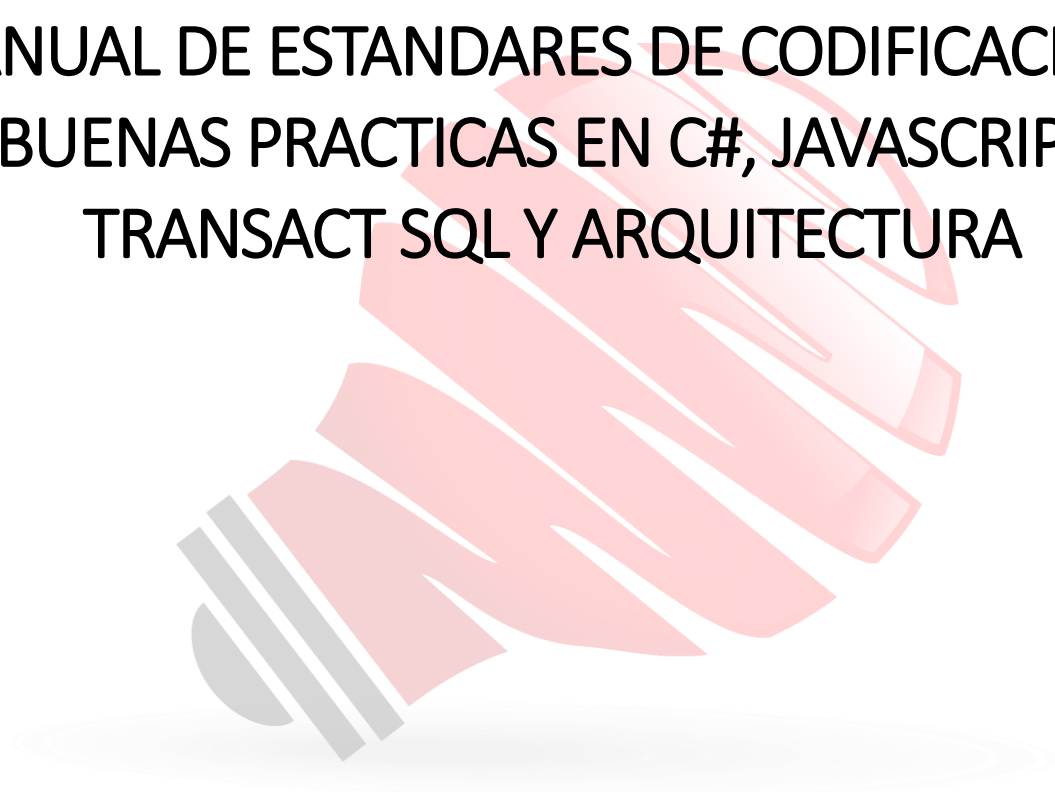


# MANUAL DE ESTANDARES DE CODIFICACIÓN Y BUENAS PRACTICAS EN C#, JAVASCRIPT, TRANSACT SQL Y ARQUITECTURA



CONFIDENTIAL

## Introducción.

Este documento está basado en las buenas prácticas definidas y recomendadas por expertos desarrolladores, así como por Microsoft y Metodologías Ágiles que deberán de cumplirse en la Fábrica de Software de Engine Core

Específicamente, este documento contiene:

- Convenciones y estándares de nombres
- Sangría, espaciado y comentarios
- Buenas prácticas de programación C Sharp
- Buenas prácticas de programación JavaScript
- Buenas prácticas de programación Base de Datos SQL Server
- Arquitectura

## 1. Objetivo.

El principal objetivo de la utilización de estándares de codificación, es institucionalizar buenas prácticas y recomendaciones de diseño, para lograr mayores niveles de calidad en los productos de a través de la Fábrica de Software de Engine Core

La utilización y seguimiento de los estándares definidos en este documento debe de dar como resultado efectos positivos en:

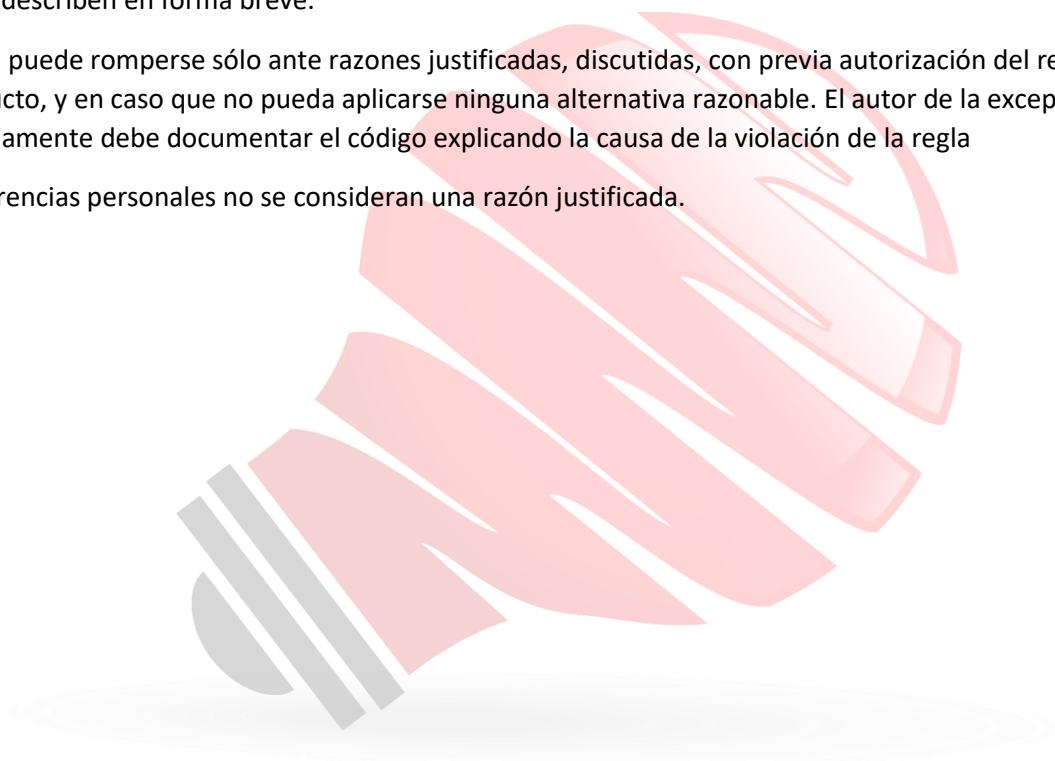
- Mantenimiento del código escrito de acuerdo a un estilo estandarizado
- Disminución de errores, especialmente en aquellos difíciles de individualizar.
- Mantenimiento del código estructurado de acuerdo a las guías de diseño.
- Performance de la aplicación desarrollada con prácticas eficientes en la codificación del lenguaje.

