**De la Salle Bajío**

**2014**

Maestría en Tecnologías web y dispositivos móviles STD – 0121.2 Programación en lenguajes

Versión 1.3



Jorge Fragoso Mora

Pedro Iracheta

Eduardo Medina

Arturo Najar Madrigal

**Descripción del documento y control de cambios**

|  |  |
| --- | --- |
| **Descripción** | El presente documento contiene el estándar que se adoptará para la codificación de las aplicaciones o componentes desarrollados en cualquier lenguaje de programación. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Control de cambios** | | | |
| **Versión** | **Fecha** | **Adiciones / Modificaciones** | **Preparado / Revisado** |
| 1.3 | 13/08/2014 | Se anexo encabezado pie de página, formato de letras. | Jfragoso |
| 1.2 | 19/07/2014 | Revisión y validación de las reglas del documento. | Jfragoso, Piracheta, Emedina, Anajar |
| 1.1 | 12/07/2014 | Se agregó buenas practicas utilizadas en algunos lenguajes. | Jfragoso |
| 1.0 | 05/07/2014 | Se agregó las definiciones de términos comunes utilizados en lenguajes de programación | Anajar |
| 1.0 | 01/07/2014 | Creación del documento | Emedina |

# CONTENIDO

[Contenido i](#_Toc274224244)

[1. Introducción 1](#_Toc274224245)

[2. Lineamientos 1](#_Toc274224245)

[3. Definiciones 2](#_Toc274224246)

[4. Estándares 4](#_Toc274224247)

[4.1. Prefijos 4](#_Toc274224249)

[4.2. Nomenclatura 5](#_Toc274224249)

[4.2.3 Estilo de codificación 8](#_Toc274224250)

[4.2.3.1 Formato 8](#_Toc274224250)

[4.2.3.1 Codigo Comentado 9](#_Toc274224250)

[4.2.3.1 Encabezdo de programas 10](#_Toc274224250)

[4.3. Buenas practicas 11](#_Toc274224249)

[4.3.1 Lenguajes de programación 11](#_Toc274224250)

[4.3.2 Web 26](#_Toc274224250)

[6. Ciclo de revisión 35](#_Toc274224256)

1. INTRODUCCIÓN

Los estándares de codificación son una parte fundamental en el desarrollo de código, ya que permiten generarlo de forma que sea fácil de entender, desarrollar y mantener lo cual trae como beneficios:

La creación de una estructura lógica de código de fácil entendimiento.

Código legible y comprensible.

Reducción de defectos.

Elaboración de instrucciones fáciles de localizar y resolver.

Mantenimiento fácil y con menor riesgo de error.

Incrementar la productividad en el proceso del desarrollo.

Trabajar bajo estándares de codificación facilita la mantenibilidad del código, ya que, facilita la adición de nuevas características, modificación de las ya existentes, depuración de errores y mejora del rendimiento del aplicativo/sistema/módulo de software, además que ayuda a implementar las mejores prácticas y hace que el conocimiento se difunda entre las áreas involucradas de una manera más fácil.

Por lo tanto, desarrollar sistemas de software bajo técnicas de programación sólidas y buenas prácticas resulta en la implementación de proyectos fáciles de entender y mantener por distintos desarrolladores, internos y externos, sin inversión de tiempo en la compresión del código.

1. Lineamientos

**2.1.1 Ambiente de Desarrollo**

Visual Studio, PHP, Java, se han designado como la herramienta de ambiente de desarrollo oficial para el desarrollo de aplicaciones, en el tercer cuatrimestre de la maestría en tecnologías web y dispositivos móviles.

