Estándares de Codificación en C# y Buenas Prácticas de Programación

@canaldenegocio

Contenido

Introducción.

Documento basado en metodologías de Microsoft + Metodologías Agiles.

Para mantener un buen nivel de cumplimiento de buenas prácticas, al finalizar un Sprint (en la retrospectiva), debes realizar reuniones de revisión de código para asegurar que cada uno está siguiendo las reglas. Tres tipos de revisiones de código son recomendadas:

a. **Peer Review:** Revisión por un Par – otro miembro del equipo revisa el código asegurándose de que el código sigue los estándares de codificación y cumple los requerimientos. Este nivel de revisión puede incluir además algunas pruebas unitarias. Cada archivo en el proyecto debe pasar por este proceso.

b. **Revisión del Arquitecto** – el arquitecto del equipo debe revisar los módulos principales del proyecto para asegurarse de que se adhieren al diseño y que no haya "grandes" errores que puedan afectar el proyecto en el largo plazo.

c. **Revisión Grupal** – aleatoriamente se selecciona uno o más archivos y se conduce una revisión de grupo finalizado un Sprint. Distribuye una copia impresa de los archivos a todos los miembros 30 minutos antes de la reunión. Permite que la lean y que lleguen con puntos de discusión. En la reunión para la revisión grupal, usa un proyecto para desplegar el contenido del archivo en la pantalla. Navega por cada una de las secciones del código y permite que cada miembro dé sus sugerencias en cómo esa pieza de código podría ser escrita de mejor manera. (No olvides apreciar al desarrollador por el buen trabajo y asegúrate de que no se sienta ofendido por el "ataque de grupo")

2. Convenciones y Estándares de Nombres

Nota: Los términos "notación Pascal" y "notación de Camell" son usados a través de este documento

Notación Pascal – El primer carácter de todas las palabras se escribe en Mayúsculas y los otros caracteres en minúsculas.

Ejemplo: ColorDeFondo

Notación de Camell – El primer carácter de todas las palabras, excepto de la primera palabra se escribe en Mayúsculas y los otros caracteres en minúsculas.

Ejemplo: colorDeFondo

1. Usa notación **Pascal** para el nombre de las Clases

```
public class HolaMundo
{
        …
}
```

2. Usa notación **Pascal** para el nombre de los Métodos

```
void DiHola(string nombre)
{
        …
}
```

3. Usa notación de **Camell** para variables y parámetros de los métodos

```
int cuentaTotal = 0;
void DiHola(string nombre)
{
        string mensajeCompleto = "Hola " + nombre;
        …
}
```

4. Usa el prefijo **"I"** con notación Pascal para las interfaces (Ejemplo: `IEntity`). Utiliza **"T"** para estructuras de tipos de datos .

```
public class TFactura {
    public String NombreDelCliente;
(…)
    public ArrayList LineasDeDetalle = new ArrayList();
}
```

5. No uses notación Húngara para el nombre de las variables.

En épocas previas muchos de los programadores les agradaba la notación Húngara – la cual especifica el tipo de dato de la variable como un prefijo en el nombre y usar el prefijo m_ para variables globales. Ejemplo:

```
string m_sNombre;
int nEdad;
```

Sin embargo, en los estándares de codificación de .NET, esto no es recomendado. El uso del tipo de dato y del prefijo m_ para representar variables globales no debe ser usado. Todas las variables deben usar una notación **Camell**.

Aun así algunos programadores prefieren usar el prefijo m_ para representar variables globales dado que no hay otra forma fácil de identificar una variable global.

6. Usa palabras entendibles y descriptivas para nombrar a las variables. No uses abreviaciones.
   Correcto:

```
string direccion;
int salario;
```

   Incorrecto:

```
string nom;
string domic;
int sal;
```

   En el caso de compartición de viejas tecnologías (RPG, COBOL …) que requería abreviaturas en la definición de variables, **campos y tablas**, usar una **Pair Name**, es decir nombrar el objeto middleware con una nomenclatura moderna y con un subguión enlazar el nombre antiguo:

   `FACCLI` → Facturas del Cliente en COBOL

```
string FacturasDelCliente_FACCLI = FACCLI;
```

7. No uses nombres de variables de un solo caracter como `i, n, s` etc. Usa nombres como `indice, temp`

Una excepción en este caso podría ser las variables usadas para iteraciones en los ciclos:

```
for ( int i = 0; i < cuantos; i++)
{
        …
```

```
}
```

Si la variable es usada solo como un contador para una iteración y no es usada en ningún otro lado dentro del ciclo, mucha gente le agrada nombrar la variable con un solo caracter (i) en vez de usar un nombre adecuado distinto.

8. No uses guiones bajos (_) para nombres de variables **locales**.

9. Todas las **variables globales** deben usar el prefijo de guión bajo (_) de tal forma que puedan ser identificadas de otras variables locales.

10. No uses palabras reservadas para nombres de variables.

11. Usa el prefijo **"Es" ó "Is"** para variables de tipo `boolean` o prefijos similares.
    Ejemplo:

```
private bool EsValido
private bool IsActivo
```

    La propuesta de **"Is"** , más aceptada, viene dada a que es coherente con las propiedades de uso global de .NET , por ejemplo: `objeto.IsEmpty();` se agrupa en el IntelliSense de una forma más coherente

12. Los nombres de los espacios de nombres deben seguir el siguiente estándar de patrón
    `<NombreDeCompañía>.<NombreDeProducto>.<MóduloSuperior>.<MóduloInferior>`

```
namespace canaldenegocio.com.GestionDeFacturacion.Facturas.Proveedores
```

13. Los nombres de Clases o Métodos y funciones deben seguir el estándar:

```
<Accion/Verbo en Inglés><descripción>
GetClientes();
AddCliente();
```

    De esta manera se agrupará mejor en el IntelliSense con el resto de métodos generales de los objetos.

14. Usa el prefijo apropiado para cada elemento de la Interfaz Gráfica de manera que puedas identificarlos para el resto de las variables.

    Hay dos distintas aproximaciones recomendadas aquí.

    a. Usa un prefijo común (`ui_`) para todos los elementos de la interfaz gráfica. Esto ayudará a agrupar todos los elementos de la interfaz gráfica y que sean fáciles de acceder desde intellisense.

b. Usa un prefijo apropiado para cada elemento de la interfaz gráfica. Una lista breve es dada a continuación. Dado que .NET tiene una gran cantidad de controles, tu puedes lograr una lista completa de prefijos estándares para cada control (incluyendo controles de terceras partes) que estés usando.

Control	Prefijo
Label	lbl
TextBox	txt
DataGrid	dtg
Button	btn
ImageButton	Imb
Hyperlink	hlk
DropDownList	ddl
ListBox	lst
DataList	dtl
Repeater	rep
Checkbox	chk
CheckboxList	cbl
RadioButton	rbt
RadioButtonlist	rbl
Image	img
Panel	pan
PlaceHolder	phd
Table	tbl
Validators	val

Una combinación de ambos es aceptable y ordenada (pero no muy usual) → `ui_lblNombre`

15. El nombre de los archivos debe coincidir con el nombre de la clase.
Por ejemplo, para la clase HolaMundo el nombre del archivo debe ser HolaMundo.cs . Y usa notación Pascal para el nombre de los archivos.

16. En Bases de Datos se recomienda:
    a. Stored Procedures deben nombrarse con prefijo "spr_". NO usar nunca "sp_", La razón porque: SQL Server reconoce el prefijo "sp_" como "System Stored Procedure", es decir, un procedimiento almacenado de Sistema y lo buscaría en la BBDD "Master".
    b. Tablas "tbl_" ….
    c. Vistas "vw_" …
    d. En caso de Imágenes o Proxy de tablas en entornos tipo COBOL, DB2… recordar usar una "Pair Name": tbl_Clientes_CLI

3. Sangría y Espaciamiento

1. Usa TAB para la sangría. No uses ESPACIOS. Define el tamaño del Tab de 4 espacios.

2. Los comentarios deben estar al mismo nivel que el código (usar el mismo nivel de sangría).
   Correcto:

```
//Formatea un mensaje y lo despliega

string mensajeCompleto = "Hola " + nombre;

DateTime horaActual = DateTime.Now;

string mensaje = mensajeCompleto + ", la hora es: " +
horaActual.ToShortTimeString();

MessageBox.Show(mensaje);
```

   Incorrecto:

```
//Formatea un mensaje y lo despliega
string mensajeCompleto = "Hola " + nombre;

DateTime horaActual = DateTime.Now;

string mensaje = mensajeCompleto + ", la hora es: " +
horaActual.ToShortTimeString();

MessageBox.Show(mensaje);
```

3. Las llaves ({}) deben estar en el mismo nivel que el código fuera de las llaves.

```
if ( … )

{

        //Haz algo
        // …

        return false;

}
```