## 2. Alcance.

Este documento aplica a todos los lenguajes de programación, sin embargo, los ejemplos mencionados en este documento están enfocados a lenguajes de programación como C-Sharp, Java Script y Transact SQL, muchas de las recomendaciones descritas pueden ser usadas independientemente del lenguaje en el que se esté trabajando, se incluyen algunos comentarios acerca de prácticas comunes y problemas conocidos de C#, las cuales se describen en forma breve.

Una regla puede romperse sólo ante razones justificadas, discutidas, con previa autorización del responsable del producto, y en caso que no pueda aplicarse ninguna alternativa razonable. El autor de la excepción, obligatoriamente debe documentar el código explicando la causa de la violación de la regla

Las preferencias personales no se consideran una razón justificada.





# ENGINE CORE

## CONVENCIÓN DE ESTANDARES

Ciudad de México  
01 de octubre de 2019

Convenciones utilizadas en este documento

Colores y énfasis	Descripción
Azul	Indica un ejemplo de código en .net c Sharp que es correcto aplicar
Rojo	Indica un ejemplo de código en .net c Sharp que no es correcto aplicar
Verde	Indica comentarios de código en .net c Sharp
Negrita	Texto con énfasis adicional que debe ser considerado importante.

Uno de los aspectos más importantes dentro de la estandarización de código fuente y uso de las buenas practicas tiene que ver con el 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 definidas en este documento. Tres tipos de revisiones de código son recomendadas:

- Peer Review:** Revisión por un Par – Donde 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.
- \* Revisión del Arquitecto** – El arquitecto de software o responsable de la Fábrica de Software deberá de revisar y validar el código fuente del interesado, checando que este alineado a los estándares de codificación y buenas practicas definidos
- 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 con previa anticipación 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. (Nunca olvides apreciar al desarrollador por el buen trabajo y asegúrate de que no se sienta ofendido por el “ataque de grupo”)

## 3. Convenciones y Estándares de Nombres.

### 1. Tipos de Codificación.

Existen varios tipos de metodologías y convenciones para la notación de código fuente como Pascal o Camell entre otros, los cuales definiremos a continuación:

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

Ejemplo: ColorDeFondo

```
public class HolaMundo(string MensajeCompleto)
{
    ...
}
```

\* **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

```
public class holaMundo(string mensajeCompleto)
{
    ...
}
```

El tipo de notación que deberá de usarse sin excepción alguna es Pascal Case

## 2. Uso de nombres entendibles.

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

### Correcto:

```
string direccion;  
int usuario;
```

### Incorrecto:

```
string nom;  
string dir;  
int pass;
```

Evitar el uso de nombres como i, a, s, t, es mejor usar nombres descriptivos a excepción de iteración en los ciclos.

```
for (int i = 0; i < numTotal; i++)
```

## 3. Variables Globales y Locales.

Variables Globales: Estas deberán de ser declaradas con un prefijo guion bajo.

```
string _VariableGlobal;
```

Variables Local: Estas deberán de ser declaradas sin un prefijo guion bajo.

```
string VariableLocal;
```

## 4. Tipos Implícitos.

Cuando se declare una variable, si esta tiene asignado un valor que es totalmente reconocible u obvio, o cuando el tipo exacto no es importante se podrá usar el tipo **var**, de lo contrario se deberá de declarar de forma explícita el tipo de variable que se estará recibiendo, el uso del tipo **var** deberá de estar restringido para información estática y no dinámica.

### 5. No usar palabras reservadas.

El nombre de las variables no puede ser igual a alguna palabra reservada del lenguaje de programación .

*Estas son algunas de las palabras reservadas más comunes en c sharp*

AddHandler	AddressOf	Alias	And	Ansi
As	Assembly	Auto	Base	Boolean
ByRef	Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar	CDate
CDec	CDBl	Char	CInt	Class
CLng	CObj	Const	CShort	CSng
CStr	CType	Date	Decimal	Declare
Default	Delegate	Dim	Do	Double
Each	Else	Elseif	End	Enum
Erase	Error	Event	Exit	ExternalSource
False	Finalize	Finally	Float	For
Friend	Function	Get	GetType	Goto
Handles	If	Implements	Imports	In
Inherits	Integer	Interface	Is	Let
Lib	Like	Long	Loop	Me
Mod	Module	MustInherit	MustOverride	MyBase
MyClass	Namespace	New	Next	Not
Nothing	NotInheritable	NotOverridable	Object	On
Option	Optional	Or	Overloads	Overridable
Overrides	ParamArray	Preserve	Private	Property
Protected	Public	RaiseEvent	ReadOnly	ReDim
Region	REM	RemoveHandler	Resume	Return
Select	Set	Shadows	Shared	Short
Single	Static	Step	Stop	String
Structure	Sub	SyncLock	Then	Throw
To	True	Try	TypeOf	Unicode
Until	volatile	When	While	With
WithEvents	WriteOnly	Xor	eval	extends
instanceof	package	var		

### 6. Uso de prefijos.

Cuando se trate de variables tipo **boolean** se debe de usar el prefijo “es”.

```
private bool esValido
private bool esActivo
```

### 7. Los espacios de nombres definidos deberán de tener el siguiente patrón.

```
<NombreDeCompañía>.<NombreDeProducto>.<MóduloSuperior>.<MóduloInferior>
namespace eCore.ERP.Usuarios;
```

### 8. Uso de prefijo para controles que provengan de interfaz gráfica

Tipo de Control	Prefijo	Tipo de Control	Prefijo
Etiqueta	Lbl	Lista	Lst
Caja de Texto	Txt	Check	Chk
Grilla	Dg	Check Lista	Chkl
Botón	Btn	Radio Botón	Rbtn
Botón Imagen	Ibtn	Radio Botón Lista	Rbtnl
Hyper Liga	Hlk	Imagen	Img
Combo	Cb	Tabla	Tbl
Validador	Val		

### 9. Nombre archivo vs clase.

El nombre de los archivos físicos deberá de coincidir con el de la clase.

**Nombre Clase:** RegistraUsuario

**Nombre Archivo:** RegistraUsuario.cs



## 4. Sangría, Espaciamiento, Comentarios.

### 1. Sangría

Usar TAB para la sangría, jamás use espacios, la sangría deberá de estar definida por 2 TABs.

### 2. Comentarios

Los comentarios deben de estar al mismo nivel que el código.