1. DEFINICIONES

|  |  |
| --- | --- |
| ***Pascal Case*** | El primer ***carácter*** de cada palabra esta en mayúscula excepto la primera palabra  Ejemplo: ***BackColor***  Esta convención es consistente con el Framework de .NET y es fácil de leer. |
| ***Camel Case*** | El primer ***carácter*** de cada palabra está en mayúscula  Ejemplo: ***backColor***  Esta convención es consistente con el Framework de .NET y es fácil de leer. |
| ***Notación Húngara*** | Consiste en prefijos en minúsculas que se añaden a los nombres de variables y que indican su tipo. EL resto del nombre indica, lo más claramente posible, la función que realizara la variable.  Ejemplo: ***iContador***  Donde i es el tipo de variable que es entero (int) |
| ***SCREAMING\_CAPS*** | Nombres de variables únicamente con mayúsculas.  Ejemplo:  ***Public static const string BACKCOLOR= “Blue”;*** |
| ***Namespaces*** | Es un ámbito que permite organizar el código y proporciona una forma de crear tipos globalmente únicos. |
| ***Assembly*** | Es una librería de código compilado que brinda funciones para construir, versionar y liberar unidades de implementación, existen dos tipos los .exe y los .dll, puede estar conformado por uno o varios archivos. En .net es un archivo ejecutable para las interfaces de Windows. |
| ***Collection*** | Permite hacer referencia a un grupo de elementos relacionados como un objeto único. Los elementos de una colección sólo necesitan estar relacionados por el hecho de existir en la colección. Los elementos de una colección no tienen que compartir el mismo tipo de datos. |
| ***Delegate*** | Método para llamar a un objeto y de forma opcional llama al método. |
| ***JIT*** | (***Just in time***) Compilador de lenguaje intermedio de Microsoft (***MSIL***), encargo de ejecutar únicamente la parte de código necesaria para la ejecución. |
| ***path*** | Ejecuta operaciones en instancias de ***String*** que contiene información de rutas de archivos o directorios. |
| ***script*** | Conjunto de instrucciones |
| ***Código Duro*** | Datos directamente en el código fuente del programa, en lugar de obtener esos datos de una fuente externa como un fichero de configuración o parámetros de la línea de comandos, o un archivo de recursos.  Ejemplo: ***if(estatus = “Activo”);***  La cadena ***“Activo”*** está siendo colocada en duro y si cambiara sería necesario recompilar el código fuente |
| ***Magic Number*** | Es cualquier número usado en el código fuente que no sean 0 y 1. Siempre se deben definir constantes con un nombre de semántica y utilizar el nombre de la semántica en el código del programa. Esto hará que el código sea más fácil de leer.  Ejemplo: ***if(password.length()>7) //Incorrecto***  ***If(passeord.length()>LongitudMaxima) //correcto*** |

1. ESTÁNDARes

* 1. Prefijos
     1. Controles

Los objetos deben ser nombrados con un prefijo consistente que haga más fácil identificar el tipo de objeto.

A continuación se listan las convenciones para algunos de los objetos.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Control** | **Prefijo** | **Control** | **Prefijo** | **Control** | **Prefijo** |
| Label | lbl | CheckBox | chk | WebControl | wc |
| TextBox | txt | RadioButton | rbt | UserControl | uc |
| DropDownList | ddl | HTML td | htd | Repeater | rpt |
| ListBox | lsb | HTML tr | htr | Connection | con |
| LinkButton | btn | HTML table | htb | DataAdapter | adp |
| ImageButton | btn | DataView | dvw | Transaction | tx |
| Button | btn | Relation | rel | Command | cmd |
| ComboBox | cmb | DataColumn | dtc | DataReader | dr |
| GridView | gdv | DataTable | tbl | DataRow | drw |
| Menu | mnu | DataSet | dst | DataBase | db |

Ejemplo: txtNombre, ddlMunicipio, btnGuardar, rptClientes.

* 1. Nomenclatura

De todos los componentes que conforman un estándar de codificación, los estándares de nomenclatura son los más visibles y podría decirse que los más importantes.

Tener un estándar para nombrar los diversos objetos en su programa le ahorrará una enorme cantidad de tiempo, durante el proceso de desarrollo por sí mismo y también durante los trabajos posteriores de mantenimiento

**4.2.1 Reglas Generales**

**4.2.1.1** Siempre usar nombres en notación ***Camel Case*** o ***Pascal Case*** para mayor referencia revisar el punto 4.2.2

**4.2.1.2** No usar nombres que inicien con un ***carácter*** numérico.

**4.2.1.3** Solo usar notación ***Húngara*** en controles

Ejemplo**: *strNombre o iContador***

**4.2.1.4** No usar abreviaciones a menos que el nombre completo sea mayor a 15 caracteres.

**4.2.1.5** No usar palabras reservadas de ningún lenguaje de programación como nombres.

**4.2.1.6** No agregue prefijos redundantes o sin sentido y sufijos a los identificadores

Ejemplo:

//Incorecto

Public enum ColoresEnum{ . . .}

Public class CVehiculo{ . . .}

Public struct EstructuraRectangulo{ . . .}

**4.2.1.7** No incluir el nombre de clase padre dentro de un nombre de propiedad

Ejemplo:

//Correcto

Cliente.Nombre

//Incorrecto

Cliente.NombreCliente

**4.2.1 Uso de nombres y sintaxis**

|  |  |  |
| --- | --- | --- |
| **Tipo** | **Descripción** | **Ejemplo** |
| **Archivo de proyecto** | ***Pascal Case***, siempre debe coincidir con el nombre del ensamblado y el espacio de nombre raíz. | **Espacio de Nombres** OBJ.CreditoHipotecario.Web.ReglasNegocio  **Ensamblado**:  OBJ.CreditoHipotecario.Web.ReglasNegocio.dll  **Nombre de Proyecto**:  OBJ.CreditoHipotecario.Web.ReglasNegocio.csproj |
| **Espacio de nombres** | ***Pascal Case***, Utilizar <NombreEmpresa>.<Producto>.<Tecnología>.<Componente>. | OBJ.CreditoHipotecario.Web.ReglasNegocio  OBJ.CreditoHipotecario.Web.Controladores |
| **Ensamblado** | Si los Ensamblados contienen solo un Espacio de nombres o están contenidos en un Espacio de nombres raíz, el nombre del Ensamblado debe de ser el mismo que el del Espacio de nombres. | OBJ.CreditoHipotecario.Web.ReglasNegocio.dll  OBJ.CreditoHipotecario.Web.Controladores.dll |
| **Clases y Estructuras** | ***Pascal Case***, las clases no deben tener el mismo nombre que el espacio de nombres en el cual residen. Cualquier acrónimo de tres o más letras deben ser en Pascal Case, no todas las letras en mayúculas. Evitar abreviaciones, y siempre usar pronombes | DocumentoXml FormaPrincipal ControlEncabezado ListaClientesDataSet (dataset) |
| **Clases Collection** | Seguir convención de nombres para clases, pero agregar “***Collection***” al final del nombre. | ***AltaCollection*** |
| **Clases Delegate** | Seguir convención de nombres para clases, pero agregar “***Delegate***” al final del nombre. | ***AltaRespuestaDelegate*** |
| **Clases Exception** | Seguir convención de nombres para clases, pero agregar “***Exception***” al final del nombre. | ***TransaccionInvalidaException*** |
| **Atributos de Clase** | Seguir convención de nombres para clases, pero agregar “***Atributo***” al final del nombre. | ***WebServiceAtributo*** |
| **Interfaces** | Seguir convención de nombres para clases, pero iniciar el nombre con "***I***" y colocar en mayúscula la siguiente letra. | ***IConsultaSocio*** |
| **Enumeraciones** | Seguir la nomenclatura de clases. No agregar "***Enum***" al final del nombre de la enumeración. Si esta representa un conjunto de banderas, finalizar el nombre con plural. | **OpcionesBusqueda (Banderas)**  **AceptarRechazarRegla (enum)** |
| **Métodos** | ***Pascal Case***, sin guiónes bajos, excepto en el manejo de eventos. Evitar abreviaciones. | **VB: Public Sub HacerAlgo(...)**  **C#: public void HacerAlgo(...)** |
| **Propiedades** | ***Pascal Case*** sin guiónes bajos. Evitar el uso de abreviaciones. | **C#: private string\_name;**  **Public int name**  **{**  **Get { retunr \_name;}**  **Set { \_name=value;}**  **}** |
| **Parámetros** | ***Camel Case***. Evitar abreviaciones. | **VB: ByRef idRegistro As Integer**  **C#: ref int grabado** |
| **Variables locales** | ***Camel Case*** | **VB: Dim idRegistro As Integer**  **C#: int idRegistro ;** |
| **Variables Protegidas** | ***Camel Case*** iniciando con guión bajo | **VB: Private \_idRegistro As Integer**  **C#: private int \_idRegistro ;** |
| **Controles en Formas** | ***Camel Case*** Utilizar el prefijo definido para cada tipo de control dentro de las formas. | **txtIdUsuario, lblEncabezado, btnSubmit** |
| **Constantes** | Mismas convenciones utilizadas para variables ***Públicas/Privadas***. Si es expuesta al público usar el ***PascalCase***. Si es privada a una Función/Sub, usar ***CamelCase.***  No usar ***SCREAMING\_CAPS***. | **NombreClase.NombreConstantePublica**  **ConstanteLocal**  **\_privateClassScopedConstant** |
| **Dependencia de Conexiones e interfaces** | lowerCamelCase  UpperCamelCase | **conBuroCredito** |

**4.2.3 Estilo de Codificación.**

El estilo de codificación es la causa de más inconsistencia y controversia entre los desarrolladores. Cada desarrollador tiene una preferencia y rara vez hay dos iguales. Sin embargo, un diseño consistente, formato y la organización son clave para crear código fácil de mantener.

**4.2.3.1 Formato**

1. No declarar más de un espacio de nombres por archivo.

2. No poner múltiples clases en un archivo simple.

3. Los ensamblados deben tener el mismo nombre que el ***espacio de nombres***

4. Generar un archivo por clase con el mismo nombre de la clase.

5. Utilizar un identado consistente: Si es con espacios, no usar tab y si es con tab, no mezclar con espacio. Ser consistente en el tamaño recomendado son 2 espacio y tabs con ancho de 2 caracteres.

6. Los comienzos y finales de bloques {} deben estar al mismo nivel del código y comenzar al mismo nivel, lo que implica que no se pueden estar en diferente columna

7. Si se usan líneas blancas para separar bloques de código, deben ser una sola línea. En lo posible no usarlas: un código bien identado no requiere de espacios para clarificarse.