4. Usa una línea en blanco para separar un grupo lógico de código
   Correcto:

```
bool DiHola ( string nombre)

{

string mensajeCompleto = "Hola " + nombre;

DateTime horaActual = DateTime.Now;
```

```
string mensaje = mensajeCompleto + “, la hora es: “ +
horaActual.ToShortTimeString();


MessageBox.Show(mensaje);
if ( … )
{
        //Haz algo
        // …

        return false;
}


        return true;
}
```

Incorrecto:

```
bool DiHola ( string nombre)
{
        string mensajeCompleto = “Hola “ + nombre;
        DateTime horaActual = DateTime.Now;
        string mensaje = mensajeCompleto + “, la hora es: “ +
        horaActual.ToShortTimeString();
        MessageBox.Show(mensaje);
        if ( … )
        {
                //Haz algo
                // …
                return false;
        }
        return true;
}
```

5. Debe haber una y solo una línea en blanco entre cada método dentro de las Clases.

6. Las llaves ({}) deben estar en una línea separada y no en la misma línea del `if, for` etc.
   Correcto:

```
if ( … )
{
        // Haz algo

}
```

Incorrecto:

```
if ( … ) {
        //Haz algo
}
```

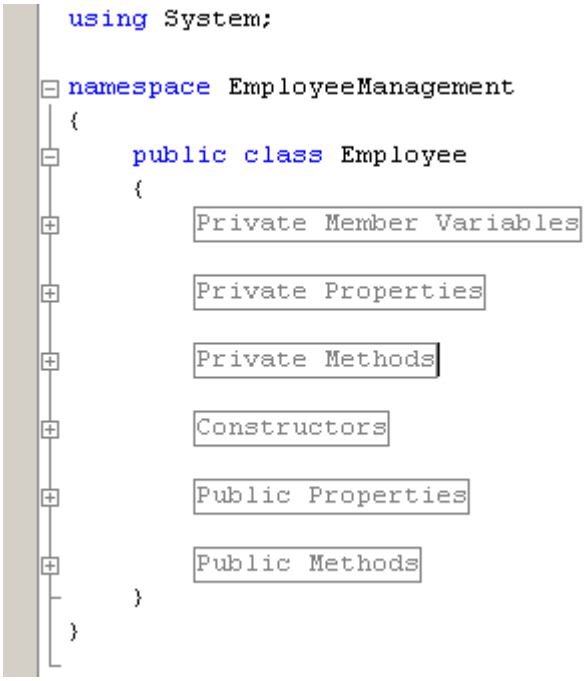
7. Usa un espacio simple antes y después de los paréntesis y los operadores.
   Correcto:

```
if ( muestraResultado == true )
{
        for ( int i = 0; i < 10; i++ )
{
        //
}
}
```

   Incorrecto:

```
if(muestraResultado==true)
{
        if(int i=0;i<10;i++)
        {
                //
        }
}
```

8. Usa `#region` para agrupar piezas de código juntas. Si tu usas una agrupación apropiada usando `#region`, la página debe verse como a continuación cuando todas las definiciones estén cerradas.



9. Mantén privadas las variables globales, las propiedades y los métodos en la parte superior del archivo y los elementos públicos en la parte inferior.

4. Buenas prácticas de programación

> "Los programas deben ser escritos para que los lean las personas, y sólo incidentalmente, para que lo ejecuten las máquinas".
>
> – Abelson and Sussman

1. Evita escribir métodos muy largos. Un método debe típicamente tener entre 1 a 25 líneas de código. Si un método tiene más de 25 líneas de código, debes considerar refactorizarlo en métodos separados.

2. El nombre de los métodos debe decir lo que hace. No uses nombres engañosos. Si el nombre del método es obvio, no hay necesidad de documentación que explique qué hace el método.

> "Está bien investigar y resolver misteriosos asesinatos, pero no deberías necesitar hacerlo con el código. Simplemente deberías poder leerlo"
> – Steve McConnell

Correcto:

```
void GuardaNumeroTelefonico ( string numeroTelefono )
{

}
```

Incorrecto:

```
//Este método guardará el número de teléfono
void GuardaDetalles ( string num )
{
	//Guarda el número de teléfono.
}
```

3. Un método debe tener solo 'una tarea'. No combines más de una tarea en un solo método, aún si esas tareas son pequeñas.

Correcto:

```
//Guarda la dirección
//Envia un email para informar que la dirección es actualizada.

SetDirección ( direccion );
EnviaCorreoAlSupervisor( direccion, email );

void SetDirección ( string direccion )
{
// Guarda la dirección.
// …
}

void EnviaCorreoAlSupervisor ( string dirección, string email )
{
// Envia un correo para informar que la dirección fue cambiada.
// …
}
```

Incorrecto:

```
// Gurada la dirección y envía un correo al supervisor para informar
que
// la dirección fue cambiada.
GuardaDireccion ( direccion, email );


void GuardaDireccion ( string direccion, string email )
{
        // Tarea 1.
        // Guarda la direccion.
        // …


        // Tarea 2.
        // Envia un correo para informar al supervisor que la
        direccion fue cambiada
        // …
}
```

4. Usa los tipos específicos de C# o VB.NET (alias), en vez de los tipos definidos en el espacio de nombres System.

```
int edad;    (no Int16)
string nombre;     (no String)
object infoContrato;     (no Object)
```

5. Siempre verifica valores inesperados. por ejemplo, si estas usando un parámetro con 2 posibles valores, nunca asumas que si uno no concuerda entonces la única posibilidad es el otro valor.
   Correcto:

```
If ( tipoMiembro == eTipoMiembro.Registrado )
{
// Usuario registrado… haz algo…
}
else if ( tipoMiembro == eTipoMiembro.Invitado )
{
// Usuario invitado… haz algo…
}
else
{
// Usuario inesperado. Lanza una excepción
      throw new Exception ( "Valor inesperado " +
tipoMiembro.ToString() + "." )
// si nosotros agregamos un nuevo tipo de usuario en el
futuro,
// podremos fácilmente encontrar el problema aquí.
}
```

   Incorrecto:

```
If ( tipoMiembro == eTipoMiembro.Registrado )
{
      // Usuario registrado… haz algo…
}
else
{
      // Usuario invitado… haz algo…

      // Si nosotros introducimos otro tipo de usuario en el futuro,
      // este código fallará y no se notará.
}
```

6. No incrustes números en el código. En vez de eso usa constantes. Declara constantes en la parte superior del archivo y úsalas en tu código.

Sin embargo, usar constantes tampoco es recomendado ☺ . Se debe usar las constantes en el archivo de configuración o en la base de datos, de tal forma que las puedas cambiar posteriormente. Declara los valores como constantes solo si tu estas totalmente seguro de que este valor nunca necesitará ser cambiado.

7. No incrustes cadenas de texto en el código. **Usa archivos de recursos**.

8. Convierte las cadenas de texto a minúsculas o mayúsculas antes de compararlas. Esto asegurará que la cadena coincida.

```
if ( nombre.ToLower() == "juan" )
{
        // …
}
```

9. Usa String.Empty en vez de "".
   Correcto:

```
if ( nombre == String.Empty )
{
        // haz algo
}
```

   Incorrecto:

```
if ( nombre = "" )
{
        // haz algo
}
```

10. Evita usar variables globales. Declara variables locales siempre que sea necesario y pásalas a otros métodos en vez de compartir una variable global entre métodos. Si compartes una variable global entre métodos, te será difícil rastrear qué método cambia el valor y cuando.

11. Usa `enum` dondequiera que sea requerido. No uses números o cadenas para indicar valores discretos.

Correcto:

```
enum TipoCorreo
{
	Html,
	TextoPlano,
	Adjunto
}

void EnviarCorreo (string mensaje, TipoCorreo tipoCorreo)
{
	switch ( tipoCorreo )
	{
	case TipoCorreo.Html:
// Haz algo
		break;
	case TipoCorreo.TextoPlano:
// Haz algo
		break;
	case TipoCorreo.Adjunto:
// Haz algo
		break;
	default:
// Haz algo
		break;
	}
}
```

Incorrecto:

```
void EnviarCorreo (string mensaje, string  tipoCorreo)
{
	switch ( tipoCorreo )
	{
		case "Html":
			// Haz algo
			break;
		case "TextoPlano":
```

```
                    // Haz algo
                    break;
                case "Adjunto":
                    // Haz algo
                    break;
                default:
                    // Haz algo
                    break;
            }
        }
```

12. No definas las variables globales públicas o protegidas. Mantenlas privadas y expón Propiedades públicas/protegidas.

13. Las rutinas que controlan los eventos (event handlers) no deben contener el código que ejecuta la acción requerida. En vez de ello, llama a otro método desde la rutina controladora.

```
ButtonVentas_Click(Object sender, EventArgs e) Handles Button1.Click
{
    GetVentasPorMes();
}
```

14. No invoques programáticamente el evento Click de un botón para ejecutar la misma acción que has escrito en el evento. En vez de ello, llama el mismo método que es invocado desde la rutina controladora del evento click.