#### Correcto:

```
//Este es un comentario de ejemplo que está alineado
string MensajeCompleto = "Hola" + nombre;
MessageBox.Show(MensajeCompleto);
```

#### Incorrecto:

```
//Este es un comentario de ejemplo que NO está alineado
string MensajeCompleto = "Hola" + nombre;
MessageBox.Show(MensajeCompleto);
```

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

```
boolean EsMayorDeEdad = false;
if (Edad > 18)
{
    return EsMayorDeEdad = true; //Validación es mayor de edad
}
```

### 3. Separador

Usar una línea en blanco como separador para dos grupos lógicos de código y deberá de haber solo una línea en blanco para separar dos métodos dentro de una clase.

```
public class clase1(string Parametro1)
{
    ...
}

public class clase2(string Parametro1)
{
    ...
}
```

### 4. Llaves.

Las llaves deben de estar siempre en líneas separadas y no en la misma línea de las sentencias if, for, while, switch, etc, esto facilita la lectura y comprensión del código.

#### Correcto:

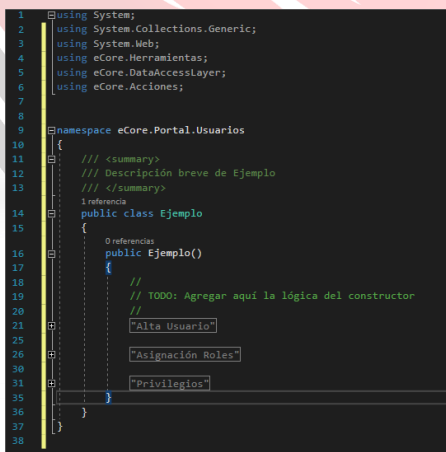
```
if (!existeError)
{
    ...
}
```

#### Incorrecto:

```
if (!existeError ) {...}
if (!existeError ) {...}
```

### 6. Uso de `#region`.

Usa `#region` para hacer una agrupación lógica de los sectores de código, esto permitirá la identificación rápida.



```
1  @using System;
2  using System.Collections.Generic;
3  using System.Web;
4  using eCore.Herramientas;
5  using eCore.DataAccessLayer;
6  using eCore.Acciones;
7
8
9  namespace eCore.Portal.Usuarios
10 {
11     /// <summary>
12     /// Descripción breve de Ejemplo
13     /// </summary>
14     public class Ejemplo
15     {
16         0 referencias
17         public Ejemplo()
18         {
19             // TODO: Agregar aquí la lógica del constructor
20             //
21             "Alta Usuario"
22             "Asignación Roles"
23             "Privilegios"
24         }
25     }
26 }
27
28
```

### 7. Ubicación de variables, propiedades y métodos.

Mantén privadas las variables globales a la clase, las propiedades y los métodos deberán de ubicarse en la parte superior del archivo y los elementos públicos en la parte inferior

## 5. Buenas prácticas de programación C Sharp.

### 1. Métodos largos.

Se debe de evitar escribir métodos muy largos en la medida de lo posible, típicamente un método deberá de tener entre 1 y 40 líneas de código, si el método contiene más de 40 líneas de código se deberá de considerar el re factorizarlo en métodos separados.

### 2. Nombres Descriptivos.

El nombre de los métodos deberá de describir de forma clara lo que ejecutara, si el nombre del método es suficientemente claro, no habrá que documentar la explicación de su función.

#### Correcto:

```
void CrearUsuario ( DataTable UsuarioDetalle )  
{  
    ...  
}
```

#### Incorrecto:

```
void SalvaDetalle (DataTable DetEmp)  
{  
    ...  
}
```

### 3. Código atómico.

El método deberá de tener solo una tarea o fin, no se debe de combinar tareas en un solo método por más pequeñas que estas sean.

#### Correcto:

```
// Crea Usuario
// Envía un email para proceso de activación de usuario
CrearUsuario(Usuario);
EnviaCorreo(uuidUsuario, CorreoContenido);

void CrearUsuario(DataTable Usuario )
{
    // Código para creación de Usuario
}

void EnviaCorreo ( string uuidUsuario, string CorreoContenido)
{
    // Envía un correo para informar que se ha guardado el detalle
}
```

#### Incorrecto:

```
// Crea Usuario
// Envía un email para proceso de activación de usuario
CreaUsuarioYEnviaCorreoActivacion(detalleEmpleado, nombreEmpleado, correo );

void CreaUsuarioYEnviaCorreoActivacion ( DataTable DetalleUsuario, string
uuidUsuario, string CorreoContenido )
{
    // Tarea 1

    // Crea usuario
    // Tarea 2
    // Envía un correo activación de cuenta
    // Tarea N
}
```

### 4. Uso de tipos específicos de variables.

Cuando se declara una variable es imperativo usar los tipos específicos de las mismas en lugar de usar los que se encuentran en el espacio de nombres System.

Usar <code>int</code> edad;	en lugar de usar <code>Int16</code>
Usar <code>string</code> teléfono;	en lugar de usar <code>String</code>
Usar <code>object</code> detalleEmpleado	en lugar de usar <code>Object</code>

### 5. Validación de valores inesperados.

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 == Usuario.Registrado )
{
    // Usuario registrado
}
else if ( TipoMiembro == Usuario.Invitado )
{
    // Usuario invitado
}
else
{
    // Usuario no esperado, se genera una excepción controlada
    throw new Exception ( "Error " + Usuario.ToString())
}
```

#### Incorrecto:

```
if ( tipoMiembro == usuario.registrado )
{
    // Usuario registrado
}
else
{
    // Usuario invitado
    // Si nosotros creamos un nuevo tipo de usuario en el futuro,
    // este código no funcionara de forma adecuada, no te darás cuenta
}
```

## 6. Comparaciones explícitas

Se considera una mala práctica evaluar o comparar una expresión booleana contra expresiones true o false.

### Correcto:

```
if (VariableBoleana)
```

### Incorrecto:

```
if ( VariableBoleana == false)  
if ( VariableBoleana != false)  
while (((VariableBoleana == true) == true) == true)
```

## 7. Uso de **Return**.

Un método solo deberá de tener una entrada y una salida de tal forma que el uso de la directiva return debe de estar al final del método, no es recomendable tener múltiples return en la rutina de código.

## 8. Números en código embebidos.

No se deben de usar números en código, es preferible usar constantes en la cabecera de la clase, aunque dependiendo el motivo es preferible usar constantes en archivos de configuración.

## 9. Cadenas de texto en código embebido.

No se deben de usar cadenas de texto en código, es preferible usar archivos de recursos externos a la funcionalidad del código, por ejemplo, un catálogo de errores.

### 10. Comparaciones entre cadenas.

Cuando se hagan comparaciones entre cadenas, es importante convertir las cadenas a minúsculas o mayúsculas antes de compararlas para que el resultado se asegure que pueda coincidir.

