VISUAL BASIC BRAIN UPGRADE

Incluye:

- Introducción a .NET
- Orientación a Objetos
- Nuevas características
- ADO.NET
- Web Forms
- Win Forms
- GDI+

VB.NET

Todo lo que necesita para migrar a esta nueva versión y más...

www.vblibros.com

Erich Bühler

Visual Basic© .NET
Brain Upgrade

Erich Bühler

Esta es una versión DEMO, mas detalles en:

www.vblibros.com

Nota de Copyright

Todos los derechos reservados. Ninguna parte de este libro puede ser reproducida, almacenada, o transmitida bajo ninguna tecnología, sin el consentimiento por escrito del autor. En el caso de que se use el contenido de este libro para cualquier artículo o documento se deberá citar la fuente.
La información ofrecida en este libro no tiene garantía implícita o explicita, y el autor no se hace responsable por los daños causados directa o indirectamente por el material contenido.
La versión de demostración del libro es de libre distribución
Este versión está sujeta a cambios.

Primera edición: Marzo-2001
Edición actual: Marzo-2001

© 2001 Erich Bühler

Información sobre Marcas registradas y/o Empresas

El autor ha se ha esforzado por proveer toda la información de compañías y productos mencionados en este libro utilizando las formas apropiadas de nomenclatura. No obstante, no se puede asegurar la exactitud de la misma.

Créditos

Autor

Erich Bühler

Revisiones Técnicas

Ing. Eduardo Carreira
Ricardo Coulthurst

Diseño de Tapa

Santiago Carreira
Erich Bühler

Agradecimientos

Ing. Eduardo Carreira
Santiago Carreira
Wilson Pais
Andy Gonzalez

Recomendaciones para utilizar este libro

Para ejecutar los ejemplos de este libro, necesitará un mínimo de:

- Pentium II de 300 MHz con 128 MB de RAM, y al menos 10 MB de espacio en disco rígido.
- Microsoft© Windows© 2000
- Microsoft© Internet Explorer 5.5 o superior
- IIS
- Motor de base de datos compatible con OLEDB
- Visual Basic©.NET

Convenciones utilizadas

En este libro se utilizan diferentes estilos de letras y composiciones, a los efectos de ayudarlo a diferenciar entre los diferentes tipos de información. Los estilos de notas importantes se presentan de la siguiente manera:

No debe olvidarse de liberar la memoria, de lo contrario puede producirse un error de protección general al salir de la aplicación.

- Las palabras importantes aparecen en **negrita**
- Siglas o palabras en ingles aparecen en *este formato*
- Los hipervínculos aparecen en este formato **http://www.vblibros.com**
- Objetos, funciones o código aparecen en `este formato`
- Las anotaciones aparecen en este formato
- Si el ejemplo es importante aparece de la siguiente forma:

```
Protected Sub RadioButton1_CheckedChanged(ByVal sender As _
Object, ByVal e As System.EventArgs)

End Sub
```

Nota del Autor

El objetivo de este libro es minimizar la curva de aprendizaje hacia la nueva versión Microsoft© Visual Basic© .NET.

Los cambios a nivel de lenguaje y arquitectura son enormes, y de hecho he tratado que el material aquí contenido pueda ser útil tanto para una persona que desee migrar a esta nueva versión, como para quien desee aprender algo mas del lenguaje.

Al final del libro encontrará un glosario tecnológico, donde se explayan nombres, siglas, y su significado.

El libro está lleno de ejemplos y prácticas que hacen del aprendizaje o migración mas ameno, no obstante, sus recomendaciones son bienvenidas y tomadas para futuras revisiones. Adicionalmente, al inicio de cada capítulo podrá encontrar el objetivo del mismo, el nivel de dificultad, y vínculos puede consultar.

He tratado de cubrir los aspectos mas importantes de esta nueva versión, pero si considera que algún punto puede ser importante de incluir, pónganse en contacto conmigo.

Atentamente,
Erich Bühler
tech@vblibros.com

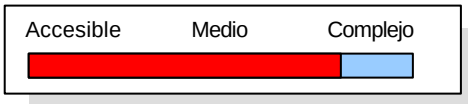
Tabla de contenidos

Capítulo 1

Introducción a la plataforma .NET

Descripción:	Este capítulo es una introducción a la plataforma .NET, así como las nuevas características de VB.NET
Vínculos adicionales:	http://www.Microsoft©.com/net http://www.Microsoft©.com/vbasic

Nivel de Dificultad



Este primer capítulo presenta la evolución del Microsoft© Visual Basic©, así como un resumen de cada una de las nuevas tecnologías incluidas en *VB.NET* .

Puede omitirlo si lo que desea es conocer las características propias de esta nueva versión de Microsoft© Visual Basic©, pero debido al gran número de cambios a nivel programático y tecnológico, es recomendable que comience por entender el nuevo paradigma, que responde a un avance o solución a problemas planteados por tecnologías utilizadas actualmente.

Un poco de historia...

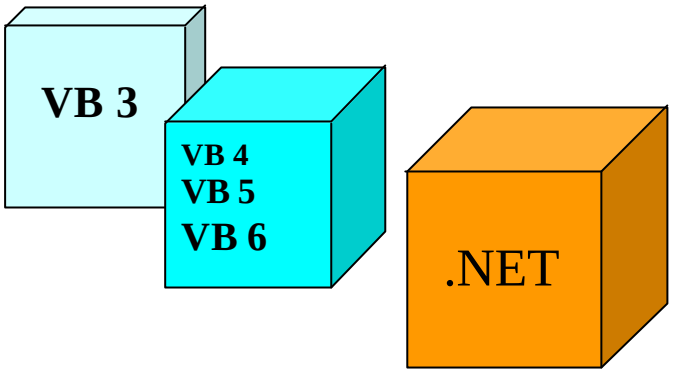
Generalmente, una nueva versión de Microsoft© Visual Basic© implica la inserción de alguna tecnología y/o extensión del lenguaje. En muchos casos los cambios tecnológicos en Microsoft© Visual Basic© no implicaban grandes modificaciones desde el punto de vista del lenguaje (programáticos), aunque sí generalmente agregaban nuevas palabras, procesos, y/o formas a la hora de implantar la solución.

De esta forma, muchos desarrolladores pudieron mantener el mismo nivel de conocimientos trabajando simultáneamente en diferentes versiones, sin que esto produjese la necesidad imperiosa de aprender las características de la nueva versión.
En muchos casos el desarrollador continuaba utilizando tecnologías de versiones anteriores en las mas recientes, ya sea por comodidad, estabilidad, o porque la nueva tecnología no le ofrece grandes beneficios.

Excepto en la migración de Microsoft© Visual Basic© 3.0 a Microsoft© Visual Basic© 4.0 en la cual el desarrollador tuvo que actualizar sus controles visuales (*VBX*) a los nuevos controles *OLE* o *ActiveX Controls* (*OCX*), prácticamente no existieron nuevos problemas de impacto en la programación al realizar las migraciones.
La versión 5.0 de Microsoft© Visual Basic© incluyó la posibilidad de crear *controles visuales Activex*, con el fin de distribuir ó reutilizar funcionalidad, cosa que hasta ahora estaba reservada solamente a los lenguajes de mas bajo nivel.
La versión 6.0 Microsoft© Visual Basic© agregó algunas nuevas palabras en el lenguaje, así como soporte nativo para *Microsoft© Activex Data Object* (*ADO*), un generador de informes, y un diseñador visual de objetos *ADO*.
Con *Microsoft© Activex Data Object* (*ADO*) se estableció una forma fácil de acceder a un origen de datos, con bajo impacto programático en caso de cambiar el motor de datos.

La tendencia de agregar funcionalidad al lenguaje manteniendo la compatibilidad con versiones anteriores -y sobre todo manteniendo compatibilidad con sus conocimientos- se mantuvo hasta la versión 6.0. Pero Microsoft© a dado un paso muy grande esta vez, y **ha decidido re-diseñar el lenguaje**.
Con el paso de las diferentes versiones, muchas características y funciones permanecieron por compatibilidad, generando de esta forma inconsistencias en el lenguaje. El advenimiento de las tecnologías *Web* implicó cambios estructurales profundos y notorios sobre la arquitectura de los lenguajes y sistemas operativos. Esto llevó a que Microsoft© considerara éste un buen momento para rehacer el lenguaje *Basic* en la versión 7.0 (o *.NET*), y de esta forma solucionar aquellas inconsistencias o características que –por mantener compatibilidad con versiones anteriores- limitaban el lenguaje.

Otra de las cosas exigidas por los desarrolladores Visual Basic© desde hace ya algunos años, es el soporte de orientación a objetos, el cual es uno de los cambios más trascendentes en esta versión.



¿Qué hay de nuevo en VB.NET ?

Microsoft© Visual Basic© 7.0 (denominada *.NET*), incluye características que hasta ahora estaban solamente disponibles en lenguajes de bajo nivel, tales como la reutilización total de código, herencia, polimorfismo, sobrecarga, constructores parametrizados, etc.

Por otra parte, la idea de utilizar un entorno de desarrollo único para todos los lenguajes, hacen que el cruzamiento Inter-lenguaje de código sea extremadamente sencillo. La inclusión de varias tecnologías *Web*, como los llamados *Web Services* –una suerte de componente de fácil utilización a través de Internet- hacen que las posibilidades se amplíen.

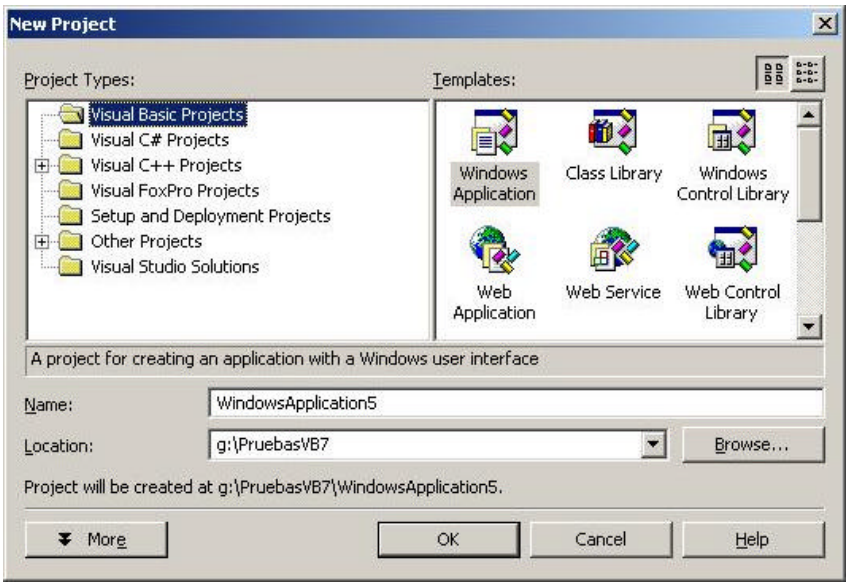
Si bien el enfoque publicitario de *VB.NET* se ha dado principalmente hacia tecnologías *Internet*, veremos que los cambios radicales van mucho mas allá.

Nueva Interfase

Originalmente la interfase utilizada en las aplicaciones Windows©, estaban orientadas hacia botones estándares, y se consideraba necesaria en cualquier aplicación que quisiera contar con cierto 'reconocimiento'. Con el advenimiento de *Internet*, las aplicaciones comenzaron a parecerse a páginas Web e hipervínculos, avanzando hacia esta dirección. Microsoft© no se ha quedado atrás, y ha rediseñado el entorno de Microsoft© Visual Basic© para incluir muchas de estas cualidades. Una de las características principales de Visual Basic©, es la contar con un entorno que hace posible la creación de la interfase gráfica, dibujando en forma fácil y rápida controles y ventanas. Para la mayoría de empresas y desarrolladores, ésta es una característica crucial a la hora de elegir el lenguaje a utilizar. De hecho, muchas personas que hoy están trabajando con Microsoft© Visual Basic©, ingresaron en algún momento atraídos por la simplicidad y velocidad con la que se construían las interfases gráficas.

Las versiones anteriores tenían solamente la característica de crear formularios y -posteriormente- controles ActiveX. Esta nueva versión tiene la posibilidad de dibujar pantallas y trabajar con código *Web*. De esta forma se incluye soporte para manejar en forma integrada el código asociado a cada uno de los eventos de los objetos *Web*,

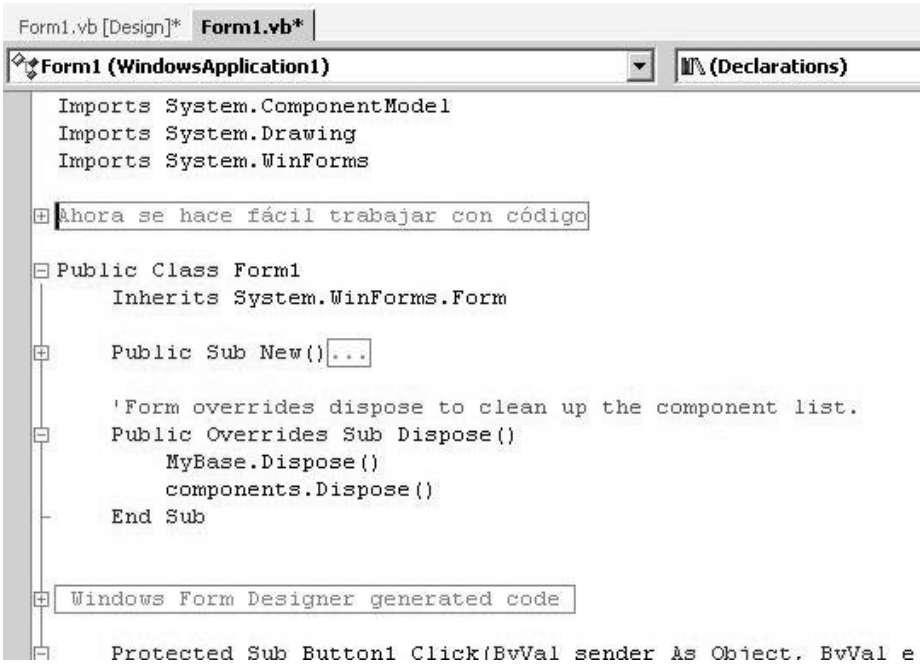
relegándonos por momentos que estamos desarrollando en una arquitectura *Internet* (3 capas).