15. Nunca incrustes en el código rutas o letras de dispositivos. Obtén la ruta de la aplicación programáticamente y usa rutas relativas a ella.

16. Nunca asumas que tu código se ejecutará desde el disco "C:". No puedes saber si algunas usuarios lo ejecutan desde la red o desde "Z:".

17. En el arranque de la aplicación, ejecuta una clase de "auto verificación" y asegúrate que todos los archivos requeridos y las dependencias están disponibles en las ubicaciones esperadas. Verifica las conexiones a la base de datos en el arranque, si es requerido. Dale un mensaje amigable al usuario en caso de algún problema.

18. Si el archivo de configuración requerido no se encuentra, la aplicación debe ser capaz de crear uno con valores predeterminados.

19. Los mensajes de error deben ayudar al usuario a resolver el problema. Nunca muestres mensajes de error como "Error en la Aplicación", "Hay un error…" etc. En vez de ello da

mensajes específicos como “Fallo al actualizar la base de datos”, sugiere lo que el usuario debe realizar: “Fallo al actualizar la base de datos. Por favor asegúrate de que la cuenta y la contraseña sean correctos”.

20. Cuando despliegues mensajes de error, adicionalmente a decirle al usuario qué está mal, el mensaje debe también decirle lo que el usuario debe hacer para resolver el problema. En vez de un mensaje como “Fallo al actualizar la base de datos.”, sugiere lo que el usuario debe hacer: “Fallo al actualizar la base de datos. Por favor asegúrate de que la cuenta y la contraseña sean correctos.”.

21. Muestra mensajes cortos y amigables al usuario. Pero registra el error actual con toda la información posible. Esto ayudará mucho a diagnosticar problemas.

22. Usa ficheros de recursos (.resx) para todos los literales de la aplicación. Nos favorece el cambio idiomático, cambio de literales rápido, unificación de mensajes de eventos, etc. En desarrollos muy compartidos, pensar en encapsular en una/s librería/s independiente/s los literales y mensajes para que pueda ser fácilmente compartido por las demás librerías.

    ¿y por qué no hacer un sistema basado XML ó Tablas en BBDD?... nadie te lo impide. Pero ten en cuenta que un fichero .resx es un sistema “específico” de .NET ideado para esta función y por tanto muy optimizado y “precompliado” (por lo tanto más rápido de acceder). XML es más lento y en cadenas largas muy poco optimo.

23. No guardes más de una clase en un solo archivo.

24. Evita tener archivos muy grandes. Si un solo archivo tiene más de 1000 líneas de código, es un buen candidato para refactorizar. Divídelos lógicamente en dos o más clases.

25. Evita métodos y propiedades públicas, a menos que ellas realmente necesiten ser accedidas desde afuera de la clase. Usa “`internal`” si ellas solo son accedidas dentro **del mismo ensamblado**. Normalmente pocos usan esta buena práctica, es muy recomendable pero por otro lado es menos “ágil” al codificar.

```
internal class Test
{
    public int MiTest;
}
```

Otro ejemplo completo:

```
// Assemby : A
```

```
namespace AssemblyA
{
    public class A
    {
        protected internal string SomeProtectedInternalMethod()
        {
            return "SomeValue";
        }
    }

    public class A2 : A
    {
        public string SomeMethod()
        {
            // We can access the method because
            // it's protected and inherited by A2
            return SomeProtectedInternalMethod();
        }
    }

    class A3 : A
    {
        public string SomeMethod()
        {
            A AI = new A();
            // We can access the method through an instance
            // of the class because it's internal
            return AI.SomeProtectedInternalMethod();
        }
    }
}


// Assemby : B
using AssemblyA;
namespace AssemblyB
{
    class B : A
    {
        public string SomeMethod() {
            // We can access the method because
            // it's inherited by A2
            // despite the different assembly
            return SomeProtectedInternalMethod();
        }
    }

    class B2
    {
        public string SomeMethod()
        {
            A AI = new A();
            // We can't access the method
            // through the class instance
            // because it's different assembly
```

```
            return AI.SomeProtectedInternalMethod();
        }
    }
}
```

26. Evita pasar muchos parámetros a un método. Si tienes más de 4~5 parámetros, es un buen candidato para definir una clase o una estructura. Lo contrario destroza el consumo en memoria, más facilidad de corrupción de datos, más castigo a los ciclos del procesador... etc

27. Si tienes un método que retorna una colección, devuelve una colección vacía en vez de null, si no hay datos que retornar. Por ejemplo, si tienes un método que retorna un ArrayList, siempre retorna un ArrayList válido. Si no tienes elementos que devolver, entonces retorna un ArrayList válido con 0 elementos. Esto hará fácil para la aplicación que llama al método verificar solamente la propiedad "Count" en vez que hacer verificaciones adicionales para "null".

28. Usa el archivo AssemblyInfo para llenar la información como el número de versión, descripción, nombre de la compañía, nota de derechos reservados etc.

29. Organiza lógicamente tus archivos dentro de carpetas apropiadas. Usa una jerarquía de carpetas de 2 niveles. Puedes tener hasta 10 carpetas en raíz y cada carpeta hasta 5 subcarpetas. Si tienes muchas carpetas que no pueden ser acomodadas en la jerarquía de 2 niveles mencionada arriba, necesitas refactorizar en distintos ensamblados.

30. Asegúrate de tener una buena clase de registro (logging) la cual puede ser configurada para registrar errores, advertencias o trazar acciones. Si configuras para registrar errores, deben ser sólo errores. Pero si la configuras para registrar las acciones, debe registrar todo (errores, advertencias y acciones). Tu clase de registro debe ser escrita de tal manera que en un futuro puedas cambiar fácilmente para trazar en el Registro de Eventos de Windows, SQL Server o enviar Correo Electrónico al administrador o a un Archivo etc sin cambiar nada en otras partes de la aplicación. Usa la clase de registro extensivamente en el código para registrar errores, advertencias y aún registrar mensajes de actividades que puedan ayudar a resolver un problema.

31. Si estas abriendo conexiones a una base de datos, sockets, archivos etc, siempre cierra dichas conexiones en el bloque `finally`. Esto asegurará que aún si una excepción ocurre después de abrir la conexión, se cerrará seguramente en el bloque `finally`.

```
int numerillo = 123;
string texto = "cosillas";
object oobjetoo = s;
```

```
try
{
    // Invalid conversion; o contains a string not an int
    numerillo  = (int) oobjetoo ;
}
finally
{
    Console.Write("numerillo  = {0}", numerillo );
}
```

32. Declara variables tan cerca como sea posible de donde son usadas por primera vez. Usa una declaración de variable por línea.

33. Usa la clase **StringBuilder** en vez de String si tienes que manipular objetos de tipo string dentro de un ciclo. El objeto String trabajar en modo extraño en .NET. Cada vez que concatenas una cadena, realmente se descarta la cadena anterior y se recrea un nuevo objeto, lo cual es una operación relativamente pesada.
    Considera el siguiente ejemplo:

```
public string ComponMensaje ( string[] lineas )
{
       string mensaje = string.Empty;

       for ( int i = 0; I < lineas.Lenght; i++ )
       {
              mensaje += lineas[i]
       }

       return mensaje;
}
```

    En el ejemplo de arriba, podría parecer que solo estamos concatenando al objeto string 'mensaje'. Pero lo que realmente está sucediendo es que, el objeto string se descarta en cada iteración y recreado con la línea agregada a él.