```
if (Nombre.ToLower() == "juan" )  
{  
    ...  
}
```

### 11. Validación de cadenas vacías.

Cuando sea necesario validar si una cadena es vacía se debe de usar `String.Empty`, o validar la longitud de la cadena en lugar de validar contra una cadena vacía "" .

#### Correcto:

```
if (Nombre == String.Empty)  
{  
    ...  
}  
  
if (Nombre.Length == 0)  
{  
    ...  
}
```

#### Incorrecto:

```
if (Nombre == "" )  
{  
    ...  
}
```

### 12. Uso de variables globales restringidas.

Evita usar variables globales a medida que sea posible, debes de declarar variables locales y pasarlas entre métodos en lugar de compartir una variable global entre varios métodos, esto ayudara a rastrear más fácil un error y se mitigara la probabilidad de que ese valor cambie cuando está en uso por otro hilo.

### 13. Uso Enumeraciones `enum`.

Usar enumeraciones donde sea requerido, no usar número o cadenas para validar o comparar.

#### Correcto:

```
enum TipoProveedorBaseDatos
{
    SqlClient,
    OracleClient,
    Odbc
}

void DataAccessLayer (TipoProveedorBaseDatos TipoProveedorBD)
{
    switch (TipoProveedorBD)
    {
        case TipoProveedorBaseDatos.SqlClient:
            // Haz algo
            break;
        case TipoProveedorBaseDatos.OracleClient:
            // Haz algo
            break;
        case TipoProveedorBaseDatos.Odbc:
            // Haz algo
            break;
        default:
            // Haz algo
            break;
    }
}
```

#### Incorrecto:

```
void dataAccessLayer ( string tipoProveedorBD )
{
    switch (tipoProveedorBD)
    {
        case "sqlClient":
            // Haz algo
            break;
        case "oracleClient":
            // Haz algo
            break;
        case "odbc":
            // Haz algo
            break;
        default:
            // Haz algo
            break;
    }
}
```



### 14. Eventos de Controles.

Las rutinas que controlan eventos (event handlers) para el caso de aplicaciones de escritorio no deben de contener el código que ejecuten la acción requerida, si no deben de hacer una llamada al método que si la ejecute desde esa rutina

#### Correcto:

```
BtnGuardaUsuario_Click(Object sender, EventArgs e) Handles btnGuardaUsuari.Click
{
    GuardarUsuario();
}
```

#### Incorrecto:

```
BtnGuardaUsuario_Click(Object sender, EventArgs e) Handles btnGuardaUsuari.Click
{
    // Código para guardar usuario
}
```

### 15. Uso de rutas o letras de acceso a unidades.

No embebas en el código rutas o letras de dispositivos obtén la ruta de la aplicación de forma dinámica y usa rutas relativas a ella.

### 16. Preparar el código por si no existe un recurso.

Si el archivo o recurso requerido no se encuentra, la aplicación deberá de ser capaz de crear uno con los valores predeterminados, por ejemplo, si una carpeta a usar no existe, el código deberá de ser capaz de crear dicha carpeta.

### 17. Mensajes de respuesta.

Los mensajes regresados por la aplicación deberán de ser descriptivos y ayudar al usuario a entender lo que ha sucedido, no se deberán regresar error que no sean controlados como "Object reference no set to an instance of an object" ó "Error en la aplicación", se deberán de dar error más específicos y entendibles como "Ha ocurrido un error al intentar crear el usuario, para mayor información contactar a mesa de servicio, error # 1257624", este número devuelto es la relación hacia el detalle del error que deberá de estar almacenado previamente en base de datos

### 18. Log de Auditoria.

Los errores a mostrar deberán de ser customizados a algo entendible para el usuario, sin embargo, es imperativo guardar en un log de auditoria el mensaje completo arrojado por la plataforma, para poder tener rastreabilidad y poder validar el error.

### 19. Ficheros `.resx` para literales de aplicación.

Los mensajes de error y en general las literales de la plataforma deberán de estar contenidas en un fichero `.resx`, esto facilitara la identificación de errores y al cambio idiomático, cambio de literales rápido, unificación de mensajes, etc., en caso de no manejarse un fichero `.resx` se deberá de concentrar en un diccionario de datos de tipo HashTable la descripción de los mensajes.

### 20. Clases por archivo.

Aunque el esquema de desarrollo de .Net permite guardar más de una clase en un archivo, la buena práctica es siempre manejar una sola clase por archivo, esto permite la identificación y orden del código.

### 21. Clases muy grandes.

Si la clase que se está desarrollando tiene más de 1000 líneas de código, es preferible re factorizar el código y separar en clases diferentes, solo en casos justificados se podrán tener clases que superen las 1000 líneas de código

### 22. Métodos y propiedades Públicas.

Se debe de evitar generar métodos y propiedades públicos, a menos que sea muy necesario el uso de ellos de forma externa, se debe de usar el tipo `“internal”` si es que ellas son accedidas desde el mismo ensamblado.