El ser una de las pocas herramienta de trazado (Debug) Visual que funciona a la perfección, y contar con la posibilidad de establecer conexiones a motores de base de datos e informes en forma gráfica, hacen que el entorno de desarrollo sea deseable para cualquier lenguaje. Es así que muchos desarrolladores que trabajaban en otros entornos cambiaron a Visual Basic© con el único fin de obtener algunas características del entorno de desarrollo. Visto que en esta nueva versión *.NET* todos los lenguajes comparten el mismo entorno de desarrollo, existen varias empresas que están trabajando en la construcción de lenguajes compatibles con el entorno y con las características .NET.

El reemplazo del explorador de Proyectos -ahora llamado **Explorador de Soluciones**- hace posible trabajar simultáneamente con varios proyectos de diferentes lenguajes al mismo tiempo. Esto no sólo ofrece la comodidad de utilizar una única interfase, sino que viene como respuesta a varios de los cambios de unificación de los lenguajes. Como ejemplo, verá que ahora es posible heredar programáticamente o binariamente componentes realizados en lenguajes diferentes, e incluso hasta heredar visualmente formularios.
Debido al aumento de código generado por algunos asistentes, el editor admite definir secciones jerárquicas de código, que pueden ser comprimidas o expandidas dinámicamente.

Gracias a esta funcionalidad, se mejora la visibilidad y claridad de la implementación de un proyecto.



En la nueva interfase, la **creación de macros** le permite controlar el entorno de desarrollo de *VB.NET* , permitiendo grabar y reproducir mas tarde aquellas acciones que utiliza frecuentemente. Veremos mas pormenorizado las nuevas características del entorno de desarrollo en el próximo capítulo.

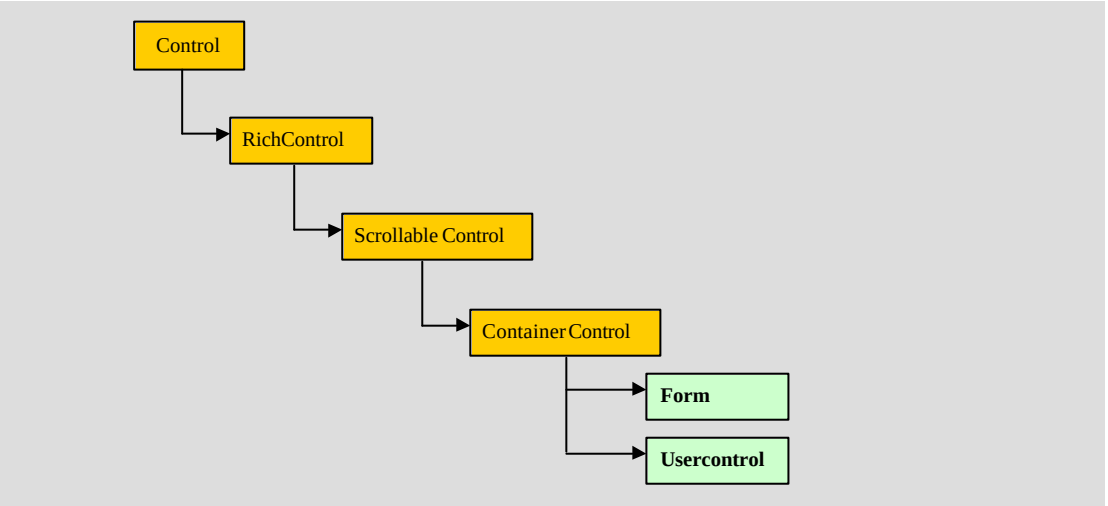
Windows© Forms

Si bien esta tecnología será vista en detalle en el capítulo 3, haremos una breve introducción a los efectos que comience a analizar el alcance e impacto de los cambios en esta nueva versión.
Como vimos anteriormente, el entorno de programación le permite generar controles y formularios en forma fácil y rápida, pero a costo de perder flexibilidad en algunos aspectos. Cuando usted dibuja un formulario, toda la información referente a los atributos del objeto (posición, cantidad de controles, colores, etc) es guardada internamente en los archivos del formulario. Esta información no puede ser accedida programáticamente, siendo gestionados automáticamente por Visual Basic© cada vez que se carga el proyecto. De esta forma, el motor de formularios interpreta estos datos y recrea la interfase gráfica al último estado almacenado.

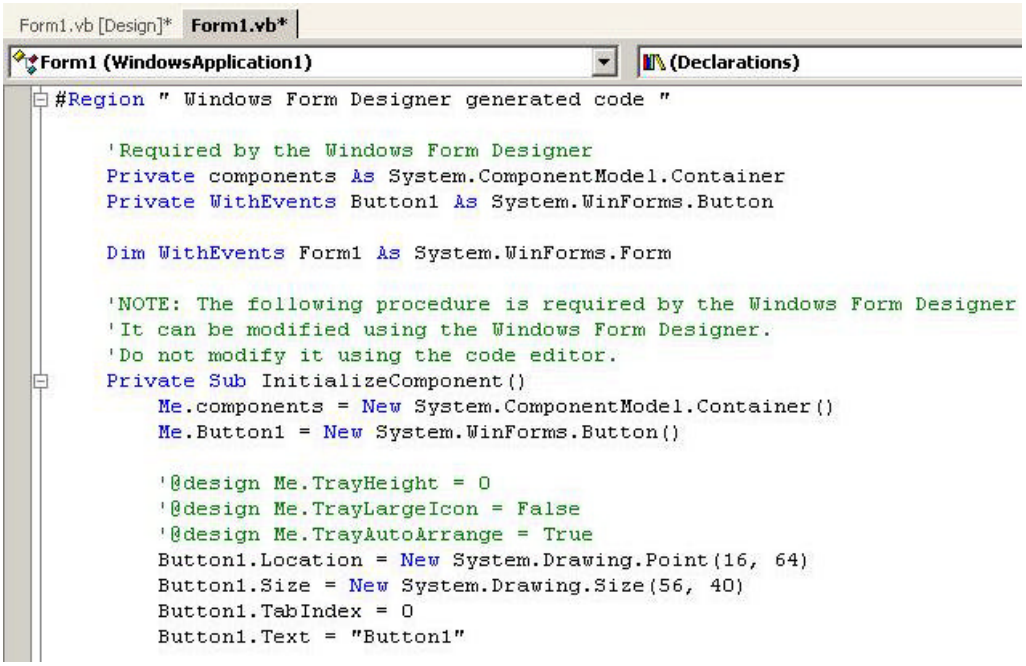
En muchos lenguajes, el diseño de la interfase gráfica es aún mas complejo, ya que debe realizarse manualmente, programando cada línea de código, que posteriormente generará en tiempo de ejecución uno a uno los controles y sus características. Esta tarea es tediosa y lenta, y requiere que el programador tenga conocimientos específicos del lenguaje. En otros lenguajes se cuenta con asistentes que permiten crear la pantalla, escribiendo por usted las líneas de código. Si comparamos la opción utilizada por Visual Basic© con la empleada por otros lenguajes, podremos observar algunas ventajas y desventajas:

Visual Basic©	Otros lenguajes
La interfase se construye en forma rápida	La interfase se construye en forma lenta
No es necesario conocer el lenguaje para construir la interfase	Generalmente, es necesario conocer el lenguaje para construir la interfase
Se pierden características avanzadas de dominio sobre la interfase gráfica.	Se tiene dominio total sobre la interfase gráfica.

De esta forma, cada modalidad cuenta con sus respectivos beneficios. Una característica importante a tener en cuenta, es la de obtener el mayor control sobre la interfase gráfica, cosa con la que Visual Basic© no cuenta. Una opción media, es la de generar un asistente visual para dibujar las pantallas, que sea capaz de escribir código por usted. De esta forma, se tendría lo mejor de cada mundo, sin perder ninguna de las ventajas arriba citadas.
Como Visual Basic© no cuenta con funcionalidad para soportar esto, fue necesario extender la capacidad del lenguaje, a los efectos de poder definir nuevas características que hagan posible el acceder a los recursos gráficos.



El manejo de esta nueva modalidad, trae como beneficio el control total de la interfase gráfica. La nueva versión de Visual Basic© escribe código cuando usted agrega, quita, o modifica las propiedades de un control.
Para ello, se ha implementado un conjunto de objetos capaz de manipular la creación y características de los controles visuales de la interfase gráfica. De hecho, todos los controles visuales se heredan de una jerarquía de clases comunes. Adicionalmente las nuevas características del lenguaje permiten capacidades tales como herencia, sobrecarga, etc, sobre los controles, así como la de heredar un formulario en otro, y la esperada característica de auto-ajuste de los objetos al tamaño de la ventana.

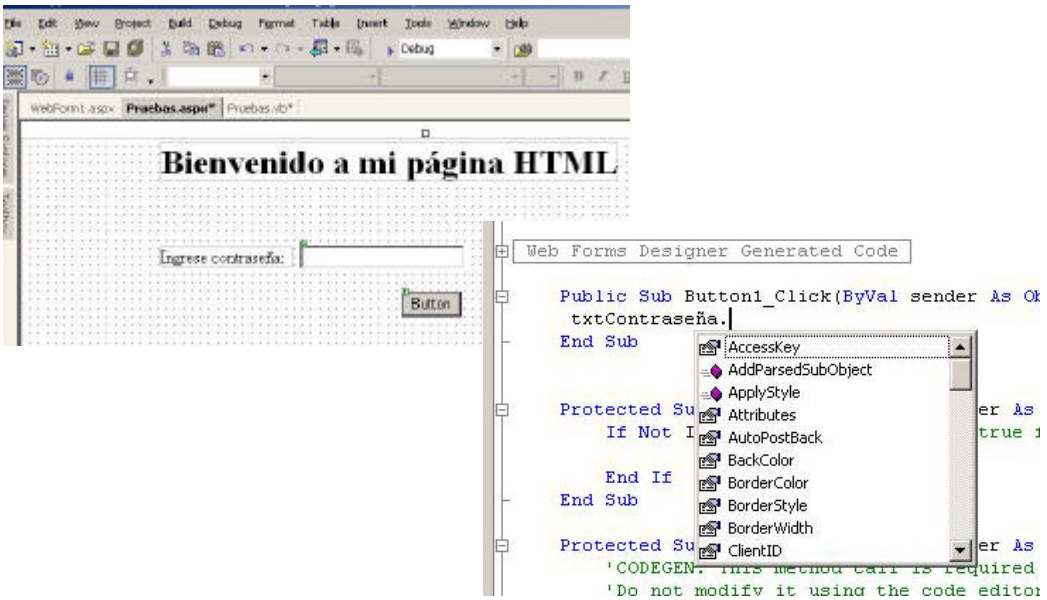


Si bien características similares ya existían en lenguajes tales como *Borland Delphi*, *Java*, o en *Visual C++* con las llamadas *Windows© Foundation Classes* (un conjunto de clases para construir la interfase Visual), ahora es posible obtener las mismas características funcionales desde todos los lenguajes de Visual Studio.NET©, con una curva de aprendizaje menor.

Formularios Web (Web Forms)

No es novedad que el advenimiento de *Internet* ha cambiado y está cambiando la forma en la que las personas y las aplicaciones hacen las cosas. Lo que si es nuevo, son las nuevas tecnologías aplicables de Visual Basic.NET© para *Internet*, denominadas ASP.NET.

En versiones anteriores de Visual Basic©, contábamos con características como Activex Documents, Web Classes, y el diseñador de DHTML. Ninguna de estas es fácil a la hora de desarrollar una aplicación, y en muchas ocasiones tampoco es efectiva, ya que muchas veces se termina realizando programáticamente –y con un gran esfuerzo- aquello que es muy fácil de hacer en otros productos.
Hoy en día el desarrollador tiene claro que para trabajar en aplicaciones para *Internet* con productos Microsoft©, debe utilizar Visual Interdev©. La versión de Visual Basic.NET© se fusiona con Visual Interdev© -por lo que este último deja de existir como tal- a los efectos de obtener características comunes.
Ahora se incluyen los llamados Formularios Web o *Web Forms*. Éstos constan de dos partes: el código HTML que contiene la representación visual de la pantalla, y el código asociado a cada uno de los eventos de la página.
La ventaja de los *Web Forms*, es que usted utiliza un diseñador para dibujar los formulario *Web*, similar al que emplea para crear formularios estándares Windows©. Es luego el entorno de desarrollo quien se encarga de separar el código que usted escribe en los eventos de los objetos, de la interfase visual.
A diferencia de la tecnología que se utilizada en las páginas *ASP (Active Server Pages)* en la cual se incluye el código Visual Basic© Script (un sub conjunto del lenguaje Visual Basic©) dentro de la página, en los formularios *Web* la representación visual es almacenada como una página HTML, y el código es compilado en una biblioteca, con el fin de lograr un mejor desempeño.



El código que se asocia a los eventos es mayoritariamente ejecutado del lado del servidor, esto quiere decir que cada vez que el usuario realiza una acción sobre la

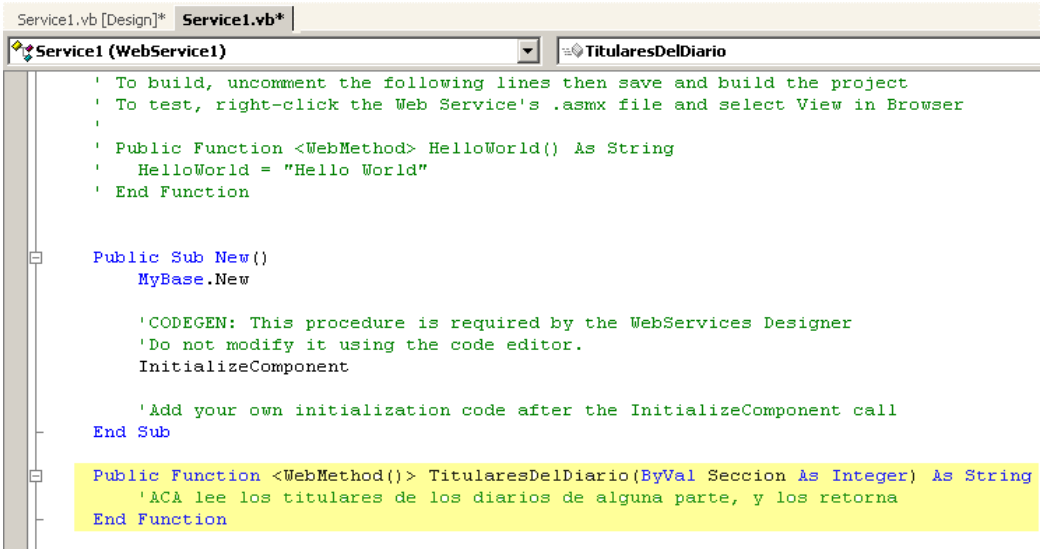
interfase Web, esta será transmitida a través de *HTTP* hacia el servidor, se ejecutará, y la pantalla resultante será enviada al cliente. Todo esto se hace en forma transparente para usted, y en muchas ocasiones se optimiza la ida y vuelta al servidor, enviando varios eventos al mismo tiempo. Como ventaja obtenemos un máximo rendimiento, con una interfase conocida, logrando un resultado multi-navegador y multi-plataforma. Créame que la tarea de programar un Formulario *Web* es muy similar a la de trabajar con formularios estándares, he incluso puede utilizar las mismas características de trazado (Debug). Similar a los controles *Activex* que puede dibujar en un formulario, los formularios *Web* cuentan con controles *Web* (*Web Controls*), que puede arrastrar desde la barra de herramienta hacia el formulario *Web*, y posteriormente configurar sus propiedades mediante la ventana de propiedades o código.
Con Visual Basic.Net© puede crear verdaderas aplicaciones para *Internet*, obteniendo el 100% de la funcionalidad del lenguaje, y reutilizando aquellas bibliotecas que ya tiene desarrolladas en versiones anteriores.