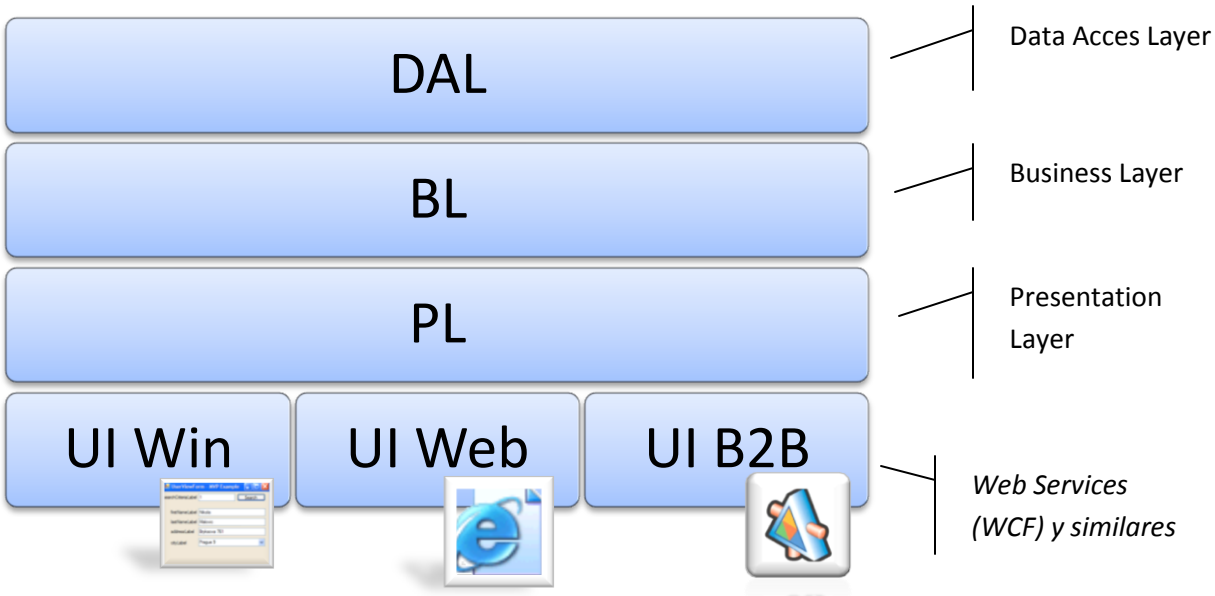
    Si tu ciclo tiene bastantes iteraciones, entonces es una buena idea usar la clase StringBuilder en vez del objeto String.

    Observa el ejemplo donde el objeto String es reemplazado con StringBuilder.

```
public string ComponMensaje ( string[] líneas )
{
        StringBuilder mensaje = new StringBuilder();

        for ( int i = 0; I < lineas.Lenght; i++ )
        {
                mensaje.Append( lineas[i] );
        }

        return mensaje.ToString();
}
```

5. Arquitectura

1. Siempre usa una arquitectura multicapa (N-Tier).



2. Evitar al máximo elementos de "terceros" en el entorno de desarrollo o arquitecturas. Comprometen la escalabilidad futura, pues esos "terceros" no saben cómo Microsoft orientará su próxima estrategia de desarrollo y puede comprometer el proyecto. Por ejemplo Microsoft ahora apuesta ahora por jQuery, por lo que los otros actores (Dojo, MooTools, Prototype…) pueden quedar fuera de juego. Hay que adoptar aquellos frameworks que Microsoft adopte **claramente** o nos quedaremos cautivos. Y la historia está llena de cadáveres tecnológicos con grandes expectativas, como por ejemplo Adobe Flash, Microsoft apuesta por HTML5 y Silverlight ¿qué hacemos ahora con las super guays aplicaciones en Flash?. En éste ejemplo, es más "seguro" apostar por Silverlight como reproductora media, que Flash, pues Silverlight "es" Microsoft y por tanto, tecnología "sostenida" a futuro "e incorporada en el sistema Operativo de serie", frente a elementos de terceros.

3. Nunca accedas a la base de datos desde las páginas de la interfaz gráfica. Siempre ten un conjunto de clases de capa de acceso a datos la cual ejecute todas las tareas relacionadas con la base de datos.

4. Usa las sentencias try-catch en tu capa de acceso a datos para atrapar todas las excepciones de la base de datos. Este controlador de excepciones debe registrar todas las excepciones desde la base de datos. Los detalles registrados deben incluir el nombre del comando siendo ejecutado, nombre del procedimiento almacenado, parámetros, cadena de conexión usada etc. Después de registrar la excepción, debe de re lanzarse para que la otra capa en la aplicación la atrape y tome las acciones apropiadas.

5. Separa tu aplicación en múltiples ensamblados. Agrupa todas las clases de utilidades independientes (acceso a archivos, validaciones genéricas, literales, constantes…) dentro de una librería (¿proyecto?) de clases separada. Todos tus archivos relacionados a la base de datos pueden ser otra librería de clases.

6. ☺ **KISS** (Keep It Simple, Stupid - Mantenlo simple, estúpido): Se recomienda el desarrollo empleando partes sencillas, **comprensibles** y con errores de fácil detección y corrección, rechazando lo enrevesado e innecesario en el desarrollo de sistemas complejos en ingeniería. En SCRUM es un pilar imprescindible.

   Si lo puede entender tu abuela… estás en el camino correcto. ☺ El código está bien formado cuando alguien que no lo ha codificado es capaz de leerlo sin necesidad de adivinar que hace y que necesita. (ver técnicas de revisión de código: **Peer Review, …etc.**)

   Recuerda que: pasado dos meses si tu código "ni tú" sabes lo que hace es que está mal conceptuado y las funciones mal definidas.

   "Buen código es aquel que hace obvio lo que está pasando"

7. **KAIZEN**, metodología de calidad en la empresa y en el trabajo, en su vertiente para los desarrolladores IT nos dice: "A los informáticos nos gusta la tecnología, nos imaginamos grandes catedrales de arquitecturas". Olvidemos las catedrales, recordemos, pues, a **KISS**.

8. Seguir los patrones S.O.L.I.D. lo mejor posible:

   a. **SRP (Single Responsibility Principle):**

      El principio de responsabilidad única nos indica que debe existir un solo motivo por el cual la clase debe ser modificada, o sea, que la clase debe tener un solo propósito. Es el principio más fácil de violar

   b. **OCP (Open-Closed Principle):**

      El principio Abierto/Cerrado indica que las clases deben estar abiertas para la extensión y cerradas para la modificación, o sea, que una clase debe poder ser extendida sin tener que modificar el código de la clase.

   c. **LSP (Liskov Substitution Principle):**

      El principio de sustitución de Liskov indica que las clases derivadas (hijas) pueden ser sustituidas por sus clases base. Se promueve que la herencia se realice en forma transparente, o sea, no se debe implementar métodos que no existan en sus clases base ya que de lo contrario se rompe el principio.

"De OCP y LSP se deduce que las clases base (abstractas o no) modelan el aspecto general y las clases heredadas modelan el comportamiento local."

d. **ISP (Interface Segregation Principle):**

El principio de segregación de interfaces indica que hay que hacer interfaces de grano fino que son específicos de clientes, dicho de otra forma, muchas interfaces muy especializadas son preferibles a una interfaz general en la que se agrupen todas las interfaces

e. **DIP (Dependency Inversion Principle):**

El principio de inversión de dependencias indica que las abstracciones no deben depender de los detalles, los detalles deben depender de las abstracciones

Atención: Aplicar los principios SOLID pero sin dejar de vista Kaizen. No podemos complicar la "lectura" del código, emborrachándonos de Clases Interfaces, heredadas y encapsulamientos complejos. **Lo que no se intuye a la vista es que está mal plateado y es un riesgo de mantenimiento a futuro**.

9. TAXONOMIZA, TAXONOMIZA, TAXONOMIZA por favor. Cuando aparece un concepto, llegar a un acuerdo sobre cómo llamarlo para todos. Crear un diccionario de conceptos :
   a. ¿e-mail? ó ¿correo electrónico?
   b. → eMail : e-mail, correo electrónico
   c. `public String eMail;`
10. Vacúnate contra las enfermedades más abituales: **Cacheitis**, **Procrastinación**, **code entropy**…
    a. **Cacheitis**: vendría a ser algo así: "desorden transitorio del arquitecto o diseñador de software por el cual el uso inadecuado de la caché provoca como resultado una solución ineficiente" ☺ . ¿Y cuál es la primera solución que se les ocurre***? Sí señoooor: más caché***. Caché por todos lados ☺. Vamos a cachear todo lo cacheable. Bien, pues, mejor que asumas que no puedes cachearlo todo. Esto es viejísimo: ya es harto conocido por ejemplo, que en el mundo de las bases de datos es absurdo tratar de indexar todo en la búsqueda del rendimiento.

Y ésta mala práctica, la cacheitis, la extiende al uso de la cache en general, pero:

i. El cacheo temprano no es bueno,
ii. Hay tener en cuenta las otras cachés.
iii. Una colección estática o global no es una caché: los desarrolladores noveles, y a veces los no tan noveles, utilizan datos estáticos o globales para simular el uso de la caché.

b. **Procrastinación**: "dejar para mañana lo que se puede hacer hoy." Básicamente, es no afrontar una tarea que tenemos pendiente, y que vamos retrasando horas, días o incluso semanas. La cuestión es que esa tarea, normalmente:
   i. No nos gusta.
   ii. No nos vemos preparados para llevarla a cabo.
   iii. Sentimos temor del resultado de esa tarea.
c. **Code Entropy**: Entropia en el código. La tendencia natural es que el código cada vez se complique más y más. Con SCRUM + Kaizen, por ejemplo, eso es más difícil que ocurra, pues te obliga a analizar el código realizado y optimizarlo.
d. …

11. Sé **ACID** ☺ (Atomicity, consistency, isolation, and durability) con los datos.

- **Atomicidad:** es la propiedad que asegura que la operación se ha realizado o no, y por lo tanto ante un fallo del sistema no puede quedar a medias.