8. Utilizar un solo espacio antes y después de cada operador.

9. Declarar cada variable independientemente no en la misma sentencia.

10. Colocar las sentencias y el espacio de nombres “**using**” junto en la parte superior del archivo

11. Agrupar las clases implementadas por tipo en el siguiente orden.

* Variables “Member” (Campos, estructuras y enumerados)
* Propiedades
* Constructores y Finalizadores
* Métodos Privados
* Métodos Públicos

12. Usar “#***region***” para agrupar fragmentos de código. Si se agrupa adecuadamente usando “#región”, la página debe mostrarse así cuando todas las definiciones se contraen

**4.2.3.2 Código comentado**

Es la documentación interna que esta formada por los comentarios que los desarrolladores escriben dentro del código fuente durante la fase de construcción.

4.2.3.2.1 Cada clase debe contar con un encabezado, esto es para indicar datos generales, como fecha de creación, creador, modificaciones, propósito, etc

4.2.3.2.2. Todos los comentarios deben ser escritos en el mismo lenguaje, ser correctos gramaticalmente y contener puntuación apropiada

4.2.3.2.3. Usar // o /// nunca usar /\* ….\*/

4.2.3.2.4 No usar marcos tipográficos.

4.2.3.2.4 Usar comentarios en línea para explicar omisiones, problemas conocidos y objetivos específicos de un algoritmo.

4.2.3.2.6 No usar comentarios en línea para explicar código obvio. Un código bien escrito es documentado por si mismo

4.2.3.2.7 Incluir comentarios usando banderas de listas de tareas clave, para permitir filtrar los comentarios

4.2.3.2.8 Se debe utilizar esquema de comentario ***XML*** para todas las definiciones de campos, propiedades, eventos, procedimientos, funciones, estructuras, enumerados, interfaces, delegados definidos en una clase o módulo, excepto los ***espacio de nombres.***

Los comentarios XML deben estar en la línea de insertado, directamente en el código fuente. Esto facilita mantenerlos actualizados a medida que evolucione el código.

Siempre que se incluyan comentarios <***summary***> incluir comentarios <***param***>, <***return***>, y <***exception***> en las secciones donde aplique

4.2.3.2.9 Los comentarios deben ubicarse encima o a un lado de la línea a la cual describen

4.2.3.2.10 Siempre agregar ***tags CDATA*** a los comentarios que contengan código y otro marcado embebido a fin de evitar problemas de codificación

**4.2.3.3 Encabezado de programas**

Se recomienda que cada programa cuente con un encabezado, esto es para indicar datos generales, como fecha de creación, creador, modificaciones, propósito, etc. Se propone el siguiente formato para el encabezado:

\\Proposito: <Describir el propósito del programa>

\\Parametros: <Nombre de parámetros y breve descripción de cada uno de ellos>

\\Fecha creación: <Fecha de creación>

\\Creador: <Nombre o siglas del programador>

\\Fecha modificación: <Fechas de modificación, separadas por comas>

\\Modifico: <Nombre o siglas de programadores que modificaron el programa, separados por comas>

\\ Dependencia de Conexiones e interfaces: <Especificar las conexiones con los diferentes sistemas >

* 1. Buenas practicas
     1. Lenguajes de Programación
     2. Siempre preferir hacer comparaciones numéricas que comparaciones de caracteres.

|  |
| --- |
| **If (strData == ””)**  **Se puede cambiar por**  **If (strData.Lenght == 0)** |

* + 1. Siempre que se pueda, evitar el uso de una negación, rescribiendo el orden de una selección o modificando el tipo de lazo.

|  |
| --- |
| **If(!Option1)  Hacer1 else  Hacer2**  **Puede rescribirse como:**  **If(Option1)  Hacer2 else  Hacer1** |

* + 1. Siempre incluir una sentencia default en switch. Si ya se contemplaron todas las opciones, colocar un mensaje de error. Un error común es un dato que no es procesado por que no coincide con lo que se esperaba.
    2. Incluir una sentencia finally en los try. Filtrar los errores en el orden de más probable ocurrencia y de más expecifico a más genérico. Procurar siempre enviar a una clase de trazado un mensaje del error ocurrido, dejando a la configuración, decidir a donde va este mensaje.

|  |
| --- |
| **public void ExceptionDemo() {  try  {  ' Calls a method in a secondary component.  SecondaryComponent.MyMethod();  }  catch(ArgumentOutOfRangeException e)  {  ' Insert code to fix this specific exception instance.  }  catch(UnauthorizedAccessException e)  {  ' Insert code to fix this specific exception instance.  ' This block will catch any other exceptions that are thrown and  ' package them inside a new exception which will be thrown to the  ' client application.  }  catch(Exception e)  {  throw new Exception("An unhandled exception occured", e);  ' The Finally block specifies code that will be executed in the event  ' of an exception.  }  finally  {  ' Insert code to save data, release references, and so on.  } }** |

