a datos con ADO.NET

El acceso a fuentes de datos es algo indispensable en cualquier lenguaje o plataforma de desarrollo. La parte de la BCL que se especializa en el acceso a datos se denomina de forma genérica como **ADO.NET**.

Si usted ha programado con Visual Basic 6.0 o con ASP, ha empleado en su código con total seguridad la interfaz de acceso a datos conocida como ADO (*ActiveX Data Objects*), puede que combinado con ODBC (*Open Database Connectivity*). Si además es usted de los programadores con solera y lleva unos cuantos años en esto es probable que haya usado RDO o incluso DAO, todos ellos métodos mucho más antiguos.

ADO.NET ofrece una funcionalidad completamente nueva, que tiene poco que ver con lo existente hasta la fecha en el mercado. Sin embargo, con el ánimo de retirar barreras a su aprendizaje, Microsoft denominó a su nuevo modelo de acceso a datos con un nombre similar y algunas de sus clases recuerdan a objetos de propósito análogo en el vetusto ADO.

ADO.NET es un modelo de acceso mucho más orientado al trabajo desconectado de las fuentes de datos de lo que nunca fue ADO. Si bien este último ofrecía la posibilidad de desconectar los *Recordsets* y ofrecía una forma de serialización de estos a través de las diferentes capas de una aplicación, el mecanismo no es ni de lejos tan potente como el que nos ofrece ADO.NET.

El objeto más importante a la hora de trabajar con el nuevo modelo de acceso a datos es el **DataSet**. Sin exagerar demasiado podríamos calificarlo casi como un motor de datos relacionales en memoria. Aunque hay quien lo asimila a los clásicos *Recordsets* su funcionalidad va mucho más allá como se verá en el correspondiente módulo.

Arquitectura de ADO.NET

El concepto más importante que hay que tener claro sobre ADO.NET es su modo de funcionar, que se revela claramente al analizar su arquitectura:

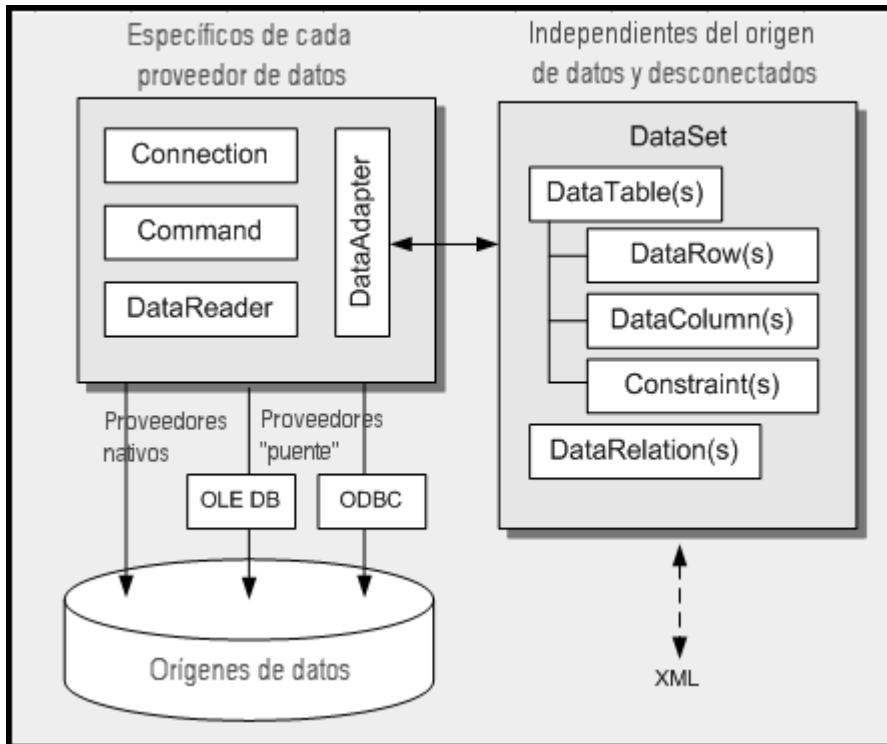


Figura 1.3.- Arquitectura de ADO.NET

Existen dos capas fundamentales dentro de su arquitectura: la **capa conectada** y la **desconectada**.