```
internal class PruebaClaseInterna  
{  
    public int MiPruebaClaseInterna;  
}
```

### 23. Número de parámetros en un método.

Si existe la necesidad de pasar más de 8~10 parámetros a un método específico, este es un buen candidato para definir una clase con parámetros y una estructura de datos tipada, de lo contrario el consumo de memoria es afectado de forma considerable, permite la corrupción de datos, y se castiga en ciclos de procesamiento al servidor lo cual pega en performance.

### 24. Estructura tipada.

Si se define una estructura tipada, se debe de generar en el constructor el `GET` y el `SET`, una estructura no puede tener solo la propiedad de escritura "`SET`" o el "`GET`" de forma independiente.

### 25. Retorno de datos complejos.

Si tienes un método que regresa una colección, es recomendable que si el resultado no trajo datos es mejor regresar una colección vacía que nula, esto hará que siempre se espere un tipo de datos y el manejo de validación o excepciones sea el más óptimo.

### 26. Versionamiento de librerías o código.

Se deberá de usar el archivo AssemblyInfo para llenar la información correspondiente al número de versión, la descripción general del ensamblado, derechos de autor, nombre de la compañía etc, en caso de clases que no generen dll se deberá de colocar al inicio de la clase dicha información.

### 26. Manejo de errores.

Se deberá de registrar en un log de base de datos el error descriptivo, el cual deberá de permitir la rastreabilidad y trazabilidad del evento, que lo ocasiono, quien lo genero, que proceso se ejecutó y en qué momento sucedió, este log deberá de ser eliminado en un periodo de tiempo optimo y deberá de ser eficiente para no afectar a las soluciones que lo están invocando.

### 27. Bitácoras de acciones.

Se deberá de registrar en una bitácora de base de datos el mensaje que se requiere almacenar, el cual deberá de permitir la rastreabilidad y trazabilidad del evento, quien lo genero, que proceso se ejecutó y en qué momento sucedió, este log deberá de ser eliminado en un periodo de tiempo optimo y deberá de ser eficiente para no afectar a las soluciones que lo están invocando.

### 28. Bloque `finally` para conexiones a base de datos, archivos, sockets, etc.

Cuando se hagan conexiones a base de datos, archivos, sockets, etc, se deberá de asegurar que en el bloque `finally` se cierren dichas conexiones a los recursos utilizados en la rutina programada, esto asegurara que, aunque una excepción no controlada exista, este código se ejecute cerrando los objetos usados no dejándolos abiertos utilizando memoria o recursos del servidor donde se está ejecutando.

### 29. Declaración de variables de forma cercana.

La declaración de las variables deberá de ser lo más cercana posible al lugar en donde es usada, se debe de hacer una declaración de variable por línea.

### 30. Concatenación de objetos tipo Cadena.

Se debe de usar la clase `StringBuilder` en lugar de `String` si es que se requiere de manipular objetos de tipo cadena en un ciclo o iteración, el objeto de tipo `String` trabaja de una forma poco eficiente ya que cada vez que concatena una cadena, internamente se descarta la cadena anterior y se crea un nuevo objeto con la concatenación de la nueva cadena, esto no sucede con `StringBuilder`.

#### Correcto:

```
public string GeneraMensaje (string[] Lineas)
{
    StringBuilder Mensaje = new StringBuilder();
    for (int i = 0; i < Lineas.Length; i++)
    {
        Mensaje.Append(Lineas[i]);
    }
    return Mensaje.ToString();
}
```

### Incorrecto:

```
public string GeneraMensaje (string[] Lineas)
{
    string Mensaje = string.Empty;
    for (int i = 0; i < Lineas.Length; i++)
    {
        Mensaje += Lineas[i]
    }
    return Mensaje;
}
```

## 31. Uso de expresiones lambda y controlador de eventos.

Si está definiendo un controlador de eventos, utilice la expresión lambda, eso hará que su código quede más compacto.

### Correcto:

```
public Form2()
{
    this.Click += (s, e) =>
    {
        MessageBox.Show(((MouseEventArgs)e).Location.ToString());
    };
}
```

### Incorrecto:

```
public Form2()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

## 32. Variables por referencia.

Si en un método se tiene la necesidad de retornar más de un parámetro se deberá de regresar dichos parámetros por referencia y la forma en que se tipe el método deberá de ser booleana, que nos debe de regresar si se ha ejecutado con éxito la rutina un **true**, de lo contrario un **false**.

```
public static extern bool DuplicateToken(IntPtr token,int levelImper, ref IntPtr hNewToken);
```

### 33. Tratamiento de Cadenas.

Para el tratamiento o validación de cadenas es recomendable usar las directivas `String.Format()`, `String.Compare()`, `String.StartsWith()` y `String.EndsWith()`.

### 34. Declaración de variables.

No se deberán declarar variables dentro de un bucle o iteraciones a menos que sea justificable su uso, como en el caso de creación de hilos o de parallel.

### 35. Parseo de datos antes de consultar a la base de datos.

Se deberá de hacer el casteo o parseo de los datos desde c sharp que no correspondan al tipo o formato definido en base de datos, ya que de no hacerlo esto genera una conversión implícita a nivel de base de datos en toda la tabla, convirtiendo los datos completamente para poder después hacer la comparación.

## 6. Buenas prácticas de programación Java Script.

### 1. Uso de operadores de comparación `===` o `==`

JavaScript utiliza dos tipos de operadores diferentes como son `===` o `==` la buena práctica dice que se debe de ocupar siempre el primero ya que toma los valores tipados y los compara, en caso de la segunda aunque los valores tengan tipos diferentes pero sean iguales en su valor los comparara.

Este segmento de código devolverá un **False**.

```
var x = 10;  
var y = "10";  
if ( x === y )
```

Mientras que este segmento de código devolverá un **True**.

```
var x = 10;  
var y = "10";  
if ( x == y )
```

Este segmento de código regresara un **True**, algo no esperado por que 10 siempre es **True**.

```
var x = 0;  
if ( x = 10 )
```

Este segmento de código regresara un **False**, algo no esperado por que 0 siempre es **False**.

```
var x = 0;  
if ( x = 0 )
```

### 2. Uso de comparaciones estrictas en un **switch**.

Se debe de considerar que cuando se usa un Switch las comparaciones son estrictas con el tipo de dato que se está usando.

Este segmento de código **SI** mostrara una alerta.

```
var x = 10;  
switch(x) {  
  case 10: alert("Hello");  
}
```

Este segmento de código **NO** mostrara una alerta.

```
var x = 10;  
switch(x) {  
  case "10": alert("Hello");  
}
```

### 3. Confusión en Suma Vs Concatenación.

```
var x = 10 + 5;      //El resultado de esta sentencia es 15
var x = 10 + "5";    //El resultado de esta sentencia es 105

var x = 10;

var y = 5;

var z = x + y;       //El resultado de esta sentencia es 15

var x = 10;

var y = "5";

var z = x + y;       //El resultado de esta sentencia es 105

var x = 10;

var y = document.getElementById("campoConValor5");

var z = x + y;       //El resultado de esta sentencia es 105
```

### 4. Errores en operaciones con punto flotante.

Todos los números de punto flotante en JavaScript se almacenan con 64 bits, en general todos los lenguajes de programación tienen dificultades con este tipo de dato y JavaScript no es la excepción

```
var x = 0.1;
var y = 0.2;
var z = x + y //El resultado de esta operación no es 0.3 si no 0.30000000000000004

if (z == 0.3) // La comparación nos dará un false ya que los dos números no son iguales
```

Ejemplo Funcional: [http://www.w3schools.com/js/tryit.asp?filename=tryjs\\_mistakes\\_floats](http://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_floats)

La recomendación es manejar los valores en entero multiplicándolos por una constante 10 y tratarlos

```
var z = (x * 10 + y * 10) / 10; //El resultado de esta operación es 0.3
```



### 5. Accediendo a un **array** con índices.

Las matrices con índices nombrados se llaman matrices asociativas o Hash, JavaScript no es totalmente compatible con este modo de declaración.