- **Consistencia:** *Integridad*. Es la propiedad que asegura que sólo se empieza aquello que se puede acabar. Por lo tanto se ejecutan aquellas operaciones que no van a romper las reglas y directrices de integridad de la base de datos.

- **Aislamiento:** es la propiedad que asegura que una operación no puede afectar a otras. Esto asegura que la realización de dos transacciones sobre la misma información sean independientes y no generen ningún tipo de error.

- **Durabilidad:** es la propiedad que asegura que una vez realizada la operación, ésta persistirá y no se podrá deshacer aunque falle el sistema.

Cumpliendo estos 4 requerimientos un sistema gestor de bases de datos puede ser considerado **ACID Compliant**.

12. **Web Service**. No abusar de ellos. Es un modelo que consume muchos recursos. Hoy, por hoy, no debería usarse de forma directa y debería ser usado tras un WCF, que nos permite hacer uso de TCP/IP mucho más ágil, rápido y reducción de consumo de recursos.  No es un sistema para conectar procesos internos, en sustitución de una librería referenciada. Toda comunicación entre librerías de aplicaciones internas se realiza con librerías referenciadas directamente, no con Web Services. Web Services es, exclusivamente, para comunica aplicaciones y procesos "de terceros" o que no podamos conectar directamente por una VPN, Gateway o similar.

13. Unificación de Interface. En un proceso de diseño de interface con usuarios u otros servicios, hay que realizar unas sesiones de común acuerdo en donde va cada elemento y la manera que se interactuará con el usuario. En el caso de interface con usuario, se recomienda seguir algún patrón de ergonomía **+ la opinión del usuario**. Crea un prototipo y muéstralo al usuario, no siempre lo "científicamente" adecuado es lo idóneo.

14. Versionado y ciclo de vida de una aplicación.
    a. http://es.wikipedia.org/wiki/Fases_del_desarrollo_de_software
    b. **Alfas**: aplicación en desarrollo, no operativo. Se nombra así: 0.0.0001
    c. **Betas**: aplicación concluida en sus funcionalidades principales. De alcance interno y de testeadores funcionales (FDD). Está en equipos de Integración para el resto de desarrolladores y testeadores. Eliminación de errores de forma activa. Se nombra: b1.0
    d. **RC** (**Release Candidate**) Candidata a Versión: Cuando todos los test han sido pasados y solo queda su distribución o implementación (está en pre-producción). De alcance público Se nombra: rc1.0
    e. **RTM** (**Release To manufacturing**), opcional. Se considera el estado del producto definitivo que puede ser usado antes de su distribución o uso masivo.
    f. **Release** Final. Producto considerado final y en producción. Se nombra 1.0
    g. **Release menor**, de mejora menor o corrección de bug una vez en producción. Se nombra 1.1

Acceso a Datos ¿Code handle, dataset tipados, Entity…?

Por orden:

a) "no me complico la vida": **dataset tipado/** Table Adapters

b) "Pata Negra": **Entity Framework** (EF)

c) "Master del Universo" o "no sé hacerlo de otro modo", según el caso :-D → **Code handle** (a manija yo me lo curro, como toda la vida)

d) Ah! Y **LINQ** … bueno, es útil para manipular objetos de datos como si fueran registros (por ejemplo un ArrayList). Pero la opción de hacer "selección de datos" en el lado código en lugar de en vistas en ó Stored Procedures en la BBDD (SQL Server, Oracle…) rompe con la metodología de las n-capas y la de "cuanto más cerca de la BBDD mejor" (por tanto la capa de manipulación de datos debe estar en la BBDD). Excepción: en el caso de, por ejemplo, teléfonos móviles que no pueden conectarse a una fuente de reglas de negocio y de capa de datos, sí sería interesante LINQ (ver apartado de smartphone), de hecho Microsoft indica este sistema para Windows Phone.

La opción es bien clara: **Entity Framework** (EF). PERO; hay que tener un gran dominio y experiencia en acceso a datos, objetos de datos… etc. Curva de aprendizaje es muy alta.

Si quieres una aplicación totalmente desligada a la fuente de datos, EF puede ser tu hombre.

Sin embargo. El 90% aplicaciones se desarrollan en entornos "estables" (desarrollos internos) y la arquitectura/repositorio de acceso a datos no cambia con facilidad. Por tanto, tenemos una opción intermedia que se llama Table Adapter - DataSet Tipados. Es el "padre" de EF. Mediante su asistente nos permite crear transformaciones y accesos a datos de una forma rápida, escalable altamente controlada. Además, si seguimos la buena práctica de usar un middlerware (como SQL Server) para acceder a cualquier fuente de datos (Oracle, Access, DB2…), EF no es necesaria y los "Table Adapter" nos soluciona todas las necesidades.

Por tanto. Descartando las otras opciones por escalabilidad, curva de aprendizaje, etc. Nos centramos en Dataset Tipados - Table Adapters:

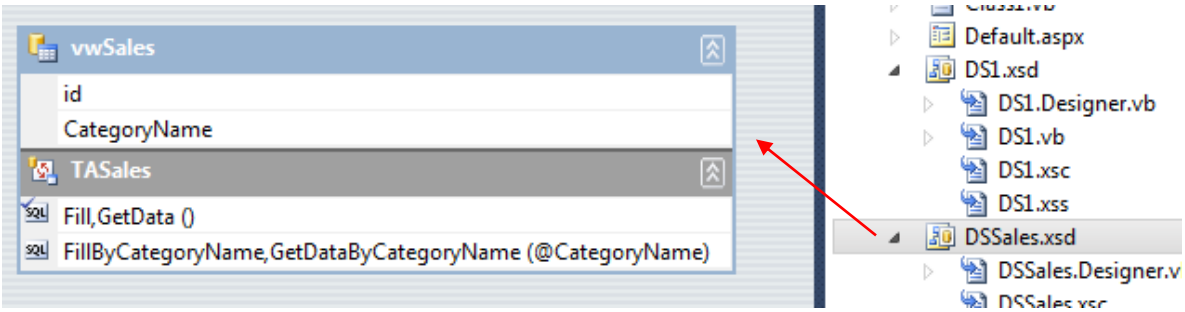
15. Usar Table Adapters y DataSet Tipados (Typed DataSet) en lugar de "hardcodear" llamadas a mano a las BBDD y no disponer de los nombres de campos. Typed DataSet nos

da mejor rendimiento, más clara la lectura de código, automatización de las conexiones a datos, más optimizadas las llamadas al servidor de datos… etc.

Usa el Wizard (asistente) de Visual Studio: TableAdapter Query Configuration . Te ahorra tiempo, complejidad; facilita optimización y control de accesos a datos, gestión de concurrencias, libera mejor las sesiones y se organiza mejor los accesos a datos, prepara el cacheo optimizadamente … etc.
¡Lo hace casi todo por ti!

☺ **Correcto:**

1) Previo: gestión de acceso a datos mediante el asistente (TableAdapter Query Configuration) y guardado de estructura en un dataset (DataSet.xsd)



2) Acceso al dataset con Table Adapters y … tipado:

```
clsSales()
{ …

DSSalesTableAdapters.TASales ta = New DSSalesTableAdapters.TASales();
DSSales.vwSalesDataTable dt = New DSSales.vwSalesDataTable();


         Public  DSSales.vwSalesDataTable GetSalesByCategory(string categoryName)
         {
          ta.FillByCategoryName(dt, categoryName)

          Return dt
         }
}

void MostrarResultado(string categoryName)
{
       clsSales Sales = New clsSales();

       DSSales.vwSalesDataTable dt;
       dt = Sales.GetSalesByCategory(categoryName);

        Console.WriteLine( dt(0).id + " " + dt(0).CategoryName );
}
```

¡¡ Tres lineas de código !! 👍

☹ Inadecuado (ojo, que no incorrecto):

```
static void GetSalesByCategory(string connectionString,
    string categoryName)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        // Create the command and set its properties.
        SqlCommand command = new SqlCommand();
        command.Connection = connection;
        command.CommandText = "SalesByCategory";
        command.CommandType = CommandType.StoredProcedure;

        // Add the input parameter and set its properties.
        SqlParameter parameter = new SqlParameter();
        parameter.ParameterName = "@CategoryName";
        parameter.SqlDbType = SqlDbType.NVarChar;
        parameter.Direction = ParameterDirection.Input;
        parameter.Value = categoryName;

        // Add the parameter to the Parameters collection.
        command.Parameters.Add(parameter);

        // Open the connection and execute the reader.
        connection.Open();
        SqlDataReader reader = command.ExecuteReader();

        if (reader.HasRows)
        {
            while (reader.Read())
            {
                Console.WriteLine("{0}: {1:C}", reader[0], reader[1]);
            }
        }
        else
        {
            Console.WriteLine("No rows found.");
        }
        reader.Close();
    }
}
```