Servicios Web (Web Services)

A mi criterio, esta es la **característica más importante** de *VB.NET* , en la cual se basaran las aplicaciones de la nueva generación.
Cuando usted tiene que integrar servicios de *Internet* y aplicaciones de escritorio, lo debe hacer mediante soluciones que en muchos casos resultan ser demasiado complejas, debido a que no existe o mejor dicho no existía una tecnología que permitiera una mezcla sencilla.
Generalmente, la arquitectura seleccionada consta de las llamadas **aplicaciones híbridas** (parte aplicaciones de escritorio Win32© y parte páginas ASP), difíciles de mantener o actualizar.

Cuando usted trabaja con una biblioteca estándar, la idea principal es la de exponer lógica (procedimientos, funciones, y eventos) a través de los protocolos estándares de Windows©. Conceptualmente, un servicio *Web* o *Web Service* es similar a una biblioteca estándar (*COM* compatible). Cuando usted trabaja con bibliotecas estándares, debe cerciorarse de agregar la referencia a antes de comenzar a utilizarla en el proyecto. Una vez hecho esto, Visual Basic© la examina, y –entre otras- obtiene la interfase (procedimientos, funciones, y eventos) de la biblioteca, a los efectos de hacerlo visibles mediante la tecnología intelisense© utilizada en el editor de código, y visible desde la aplicación. Si bien los componentes estándares son muy efectivos –y de hecho lo seguirán siendo-, existen algunos casos en los cuales no ofrecen un marco de trabajo sencillo. Los componentes estándares no fueron pensados para ser accedidos o utilizados a través de *Internet*, debido a que utilizan la tecnología basada en estándares de Microsoft© (y por ende de Windows©) denominada *COM* o *Component Object Model*. Ésta hace posible la interconexión en forma eficiente entre componentes. Por otra parte, el formato en el cual se gestiona la comunicación entre ellos es binario, lo que lo hace muy óptimo en la familia de entornos Windows©. Lamentablemente si la biblioteca necesita ser accedida través de *Internet*, este formato sufre de graves inconvenientes. Uno de ellos es el de no ser de fácil interpretación o manipulación por aplicaciones o sistemas operativos no-Windows©.

A diferencia de una biblioteca estándar, un servicio *Web* esta concebido para exponer la interfase a través de *Internet*, utilizando protocolos estándares, y que han sido suficientemente probados en estos últimos años.



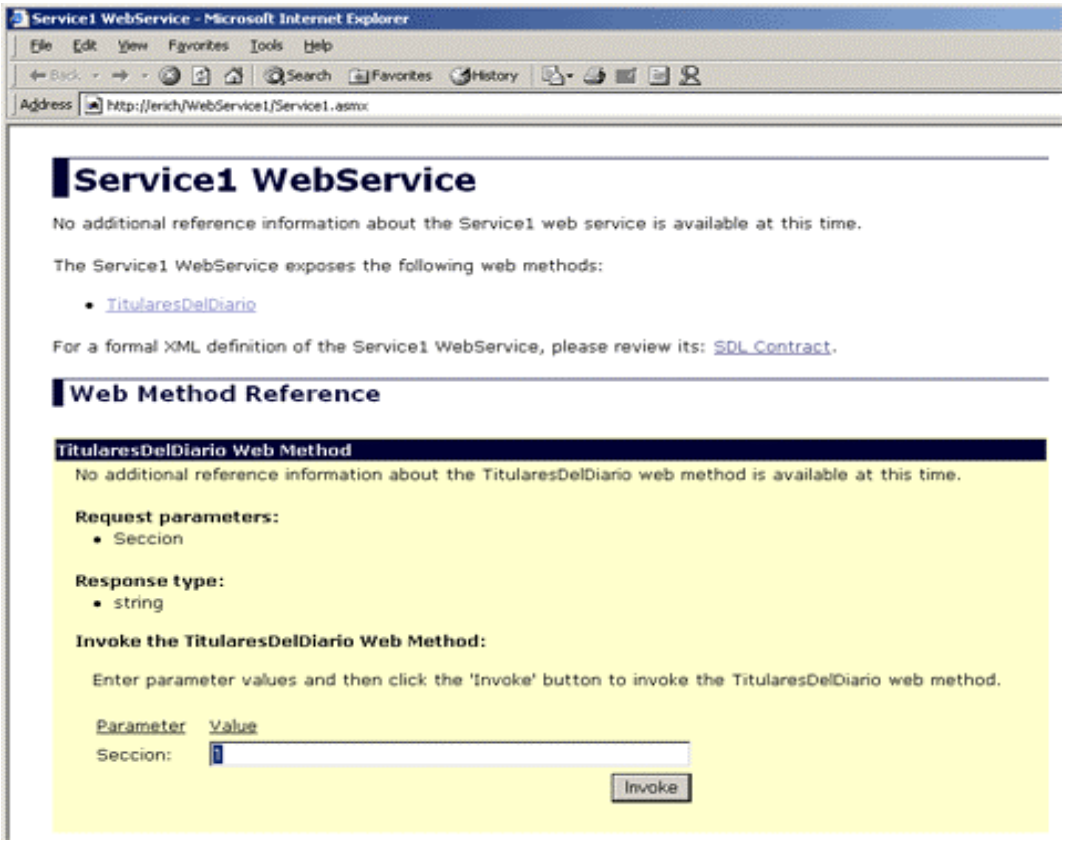
Esto significa un cambio radical, ya que ofrece acceso a cualquier aplicación que sea capaz de utilizar el protocolo *HTTP*. Adicionalmente, la información es transferida en formato texto, y más específicamente en *XML*.
XML es un formato de texto que cumple con ciertas reglas estándares, además de ser abierto, y de fácil interpretación por cualquier lenguaje o sistema operativo. Tiene características similares a *HTML*, pero a diferencia, *XML* describe solamente información, mientras que *HTML* describe presentación.

Otra ventaja, es que los servicios *Web* pueden ser ejecutados desde un navegador, una aplicación no-Windows© que tenga acceso *Internet*, o cualquiera que sea capaz de entender *XML*.
Veamos algunas características de ambos:

Bibliotecas *COM* estándares	Servicios Web
Están centradas en el paradigma Windows©	Están centradas en el paradigma *Internet*
Utiliza formato binario de interconexión	Utiliza *XML* como meta-lenguaje de comunicación.
Son principalmente utilizadas en entornos Windows©	Pueden ser interpretados por cualquier sistema operativo o lenguaje
Es muy óptimo en desempeño de comunicación	Es medianamente óptimo

Si lo vemos desde el punto de vista de *VB.NET*, la utilización de un servicio *Web* no difiere programáticamente de la utilización de una biblioteca estándar, aunque la biblioteca Web exhiba su interfase a través de Internet. Internamente, *VB.NET* se encarga de la comunicación, transformando el llamado a la biblioteca en *XML*, serial

izando y enviándola posteriormente, y por último ejecutando en forma remota, y enviando la respuesta de retorno.



Es luego *ASP.NET* quien se encarga de traducir la resultante a un formato entendible por *VB.NET* . Todo esto se realiza en forma transparente para usted, internamente utilizando un protocolo llamado *SOAP*, una página activa *ASP*, y la implementación de su biblioteca.

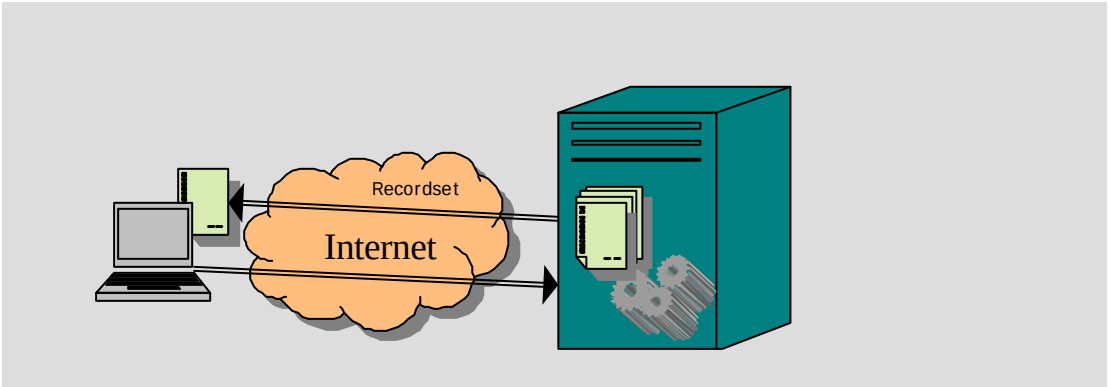
Por otra parte, la gestión de seguridad es mucho más fácil de manejar, ya que basta con restringir el acceso a la carpeta que contiene el servicio *Web* para controlar su accesibilidad.

La **integración de servicios y aplicaciones mediante los servicios *Web***, constituye un gran paso en el desarrollo de aplicaciones distribuidas. Usted debe visualizar la implantación mediante este paradigma como un gran puzzle: usted incorpora la funcionalidad que requiera y descarta aquella que no necesita.

De esta forma, **su aplicación consume servicios** que están a través de *Internet*, sin importar dónde se encuentre, ni quien es el proveedor.

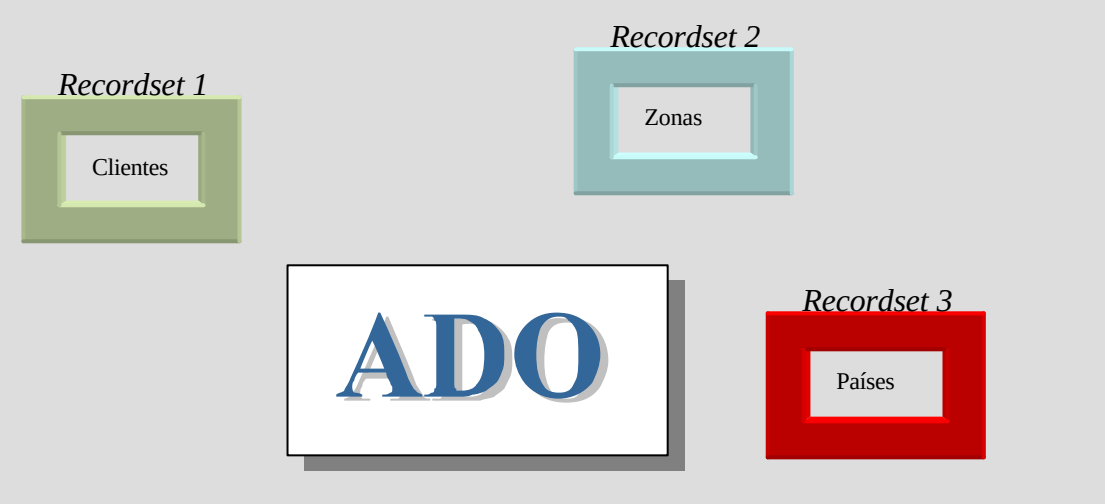
ADO.NET

Inicialmente el acceso a datos desde Visual Basic© se realizaba llamando a funciones propias del manejador de cada base de datos. Esto hacía que el desarrollador tuviese que aprender el conjunto de funciones que ofrecía cada proveedor.
Con el paso de las versiones de Visual Basic©, se instauraron diferentes tecnologías que hicieron mas fácil el acceso a datos.
La primer versión de estos objetos se denominó *DAO* (*Data Access Object*), y estuvo originalmente ideado para interactuar con bases de datos de Microsoft© Access. Desde Visual Basic© son exhibidas como un conjunto de clases que encapsulan las características necesarias para acceder a la estructura y al contenido de la base de datos, realizando así una abstracción de las funciones originales del manejador.
La necesidad de integración de Visual Basic© con motores de datos relacionales, generó que Microsoft© desarrollara un segundo conjunto de objetos denominado *RDO* (*Remote Data Access*), ideado para acceder principalmente a motores relacionales de datos. De esta forma el desarrollador accedía en forma optima a motores de datos, pero carecía de eficiencia a la hora de utilizar archivos de Microsoft© Access. Adicionalmente, RDO tenía ciertas características que variaban de motor a motor, y que hacían difícil construir una aplicación motor-independiente.
Fue así que hace algún tiempo atrás, se comenzó el desarrollo de un conjunto de tecnologías –entre las cuales se encuentra *OLE-DB*, el reemplazo de *ODBC*- y objetos mas avanzados, que tuviesen las mismas características de programación sin importar el motor de datos con el cual se deseara trabajar. Para ello se tomó lo mejor de las tecnologías *DAO* y *RDO*, y se produjo un conjunto de objetos –más fácil de utilizar y más eficiente que *RDO*- denominado ***ADO*** (*ActiveX Data Object*). La primer herramienta que lo incluyó fue Visual Interdev©, y mas tarde se agregó a todos los lenguajes de Visual Studio. Hoy en día, *ADO* es usado también por otros lenguajes no Microsoft©.
Alguna de las características de *ADO* ofrecen facilidades a la hora de utilizarlas en desarrollos distribuidos. Una de ellas, es la denominada **Cursores Desconectados** (*Desconected Recordset*), la cual permite a una aplicación conectarse a un origen de datos, obtener información de una tabla, y luego gestionarlos en forma desconectada. Al reanudar la conexión, es posible hacer efectivo los cambios realizados.