* + 1. Para recorres un array es preferible un for ya que es más rápido que un foreach. En general un foreach requiere GetEnumerator, MoveNext, Reset, y Current implementados en la colección o array, de allí que tienda a ser más lento. Sólo se aconseja utilizar cuando la claridad del código supera la necesidad de alto rendimiento.
    2. No utilizar bloques de decisión para sustituir asignaciones binarias:

|  |
| --- |
| **if(strData == "")**  **bVacio = true;**  **else**  **bVacio = false;)** |

Dentro del if() va una expresión booleana y si al ser true, se asigna true y al ser false se asigna false, entonces lo correcto es hacer la asignación directamente:

|  |
| --- |
| **bVacio = strData == "";** |

Nota: C# evalúa primero el signo ==, por lo que no es necesario los paréntesis.

* + 1. Si se requiere hacer casting, preferir no generar una excepción:

|  |
| --- |
| **string s1 = (string)o; // Casting utiliza 'castclass' y genera una excepción si no es del tipo correcto.**  **string s2 = o as string; // 'as' utiliza 'isinst', por lo que no genera excepción.** |

Nota: Expresión como 'is', tampoco genera excepción.

|  |
| --- |
| **if(o is string) // 'is' también utiliza 'isinst'**  **{**  **// ...**  **}** |

* + 1. Cuando se asignen propiedades con valore enum, generar la excepción adecuada, ya que un error de asignación no es validada por C# directamente. Esto es importante cuando se asignan valores provenientes de bases de datos.

|  |
| --- |
| **enum Color : byte**  **{**  **Red,**  **Green,**  **Blue**  **}**  **//Este código retorna sin error, pese a que color varia de 0 a 2.**  **Color ColorFalso()**  **{**  **return (Color)3;**  **}**  **void PintarConColor(Color c)**  **{**  **// Validar que el color es correcto**  **if (!Enum.IsDefined(typeof(Color), c))**  **throw new ArgumentOutOfRangeException("c");**  **//Se procede normalmente, ya que fue validada.**  **}**  **Esta función generará la excepción adecuada**  **PintarConColor(ColorFalso());** |

* + 1. Cuando se ejecute una operación y se llegue a una condición que no debería ocurrir nunca, es posible que continuar operando genere más daño que abortar, en ese caso hacerlo con un 'Fail Fast' .

|  |
| --- |
| **void Operacion()**  **{**  **if (/\*...\*/)**  **{**  **// Algo muy malo ocurrió, es necesario abortar.**  **Environment.FailFast("...Descripción del problema...");**  **}**  **// ...**  **}** |

* + 1. Utilizar excepciones de doble pase, en casos en que se cambien contextos de seguridad:

|  |
| --- |
| **impersona a una cuenta de administrador... IntPtr administratorToken = /\*...\*/; WindowsImpersonationContext context = WindowsIdentity.Impersonate(administratorToken); try {  // Se realiza algo con la cuenta de administración. } finally {  // Se elimina el contexto al anterior para retormar el usuario sin derechos administrativos...  if (context != null)  context.Undo(); }**  **// Se impersona a una cuenta de administrador... IntPtr administratorToken = /\*...\*/; WindowsImpersonationContext context = WindowsIdentity.Impersonate(administratorToken); try {  // Se realiza algo con la cuenta de administración. } finally {  // Se elimina el contexto al anterior para retormar el usuario sin derechos administrativos...  if (context != null)  context.Undo(); }** |

Si durante el try ocurre un error, es posible que sea capturado por un Catch colocado en una función superior en el stack y ya que el los filtros catch se ejecuta primero que el finally, este se ejecuta con la cuenta administradora, lo que permite un ataque de elevar el tipo de privilegios.

|  |
| --- |
| **// Se impersona a una cuenta de administrador...**  **IntPtr administratorToken = /\*...\*/;**  **WindowsImpersonationContext context = WindowsIdentity.Impersonate(administratorToken);**  **try**  **{**  **try**  **{**  **// Se realiza algo con la cuenta de administración.**  **}**  **finally**  **{**  **// Se elimina el contexto al anterior para retormar el usuario sin derechos administrativos...**  **if (context != null)**  **context.Undo();**  **}**  **}**  **catch**  **{**  **throw;**  **}** |

Esto fuerza a ejecutar el catch exterior, luego el finally interno y luego hace el relanzamiento de la excepción, no permitiendo que se realice con la cuenta administrativa.