¡¡ 13 lineas de código !! 👎

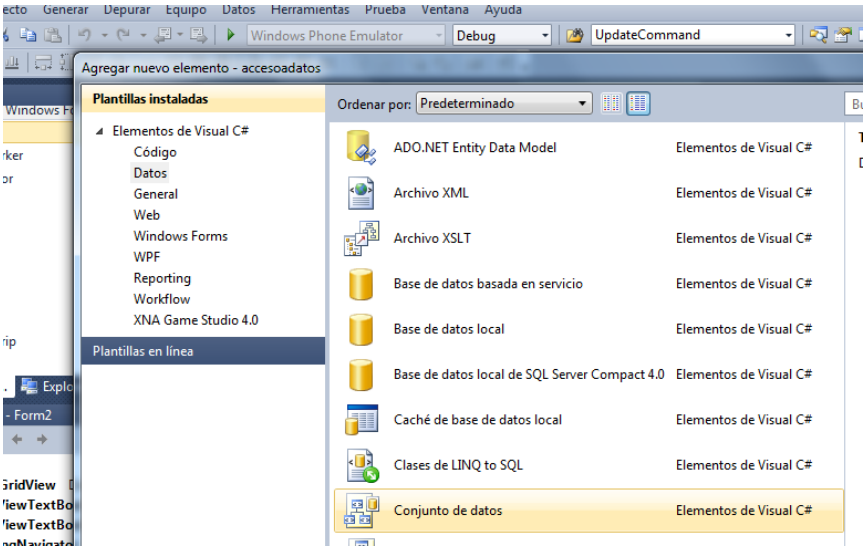
Podemos hacer uso de Entity Framework, si nuestro programa tiene que ser **fuertemente** agnóstico de la fuente de datos y de la capa de negocio… pero no es recomendable si no es preciso o nuestro dominio de ésta técnica no es muy alto y si la tecnología del lado fuente de datos y/o del lado capa de negocio son de la misma arquitectura (.NET…).

16. Eso sí, esto NUNCA:

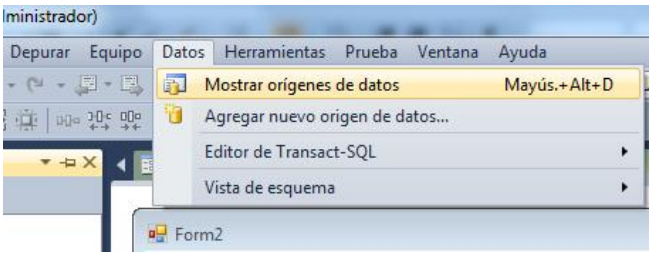
```
sql = "SELECT Nombre, Apellidos, id FROM Clientes Where Nombre = " +
dt.Nombre + " AND Apellido=" + dt.Apellido
command.CommandText = sql
(…)

 connection.Open();
 SqlDataReader reader = command.ExecuteReader();
(…)
```

Encadenamientos de string para construir una consulta

17. **Hacer uso de las herramientas y asistentes que Visual Studio ofrece**. No por eso se es "peor" programador o "menos cool" :-P. Están ahí para facilitar el trabajo y ser más "ágiles". Se recomienda el uso de:
    a. "TableAdapter Query Configuration" del "Conjunto de Datos" (DataSet)



    b. del "Origen de Datos"

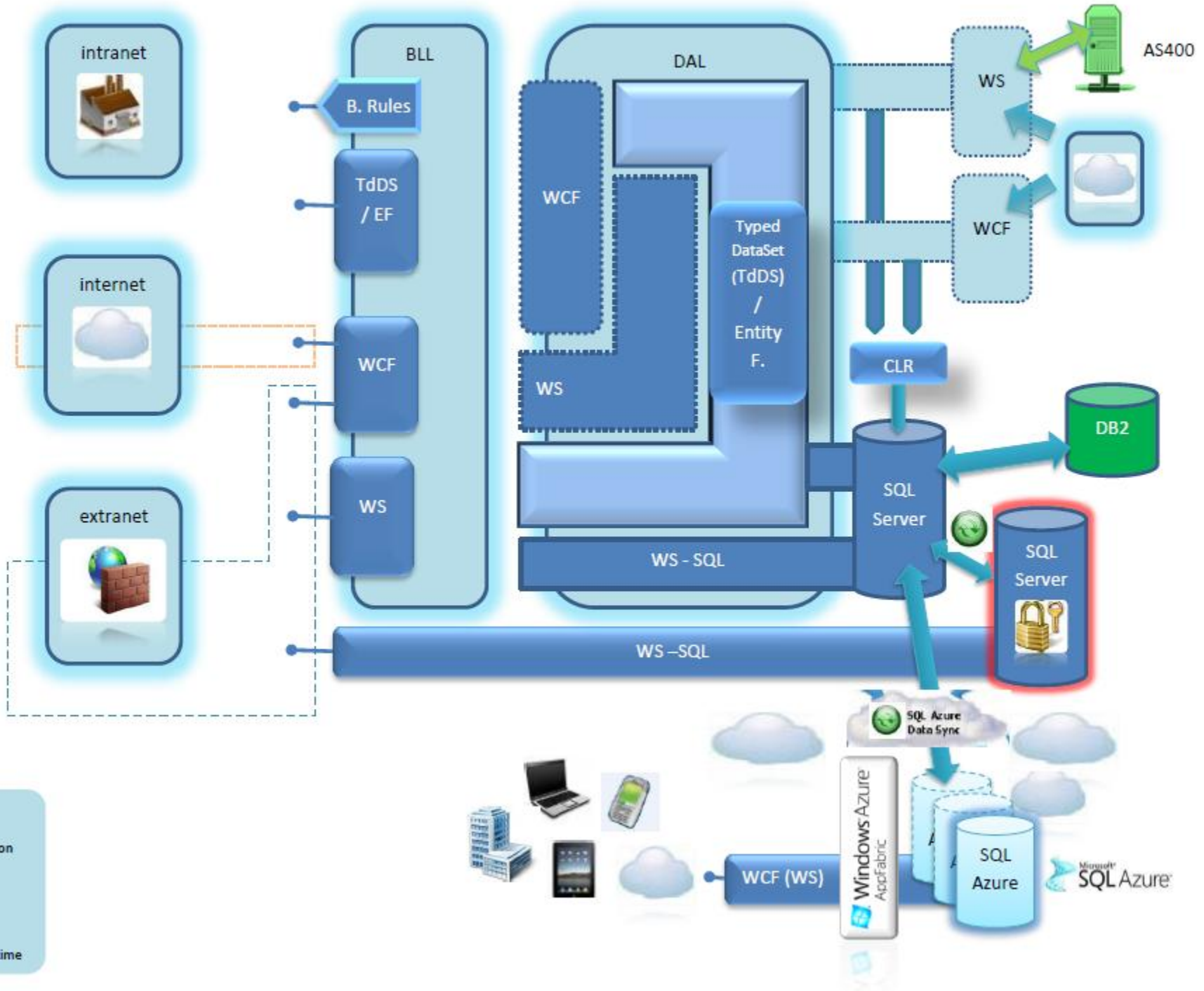


18. La "selección y estructura" de datos, cuanto más cerca del "motor" de la BBDD mejor. Uso de Stored Procedures y Vistas (entronca con el punto anterior), **y nunca acceso directo a la tabla**. En el caso de SQL Server, por ejemplo, existe la opción más extrema (y radicalmente más efectiva) llamada *integración CLR* que, además, se integra muy bien con el control de versionado en TFS.

19. Minimizar el número de arquitecturas. Si disponemos de más de una base de datos, usar una como middleware, siempre y cuando eso no sea una desventaja por motivos de rendimiento u operatividad que no pueda ser soportado por la middleware. **SQL Server es una excelente middleware**, y si los programas de acceso son .NET, mucho más recomendable por homogeneidad.
    De forma genérica, encapsular otras plataformas (frameworks) en la predominante u oficial de la empresa.

20. Al crear tablas en la BBDD Tener en cuenta una serie de campos extra de mucha utilidad para el control de registros:
    a. DataTime Ultima acción y proceso (usuario + proceso). Siempre viene bien saber quién y cuando dejó en ese estado el registro. ¿ha sido un usuario?¿ha sido un proceso?¿cuándo?... las típicas preguntas necesarias cuando ha ocurrido algún incidente ;-)
    b. Eliminación de registro. Es recomendable, NO eliminar físicamente un registro de primeras. Pasarlo a un estado de "baja lógica" y posteriormente, en un proceso manual y/o automático (Data Garbage Collection), periódico, eliminarlo físicamente y/o pasarlo a un histórico. Usar Triggers, por ejemplo, para interceptar acciones de eliminación y convertirlas a updates del estado de ese campo.
    c. Notas simples. Un campo de uso genérico conteniendo observaciones que permitan identificar cualquier circunstancia. El "Post-it" del DBA ☺!!