#### Correcto:

```
var persona = [];  
persona [0] = "Juan";  
persona [1] = "Montes";  
persona [2] = 32;  
var x = persona.length;    // person.length regresara el valor 3  
var y = persona [0];       // persona[0] regresara el valor "Juan"
```

#### Incorrecto:

```
var persona = [];  
persona ["nombre"] = "Juan";  
persona ["apellidoPaterno"] = "Montes";  
persona ["edad"] = 32;  
var x = persona.length;    // person.length regresara el valor 0  
var y = persona [0];       // persona[0] regresara el valor undefined  
var y = persona ["nombre"]; // persona[0] regresara el valor "Juan"
```

### 6. Reducción de actividad en ciclos o iteraciones.

#### Correcto:

```
numeroPersonas = arr.length;  
for ( i = 0; i < numeroPersonas; i++) {
```

#### Incorrecto:

```
for (i = 0; i < arr.length; i++) {
```

### 7. Reducción de acceso a objetos DOM.

El acceso al HTML DOM es muy lento en comparación a otras sentencias de JavaScript por lo que se recomienda declarar un objeto que contenga lo necesario del DOM y se trate el objeto asignado.

#### Correcto:

```
obj = document.getElementById("texto");  
obj.innerHTML = "Hola Mundo";  
obj.style.color = "red";
```

#### Incorrecto:

```
document.getElementById("texto").innerHTML = "Hola Mundo";  
document.getElementById("texto").style.color = "red";
```

### 8. Evitar variables innecesarias.

#### Correcto:

```
document.getElementById("nombreCompleto").innerHTML = nombre + " " + apellidoPaterno + " " + apellidoMaterno
```

#### Incorrecto:

```
var nombreCompleto = nombre + " " + apellidoPaterno + " " + apellidoMaterno;  
document.getElementById("nombreCompleto ").innerHTML = nombreCompleto;
```

### 9. Construcción de Cadenas desde un Array.

#### Correcto:

```
var ciudades = ["CDMX", "Guadalajara", "Mazatlán", "Tepic"];  
  
var lista = ciudades.join(",");  
document.write(lista);
```

### 10. Usar `{}` en vez de `New Object()`.

Existen muchas formas de crear objetos en java script, a continuación veremos cómo sería la forma correcta o incorrecta de hacerlo, ya que es recomendado usar el método literal que es más robusto que instanciar un Object.

#### Correcto:

```
var o = {  
  nombres: 'Gonzalo',  
  apellidos: 'Chacaltana',  
  algunaFuncion : function() {  
    console.log(this.nombres);  
  }  
};
```

#### Incorrecto:

```
var o = new Object();  
  
o.nombres = 'Gonzalo';  
  
o.apellidos = 'Perez';  
  
o.algunaFuncion = function() {  
  console.log(this.nombres);  
}
```

### 11. Usar `[]` en lugar de `New Array()`.

#### Correcto:

```
var cursos = ['Lenguaje de Programación', 'Administración de Proyectos'];
```

#### Incorrecto:

```
var cursos = new Array();
```

```
a[0] = "Lenguaje de Programación";  
a[1] = 'Administración de Proyectos';
```

### 12. Omitir el uso de la palabra clave `var` varias veces para la declaración de múltiples variables

#### Correcto:

```
var edad = 14, peso = 56, talla = 166;
```

#### Incorrecto:

```
var edad = 14;  
var peso = 56;  
var talla = 166;
```

### 13. Siempre deberá de finalizar una línea de código con punto y coma “;”

Siempre que se finalice una línea de código deberá de colocar un punto y coma “;” cuando la función de la línea de comando concrete su función, esto evitara muchos errores y hará más fácil la ubicación de un error.

#### Correcto:

```
var cadenaEjemplo = 'cadena de ejemplo';  
  
function obtenUsuario() {  
    return 'Pedro Lopez';  
}
```

#### Incorrecto:

```
var cadenaEjemplo = 'cadena de ejemplo  
  
function obtenUsuario() {  
  
    return 'Pedro Lopez'  
}
```

### 14. Uso de `Eval()` restringido.

La ejecución al vuelo de funciones mediante `Eval()` deberá de estar restringido por temas de performance, seguridad y posibles errores que se generen en el JIT(Just In Time), hay ocasiones en las que es indispensable su uso en entornos de arquitectura que requieren pasar funciones como parámetro para su ejecución, ahí está justificado su uso

### 15. Uso de `SetInterval()` o `SetTimeout` restringido.

El uso de `SetInterval()` o `SetTimeout` deberá de estar muy justificado, ya que invocar la ejecución de código cada cierto tiempo o con el delay de un tiempo previamente definido puede causar problemas de performance y de secuencia de ejecución del código

### 16. Los sectores de código en Java Script al final del HTML.

El código en JavaScript deberá de generarse en la parte final del documento HTML, esto porque los navegadores ejecutan de arriba hacia abajo una página, si se hace el llamado a un objeto que aún no existe generara errores, además si primero se carga el JavaScript hasta el final cargara la página dando una sensación de lentitud en el despliegue

### 17. `undefined` no es `null`.

En Java Script el valor `null` es para los objetos y `undefined` para variables, propiedades o métodos, es decir para ser `null` un objeto tiene que estar definido, de lo contrario este objeto sería un `undefined`.

Si desea probar si existe un objeto, esto generará un error si el objeto no está definido:

#### Correcto:

```
if (typeof myObj !== "undefined" && myObj !== null)
```

#### Incorrecto:

```
if (myObj !== null && typeof myObj !== "undefined")
```

## 18. Operador typeof

Este operador nos es funcional para obtener el tipo de dato de una variable.

```
typeof "Juan"           // Returns string
typeof 3.14             // Returns number
typeof NaN              // Returns number
typeof false            // Returns boolean
typeof [1,2,3,4]        // Returns object
typeof {nombre:Juan, edad:34} // Returns object
typeof new Date()        // Returns object
typeof function () {}   // Returns function
typeof myCar             // Returns undefined (si mi myCar no está declarado)
typeof null             // Returns object
```

## 19. Uso de **try**, **catch**, **finally** y **throw** para manejo de excepciones.

Al igual que en otros lenguajes del lado del servidor, JavaScript permite un buen manejo de excepciones por lo tanto es recomendable su uso.

```
function myFunction() {
  var message, x;
  message = document.getElementById("message");
  message.innerHTML = "";
  x = document.getElementById("demo").value;
  try {
    if(x == "") throw "is empty";
    if(isNaN(x)) throw "is not a number";
    x = Number(x);
    if(x > 10) throw "is too high";
    if(x < 5) throw "is too low";
  }
  catch(err) {
    message.innerHTML = "Error: " + err + ".";
  }
  finally {
    document.getElementById("demo").value = "";
  }
}
```

## 20. Uso JQuery.

Se deberá de hacer uso de JQuery que es un framework reconocido y avalado en el mercado por el mismo Microsoft que envuelve funcionalidades para que sean sencillas de utilizar siempre alineado a las mejores prácticas y actualizado de forma constante en el cambio del tiempo <https://jquery.com/>

## 21. No usar palabras reservadas Java Script.

El nombre de las variables no puede ser igual a alguna palabra reservada del lenguaje de programación.