* + 1. Utilizar tryParse para evitar excepciones durante las conversiones de string a valores:

|  |
| --- |
| **string qty = Request["Quantity"]; int parsedQty; try {  parsedQty = int.Parse(qty); } catch (FormatException) {  // Error en formato de la cadena... } Este código usa una excepción para controlar el flujo, lo cual es una mala practica, lo correcto es:**  **string qty = Request["Quantity"]; int parsedQty; if (!int.TryParse(qty, out parsedQty))  // Decir al usuario que se requiere un número...** |

Nota: Este error de concepto fue corregido en, pero lamentablemente aún es necesario en .NET 1.0 y 1.1.

* + 1. Si se va a pedir mucha memoria, prevenir condiciones de Out of Memory o OOM Para ello utilizar la clase MemoryFailPoint

|  |
| --- |
| **using (MemoryFailPoint gate = new MemoryFailPoint(100)) {  // Operación que requiere de 100MB de memoria... }** |

* + 1. Para mejorar el rendimiento de un assembly, evitar que se compile cada vez que se cargue, ya que siempre se generará el mismo código en la misma máquina. Para esto usar Native Image Generation o NGen.

El utilitario ngen.exe esta situado en el directorio del framework y permite colocar el assembly en un sitio denominado Native Image Cache. El cargador del Framework busca allí primero antes de cargar un assembly.

La línea de comando:

|  |
| --- |
| **ngen install foo.dll** |

instalará una versión en código nativo que cargará mucho más rápido y no tendrá el retardo asociado a la primera llamada de cada método.

* + 1. Si se tienen assemblys compilados en forma nativa, usar la tecnica de Hard Binding para que se cargen todos los assemblys desde el principio de la ejecución, ejemplo:

|  |
| --- |
| **using System.Runtime.CompilerServices; [assembly: Dependency("Bar", LoadHint.Always)] [assembly: Dependency("Baz", LoadHint.Sometimes)] class Foo { /\*...\*/ }** |

* + 1. Se puede mejorar la carga de un Assembly nativo aplicando la técnica del String Freezing Esto carga los String de los Assembly en un área especial del Gargabe Collector y evita los alineamientos en tiempos de ejecución haciendo más rápida la carga. Sólo funciona con NGEN, no hace nada con Assembly que se ejecuten en JIT.

|  |
| --- |
| **ejemplo: using System; using System.Runtime.CompilerServices;  [assembly: StringFreezing] class Program { /\*... \*/ }** |

* + 1. Utilizar la comparación con .Equal para evitar el proceso de Boxing. La función funcion2() es al menos dos veces más rápida que la primera, en versión release.

|  |
| --- |
| **private void funcion1()**  **{**  **bool b = str1 == str2;**  **}**  **private void funcion2()**  **{**  **bool b = string.Equals(str1, str2);**  **}** |

Nota: Esto es conveniente en casos de que se comparen objetos en que se halla modificado el operador ==.

* + 1. Si una clase implemente el IDispose, debe crearse en un bloque using:

|  |
| --- |
| **using (MyClass mc = /\*...\*/)**  **{**  **// Use 'mc'.**  **}**  **Ya que esto es compilado como:**  **MyClass mc = /\*...\*/**  **try**  **{**  **// Use 'mc'.**  **}**  **finally**  **{**  **if (mc != null)**  **mc.Dispose();**  **}** |

Lo que asegura se llame a su Dispose para liberar recursos.

* + 1. Usar System.Text.StringBuilder para concatenar cadenas de caracteres en un lazo. Pero no usarlo para un número corto de cadenas.

Nota: no resulta práctico para concatenaciones cortas, especialmente si se hacen en un sola sentencia. Siendo Str1 y Str2 String pertencientes a la clase, la funcion2 será más rápida ya que se compila como una sola concatenación.

|  |
| --- |
| **private void funcion1()**  **{**  **StringBuilder sw = new StringBuilder();**  **sw.Append("Hola");**  **sw.Append(" como");**  **sw.Append(" estas.");**  **str1 = sw.ToString();**  **}**  **private void funcion2()**  **{**  **str1 = "Hola" + " como" + " estas.";**  **}** |