Arquitectura desarrollo



WS: Web Service
WCF: Windows Communication
TdDS: Typed DataSet
E.F.: Entity Framework
BLL: Business Logic Layer
DAL: Data Access Layer
CLR: Common Language Runtime

6. UI (ergonomía de interface de usuario)

(*documento incompleto, en preparación*)

Referencias:

- Aplicación de criterios de usabilidad WEB
  - http://www.canaldenegocio.com/docs/AplicaciondecriteriosdeusabilidadWEB.pdf
- Bases para la implementación de un UI WinForms
  - http://www.canaldenegocio.com/docs/BasesdeUI.pdf
- Informe APEI de Usabilidad.
  - http://www.canaldenegocio.com/docs/informeapeiusabilidad.pdf

7. ASP.NET

1. No uses variables de sesión a través del código. Usa variables de sesión solo dentro de las clases y expón métodos para acceder el valor almacenado en las variables de sesión. Una clase puede acceder la sesión usando `System.Web.HttpContext.Current.Session`

2. No almacenes objetos grandes en la sesión. Almacenar objetos grandes en la sesión puede consumir mucha memoria del servidor dependiendo del número de usuarios.

3. Siempre usa hojas de estilo para controlar el look and feel de las páginas. Nunca especifiques nombre de las fuentes y tamaño en ninguna de las páginas. Usa clases de estilo apropiadas. Esto asegurará que puedas cambiar la interfaz gráfica de tu aplicación fácilmente en el futuro. También, si deseas soportar la personalización de la interfaz gráfica para cada cliente, es solo cuestión de desarrollar otras hojas de estilo para ellos.

4. Interface y Usabilidad de las páginas Web (ver apartado anterior 6)

5. Hacer referencia las librerías de JQuery y similares usando los **CDN** (Content Delivery Network) Es un grupo de servidores repartidos por todo el mundo en puntos estratégicos y pensados para la distribución de ficheros. Hay varios **CDN's gratuitos**, entre ellos el de Google, el de Microsoft y el de Edgecast. Tambien hay CDN's de pago como Akamai o Amazon CloudFront La idea es que, en vez de cargar las librerías desde tu servidor de hosting, las cargues directamente desde el CDN. De este modo cuando se haga la petición se cargará la librería (o librerías solicitadas) desde el nodo más cercano al cliente con lo que se cargará más rápido.

   Para cargar jQuery desde el CDN de Microsoft, pondríamos en el HEADER de nuestro HTML lo siguiente:

```
   <script src="http://ajax.microsoft.com/ajax/jquery/jquery-1.6.2.js"
type="text/javascript">
```

Usar CDN tiene varias ventajas:

- Liberas a tu servidor de la carga de estos archivos
- Incrementas las posibilidades de que el fichero esté cacheado, ya que otros sitios que usen tu CDN enlazaran al mismo fichero
- Un CDN muy probablemente servirá el fichero más rápido que desde tu propio servidor de hosting

¿Qué CDN me conviene utilizar?

- El CDN de Google es el más lento de los 3 en America del Norte y en Europa
- **En Europa** el CDN de **Microsoft** es el más rápido
- En América del Norte, el CDN de Edgecast es el más rapido
- El CDN de Edgecast gana en términos de rendimiento medio.

6. (*documento incompleto, en preparación*)

¿MVC o no MVC?

… de pende. MVC es otro más de los movimientos estratégicos de Microsoft (como lo fue J#, C#, AJAX…) para "captar" la atención de los desarrolladores de otras plataformas/lenguajes, como Java -JSP, que en este caso son Spring, Struts, Tapestry, WebWork, Hibernate, Symfony… entre otras muchas.

De la misma manera que un desarrollador de VB.NET no necesita "pasarse" al C# para nada, un desarrollador "bien arquitecturizado" no necesita MVC, pues "ya implementa" la orientación a capas (datos, reglas, interface ←→ Modelo, Controlador, Vista) de forma adecuada.

Si desarrollas **solo** Webs, y partes de cero, puede resultar interesante. Si tienes que estar dentro de un ecosistema empresarial con interacciones con otros procesos con modelos de reglas comunes, productos ya realizados, etc… NO es adecuado.

Desventajas:

- Radical cambio de forma de platear y desarrollar. La curva de aprendizaje de patrón de diseño es muy alta. Nuevos lenguajes a aprender (por ejemplo Razor, que es el ASP Classic de otra manera, no aporta nada en especial :-P)…

  ASP.NET MVC requiere un conocimiento más profundo del entorno web y sus tecnologías subyacentes, puesto que a la vez que ofrece un control mucho más riguroso sobre los datos que se envían y reciben desde el cliente, exige una mayor responsabilidad por parte del desarrollador, ya que deberá encargarse él mismo de mantener el estado entre peticiones, maquetar las vistas, crear las hojas de estilo apropiadas, e incluso los scripts.

¿Tienes un equipo nuevo en éste ambiente? pues entonces… ¡¡ni lo sueñes!!. Si el equipo de desarrollo tiene ya experiencia creando aplicaciones con Webforms y no poseen grandes conocimientos sobre programación web de más bajo nivel ni experiencia previa trabajando con el patrón MVC (de otros frameworks como Struts, Spring… etc), deberíamos pensárnoslo antes de dar el salto a ASP.NET MVC, puesto que la productividad, va a caer.

- Compromiso arquitectónico. Inflexible, no hay vuelta atrás. Los desarrolladores nunca podrán ignorar por completo el punto de "vista del modelo", incluso si están desconectados :-S Uff!!!.
  Con este modelo, como con los otros, ¡te casas con él de por vida! Y el divorcio es muy duro y complicado.. ¿Has pensado si dentro de un año aparece otro modelo más "chulo" y pueda dejar obsoleto a éste de la misma manera que a **Spring**, por ejemplo? ¡¡ Hay que pensar en términos de independencia de modelos de terceros **lo más posible** !!
- Tecnología "incipiente" en Microsoft. Microsoft hace muchos experimentos y **puentes** para captar a desarrolladores de otros frameworks y eso no quiere decir que adopte esa metodología, son movimientos para "atraerlos a su framework". Este es un experimento-puente más **y como pasó con su ASP.NET AJAX, J#... etc , con peligro de quedar en la cuneta**. Riesgo estratégico innecesario.
- La separación de conceptos en capas agrega complejidad. El proceso de desarrollo aislado de los autores de interfaz de usuario (Vista), los autores de la lógica de negocio (Modelo) y los autores del controlador, puede ocasionara retrasos en su desarrollo de los módulos respectivos.
- Es un paso hacia atrás en términos de productividad y facilidad de uso. Digamos que no es muy "ágil" en términos de implementación ¿nuestro equipo es SCRUM?.
- La cantidad de archivos a mantener y a desarrollar incrementa considerablemente.
- Si la aplicación es muy orientada a datos y necesitamos estados, no es la opción adecuada.
- No es la opción adecuada si tenemos que usar controles ya generados o crear prototipos rápidos. Para eso, mejor Webforms. Necesitas un control total sobre HTML, JavaScript y CSS, ASP.NET MVC significa que se introducen manualmente los elementos Web.
- Es importante tener una capa de servicios, sino, al final, terminas con montones de métodos de los controladores que NO deberían estar allí violando el SRP (Single Responsibility Principle).

¿Recuerdas el principio de K.I.S.S.?¿y de S.O.L.I.D.?
Recordemos lo que nos dice la metodología Kaizen: "No hagamos catedrales" ☺.

- ASP.NET MVC No implementa nada que no se pueda hacer hoy ya desde Webforms (REST, jQuery…). Es otra forma de hacer lo mismo.

- Incompatible con los componentes ASP.NET AJAX Tool Kit ☹ actuales
- No puedes hacer uso de <… runat=server> perdiendo el gran control del lado cliente que hace esta posibilidad y por tanto, de l agilidad del "código compilado".
- Tener que ceñirse a una estructura predefinida, lo que a veces puede incrementar la complejidad del sistema.

  Hay problemas que son **más difíciles de resolver** respetando el patrón MVC.

- ASP.NET MVC es "muy TDD"… por tanto "poco" SCRUM y FDD y nos obliga a un gran dominio de Simulacro de Marcos y por tanto de NUnit, xUnit.NET, o similares :-P

Por otro tanto hay que preguntarse: ¿MVC es "progreso" sólo desde el punto de vista de la escritura de la Vista? ASP.NET MVC se remonta a los años del ASP Classic (si, claro, por supuesto añade algunos ayudantes en HTML… y poco más :-P)

De hecho, la característica fundamental de ASP.NET MVC se encuentra en:

- El aislamiento de capas (MVC).
- El modo de invocación de la lógica de negocio.
- Y el nuevo estilo URL de enrutamiento (REST), en lugar del modo de plantilla.