*Estas son algunas de las palabras reservadas más comunes en JavaScript*

abstract	arguments	boolean	break	byte
case	catch	char	class*	const
continue	debugger	default	delete	do
double	else	enum*	eval	export*
extends*	false	final	finally	float
for	function	goto	if	implements
import*	in	instanceof	int	interface
let	long	native	new	null
package	private	protected	public	return
short	static	super*	switch	synchronized
this	throw	throws	transient	true
try	typeof	var	void	volatile
while	with	yield		

*Estas son algunas de las palabras reservadas de Objetos, Propiedades y Métodos en JavaScript.*

Array	Date	eval	function	hasOwnProperty
Infinity	isFinite	isNaN	isPrototypeOf	length
Math	NaN	name	Number	Object
prototype	String	toString	undefined	valueOf

*Estas son algunas de las palabras reservadas de Java.*

getClass	java	JavaArray	javaClass	JavaObject
----------	------	-----------	-----------	------------

*Estas son algunas de las palabras reservadas de Windows.*

alert	all	anchor	anchors	area
assign	blur	button	checkbox	clearInterval
clearTimeout	clientInformation	close	closed	confirm
constructor	crypto	decodeURI	decodeURIComponent	defaultStatus
document	element	elements	embed	embeds
encodeURIComponent	encodeURIComponent	escape	event	fileUpload
focus	form	forms	frame	innerHeight
innerWidth	layer	layers	link	location
mimeTypes	navigate	navigator	frames	frameRate
hidden	history	image	images	offscreenBuffering
open	opener	option	outerHeight	outerWidth
packages	pageXOffset	pageYOffset	parent	parseFloat
parseInt	password	pkcs11	plugin	prompt
propertyIsEnumerable	radio	reset	screenX	screenY
scroll	secure	select	self	setInterval
setTimeout	status	submit	taint	text
textarea	top	unescape	untaint	window

## 22. Uso de JSLint.

[JSLint](#) es un depurador escrito por [Douglas Crockford](#). Simplemente pegue su script, y la herramienta va a escanear rápidamente cualquier problema y errores que tuviera su código.



## 7. Base de Datos

### 1. Uso de **SELECT \***

No se debe de usar la sentencia "**SELECT \* FROM TABLA**", el traer todas las columnas de una tabla provoca un desbordamiento de IOPS en un servidor de base de datos, se debe de traer única y exclusivamente la información indispensable

### 2. Uso de Procedimientos almacenados.

Se deberá mandar llamar procedimientos almacenados, no hay que enviar declaraciones Select, Insert, Delete o Update a la base de datos a menos que la consulta sea muy simple y las tablas donde se consulte tengan menos de 1000 registros indizados, es decir que el plan de ejecución de SQL sea eficiente; en vez de eso, siempre hay que llamar procedimientos almacenados pasándole los parámetros correspondientes.

El motivo de esta mejor práctica es que cuando SQL Server recibe una consulta, como una declaración Select, lo primero que hace es compilarla, crear un plan de ejecución, y finalmente ejecutarlo; todos estos pasos consumen tiempo, cuando se invoca un procedimiento almacenado, este procedimiento almacenado puede ser compilado si es la primera vez que es llamado, o si cambian las estadísticas que le afecten, pero en caso contrario no es compilado y es almacenado en el caché; el plan de ejecución también es almacenado en el caché. El llamar un procedimiento almacenado ahorra tiempo de ejecución y recursos

### 3. Nombre de Stored Procedure.

No guardar los procedimientos almacenados con un nombre con prefijo "sp\_". Cuando el nombre de un procedimiento almacenado comienza con "sp\_", SQL Server lo busca en el siguiente orden:

En la base de datos maestra → En la base de datos determinada por los calificativos proporcionados (nombre de la base de datos o su dueño) → En cada base de datos que tenga dbo como dueño, si el dueño no fue proporcionado.

### 4. **UNION VS JOIN.**

Usar la cláusula Join con estándar ANSI. Para unir tablas es mejor usar la cláusula Join que hacer una unión por medio de la cláusula Where. A pesar de que a partir de SQL Server 7.0 las uniones de tablas usando Where pueden ser traducidas por el plan de ejecución a uniones explícitas, el hecho es que el compilador es quien hace esa conversión, lo cual le toma tiempo y recursos

### 5. Uso de Cursores en Stored Procedure.

Evitar el uso de cursores en los procedimientos almacenados. Los cursores en SQL Server son recursos muy caros, lo cual hace mas lento el desempeño de las consultas. Se debe evitar en lo posible el uso de cursores, mejor desarrolle la funcionalidad en la capa de desarrollo en c Sharp

### 6. Uso **SET NOCOUNT ON**.

Utilizar SET NOCOUNT ON. Al crear procedimientos almacenados, se puede mejorar el desempeño de ADO eliminando los valores innecesarios de la cantidad de renglones afectados, del conjunto de datos de salida, con solo agregar la instrucción SET NOCOUNT ON en el procedimiento almacenado

### 7. Uso **WITH(NOLOCK)**.

Cuando se ejecute una consulta de tipo Selección se deberá de hacer siempre usando la sentencia **WITH(NOLOCK)**, esto hace que la consulta no tenga que esperar a que la tabla se libere de afectaciones, pudiendo recuperar los datos con una fotografía del momento que se ejecutó, por otro lado mitiga interbloqueos entre proceso

### 8. Palabras reservadas en Mayúsculas.

Las sentencias propias del lenguaje TRANSACT SQL o SQL nativo deberán de estar escritas en mayúsculas sin excepción, el nombre de las tablas y otros objetos deberán de seguir la nomenclatura vista en la sección de codificación

### 9. Uso de sentencia **COALESCE** VS **ISNULL**.

Se deberá de usar por estándar la sentencia **COALESCE**.