* + 1. Cuando se utilice Array, usar Length para obtener el tamaño máximo del Array y GetLength para obtener el tamaño actual del Array. Nota: Length siempre retorna el tamaño que se dimensionó el array, no cuantos elementos están allí al hacer la llamada, cosa que si hace el GetLength.
    2. Usar Pascal Case para el nombre de los Namespace, las clases y los métodos. No deben tener guión bajo.
    3. Usar Camel Case para nombres de variables y parámetros de métodos.
    4. Los Assembly’s deben tener el mismo nombre que el Namespace.
    5. Evitar usar abreviaciones como nombre de variables y funciones.
    6. Generar un archivo por clase con el archivo con el mismo nombre de la clase.
    7. Los comentarios deben estar en el mismo nivel que el código que describen.
    8. Utilizar un identado consistente: Si es con espacio, no usar tab y si es con tab, no mezclarla con espacio. Ser consistente en el tamaño. Recomendado son 2 espacios, 3 espacios y tabs con ancho de 3 caracteres.
    9. Los comienzos y finales de bloques {} deben estar al mismo nivel del código y comenzar al mismo nivel, lo que implica que no pueden estar en diferente columna.
    10. Si se usa líneas blancas para separar bloques de códigos, debe ser una sola línea. En lo posible no usarlas: un código bien identado no requiere de espacios para clarificarse.
    11. Utilizar un sólo espacio antes y después de cada operador.
    12. Ningún archivo debería tener más de 400 líneas de código. Si es así debería refactorizarse.
    13. Los nombres de los métodos deben decir lo que hace el método y el de las funciones, lo que retorna.
    14. No colocar en HardCode números o cadenas. Deben usarse constantes.
    15. Evitar campos públicos.
    16. No debe estar en HardCode path y nombres de archivos.
    17. Si falta o esta errado un parámetro de configuración, debe generarse un mensaje de error claro que identifique el problema.
    18. Todo comentario debe estar correctamente escrito, con la puntuación correcta.
    19. Evitar hacer un catch genérico, ya que puede esconder problema de codificación.
    20. Evitar try-catch muy extensos, ya que no permiten un manejo acertado de los errores.
    21. Usar Array para listas constantes y Colecciones para listas dinámicas.
    22. Las clases no deben tener el mismo nombre que el Namespace donde se encuentran.
    23. Las colecciones siguen las normas de nomenclatura de las clases, deben terminar en Collection.
    24. Los delegados siguen las normas de nomenclatura de las clases, deben terminar con Delegate.
    25. Las excepciones siguen las normas de nomenclatura de las clases, terminadas en Exception.
    26. Los atributos siguen las normas de nomenclatura de las clases, terminadas en Atributo.
    27. Las interfaces siguen las normas de nomenclatura de las clases, pero comienzan con “I” mayúsculas seguidas de una letra en mayúscula.
    28. Los eventos siguen las normas de nomenclatura de los métodos, pero pueden tener guión bajo.
    29. No debe existir funciones y variables que se diferencien de otras por el Case de su nombre.
    30. Al trabajar con archivos usar System.IO.Path.DirectorySeparatorChar en vez de "\\" o @"\".
    31. Usar string.Empty en vez de "".
    32. Utilizar clases para almacenar constantes, como la de aplicaciones. Ejemplo:

|  |
| --- |
| **Settings.ModuloPrincipal**  **donde ModuloPrincipal es**  **public string ModuloPrincipal**  **{**  **get**  **{**  **return ConfigurationSettings.AppSettings[AppSettingsKeys.ModuloPrincipal];**  **}**  **}**  **y AppSettingsKeys.ModuloPrincipal es valor de una clase** |

En vez de:

|  |
| --- |
| **ConfigurationSettings.AppSettings["ModuloPrincipal"]** |

Esto facilita la modificación de la aplicación, pero no abusar, ya que existe penalidad de ejecución.

* + 1. Evitar hacer un rethrow de la forma "throw e" ya que se pierde la información del stack. Usar 'throw'. Esto evita el típico error de "exception hiding", que hace la depuración más difícil.

No se dejan objetos abiertos por excepciones:

|  |
| --- |
| **void ReadFromFile(string filename)**  **{**  **string line;**  **StreamReader reader = File.OpenText(filename);**  **while ((line = reader.ReadLine()) != null)**  **{**  **DoSomething(line);**  **}**  **reader.Close();**  **}**  **¿Qué pasa si DoSomething lanza una excepción? se deja abierto el Reader y como la variable sale de contexto, queda un recurso perdido...**  **void ReadFromFile(string filename)**  **{**  **StreamReader reader = File.OpenText(filename);;**  **try**  **{**  **string line;**  **while ((line = reader.ReadLine()) != null)**  **{**  **DoSomething(line);**  **}**  **}**  **finally**  **{**  **if (reader != null) ((IDisposable) reader).Dispose();**  **}**  **}**  **o lo que es lo mismo, pero mejor escrito:**  **void ReadFromFile(string filename)**  **{**  **using(StreamReader reader = File.OpenText(filename))**  **{**  **string line;**  **while ((line = reader.ReadLine()) != null)**  **{**  **DoSomething(line);**  **}**  **}**  **}** |

* + 1. No hacer nulo una variable local. Esto es complicar el código para algo que ya va a ocurrir cuando se salga de contexto del método o función. Sólo tiene sentido si se esta en un lazo de mucha duración y se prevé que el GC (Garbage Collector) pueda procesar mientras se realiza el lazo.
    2. Si se implementa IDispose en la función Dispose debe incluirse GC.SuppressFinalize

|  |
| --- |
| **protected virtual void Dispose(bool disposing)**  **{**  **if(disposing)**  **GC.SuppressFinaliza(this);**  **LiberarRecursos();**  **}** |

Esto evita que se llame este método múltiples veces: Cuando lo llame el usuario explícitamente o mediante un using, y cuando el GC elimine el objeto de memoria.