Pero, no es cierto decir que el nivel de Vista MVC es más claro que en ASP.NET Web Forms, ya que también se puede escribir de modo muy similar a ASP.NET MVC con Formulario web aspx ASP.NET.

VENTAJAS:

Las ventajas son muchas, también, pero para eso ya existen los "fan-boys" ☺ ¿no? y ahora MVC está "de moda", y como toda moda mejor valorarla cuando se pase la fiebre y quedarse con lo bueno que deje.

Siempre hay que mirar las des-ventajas primero y valorar si los "impedimentos" pueden poner en una situación de riesgo a nuestro trabajo. El principio de: "lo primero coger al toro por los cuernos". De nada sirven las ventajas si las des-ventajas entorpecen nuestro resultado.

Si las des-ventajas no suponen riesgos… adelante!!!.

8. Smartphones

1. Las pantallas deben ser “elásticas”: adaptarse a las multiples resoluciones existentes :
    a. Los tamaños de letras en medidas de proporcionalidad (em), para adaptarse al dispositivo.
    b. Las Imágenes “elastizarlas” usando Width 100%
    c. …
2. La navegabilidad, principalmente, en horizontal. Limitar al máximo el scroll vertical.
    a. Hay que tener presente que el usuario usa su pulgar y su movimiento natural es de derecha a izquierda. Si tiene que usar la otra mano “incomoda”.
    b. Esto “aligera” enormemente la carga de pantallas.
    c. Limita la fatiga visual del usuario.
3. Los colores con el contraste adecuado para ver en condiciones de alta luminosidad.
4. Controlar el tipo de dispositio/explorador que nos llega… para luego tomar decisiones de presentación de pantallas.
5. (*documento incompleto, en preparación*)

9. Comentarios

> "Comentar el código es como limpiar el cuarto de baño; nadie quiere hacerlo, pero el resultado es siempre una experiencia más agradable para uno mismo y sus invitados"
> – Ryan Campbell

Cometarios buenos y entendibles hacen el código más mantenible. Sin embargo,

1. No escribas comentarios para cada línea de código o para cada variable declarada.

2. Usa `//` o `///` para comentarios. Evita usar `/* … */`

3. Escribe comentarios siempre que sea requerido. Pero un buen código requerirá muchos menos comentarios. Si todos los nombres de variables y métodos son entendibles, eso hará que el código sea muy legible y no necesitará muchos comentarios.

4. No escribas comentarios si el código es fácilmente entendible sin comentarios. La desventaja de tener muchos comentarios es que, si cambias el código y olvidas cambiar el comentario, este provocará más confusión.

5. Pocas líneas de comentarios hará el código más elegante. Pero si el código no es claro/legible y hay pocos comentarios, será peor.

6. Si tienes que usar una lógica **compleja o extraña** por cualquier razón, documéntala muy bien con suficientes comentarios.

7. si inicializas una variable numérica para un número especial distinto a 0, -1 etc, documenta la razón para escoger dicho valor.

8. **El objetivo final es**, escribe código claro y legible de tal manera que no necesitas ningún comentario para entenderlo.

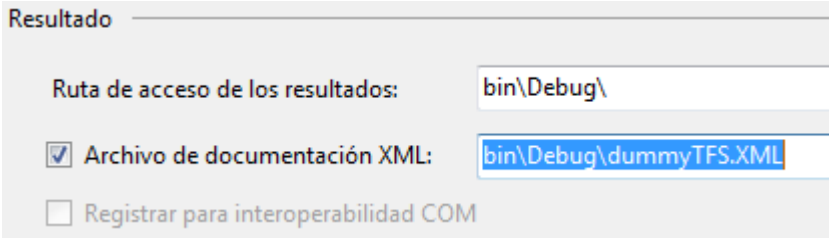
9. Ejecuta una verificación de ortografía en los comentarios y asegúrate de usar una gramática y puntuación apropiada.

10. Usar el comentario `//TODO` para poder rastrearlo con el control de tareas de VS

```
// TODO: revisar el parametro si es el nombre adecuado
   SqlParameter parameter = new SqlParameter();
   parameter.ParameterName = "@CategoryName";
```

11. Documentador de las funciones comentadas. Podemos activar que el documentador:

1. Open the project's **Properties** page.
2. Click the **Build** tab.
3. Modify the **XML documentation file** property.



10. Manejo de Excepciones

1. Nunca hagas un 'atrapa un error no hagas nada'. Si escondes una excepción, nunca sabrás si la excepción sucedió o no. Muchos desarrolladores usan este método práctico para ignorar errores no significantes. Siempre debes de tratar de evitar excepciones verificando todas las condiciones de error programáticamente. En ningún caso, atrapar una excepción y no hacer nada está permitido. En el peor de los casos debes registrar la excepción y continuar.

2. En caso de una excepción, envía un mensaje amigable al usuario, pero registra el error actual con todo el detalle posible acerca del error, incluyendo la hora en que ocurrió, el método y nombre de la clase etc.

3. En lo posible atrapa sólo excepciones específicas, no excepciones genéricas.

Correcto:

```
void LeeDesdeArchivo (  string nombreArchivo )
{
        try
        {
// lee del archivo
        }
        catch ( FileIOException ex)
        {
// registra error.
// relanza la excepción dependiendo del caso.
        throw;
        }
}
```

Menos correcto:

```
void LeeDesdeArchivo (  string nombreArchivo )
{
        try
        {
                // lee del archivo
        }
        catch ( Exception ex)
        {
                // Capturar excepciones generales, es malo… nunca
        sabremos si fue un error del archivo u otro error.
                // Aquí estas escondiendo la excepción.
                // En este caso nadie sabrá que una excepción sucedió.
                return “”;
        }
}
```

4. No necesitas atrapar la excepción general en todos tus métodos. Déjalo abierto y permite que la aplicación falle. Esto ayudará a encontrar la mayoría de los errores durante el ciclo de desarrollo. Puedes tener un controlador de errores al nivel de la aplicación (thread level) donde puedas controlar todas las excepciones generales. En caso de un ‘error general inesperado’, este controlador de errores debe atrapar la excepción y debe registrar el error además de dar un mensaje amigable al usuario antes de cerrar la aplicación, o permitir que el usuario ‘ignore y continúe’.

5. Cuando lanzas una excepción, usa la sentencia `throw` sin especificar la excepción original. De esta manera, la pila original de llamada se preserva.
   Correcto:

```
catch
{
//haz lo que se requiera para manejar la excepción

      throw;
}
```

   Incorrecto:

```
catch (Exception ex)
{
      //haz lo que se requiera para manejar la excepción

      throw ex;
}
```

6. No escribas rutinas try-catch en todos tus métodos. Úsalas solo si hay una posibilidad de que una excepción específica pueda ocurrir y no pueda ser prevenida por ningún otro medio. Por ejemplo, si deseas insertar un registro si no existe en la base de datos, debes intentar encontrar el registro usando la llave primaria. Algunos desarrolladores intentarán insertar un registro sin verificar si ya existe. Si una excepción ocurre, asumirán que el registro ya existe. Esto está estrictamente prohibido. Debes siempre verificar explícitamente por errores en vez de esperar que ocurran excepciones. Por el otro lado, debes siempre usar controladores de excepción cuando te comunicas con sistemas externos tales como redes, dispositivos de hardware etc. Tales sistemas son sujetos a fallas en cualquier momento y verificar errores no es usualmente práctico. En esos casos debes usar controladores de excepciones para recuperar la aplicación del error.

7. No escribas bloques try-catch muy grandes. Si es requerido, escribe bloques try-catch para cada tarea que ejecutes y encierra solo las piezas de código específicas dentro del try-catch. Esto ayudará a que encuentres qué parte del código generó la excepción y puedas mostrar un mensaje de error específico al usuario.

documento incompleto, en preparación

ULTIMAS ACTUALIZACIONES DESCARGAR EN:
http://www.canaldenegocio.com/docs/MejoresPracticasDotNet.pdf

AUTORES:
Josep Ribal @Jribal (@canaldenegocio) MS Movile Specialist, e-Logistic Expert.
Alberto Fernández @AlbertoFdez (@canaldenegocio) MS Architect Specialist, MeBA
Rafa Mendieta. DBA Expert, MS SQL Server Specialist.
documento incompleto, en preparación

REVISONES:
Victor Martin Espuelas (MVP - MS)

FUENTES:
Varias (principalmente Microsoft)
(documento incompleto, en preparación)

RECURSOS:
http://www.microsoft.com/download/en/details.aspx?id=18867