### 10. Ejecución de selección sentencia **WHERE**.

Cuando se generen los criterios o condiciones de búsqueda en una sentencia SELECT dentro del WHERE se deberá de considerar que el orden de los filtros deberá de estar de izquierda a derecha ponderando los campos que abarquen un mayor universo a lo que abarquen menos, esto hará que la información se divida en grupos de mayor cantidad de registros a grupos con menos cantidad de registros y el resultado por ende estará en menor tiempo, así como

considerar siempre y ponderar los campos o combinación de campos que ya se encuentran indizados.

### 11. Uso de Transacciones.

Se deberá de considerar siempre iniciar una transacción dentro de SQL manejando los errores y haciendo ROLLBACK en caso de que algo falle, esto nos permitirá mantener la integridad de lo que estamos afectado.

```
BEGIN TRANSACTION;  
BEGIN TRY  
    -sentencia  
  
COMMIT;  
END TRY  
BEGIN CATCH  
    ROLLBACK;  
    SELECT ERROR_MESSAGE();  
END CATCH;
```

### 12. Tablas Temporales.

Minimizar el uso de tablas temporales. Aunque las tablas temporales generalmente son una estructura en memoria, lo cual puede parecer que es una solución de acceso rápido, eso no significa que este enfoque mejore el desempeño; de hecho, esto empeorara el desempeño. El motivo de esto es que la estructura de una tabla temporal no la conoce de antemano el optimizador de consultas, por lo tanto el optimizador necesita recompilar el plan de ejecución una vez que la conoce; esto es, después de que la tabla temporal es creada. Muchas veces, el tiempo que le toma recompilar el procedimiento es mayor que el tiempo de la ejecución misma.

### 13. Consulta en tablas de forma derivada.

Usar tablas derivadas siempre que sea posible. Las tablas derivadas tienen un mejor desempeño. Considerando la siguiente consulta para encontrar el segundo salario más alto de la tabla de Empleados:

```
SELECT MIN(Salary) FROM Employees WHERE EmpID IN ( SELECT TOP 2 EmpID FROM Employees  
ORDER BY Salary DESC )
```

La misma consulta puede ser re-escrita usando una tabla derivada, como se muestra a continuación, y será el doble de rápida que la consulta anterior.

```
SELECT MIN(Salary) FROM ( SELECT TOP 2 Salary FROM Employees ORDER BY Salary DESC ) AS  
A
```

### 14. Sentencia **LIKE**.

Evitar el uso de caracteres comodín al inicio de una palabra al usar el identificador LIKE.

Se debe intentar evitar el uso de caracteres comodín al inicio de una palabra al hacer una búsqueda usando el identificador LIKE, ya que eso ocasiona un rastreo en el índice (index scan), lo cual se contrapone con el objetivo de usar índices. El primero de los siguientes códigos genera un rastreo en el índice, mientras que el segundo genera una búsqueda en el índice (index seek).

```
SELECT IdEmpresa FROM Empresas WHERE nombre LIKE '%pples%'
SELECT LocationID FROM Empresas WHERE nombre LIKE '%s%'
```

También se deben evitar las búsquedas utilizando operadores de no igualdad (<> y NOT) ya que éstos resultan en rastreos de índices y tablas.

### 15. Uso restringido de tipo de dato **CHAR**.

Usar el tipo de datos CHAR para una columna solamente cuando no pueda contener valores nulos. Si una columna CHAR puede contener valores nulos, es tratada como una columna de ancho fijo en SQL Server 7.0+. Así que un CHAR (100) cuando sea nulo ocupara 100 bytes, resultando en un desperdicio de espacio. Para esta situación es mejor usar VARCHAR(100). Ciertamente las columnas de ancho variable tienen un poco más de overhead de procesamiento en comparación con las columnas de ancho fijo. Se debe escoger con cuidado entre CHAR y VARCHAR dependiendo del ancho de los datos que se van a almacenar.

### 16. **VARCHAR(MAX)**.

Los campos que sean de tipo cadena, deberán de tener una constante como máximo de caracteres que se pueden admitir, está restringido usar la directiva MAX.

### 17. Estándares de codificación, comentarios, sangrías y buenas prácticas.

El estándar para codificación, comentarios, sangrías, buenas prácticas visto en los apartados anteriores pueden ser aplicados también a el estándar de base de datos según sea el caso.



## ENGINE CORE

### CONVENCIÓN DE ESTANDARES

Ciudad de México  
01 de octubre de 2019

## 8. Arquitectura.

1. Siempre se deberá de usar una arquitectura multi capa.

Siempre que se comience con una nueva plataforma, se deberá de considerar genera una arquitectura multi capa.



### 2. No tercerizar componentes.

Evitar al máximo elementos de “terceros” en el entorno de desarrollo o arquitecturas, ya que 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 podríamos quedar fuera del juego, la historia está llena de casos tecnológicos con grandes expectativas, como por ejemplo Adobe Flash, Microsoft ha apostado por HTML5.

### 3. Acceso a base de datos.

Nunca se deberá de acceder a los datos desde una interface gráfica, siempre se deberá de acceder por medio de la capa de acceso a datos, esto permite unificar la forma de acceder a los datos de una forma segura y centralizada.

### 4. Uso de sentencia **Try, Catch, Finally**

Uso de sentencias Try y Catch en las diferentes capas de la arquitectura, esto te permitirá por atrapar todas las excepciones que pueden generarse.

### 5. Encapsular código.

Se debe de separar la aplicación en múltiples ensamblados y métodos, esto permitirá la re utilización de código de forma sencilla, y dejara organizado el sistema por funcionalidades más específicas.

### 6. KISS (Keep It Simple Stupid).

Mantén el código lo más simplemente estúpido, se recomienda el desarrollo empleando partes sencillas, comprensibles, con errores fáciles de detectar y corregir, rechazando lo complejo, innecesario que hacen un sistema complejo en ingeniería, el propósito es que cualquier desarrollador lo pueda entender sin recurrir a quien lo creó, ver técnicas de Peer Review y Architect Review.

## 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 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 agrupan 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.

## 9. Taxonomiza

Como llamar a los conceptos de una forma convencional y que todos los involucrados conozcan, por ejemplo, EMail o CorreoElectronico lo correcto es general un diccionario de términos que diga.

¿E-Mail? = Correo Electrónico, e-mail

## 10. 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.

## 11. Estandarización de la interface

Se deberá de generar un documento que identifique de forma clara los lineamientos y estándares de presentación en una interface, con la finalidad de que generar pantallas de alta usabilidad.