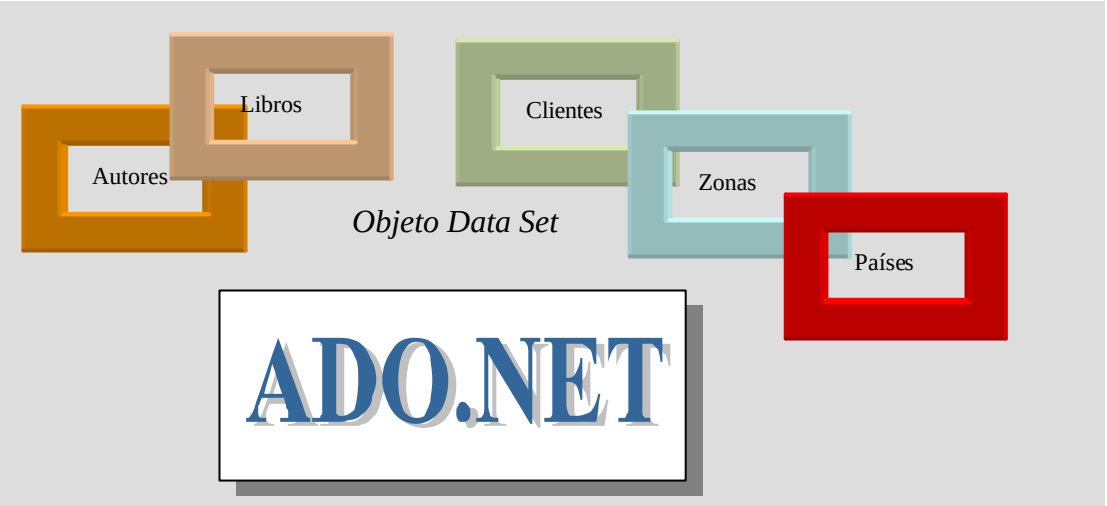


La gestión del cursor puede ser implementada por una aplicación *Win32*© o *HTML* (mediante Internet Explorer©), ya que ambas cuentan con bibliotecas de conectividad *ADO* incluidas. Por otra parte, la transferencia de información a través de una red se realiza en formato binario o *XML*, lo que lo hace sumamente eficiente.

Otra característica de *ADO*, es la de conectarse a cualquier sistema no relacional de datos, compatible con *OLE-DB* (Ej. archivos, correo electrónico, etc) mediante un esfuerzo mínimo.
Con el advenimiento de *Internet*, los sistemas requirieron ser cada vez mas distribuidos, interconectados a través de redes, las cuales en muchas ocasiones determinaban una velocidad de transferencia demasiado lenta, o con conexiones intermitentes. Debido a esto, se gestó la necesidad de mantener estructuras cada vez mas complejas en forma desconectada, siendo poco útil el manejo de tablas 'sueltas' como lo hacía hasta el momento *ADO* -sin la posibilidad de mantener relación de integridad entre ellas- aunque esta última podía ser realizada programáticamente.



La nueva versión de *ADO* denominada *ADO.NET* (antes *ADO+*), permite la manipulación de verdaderas estructuras relacionales complejas en forma desconectada.



Por otra parte, la transferencia de información de los cursores no se realiza mediante un formato binario. Si bien ésta puede ser mas eficiente, cuenta con la desventaja de no poder ser interpretado en forma fácil por sistemas operativos no Windows© Para

ello, *ADO.NET* utiliza como formato de transferencia el meta-lenguaje de marcas extendidas denominado *XML* o **eXtended Meta Language**.
XML es texto que cumple con ciertas reglas estándares de notación, y que permite la descripción de cualquier cosa.
Veamos un ejemplo de *XML*:

```
<?xml version="1.0"?>
<Order o_id="7845" o_date="2000-02-01" o_cur="$" c_id="JUAN" >
  <ShippingAddress sa_city="Erie" sa_state="PA" sa_pcode="19130">
    <Line l_text="Jueguetes Aleph" />
    <Line l_text="Almería 4759" />
    <Line l_text="3er Piso" />
  </ShippingAddress>
  <OrderDetails>
    <Item p_id="325" p_name="Peluche" od_qty="5" od_unitp="15"/>
    <Item p_id="326" p_name="Zapo" od_qty="2" od_unitp="10.2"/>
    <Item p_id="327" p_name="Pelota" od_qty="5" od_unitp="15.5"/>
  </OrderDetails>
  <Comments>
    Esta orden reemplaza la 9873.
  </Comments>
</Order>
```

Como vimos anteriormente, *XML* es fácil de leer y de procesar por cualquier sistema que capaz de leer ASCII, por lo que significa que cualquier sistema operativo puede interpretar la información. De esta forma, un conjunto de registros obtenidos mediante *ADO.NET* en la plataforma Windows©, pueden ser recepcionados por otras plataformas (*Linux, Unix, etc.*) leyendo simplemente el documento *XML*.

Evidentemente las nuevas características de *ADO.NET* van a brindarle varias ventajas en sus aplicaciones distribuidas. Veremos mas en el capítulo 7.

La realidad Distribuida es Internet

Para completar nuestro entendimiento de la arquitectura *.NET*, debemos dar un breve vistazo a las posibilidades existentes, así como sus problemas y soluciones que estas plantean. Muchas de los paradigmas actuales, ofrecen soluciones a problemas comunes, pero traen como consecuencia un conjunto de restricciones. Hagamos un poco de historia...

Visual Basic© como lenguaje alternativo

En los comienzos, Microsoft© Visual Basic© surgió como un lenguaje alternativo para palear las dificultades que otros lenguajes de bajo nivel –como el caso de Microsoft© Visual C++, etc- tenían a la hora de crear aplicaciones con entorno visual para Windows©.
Así fue que surgió Microsoft© Visual Basic©, un lenguaje elegido por Microsoft© debido a que no era necesario el pago de derechos de autor (*royalties*) para su utilización y/o distribución. Las primeras versiones fueron concebidas como una herramienta de diseño de interfases, programación, y acceso a datos. En algunos casos se ofrecía reutilización, pero solamente mediante bibliotecas implementadas en otros lenguajes, y compatible binariamente con Microsoft© Visual Basic©.

Alrededor del 95 % de las aplicaciones existentes acceden de alguna forma a datos, por lo que Microsoft© apostó y apuesta constantemente a mejorar y simplificar la conectividad a datos. Las aplicaciones distribuidas han cambiado la forma en la que se desarrollan e instalan los programas. La idea principal, es la de separar una aplicación en partes, a los efectos de hacer más fácil su reutilización y actualización.

"Divide y reina"

Un concepto importante a tener en cuenta, es que el mundo de los negocios y el mundo de la informática trabajan en conjunto, y en muchas ocasiones es difícil acompasar los cambios que el mercado fuerza sobre las aplicaciones informáticas, con la velocidad exigida. Generalmente, un cambio se traduce en un requerimiento sobre una aplicación, y el lenguaje elegido y la modalidad de trabajo, deben cumplir con un buen tiempo de respuesta.

La división en servicios

Como vimos anteriormente, las tecnologías *.NET* ayudan a que una aplicación pueda ser particionada. La división conceptual de una aplicación, ha hecho más fácil la comprensión y el desarrollo, así como la organización de los recursos en una empresa informática.

Generalmente esta división se realiza en 3 partes:

Presentación
Lógica
Datos

Cualquier aplicación es factible de separarse en estas tres partes o capas. Llamamos a esto "conceptual", ya que en muchas ocasiones estas capas pueden no estar relacionadas con la estructura física.

Capa de Presentación

La primer capa es la denominada capa de Presentación, y es aquella parte de la aplicación (servicios y/o funciones) que se encarga de dibujar la interfase gráfica o texto de la aplicación

Capa de Negocios

La segunda capa denominada capa de Negocios o lógica, es aquella parte de la aplicación (servicios y/o funciones) que realizan la parte lógica. La lógica de negocios es el conjunto de rutinas que realiza y representa los procesos lógico-funcionales de una empresa.

Capa de Datos

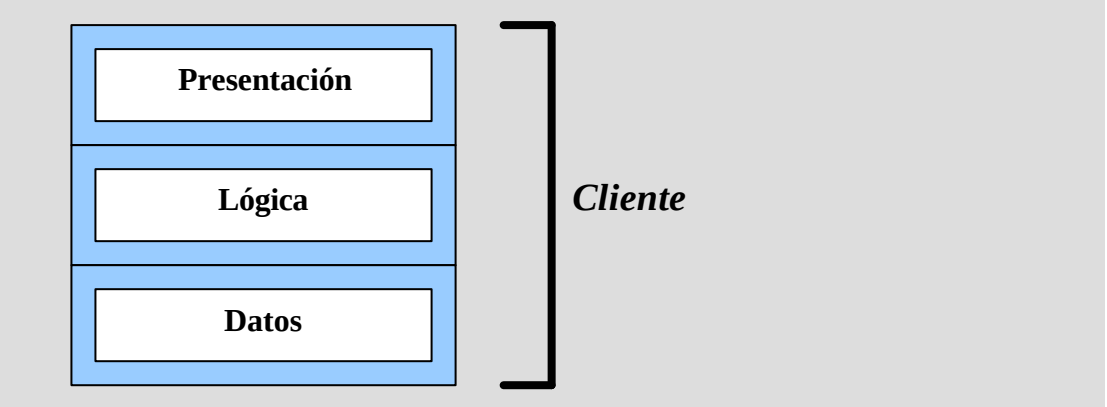
La tercera capa es la de datos, y es la encargada de proveer el acceso y manejo de información (en algunos casos persistente, Ej. Bases de datos, archivos, etc.).

Así es que una misma aplicación puede ser dividida en 3 partes. Es importante como desarrollador, que cuente con un conjunto de tecnologías específicas que den soporte a cada una de las partes (capas), y permitan tratar a cada una en forma concreta. En Visual Basic© .NET se incluye el soporte de tecnologías necesario para cubrir cada una de las capas, logrando así que puedan interactuar cooperativamente aunque estén distribuidas en diferentes espacios físicos (servidores).

En algunas, ocasiones no es necesario partir la aplicación en 3 partes (capas) físicas, pudiéndose tener en la misma aplicación en una capa física.

Aplicaciones en 1 capa: 3 servicios en 1

La aplicación en una capa es la forma más antigua de representar una aplicación, y es la más monolítica. La aplicación en 1 capa contiene las 3 partes lógicas en el mismo aplicativo. Las aplicaciones de escritorio, son generalmente un buen ejemplo de aplicaciones en 1 capa. Habitualmente en el mismo aplicativo (ejecutable) se concentran los servicios encargados de generar la interfase gráfica (pantallas), realizar la lógica, y acceder a datos. Una plantilla de Microsoft© Excel puede ser utilizada en uno o más equipos, pero incluye en sí misma toda la funcionalidad, sin la posibilidad particionar los servicios en presentación, lógica, o datos.

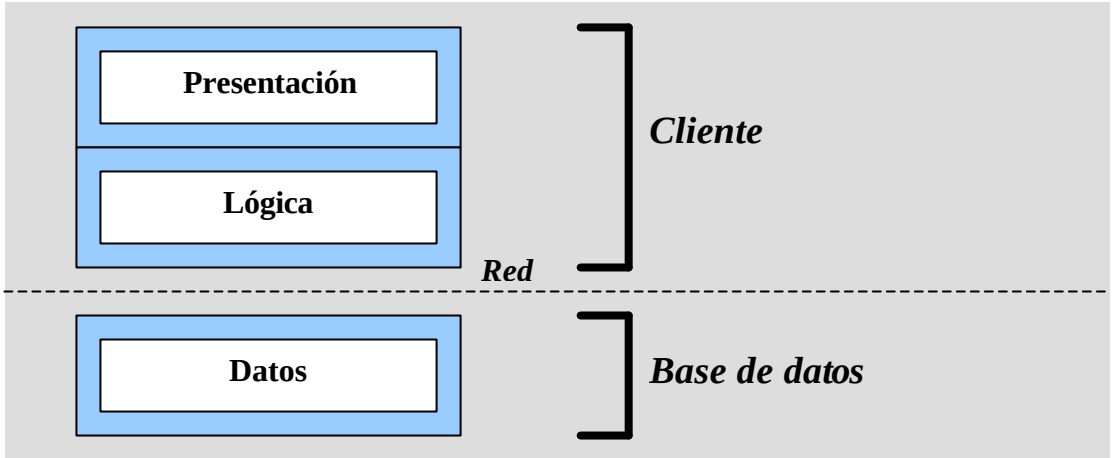


Aplicaciones en 2 capas

Las aplicaciones en 2 capas (también llamadas cliente/servidor), agrupan los servicios de presentación y lógica en la máquina cliente, y acceden a una fuente de datos compartida. En algunos casos, parte de la lógica puede ser puesta en la capa de datos, a los efectos de poder compartir lógica desde varias aplicaciones.

Un buen ejemplo de una aplicación en 2 capas es una aplicación bancaria. En este modelo, los servicios de presentación y lógica están locales, pero al realizar una transacción de dinero, el resultado será almacenado en una base de datos compartida por todas las sucursales, atendido por un servidor exclusivo de datos.

Este tipo de arquitectura tiene como ventaja la centralización de información y funciones, facilitando la actualización, mejorando la consistencia, y eliminando la duplicación de información.



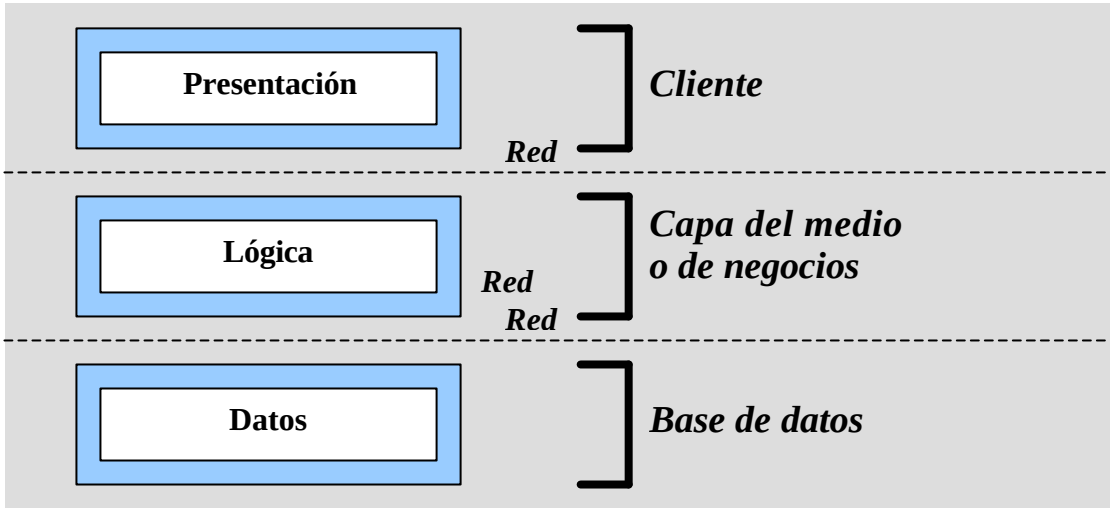
Las aplicaciones en 2 capas están aptas para empresas que no tengan una altísima demanda de información, ya que esto podría derivar en una saturación del servidor de datos, haciendo de este paradigma inservible.