Capa conectada

La primera de ellas contiene objetos especializados en la conexión con los orígenes de datos. Así, la clase genérica **Connection** se utiliza para establecer conexiones a los orígenes de datos. La clase **Command** se encarga de enviar comandos de toda índole al origen de datos. Por fin la clase **DataReader** está especializada en leer los resultados de los comandos mientras se permanece conectado al origen de datos.

La clase **DataAdapter** hace uso de las tres anteriores para actuar de puente entre la capa conectada y la desconectada.

Estas clases son abstractas, es decir, no tienen una implementación real de la que se pueda hacer uso directamente. Es en este punto en donde entran en juego los **proveedores de datos**. Cada origen de datos tiene un modo especial de comunicarse con los programas que los utilizan, además de otras particularidades que se deben contemplar. Un proveedor de datos de ADO.NET es una implementación concreta de las clases conectadas abstractas que hemos visto, que hereda de éstas y que tiene en cuenta ya todas las particularidades del origen de datos en cuestión.

Así, por ejemplo, las clases específicas para acceder a SQL Server se llaman **SqlConnection**, **SqlCommand**, **SqlDataReader** y **SqlDataAdapter** y se encuentran bajo el espacio de nombres **System.Data.SqlClient**. Es decir, al contrario que en ADO clásico no hay una única clase **Connection** o **Command** que se use en cada caso, si no que existen clases especializadas para conectarse y recuperar información de cada tipo de origen de datos.

Nota:

El hecho de utilizar clases concretas para acceso a las fuentes de datos no significa que no sea posible escribir código independiente del origen de datos. Todo lo contrario. La plataforma .NET ofrece grandes facilidades de escritura de código genérico basadas en el uso de herencia e implementación de interfaces. De hecho la versión 2.0 de .NET ofrece grandes novedades específicamente en este ámbito.

Existen **proveedores nativos**, que son los que se comunican directamente con el origen de datos (por ejemplo el de SQL Server o el de Oracle), y **proveedores "puente"**, que se utilizan para acceder a través de ODBC u OLEDB cuando no existe un proveedor nativo para un determinado origen de datos.

Nota:

Estos proveedores puente, si bien muy útiles en determinadas circunstancias, ofrecen un rendimiento menor debido a la capa intermedia que están utilizando (ODBC u OLEDB). Un programador novel puede sentir la tentación de utilizar siempre el proveedor puente para OLEDB y así escribir código compatible con diversos gestores de datos de forma muy sencilla. Se trata de un error y siempre que sea posible es mejor utilizar un proveedor nativo.

Capa desconectada

Una vez que ya se han recuperado los datos desde cualquier origen de datos que requiera una conexión ésta ya no es necesaria. Sin embargo sigue siendo necesario trabajar con los datos obtenidos de una manera flexible. Es aquí cuando la capa de datos desconectada entra en juego. Además, en muchas ocasiones es necesario tratar con datos que no han sido obtenidos desde un origen de datos relacional con el que se requiera una conexión. A veces únicamente necesitamos un almacén de datos temporal pero que ofrezca características avanzadas de gestión y acceso a la información.

Por otra parte las conexiones con las bases de datos son uno de los recursos más escasos con los que contamos al desarrollar. Su mala utilización es la causa más frecuente de cuellos de botella en las aplicaciones y de que éstas no escalen como es debido. Esta afirmación es especialmente importante en las aplicaciones Web en las que se pueden recibir muchas solicitudes simultáneas de cualquier parte del mundo.

Finalmente otro motivo por el que es importante el uso de los datos desconectado de su origen es la transferencia de información entre capas de una aplicación. Éstas pueden encontrarse distribuidas por diferentes equipos, e incluso en diferentes lugares del mundo gracias a Internet. Por ello es necesario disponer de algún modo genérico y eficiente de poder transportar los datos entre diferentes lugares, utilizarlos en cualquiera de ellos y posteriormente tener la capacidad de conciliar los cambios realizados sobre ellos con el origen de datos del que proceden.

Todo esto y mucho más es lo que nos otorga el uso de los objetos **DataSet**. Es obvio que no se trata de tareas triviales, pero los objetos *DataSet* están pensados y diseñados con estos objetivos en mente. Como podremos comprobar más adelante en este curso es bastante sencillo conseguir estas funcionalidades tan avanzadas y algunas otras simplemente usando de manera adecuada este tipo de objetos.

Nota:

Otra interesante característica de los *DataSet* es que permiten gestionar simultáneamente diversas tablas (relaciones) de datos, cada una de un origen

diferente si es necesario, teniendo en cuenta las restricciones y las relaciones existentes entre ellas.

Los *DataSet*, como cualquier otra clase no sellada de .NET, se pueden extender mediante herencia. Ello facilita una técnica avanzada que consiste en crear tipos nuevos de *DataSet* especializados en la gestión de una información concreta (por ejemplo un conjunto de tablas relacionadas). Estas nuevas tipos clases se denominan genéricamente ***DataSet Tipados***, y permiten el acceso mucho más cómodo a los datos que representan, verificando reglas de negocio, y validaciones de tipos de datos más estrictas.

Aplicaciones de Windows Forms

Las aplicaciones de escritorio son aquellas basadas en ventanas y controles comunes de Windows que se ejecutan en local. Son el mismo tipo de aplicaciones que antes construiríamos con Visual Basic 6 u otros entornos similares.

En la plataforma .NET el espacio de nombres que ofrece las clases necesarias para construir aplicaciones de escritorio bajo Windows se denomina ***Windows Forms***. Este es también el nombre genérico que se le otorga ahora a este tipo de programas basados en ventanas.

Windows Forms está constituido por multitud de clases especializadas que ofrecen funcionalidades para el trabajo con ventanas, botones, rejillas, campos de texto y todo este tipo de controles habituales en las aplicaciones de escritorio.

Visual Studio ofrece todo lo necesario para crear visualmente este tipo de programas, de un modo similar aunque más rico al que ofrecía el entorno de desarrollo integrado de Visual Basic.