A menudo, parte de la lógica de la aplicación se introduce en la base de datos en **procedimientos almacenados**, como forma de compartir código entre aplicaciones. Pero ello obliga a las empresas a contar con personal capacitado para escribir las rutinas compartidas en el lenguaje específico del motor de base de datos.

Aplicaciones en 3 capas

Las aplicaciones en 3 capas (también llamadas **distribuidas**) ofrecen actualmente los mejores beneficios para desarrollos de gran porte. Como lo dice su nombre, estos separan su funcionalidad en 3 partes. Los servicios que generan la interfase gráfica se sitúan en la capa presentación, mientras que los servicios que realizan la lógica se concentran dentro de la capa de negocios, y por último, los servicios de información en la capa de datos.

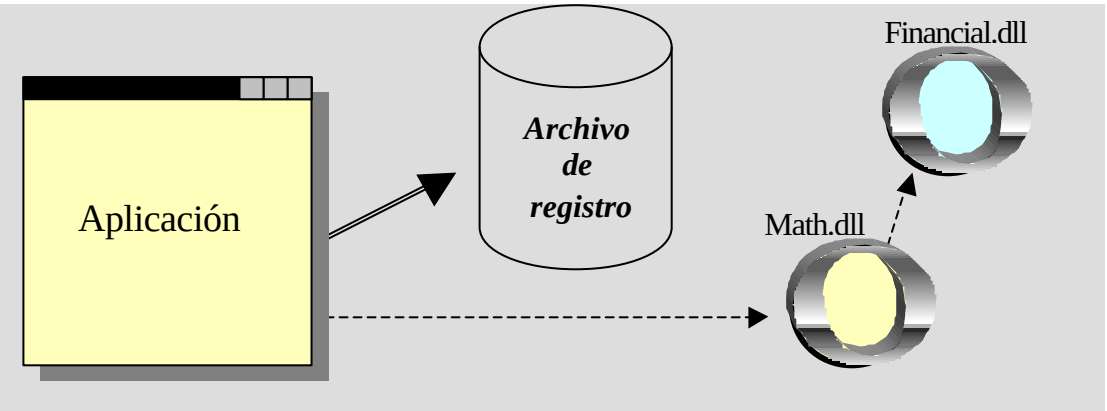
Las ventajas de las aplicaciones en 3 capas, son la de poder ser distribuidas a través de sistemas heterogéneos. A su vez, las aplicaciones distribuidas funcionan como piezas de encastrar, en donde solamente se agregan aquellos servicios que se van a utilizar.
Un ejemplo de aplicaciones distribuidas son las aplicaciones de *Internet*, donde la capa de presentación pertenece a una página *HTML* corriendo dentro de un navegador, mientras que la capa del medio (Negocios) contiene a la lógica específica del aplicativo, y por último la capa de datos realiza la persistencia de información. Las aplicaciones distribuidas tienen la ventaja de ser fáciles de instalar y actualizar, ya que el componente existe en un solo lugar, sin necesidad de tenerlo en varias partes. Por otro lado, es el paradigma que ofrece mayores facilidades para compartir funcionalidad entre aplicaciones.



Si se va a trabajar en entornos Microsoft©, cuenta con un marco de trabajo denominado *Microsoft© Distributed Internet Applications* o *DNA*. *DNA* constituye para Microsoft© un conjunto de tecnologías desarrolladas por él, para realizar aplicaciones distribuida. Actualmente *DNA*, es el paradigma que mas se ve afectado por la nueva versión de Visual Basic©.

Hablemos de COM y Visual Basic©

Hace varios años, Microsoft© implementó un estándar de comunicación entre objetos denominado *Component Object Model* (vulgarmente conocido como *COM*). *COM* hace posible que una aplicación o biblioteca pueda llamar a otra sin preocuparse por aspectos físicos (ruta, protocolo de red, etc), o incluso del lenguaje en que ésta haya sido construida. Para ello, requiere que la interfase de un componente (definición de funciones, procedimientos, y eventos) sea almacenada en el archivo de registro, mientras que se mantiene la implementación de la biblioteca en un archivo físico.



COM puede ser visto como un contrato entre una aplicación cliente y una aplicación servidora de intercomunicación. De esta forma, la mayoría de aplicaciones comparten bibliotecas de otros aplicativos o del sistema operativo en forma natural. Por otra parte, el sistema operativo utiliza internamente y mayoritariamente esta tecnología.
Si bien cuenta con muchas ventajas, en algunas ocasiones *COM* ofrece un marco de trabajo frágil, que obliga al desarrollador –por ejemplo- a preocuparse de la gestión de las referencias de los objetos.

Generalmente –y dependiendo del lenguaje- podemos encontrar los siguientes pasos:

1. Se solicita una referencia a un objeto
2. Se asigna memoria para el objeto
3. Se carga el objeto en memoria y se inicializa
4. Se utiliza
5. Se destruye la referencia
6. Se libera la memoria

Es común que un mismo objeto sea utilizado al mismo tiempo por varias aplicaciones, esto quiere decir que varias referencias pueden apuntar a la misma biblioteca desde la misma o diferentes aplicaciones. Debido a ello, es importante que el sistema operativo pueda determinar si debe o no liberar la memoria ocupada por el componente.
Para resolver esto, *COM* utiliza una operativa sencilla, manteniendo un contador de referencias a un objeto. Cada vez que se solicita una referencia a un objeto, se incrementa el contador asociado a esa biblioteca, y cuando la referencia es liberada (asigna *Nothing* a la variable) se decrementa la cuenta. Si el contador llega a cero se libera la memoria, puesto que esto significa que nadie está utilizando el componente.

Lamentablemente este esquema no es muy claro en muchos casos, generando que el desarrollador tome decisiones erróneas de cuando liberar o no una referencia.
Veamos el siguiente ejemplo en una versión 6.0 de Microsoft© Visual Basic©.

```
Sub Test()
'Declara una variable de la clase especificada.
Dim MiVariable as Objeto.Clase

'Instancia la clase, e incrementa el contador de referencias a 1
Set MiVariable = new Objeto.Clase

'Se libera la referencia, y se decrementa el contador. Si el
contador es igual a 0 se libera la memoria
Set MiVariable = Nothing

End Sub
```

En la primer línea se define un objeto de la clase especificada, mientras que en la segunda línea se solicita una referencia al objeto y se inicializa. A continuación se accede a sus métodos y propiedades. Cuando este no está siendo utilizado, es responsabilidad del programador la liberación de la referencia.
Si bien es un método sencillo, puede olvidar la última línea que libera la referencia. ¿Qué pasa si olvida la línea de liberación de referencia?

```
Sub Test()
'Declara una variable de la clase especificada.
Dim MiVariable as Objeto.Clase

'Instancia la clase, e incrementa el contador de referencias a 1
Set MiVariable = new Objeto.Clase

End Sub
```

La utilización de la palabra *Nothing* no es necesaria en este caso, ya que al finalizar un procedimiento las variables cesan su alcance, destruyéndose así las referencias, y por ende liberando la memoria.

Si bien veremos mas adelante las referencias circulares, veamos un ejemplo de impacto con referencias circulares:

```
Sub Test()

'Declara una variable de la clase especificada.
Dim X as Objeto.Clase
Dim Y as Objeto.Clase

'Instancia la clase, e incrementa el contador de referencias a 1
Set X = new Objeto.Clase
Set Y = new Objeto.Clase

Set X.MiRef = Y
Set Y.MiRef = X

Set X = Nothing
Set Y = Nothing
End Sub
```

En este ejemplo, -y pese a la correcta utilización de la palabra *Nothing*- los el contador de referencias de X e Y no son nunca liberadas, ya que constituyen referencias circulares (X aunta a Y e Y a X).

Veamos otro ejemplo:

```
'Declara una variable de la clase especificada.
Public A as Objeto.Clase

Sub Test()
Dim Z as Objeto.Clase

'Instancia la clase, e incrementa el contador de referencias a 1
Set Z = new Objeto.Clase

'Incrementa contador de referencias a 2
Set A = Z

'Decrementa contador de referencias a 1
Set Z = Nothing
End Sub
.
.
Referencia = 1
```

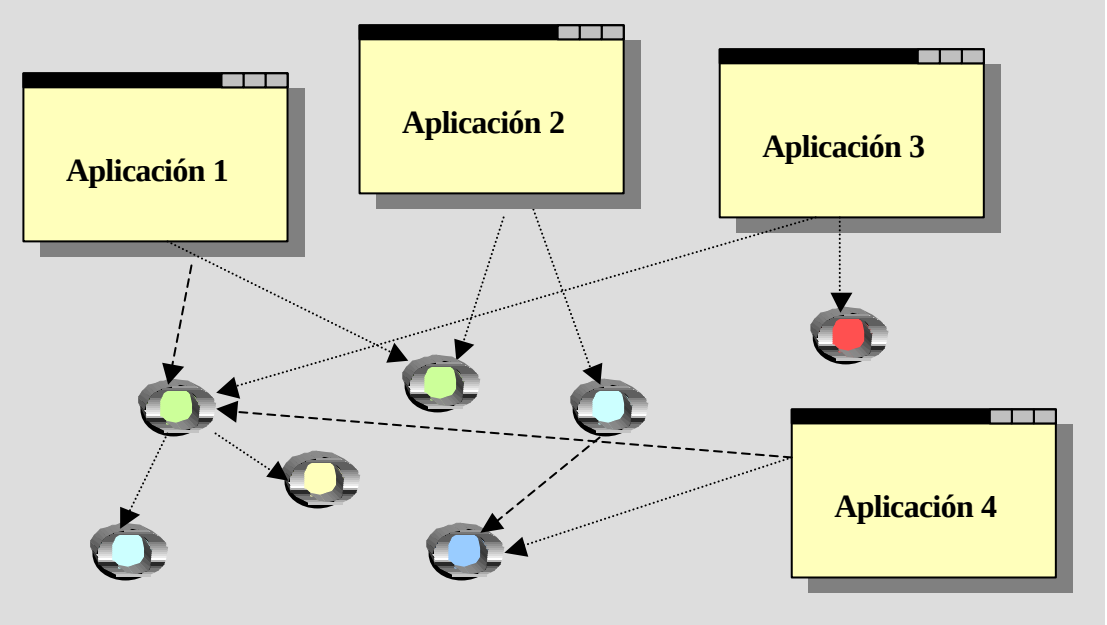
En este caso si es útil la utilización de la palabra *Nothing*, ya que de lo contrario, la referencia a la variable global será destruida al finalizar la aplicación, pero no se liberará la memoria ya que el contador no es decrementado.

Si bien el manejo de la palabra *Nothing* en Visual Basic© puede ser sencillo, en muchos casos el umbral de utilización no esta demasiado claro, y uno puede omitirsee esta línea en un caso indispensable, llevando así a que no se libere la memoria

solicitada para un objeto. Si a esto le sumamos que existen cientos de aplicaciones y bibliotecas en su sistema operativo utilizando *COM* para comunicarse, podemos tener un panorama que puede llegar a ser problemático. Veremos mas adelante, como la nueva tecnología de *Colector de Basura* resuelve este problema.

En el infierno DLL

El infierno *DLL* refiere a un conjunto de problemas causados por la utilización de bibliotecas compartidas. En general, la instalación de una aplicación consta de varios pasos, y típicamente envuelve copiar un número de componentes al disco, y crear un conjunto de información en el archivo de registro que describe los componentes instalados. De esta forma, existe una separación entre las entradas en el archivo de registro describiendo el componente, y los archivos físicos (que son los que contienen la implementación). Esto hace difícil de actualizar y desinstalar la aplicación, ya que se debe tener coordinado el archivo de registro con los archivos físicos.
La tecnología utilizada por Microsoft© en sus sistemas operativos tiene como característica la de centralizar en una base de datos la información de todas aquellos objetos, bibliotecas (Software), o configuraciones de Hardware, que el sistema operativo puede utilizar. Esta centralización tiene como objetivo la existencia de un repositorio único de información. Adicionalmente, cada aplicación o biblioteca es vista por el sistema operativo como un objeto, factible de registrar en la base de datos del sistema (también denominado archivo de registro o *registry*).
Cada aplicación puede constar de un solo ejecutable, así como también de un conjunto de ejecutables y bibliotecas relacionadas entre sí.



Cuando la aplicación comienza su ejecución, el sistema operativo debe localizar los componentes necesarios para su funcionamiento, y es allí donde interviene la información del archivo de registro.
La capacidad de compartir las llamadas **bibliotecas de enlace dinámico** o *Dynamic Link Libraries*, es la principal forma que tienen los lenguajes Microsoft© de re-utilizar un conjunto de funciones. Una biblioteca de enlace dinámico es un archivo binario factible de re-utilizar, pero sin necesidad de ser incluido dentro del mismo ejecutable.
Inicialmente, Windows© comenzó creando bibliotecas de enlace dinámico con el fin de ser compartidas solamente entre sus aplicaciones, posteriormente se agregaron algunas para ser utilizadas por desarrolladores.

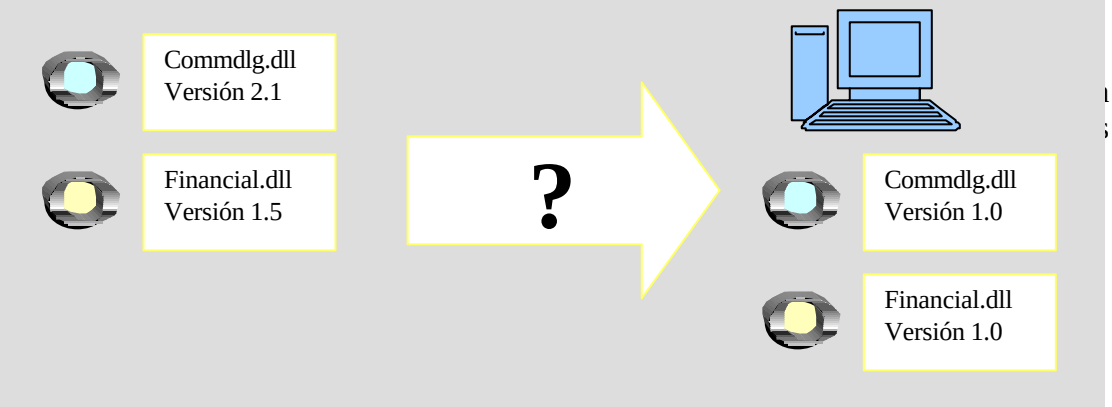
Estas bibliotecas contienen grupos de funciones y procedimientos, eliminando así la necesidad de implementar la misma funcionalidad en forma independiente. Utilizar una biblioteca con funciones reutilizables en un entorno Microsoft© es tan fácil como incluir las rutinas dentro de la misma aplicación.

Distribución de una aplicación

Al distribuir una aplicación se debe tener en cuenta incluir todas aquellas bibliotecas que su programa utilice, aunque en algunos casos sean compartidas o de propiedad de otras aplicaciones.

Un ejemplo de esto es la biblioteca *commdlg.dll*. Esta biblioteca define un grupo de cajas de diálogos comunes que pueden ser utilizadas por cualquier aplicación Windows©. Generalmente, está incluida en el sistema operativo por lo que no es necesario agregarla dentro del paquete de distribución.

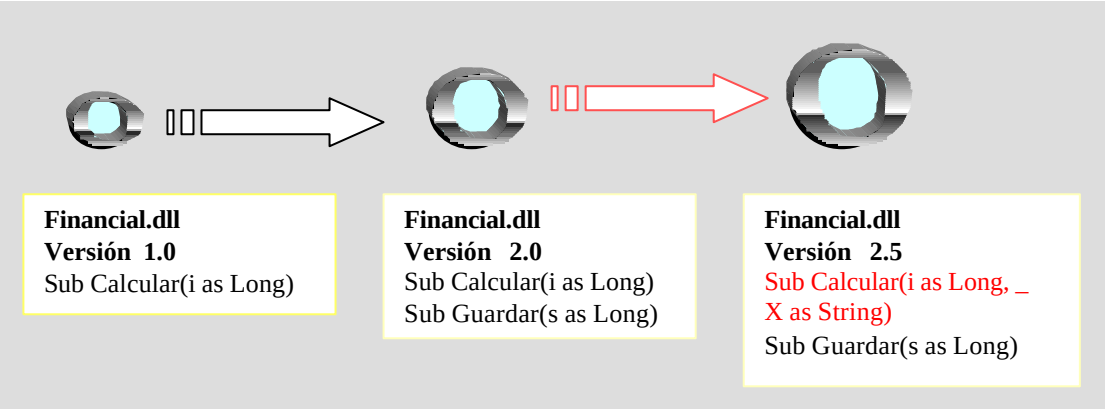
Microsoft© ha ido actualizando esta biblioteca, generando en muchas ocasiones que la versión de biblioteca que se encuentra en el sistema operativo sea diferente a la que su programa espera.



La forma mediante la cual COM resuelve esto, es mediante el recurso de versión.

Recurso de versión

Cada biblioteca incluye información denominada **recurso de versión**. Esta información de versión es utilizada por el sistema operativo para conocer las características de la misma.
Aunque no entraremos en detalle en esta utilidad, es bueno que conozca que la primer versión de una biblioteca es generalmente 1.0, y se ve incrementado en cada compilación que pueda afectar su interfase.



El archivo de recurso es consultado automáticamente para saber si la versión de biblioteca a utilizar es compatible con la esperada, o si la versión de biblioteca que existe en el paquete a instalar es mas nueva que la ya existente en el sistema operativo, y por ende se debe sobre-escribir.
Si bien se puede asegurar que una versión más actual de biblioteca contenga un conjunto de interfases que coincidan con la anterior, no puede aseverarse que la nueva implementación sea totalmente compatible, por lo que la actualización de una aplicación que utiliza bibliotecas compartidas puede ocasionar la modificación o colapso de características de otra aplicación.
Si bien en mucho casos las aplicaciones son probadas con diferentes versiones de bibliotecas comunes, no siempre se puede asegurar que el comportamiento será totalmente compatible.

La solución basada en componentes reutilizables tiene un efecto sobre la distribución de aplicaciones (en nuestro caso Visual Basic©). Si bien la generación de paquetes de instalación se hace en forma automatizada mediante asistentes, en muchos casos puede resultar caótica. En el mundo real contamos con miles de archivos compartidos *OCX, VBX, DLL,* que pueden ser utilizados por cientos de aplicaciones. En caso que tan solo una biblioteca faltara o se presente en una versión incompatible a la esperada, haría que una o varias aplicaciones fallaran o se comportaran en forma errática. Generalmente la falla es difícil de identificar, y en muchos casos los síntomas suelen ser un error de protección general o una excepción de memoria.

Con el advenimiento de los sistemas operativos Windows© 2000, se implementó la posibilidad de poder copiar una determinada biblioteca en un directorio específico, a los efectos de ser utilizada por una única aplicación (veremos mas de esto en la sección de **Ensamblados**) Si bien esta funcionalidad mejora el marco de trabajo mediante la utilización de bibliotecas privadas, en muchos casos ocasiona que si la

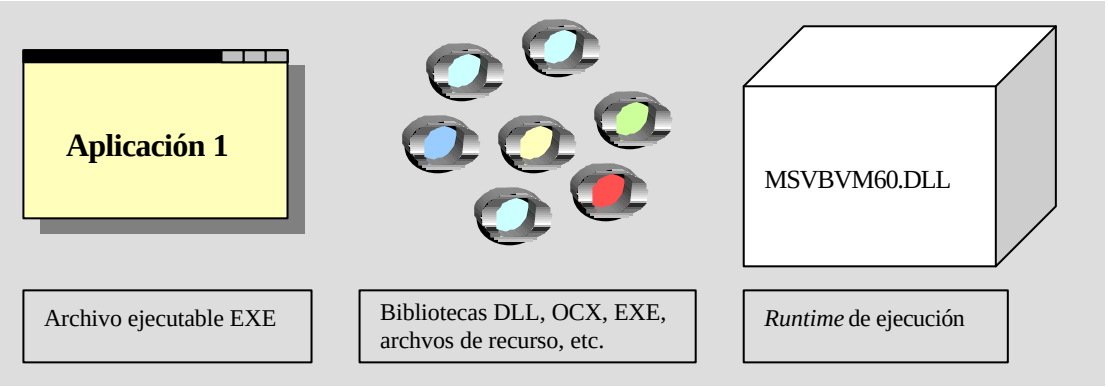
biblioteca no es compatible el problema puede ser peor aun, ya que hay que evaluar el orden en que las aplicaciones son ejecutadas, o a la última versión de componente que fue registrada, generando así fallas intermitentes y de difícil localización. En muchas ocasiones el tiempo que toma detectar la falla real no justifica, y se termina por re-instalar el aplicativo o sistema operativo.

Veremos mas adelante como la plataforma *.NET* de Microsoft© resuelve esta problemática, mediante cambios sustanciales en la estructura y forma en que se distribuye una aplicación.

Dime en que programas y te diré quien eres

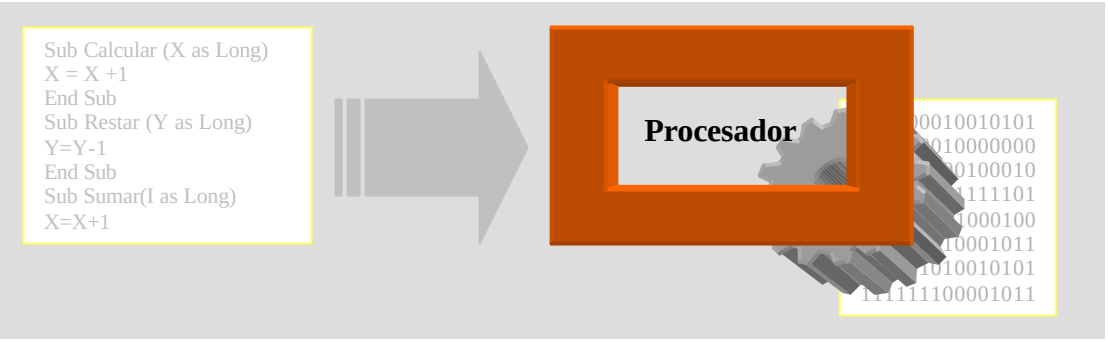
Como vimos anteriormente, la plataforma *.NET* ofrece un conjunto de tecnologías y características compartidas por todos los lenguajes Microsoft©. En esta nueva versión, usted deberá pensar en reestructurar sus aplicaciones, y no tanto en una simple actualización tecnológica Es evidente que no en todas las aplicaciones son factibles de migrar, y dependerá de usted el decidir en cada caso si las tecnologías lo benefician.

Cuando hablamos de una aplicativo Visual Basic©, generalmente nos referimos a una aplicación principal –compilada- con extensión EXE, algunas bibliotecas (extensión .DLL), así como controles Activex (.OCX), además de la biblioteca principal (*Runtime*) de Visual Basic©. Este último se extiende desde las versiones mas antiguas de Basic, y es el encargado de darle vida a las aplicación, sin ella la ejecución fallaría..



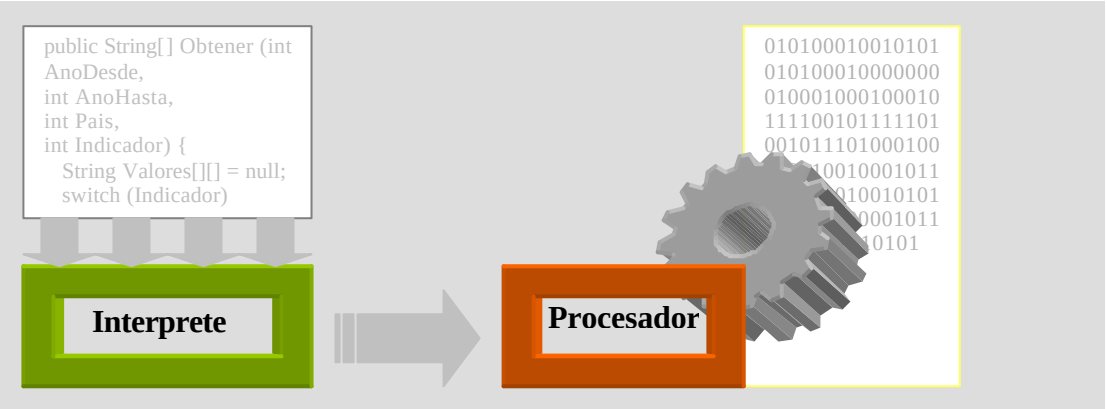
A grandes rasgos, cuando se ejecuta la aplicación, el sistema operativo carga el ejecutable compilado en memoria, analiza los vínculos (bibliotecas necesarias), y en caso de ser requerido las carga también, ejecutando posteriormente la aplicación.

Visto que el ejecutable y las bibliotecas están compiladas en lenguaje máquina, puede ejecutarse directamente sobre el procesador sin necesidad de interpretes (aunque en algunos casos, necesite de las funciones situadas en el la biblioteca principal o *Runtime*).



En otros lenguajes como *JAVA*, el proceso es significativamente diferente debido a su arquitectura. Cuando *Java* fue concebido, se pensó en un lenguaje capaz de traspasar las fronteras de un sistema operativo o procesador. Debido a que un ejecutable *JAVA*

no es compilados en lenguaje máquina, es posible su ejecución en otras plataformas diferentes a Windows©. Visto que *Java* es un lenguaje interpretado, sus rutinas son compiladas en un lenguaje intermedio llamado *ByteCode*. Este es un conjunto de instrucciones factibles de ser interpretadas por una **Máquina Virtual** o **Interprete**. El interprete implementa un procesador mediante *Software*, y virtualmente una aplicación *JAVA* puede correr sobre cualquier sistema operativo que implemente este Interprete.



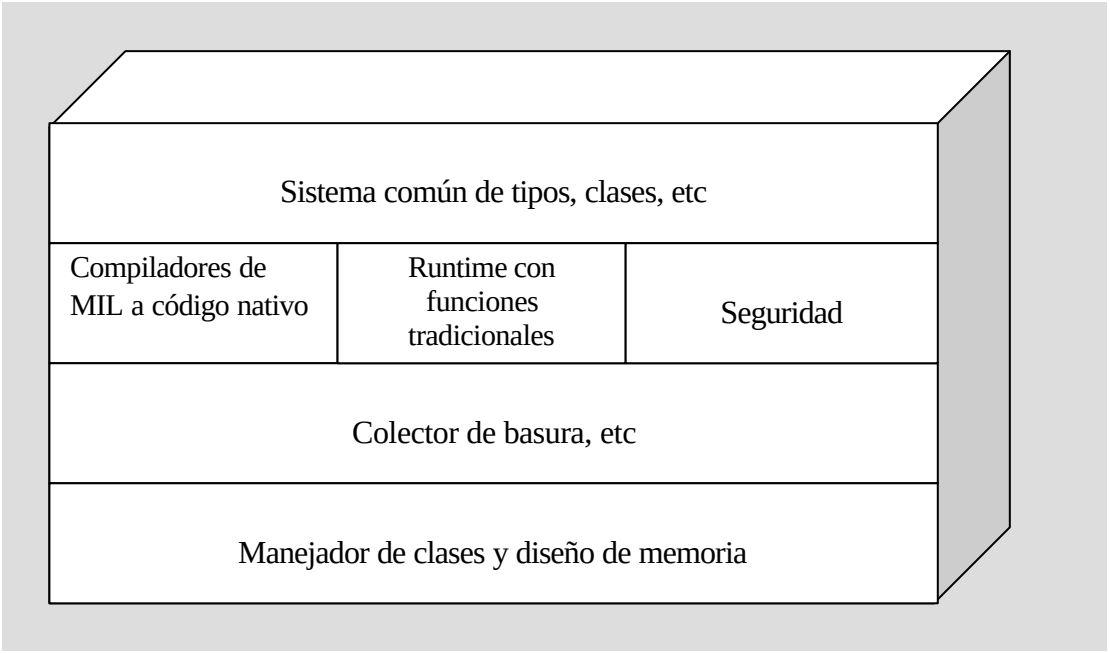
Una vez compilada la aplicación, el proceso de carga es sustancialmente diferente al del código de compilado real. La aplicación se carga en memoria, y allí entra en acción el Interprete que traduce cada instrucción del *ByteCode* a código máquina.
Si bien el compilado real tiene la ventaja de ofrecer mejor performance debido a que interactúa directamente con el procesador, el compilado a *ByteCode* tiene la posibilidad de correr bajo cualquier sistema operativo y procesador que implemente el Interprete. De esta forma ofrece mayor control y seguridad, pero a costo de una pobre performance en comparación con la compilación real.