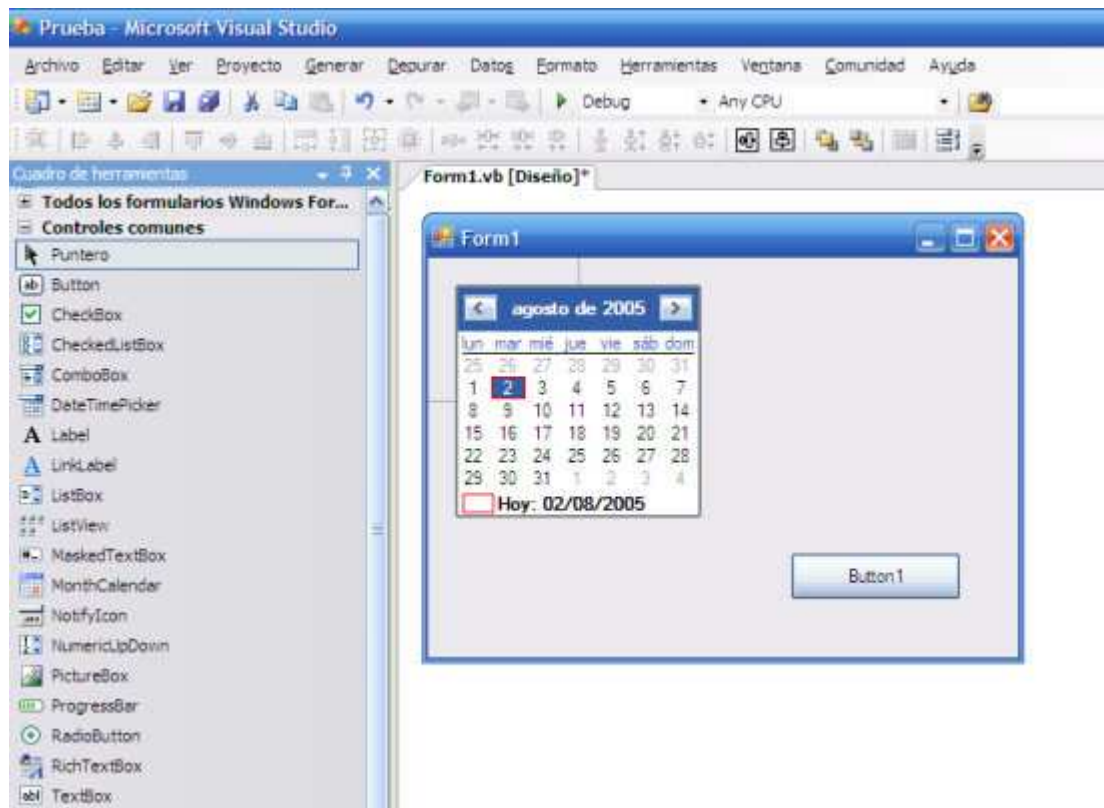


Figura 1.4.- Diseñador de interfaces de aplicaciones de escritorio con Windows Forms en Visual Studio.

Al contrario que en VB6, .NET proporciona control sobre todos los aspectos de las ventanas y controles, no dejando nada fuera del alcance del programador y otorgando por lo tanto la máxima flexibilidad. Los formularios (ventanas) son clases que heredan de la clase base **Form**, y cuyos controles son miembros de ésta. De hecho se trata únicamente de código y no es necesario (aunque sí muy recomendable) emplear el diseñador gráfico de Visual Studio para crearlas.

Este es el aspecto que presenta parte del código que genera la interfaz mostrada en la anterior figura:

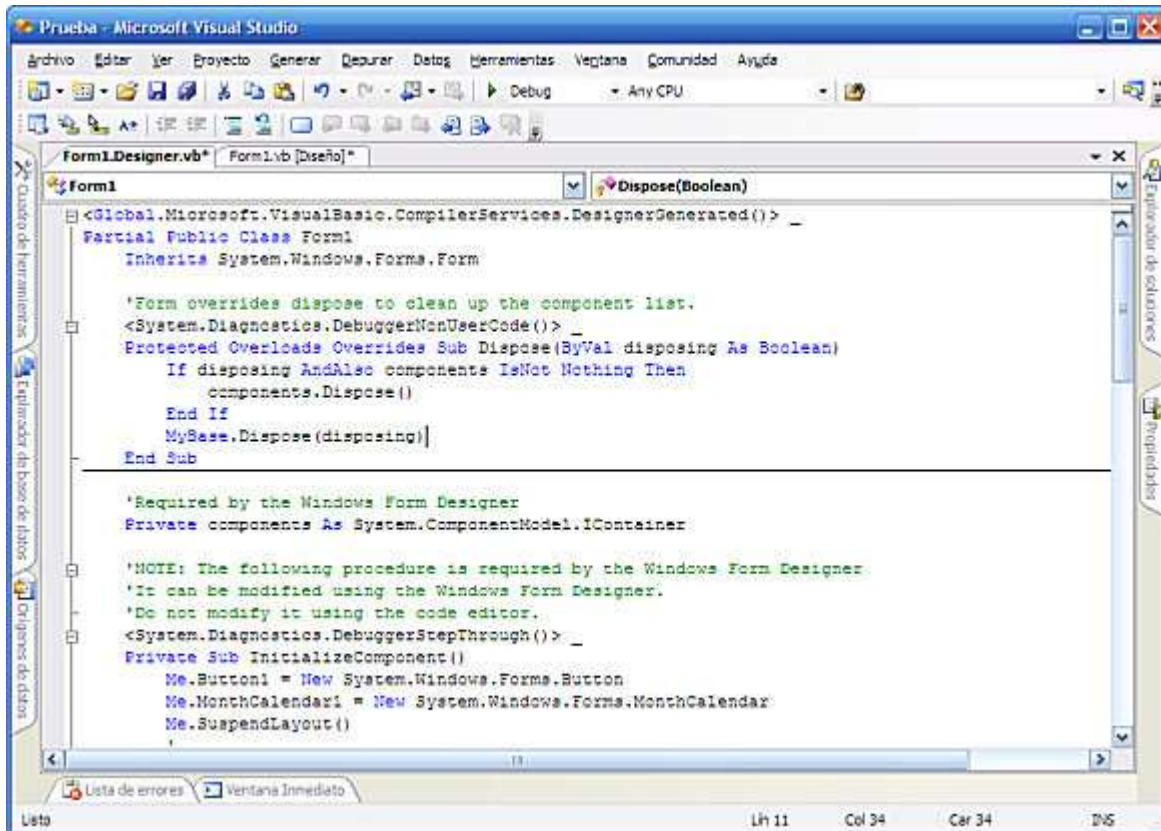


Figura 1.5.- Código autogenerated por Visual Studio para crear la interfaz de la figura anterior.

Al contrario que en Visual Basic tradicional, en donde siempre existían instancias por defecto de los formularios que podíamos usar directamente, en .NET es necesario crear un objeto antes de poder hacer uso de los formularios:

```
Dim frm As New MiFormulario  
frm.Show()
```

Todos los controles heredan de una clase **Control** por lo que conservan una serie de funcionalidades comunes muy interesantes, como la capacidad de gestionarlos en el diseñador (moviéndolos, alineándolos...), de definir márgenes entre ellos o hacer que se adapten al tamaño de su contenedor.

Aplicaciones Web Forms

Tradicionalmente las aplicaciones Web se han desarrollado siguiendo un modelo mixto que intercalaba código HTML y JavaScript propio de páginas Web (parte cliente), junto con código que se ejecutaría en el servidor (parte servidora). Este modelo contrastaba por completo con el modelo orientado a eventos seguido por las principales herramientas de desarrollo de aplicaciones de escritorio.

En el modelo orientado a eventos se define la interfaz de usuario colocando controles en un contenedor y se escribe el código que actuará como respuesta a las interacciones de los usuarios sobre estos controles. Si conoce el diseñador de VB6 o de Windows Forms mencionado en el apartado anterior sabe exactamente a qué nos referimos.

Hacer esto en una aplicación de escritorio no tiene mayor dificultad ya que todo el código se ejecuta en el mismo lugar. La principal característica de las aplicaciones Web sin embargo es que se la interfaz de usuario (lo que los usuarios de la aplicación ven) se ejecuta en un lugar diferente al código de la aplicación que reside en un servidor. Para mayor desgracia estas aplicaciones se basan en el uso del protocolo HTTP que es un protocolo sin estado y que no conserva la conexión entre dos llamadas consecutivas.

Por ejemplo, el siguiente código ilustra el código que es necesario escribir en ASP para disponer de una página que rellena una lista de selección con unos cuantos nombres (podrían salir de una base de datos y aún sería más complicado), y que dispone de un botón que escribe un saludo para el nombre que se haya elegido de la lista.

A screenshot of a code editor window displaying ASP code. The code is written in a mix of HTML and VBScript. It starts with a standard HTML document structure, followed by a VBScript section that initializes an array of names: "Antonio", "Jose", "Alberto", "Luis", and "Benito". The code then creates an HTML form with a submit button labeled "Di hola". Inside the form, there is a select dropdown menu. A loop in VBScript populates this dropdown with the names from the array. After the form, there is another VBScript section that checks if a name was selected and, if so, displays a greeting "Hola, " followed by the selected name. The code is color-coded: HTML tags are in blue, VBScript comments and keywords are in purple, and string literals are in green. The editor's status bar at the bottom shows "Ln 30, Col 1, Sel 0", "675 Bytes", "AMST", "C:\J.F.\TNS", and "Web Sources".

```
<html>

<%@ Language=VBScript %>
<%
nombres = array("Antonio",
                "Jose", "Alberto",
                "Luis", "Benito")
%>

<body>
<p>seleccione su nombre:</p>
<form method="POST" action="HolaMundo.asp">
  <p><select name="nombre" size="5">
    <% for i = 0 to UBound(nombres) %>
      <option
        <% if Request.Form("nombre") = nombres(i) then %>
          selected <% end if %> >
        <%=nombres(i) %></option>
    <% next %>
  </select><br><br>

  <input type="submit" value="Di hola"></p>
</form>
<% if Request.Form("nombre") <> "" then %>
<p>Hola, <%=Request.Form("nombre") %></p>
<% end if %>

</body>
</html>
```