Usted se preguntará que relación tiene la tecnología *JAVA* con la plataforma *.NET*. Bueno, si bien la tecnología *.NET* adquiere personalidad propia, hay muchas de las características de *JAVA* que nos van a ser útiles a la hora de entender el nuevo conjunto de tecnologías que encapsula la palabra .NET, entre ellas la biblioteca común a los lenguajes ***CLR*** *(Common Language Runtime).*

¿Qué es Common Languaje Runtime?

La plataforma *.NET* es un conjunto de tecnologías que facilitan y amplían el espectro de su desarrollo. Esta versión de Visual Basic© cambia radicalmente la metodología de ejecución, ya que cuenta con un motor de ejecución llamado *CLR (Common Language Runtime)*. Este motor de ejecución esta situado entre su aplicación y el procesador. A diferencia de otras tecnologías similares, el ***motor o Interprete*** es compartido por todos los lenguajes que integran la plataforma .NET (VB,C, C# -un nuevo lenguaje de la familia Microsoft©-, etc).

El código escrito para el la plataforma *.NET* es ejecutado sobre el control del **Interprete** de ejecución común. *CLR* ha sido instrumentado para reemplazar, ampliar, o mejorar servicios y tecnologías existentes, como ser *COM, Microsoft© Transaction Server, Microsoft© Message Queue, y COM+*.



Al utilizar el motor compartido, todos los lenguajes ofrecen características y facilidades comunes, como ser el soporte para orientación a objetos, el **colector de basura**, y el manejo de **ensamblados** (el cual vera en este capítulo).

Código Manipulado

El código manipulado es aquel que está compilado para ser traducido por el motor *CLR*. El código construido en las versiones anteriores de Visual Basic© es considerado **Código No Manipulado** (o *Unmanaged Code*), debido a que interactúa directamente con el procesador. Si bien otros lenguajes de *.NET* pueden generar código no manipulado, Visual Basic© .NET es capaz de generar sólo **código manipulado**. De esta forma, cualquier biblioteca o ejecutable producido en esta nueva versión correrá bajo el dominio y control del motor *CLR*:
Veamos algunas diferencias entre la versión 6.0 y .NET de Visual Studio:

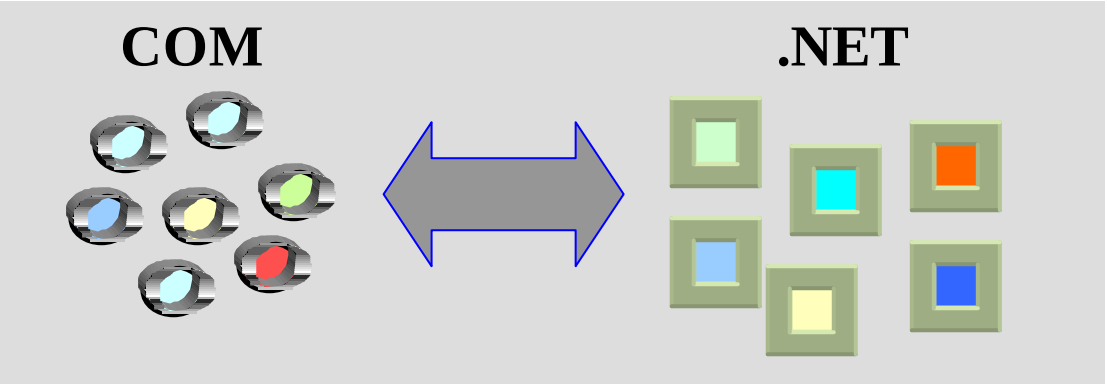
	Windows© NT 4.0 y VB6	**Windows© 2000 y VB6**	**Plataforma .NET**
Código	Código **no manipulado**, producido por VB 6.0	Código **no manipulado**, producido por VB 6.0	Código **manipulado** de Visual Basic© .NET
Capa específica del lenguaje	Utiliza la biblioteca principal de Visual Basic© MSVBVM60.DLL	Utiliza la biblioteca principal de Visual Basic© MSVBVM60.DLL	Utiliza motor de CLR MSCORE.DLL MSCORLIB.DLL
Contexto Concurrencia Transacciones	Utiliza la biblioteca de Microsoft© Transaction Server MTXEX:DLL	Utiliza bibliotecas de COM+ OLE32.DLL OLEAUT32.DLL	Utiliza motor de CLR MSCORE.DLL MSCORLIB.DLL
Manipulación de clases	Utiliza bibliotecas de COM OLE32.DLL OLEAUT.DLL	Utiliza bibliotecas de COM+ OLE32.DLL OLEAUT32.DLL	Utiliza motor de CLR MSCORE.DLL MSCORLIB.DLL

Usted se preguntará que ventajas tiene la de reemplazar una tecnología que produce ejecutables que interactúan directamente con el procesador, a una tecnología que requiere un seudo compilador para funcionar. Veremos la respuesta a continuación.

CLR fue diseñado con el objetivo de brindar un nivel sin precedentes de integración entre lenguajes, ya que el código compilado para éste sigue un conjunto de reglas comunes. Visto que *CLR* es totalmente orientado a objetos, los lenguajes heredan esta capacidad. Ahora todo en Visual Basic© es considerado un objeto, y contiene las características inherentes, como ser herencia, sobrecarga, polimorfismo, etc. A su vez, se reemplaza el manejo de errores por un esquema de captura de errores orientado a objetos. Todos los lenguajes heredan las características de *CLR*, por que lo estos se ven sustancialmente modificados, a los efectos de adaptarse a él.

Cuando usted compila una aplicación en *.NET*, el resultado no será código máquina, sino que será un metalenguaje denominado **MSIL** o **IL** *(Microsoft© Intermediate Language)*. **MSIL** es un formato compilado, que contiene instrucciones de bajo nivel que pueden ser solamente entendidas por el compilador incluido en *.NET* llamado *JIT (Just in Time Compiler)*. A diferencia de *JAVA*, cuando se ejecuta una aplicación compilada para *CLR*, *JIT* verifica que el código *MSIL* sea completamente seguro (esto quiere decir, que no realice conversión de punteros en forma ilegal, o todo aquello que pueda ocasionar un error de falla de memoria), y a continuación lo transforma en código máquina y lo ejecuta.

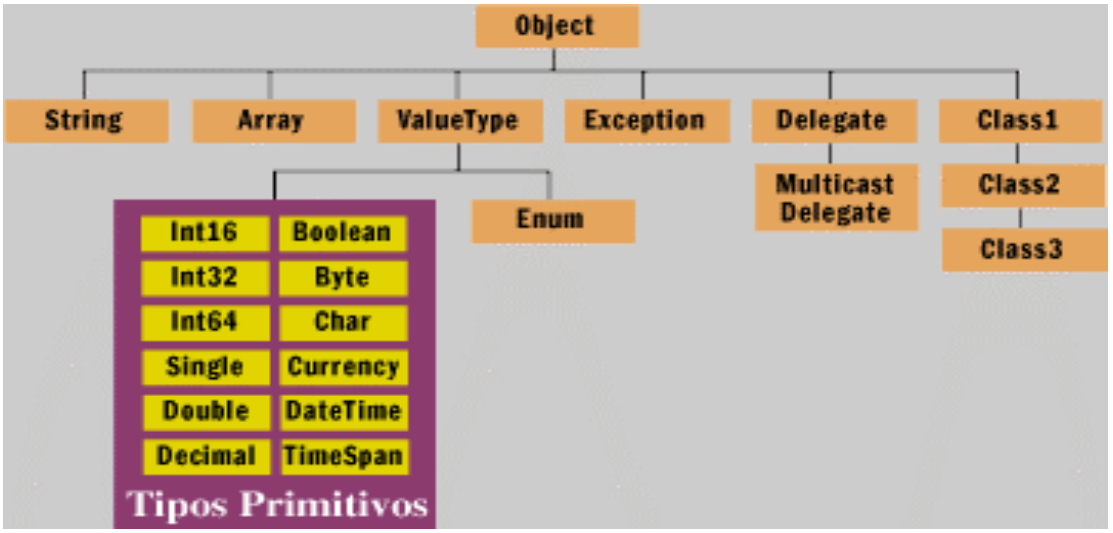
De esta forma se asegura un buen rendimiento, ya que el código final sigue siendo código máquina y no interpretado. A su vez, el código controlado puede interactuar con código no controlado, como ser un control *ActiveX* o biblioteca realizada en una versión anterior a la 7.0.
Visto que *.NET* tiene su propia forma de manejar componentes, descarta la utilización de C*OM*, pero igualmente puede utilizar componentes que soporten esta tecnología.



De hecho, *.NET* es tecnología similar a *COM* en algunos aspectos, pero con sus características negativas resueltas.

Interacción entre lenguajes

En algunas ocasiones, los desarrolladores *C++* producían bibliotecas que utilizaban características no compatibles con *VB*. En estos casos, los programadores Visual Basic© perdían la posibilidad de acceder dichas bibliotecas, provocando muchas veces la reprogramación de las mismas. Uno de los objetivos de *CLR* es la de asegurar un alto nivel de compatibilidad entre lenguajes, para ello implementa el llamado **Sistema Universal de Tipos** (*Universal Type System*).



El *Sistema Universal de Tipos* es un conjunto de tipos de datos y clases con funcionalidad (bibliotecas), que pueden ser soportados y utilizados por todos los lenguajes de *.NET*. Si bien no todos los tipos de datos definidos en *CLR* son utilizados desde todos los lenguajes, existe un mínimo accesible a todos. De esta forma, una biblioteca que es desarrollada en *C++* puede ser accesible desde *VB*, y la funcionalidad fácil de invocar. Esto es válido para todos los lenguajes de la plataforma, incluso para los lenguajes controlados por *.NET* que están siendo desarrollados por otras empresas (*APL, COBOL, PASCAL, Perl,* etc.)

Los Dominios

Unos de los problemas con los que cuenta la tecnología actual, es la de realizar aislamiento de procesos por hardware. Esto quiere decir que cada aplicación corre dentro de un espacio de memoria separado. Si bien esto tiene la ventaja de proteger contra fallas la aplicación –si se cae una aplicación lo hará sólo esta, y no todas las demás- tiene la desventaja de ser muy pesado a nivel de recursos. Imagine un servidor *Web* creando y destruyendo cientos de procesos por segundo!

Una de las características principales *CLR* es la de proveer aislamiento de procesos por *software* mediante los **Dominios**. Esto quiere decir que las aplicaciones corriendo bajo un mismo espacio de memoria están aisladas entre si. Los Dominios controlan la visibilidad y la tolerancia de fallas. Si por error cayera una aplicación, solamente afectará a este dominio y no al proceso entero.

De esta forma, se obtienen las mismas características que utilizando procesos, pero con un gasto de recursos muy inferior.

Por otra parte, cada vez que un componente o aplicación necesita conectarse con otra utiliza la tecnología *COM*, lo que implica un consumo alto en recursos. Bajo *CLR* la comunicación se realiza entre Dominios y utilizando tecnologías *.NET*, sin utilizar *COM*, haciendo esto mas eficiente.
Recuerde que cada aplicación corre en el contexto de un **Dominio** (*Domain*), y ésta es la **nueva unidad mínima de proceso.**

El colector de Basura

El colector de basura es el servicio de la plataforma *.NET* para **código manejado**, que se encarga de liberación de la memoria de aquellos objetos que no está siendo utilizados.
Los colectores de basura datan de los años 50, cuando John McCarthy (creador de *LISP*), escribió la primer especificación de un colector de basura de memoria.
Ahora, Microsoft© ha construido uno propio, a ser utilizado por todos los lenguajes de la plataforma *.NET*, porque –de nuevo- la solución que requiere un verdadero lenguaje orientado a objetos de asignar y liberar memoria constantemente es un colector.

Cuando usted utiliza un objeto, debe cerciorarse de realizar los siguientes pasos:

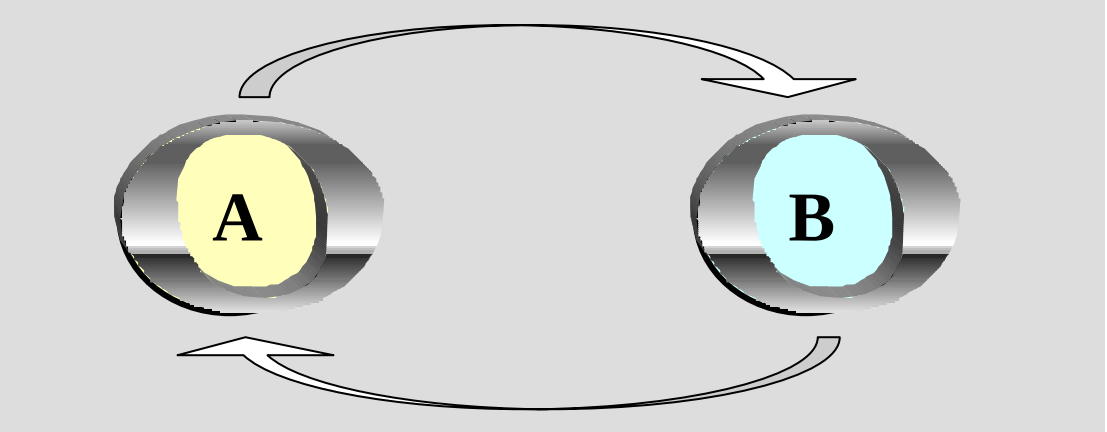
1. Se solicita una referencia a un objeto
2. Se asigna memoria para el objeto
3. Se carga el objeto en memoria y se inicializa
4. Se utiliza
5. Se destruye la referencia
6. Se libera la memoria

*En otros lenguajes el proceso puede ser aun más complejo.