Figura 1.6.-
Código ASP
sencillo que
genera una
lista de
selección y
saluda al
presionar un
botón.

Obviamente se podría haber simplificado sin enviar el formulario al servidor usando JavaScript en el cliente para mostrar el saludo, pero la intención es ilustrar la mezcla de código de cliente y de servidor que existe en este tipo de aplicaciones.

Las principales desventajas de este tipo de codificación son las siguientes:

1. **No existe separación entre el diseño y la lógica de las aplicaciones.** Si queremos cambiar sustancialmente la apariencia de la aplicación Web lo tendremos bastante complicado puesto que el código del servidor está mezclado entre el HTML.
2. **En ASP clásico no existe el concepto de control** para la interfaz de usuario. Lo único que hay es HTML y JavaScript que se deben generar desde el servidor. En el ejemplo de la figura para generar un control de lista con unos elementos no podemos asignar una propiedad de la lista (porque no existe tal lista), sino que tenemos que crear un bucle que genere los elementos HTML necesarios para generarla. Tampoco disponemos de un diseñador visual que nos permita gestionar los controles y elementos HTML existentes, y menos cuando éstos se encuentran mezclados con el código del servidor.
3. **No disponemos de forma de detectar en el servidor que se ha realizado algo en el cliente.** El cliente se encuentra desconectado desde el momento en que se termina de devolver la página. Sólo se recibe información en el servidor cuando se solicita una nueva página o cuando se envía un formulario tal y como se hace en el ejemplo, debiéndonos encargar nosotros de averiguar si la petición es la primera vez que se hace o no, y de dar la respuesta adecuada. En cualquier caso es mucho menos intuitivo que el modelo de respuesta a eventos de una aplicación de escritorio.
4. **No existe constancia del estado de los controles de cada página entre las llamadas.** En cada ejecución de la página tendremos que recrear completamente la salida. Por ejemplo si presionamos en el botón "Di Hola" tenemos que escribir además de la etiqueta "Hola, nombre" el resto de la pantalla, incluyendo la lista con todos los nombres dejando seleccionado el mismo que hubiese antes. Si estos nombres viniesen de una base de datos esto puede ser todavía más ineficiente y tendremos que buscar métodos alternativos para generarlos ya que en ASP tampoco se deben almacenar en los objetos de sesión y/o aplicación Recordsets resultado de consultas.
5. **No existe el concepto de Propiedad de los controles.** En una aplicación Windows asignamos el texto de un campo usando una propiedad (por ejemplo Text1.Text = "Hola") y ésta se asigna y permanece en la interfaz sin que tengamos que hacer nada. En una aplicación Web clásica tenemos que almacenarlas en algún sitio (una variable de sesión o un campo oculto) para conservarlas entre diferentes peticiones de una misma página.
6. **Los controles complejos no tienen forma de enviar sus valores al servidor.** Si intentamos crear una interfaz avanzada que utilice tablas y otros elementos que no son controles de entrada de datos de formularios de HTML tendremos que inventarnos mecanismos propios para recoger esos datos y enviarlos al servidor.

La principal aportación de ASP.NET al mundo de la programación es que ha llevado a la Web el paradigma de la programación orientada a eventos propia de aplicaciones de escritorio, ofreciendo:

- Separación entre diseño y lógica.
- Componentes de interfaz de usuario, tanto estándar como de terceras empresas o propios.
- Diseñadores gráficos.
- Eventos.
- Estado.
- Enlazado a datos desde la interfaz.