Como vimos anteriormente, si por error alguno de los puntos no se realiza, o se efectúa en un orden diferente, las consecuencias pueden variar desde un consumo de memoria elevado, hasta la corrupción de objetos y el comportamiento de la aplicación en forma inesperada, en momentos imprevisibles. Si bien existen algunas herramientas que nos permiten detectar este tipo de errores (como ser el *Task Manager*, *System Monitor ActiveX Control*, *Compuware's BundsChecker*, etc), ésta continúa siendo una equivocación muy frecuente, y de un alto costo e impacto para las aplicaciones actuales.
El colector de basura está destinado a reemplazar el sistema tradicional de manejo de memoria, que consiste en liberar en forma manual los recursos, y responde a ser la única forma que *COM* ofrece. Generalmente los objetos *COM* son grandes, y el proceso de asignar y liberar la memoria es pesado para esta tecnología. En los nuevos lenguajes donde todo debe ser tratado como un objeto y constantemente se solicita o libera memoria, es un costo muy alto a pagar.

Como ya vimos, una de las características de *COM* es la del manejar un contador interno de referencias. Cada vez que una referencia a un objeto es solicitada, el contador es incrementado, y cuando la referencia es liberada el contador es decrementado. Cuando el contador de referencias llega a 0, la memoria es liberada.

Este proceso resuelve bien muchas situaciones, pero como vimos anteriormente, tiene un serio inconveniente: las referencias circulares.



Los programadores frecuentemente construyen complejos sistemas que constan de cientos de componentes (objetos) relacionados entre sí. Como resultado de esto, a menudo un objeto **A** contiene una referencia a un objeto **B**, y un objeto **B** puede contener una referencia al objeto **A**. A esto se le llama referencia circular, y usted necesita código especial para tratar este caso. Por ejemplo, debe asegurarse que el objeto sea liberado en el orden adecuado, de la forma correcta, y determinar cuando ya no esta siendo utilizado. De hecho, escribir código especial para este tipo de ocasiones es una mala idea, ya que se debe considerar -incluso cuando los objetos actuales no tengan previsto utilizar referencias circulares- esta casuística. De esta forma, las referencias circulares han generado cientos de artículos y formas de remediar este problema. El recolector de basura ha encontrado una solución a las referencias circulares desde sus comienzos. Los sistemas con colector de basura pueden asignar y liberar la memoria en forma eficiente y automática, equilibrando los recursos, y liberando al programador de esta tarea. De esta forma, el colector de basura puede ser visto como un servicio –invisible para usted- que se ocupa de liberar la memoria cuando no esta siendo utilizada. Esto ha traído consigo un cambio en la modalidad de solicitud y liberación de objetos en Microsoft© Visual Basic©, ya que ahora la utilización de la palabra *Nothing* es opcional.
Otra de las características compartidas por los colectores de basura –y quizá el punto más importante a tener en cuenta- es que la memoria no es liberada en el momento que los objetos dejan de estar en uso. En las versiones anteriores de Visual Basic©, usted podía conocer el momento exacto en el cual un objeto era liberado (se asigna *Nothing* a la variable de tipo objeto, y el contador de referencias es 0). A esta característica se le denomina **finalización determinada**, ya que el desarrollador tiene el control total sobre dicho evento.

La perdida de esta característica ha causado algunos cambios importantes en el lenguaje, como el no soporte para el evento de finalización de una clase (*Class_Terminate*), debido a que se desconoce el momento exacto en el cual el colector lo destruirá. Para solucionar esto, el *código manejado* cuenta con el evento *Finalize* a ser lanzado cuando el colector de basura libere la memoria ocupada por el objeto.

A medida que continúe con el libro olvidará la tediosa tarea de liberar referencias en Visual Basic©, pero deberá pensar en el impacto que esto tendrá sobre sus aplicaciones actuales.

La solución al infierno DLL: Ensamblados (Assembly)

En algunas ocasiones, el recuerdo de los sistemas operativos monotarea y las instalaciones de aplicaciones que constaban de copiar un conjunto de archivos a un directorio (ahora llamada carpeta), invade mi mente con un dejo de melancolía. Como vimos anteriormente, la gran parte de los desarrolladores están familiarizados con las características de las bibliotecas compartidas y sus posteriores consecuencias (como ser **'El infierno DLL'**).

El infierno DLL refiere a un conjunto de problemas causados por la utilización de archivos compartidos. Con la arquitectura actual (*COM*) se hace complejo construir bibliotecas que no sean compartidas, y que estén aisladas de las demás aplicaciones. Por otra parte, la imposibilidad de mantener activa o correr varias versiones de la misma biblioteca al mismo tiempo, produce otro conjunto de restricciones. La plataforma *.NET* ofrece una solución 'definitiva' a este problema, mediante los llamados **Ensamblados** (*Assembly*).

En general, la instalación de una aplicación consta de varios pasos, y típicamente envuelve copiar un número de componentes al disco, y crear un conjunto de información en el archivo de registro que describe los componentes instalados. De esta forma, existe una separación entre la información del archivo de registro que describe al componente, y los archivos físicos (que son los que contienen la implementación), haciendo difícil coordinación de ambos.

Otro problema común es el de actualizar un componente de una aplicación mientras esta se ejecuta. En los casos más problemáticos (como ser aplicaciones *Web* de alta demanda) el servicio *Web* debe ser parado e iniciado nuevamente para actualizar el componente *COM*.

Como vimos, todos estos problemas son causados porque la descripción del componente está separada, esto indica que los componentes no son auto-descriptivos.

Los ensamblados (*Assembly*) son la tecnología principal de *.NET* utilizada para resolver los problemas de versión e instalación. En la mayoría de los casos, un ensamblado equivale a un *DLL*, y en esencia, ellos son bibliotecas lógicas (o *DLL's* lógicos). Los ensamblados son auto-descriptivos o auto-contenidos, esto quiere decir que no necesitan agregar información en el archivo de registro.

Los ensamblados contienen el llamado **manifiesto** (*Manifest*), que detalla su versión y toda aquella información para que este pueda ser identificado únicamente.

Las versiones de componentes en la plataforma *.NET* se realiza a nivel de ensamblados, siendo ésta la unidad mas pequeña. Incluso la seguridad se maneja mediante el Ensamblado como núcleo principal del modelo de seguridad, de esta forma, el desarrollador de un ensamblado puede registrar en el manifiesto el conjunto de permisos necesarios para ejecutar dicha biblioteca, y el administrador podrá garantizar los privilegios requeridos por el ensamblado mediante la política de seguridad.

En *.NET* existen los **ensamblados** denominados **lado a lado** (*Side by Side)*, y son ellos el corazón de la nueva historia en lo que a versiones se refiere. **Lado a Lado** es la habilidad de correr múltiples versiones del mismo componente en la misma máquina al mismo tiempo. Debido a esto, diferentes aplicaciones pueden estar utilizando simultáneamente diferentes versiones de la misma biblioteca. Con los componentes que soportan **Lado a lado**, los autores no están atados a una estricta compatibilidad con versiones anteriores, ya que las aplicaciones son libres de utilizar

diferentes versiones del mismo componentes. Por otra parte, aislando un componente se asegura que será accedido y utilizado por una única aplicación. El aislamiento ofrece al desarrollador el control absoluto sobre el código que es empleado por su aplicación, y es la modalidad por defecto utilizada en la plataforma *.NET*.
Como vimos anteriormente, el aislamiento de bibliotecas comienza con Windows© 2000, con la introducción del archivo con extensión **.local**. Este archivo es utilizado para causar que el sistema operativo y *COM* observen primero al directorio de la aplicación, e intenten cargar la biblioteca allí existente (en vez de la especificada por el archivo de registro).

Los **ensamblados privados**, son instalados con la estructura de directorios de la aplicación, y es *CLR* quien se encarga de buscar la biblioteca dentro del directorio. Para ello utiliza un sistema denominado *probing*, que consta de tomar el nombre del ensamblado, agregarle la extensión .dll, y buscarlo dentro de los directorios de la aplicación. A su vez, algunas características pueden variar de acuerdo al lenguaje del sistema operativo donde se ejecuta el ensamblado (a esto se le llama *cultura*). La cultura es útil cuando se desea que los *ensamblados* con recursos pertenecientes un lenguaje se utilicen en ciertas situaciones, por ejemplo que cuando el lenguaje sea español se busquen en el directorio ES, mientas que los ensamblados pertenecientes al lenguaje inglés se busquen en un directorio llamado EN.
Por otra parte, las aplicaciones de la plataforma *.NET*, cumplen con las siguientes características:

- **Son auto-descriptivas**
  Al ser auto-descriptivas, eliminan la dependencia con el archivo de registro, permitiendo un impacto cero sobre la instalación y desinstalación.

- **La Información de versión esta registrada y es forzosamente mantenida**.
  La plataforma implementa compatibilidad de versiones, a los efectos de cargar la versión adecuada en tiempo de ejecución.

- **Contienen la última configuración válida**
  Cuando una aplicación es ejecutada, la plataforma provee la capacidad de recordar el conjunto de componentes, incluyendo sus versiones, a los efectos de almacenar la información de todos los componentes que trabajan en conjunto.

- **Soportan las características *lado a lado (Side-by-side)***
  Permite que múltiples versiones de un mismo componente puedan ser instaladas y ejecutadas simultáneamente, y a las aplicaciones cliente especificar que versión desea cargar, en vez que la versión sea forzada a utilizar por el sistema operativo.

- **Soportan aislamiento de aplicación**
  Las aplicaciones no son afectadas por los cambios como consecuencia de la instalación de otra aplicación.

Como vimos anteriormente, los **ensamblados** contienen una sección denominada manifiesto, donde se especifica información descriptiva del mismo.

Veamos parte de la información allí almacenada:

- **Identidad**
  La identidad de un ensamblado consta de tres partes: nombre, versión, y cultura.

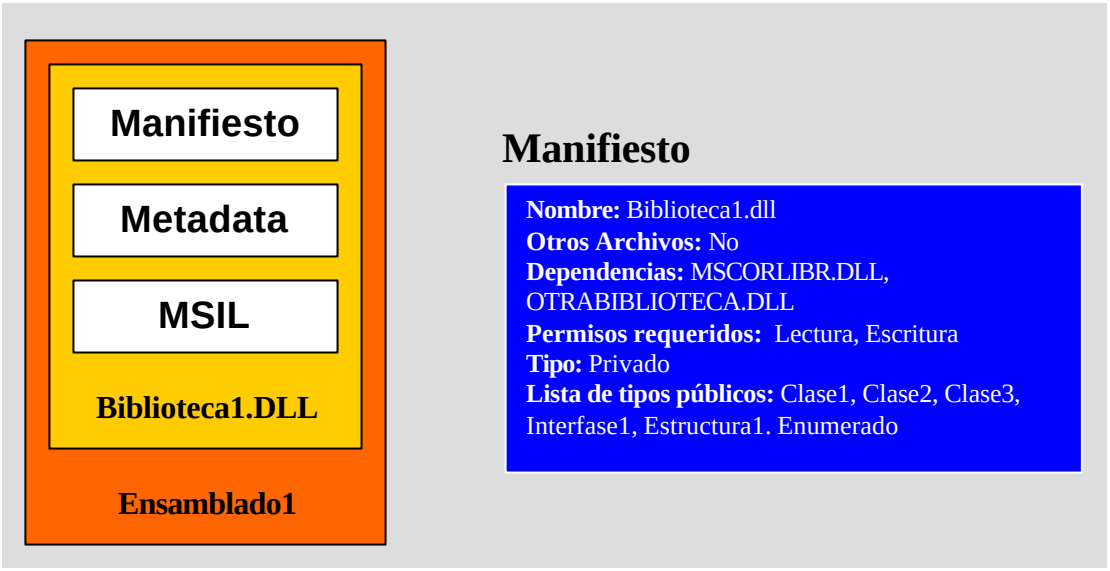
- **Lista de archivos**
  Incluye la lista de todos los archivos que contiene el **Ensamblado**.

- ***Ensamblados* referenciados**
  Lista de dependencias de todos los **Ensamblados** utilizados por el presente **Ensamblado.**

- **Tipos y recursos exportados**
  Entre otras, se especifica la visibilidad disponible para los tipos y recursos que integran el **Ensamblado.**

- **Solicitudes de permiso**
  Indica información inherente a la seguridad requerida por el **Ensamblado**.

No es necesario que conozca la estructura física de un ensamblado, pero es bueno que distinga las cuatro partes que lo componen: el manifiesto, la metadata que describe los tipos, el código en lenguaje intermedio (MSIL), la implementación de los tipos, y la lista de recursos. Si bien existen varias modalidades de empaquetar estos cuatro elementos, estos pueden estar contenidos dentro de una única biblioteca DLL, así como partido en múltiples archivos.



Al igual que las bibliotecas compartidas utilizadas mediante la tecnología *COM*, los ensamblados también pueden ser compartidos por varias aplicaciones. Los **ensamblados compartidos** son guardados en el **almacén de ensamblados globales**.

La utilización del almacén no es obligatoria, pero puede brindarle un buen marco para la administración de los mismos. Generalmente los ensamblados compartidos son agregados al almacén por el instalador de la aplicación. Un ensamblado compartido debe especificar en forma muy estricta la información de versión, compatibilidad, y dependencias. Al igual que en la tecnología *COM*, los ensamblados compartidos deben tener un nombre global único a nivel del sistema operativo, para que de esta forma pueda ser identificado únicamente. Para ello, se utiliza criptografía de clave pública y privada, que adicionalmente le asegura al autor de un componente que nadie mas que él podrá generar nuevas versiones del mismo.

Veamos con que ventajas cuenta la utilización de bibliotecas compartidas mediante el almacén de ensamblados:

Mejor rendimiento – Si el ensamblado ya ha sido cargado en memoria, no se verifica nuevamente su integridad si otra aplicación requiere una instancia del mismo. Por otra parte, el sistema operativo utiliza una única instancia para servir a todas las aplicaciones.

Chequeo de Integridad – Cuando un ensamblado es agregado al almacén, un chequeo de integridad es realizado sobre todos los archivos que la contienen.

Seguridad de archivos - Solamente usuarios con permisos de administrador pueden eliminar archivos del almacén. Debe recordar que generalmente un instalador se ejecuta con permisos de administrador.

Versiones – Permite que múltiples versiones del mismo ensamblado pueden existir en el almacén.

Como vimos, los ensamblados tienen varias ventajas sobre la arquitectura de bibliotecas compartidas de *COM*. Mas adelante, crearemos varios ensamblados desde Visual Basic©.

¿Qué verá en los próximos capítulos?

A mi criterio, el entendimiento de este capítulo constituye un paso muy importante en la comprensión del resto del libro, y del alcance de las tecnologías *.NET*.

En el próximo capítulo veremos las características principales de la nueva interfase de desarrollo, y posteriormente nos avocaremos a analizar en detalle las características aquí citadas.