* + 1. Evitar query’s dinámicos que reciben entradas del usuario.
    2. No almacenar data sensible en el ViewState.
    3. No almacenar data sensible en los log de la aplicación.
    4. Al ocurrir una excepción, no revelar información detallada al cliente, pero almacenar esta en el server.
    5. Aplicar instrumentación a las aplicaciones: Esto permite detectar actividades extrañas como muchos login fallidos, etc.
    6. Utilizar parámetros de salida en StoreProcedure en combinación con ExecuteNonQuery, en vez de un select pequeño que retorna una sola fila.
    7. Validar todas las entradas, independiente de su fuente por tipo, tamaño, formato y rango.
    8. No enviar datos no validados de nuevo al usuario: esta es la típica fuente de cross-site scripting attack.
    9. No aplicar cache en data sensible.
    10. No dejar espacio entre el paréntesis y la sentencia de bloque:

|  |
| --- |
| **[if()**  **while()**  **NombreFuncion()**  **switch(Errores.Tipo)** |

* + 1. Los bloques deben comenzar en su propia línea a menos que todo se pueda escribir en una, y siempre comienza al mismo nivel de identado que este:

|  |
| --- |
| **if(Validado)**  **{**  **...**  **}**  **while(Cuenta++ > 0)**  **{**  **...**  **}**  **get { return Monto; }**  **set { Monto = value; }**  **--En SQL BEGIN END son equivalentes a {} en C# por lo que aplican las mismas normas.**  **IF(@Data > 0)**  **BEGIN**  **..**  **END** |

* + 1. Entre operadores y un operando o más, debe haber un sólo espacio.

|  |
| --- |
| **if(K == 5) //no debe ser (K ==5) ni (K==5) ni (K == 5)**  **while(i > 0)** |

* + 1. Siempre que se realize un Assert via código, hacerlo dentro de un try-finally, y en el finally revertir la llamada al assert con un RevertAssert tan pronto termine la necesidad del permiso.

|  |
| --- |
| **[SecurityPermission p = new SecurityPermission(SecurityPermissionFlag.UnmanagedCode);**  **p.Assert();**  **try**  **{**  **// Make a P/Invoke call (safely)**  **MyPInvokeFunction();**  **}**  **finally**  **{**  **CodeAccessPermission.RevertAssert();**  **}** |

No hacerlo de este modo puede abrir vulnerabilidades de seguridad.

* + 1. Nunca utilizar un usuario con permiso de modificar bases de datos como sa, en un ambiente de producción para obtener data.
    2. No utilizar [assembly:AllowPartiallyTrustedCallers] en un assembly con strong name a menos que se haya realizado una verificación profunda de su seguridad. Esto abre posibilidades para que un código sin fulltrust realice llamadas al assembly y pueda abrir un hueco de seguridad.
    3. No seguir el Antipatten conocido como "Boat anchor": Dejar código inservible dentro del código fuente. Esto conduce a mayor tiempo de compilación y entorpece el mantenimiento. Un código considerado inútil debe ser removido del código fuente.
    4. Evitar el Antipattem denominado "Accumulate and fire": Crear una serie de variables globales de los cuales dependan muchas funcionalidades.
    5. No usar Magic Number: Siempre clarificar el por qué una constante recibe un determinado valor.
    6. Evitar los Singletons cuando no son necesarios: el abuso de este patrón, que busca reducir recursos y tiempo al crear un objeto compartido, ocasiona problemas de concurrencia. No es una panacea y hay que usarlo cuando sólo cuando convenga.
    7. Cuando se separen líneas muy largas, identarlas de modo que el inicio de los parámetros coincidan:

|  |
| --- |
| **public string ValidarFecha(int intDAnio, int intDMes,**  **int intDDia, int intHAnio,**  **int intHMes, int intHDia)** |

* + 1. El uso de la coma (,) para separar parámetros o listar elementos sigue la norma común en las lenguas de origen latino: no se puede dejar espacio antes y se debe dejar un sólo espacio después:

|  |
| --- |
| **ConsultarCambios("at", "id", true);** |

* + 1. La ortografía debe respetarse en los comentarios y regiones.
    2. Los comentarios deben ubicarse encima o a un lado de la línea a la cual describen.

|  |
| --- |
| **Es un error hacer:**  **public string Nombre**  **// Propiedad que contiene nombre**  **{**  **Debe ser:**  **// Propiedad que contiene nombre**  **public string Nombre**  **{** |

* + 1. Web
    2. Evitar ataques de one-click validando por un viewstateuserkey: se encripta el tiempo y sesión del usuario, de modo que esta página no pueda enviarse en nombre de otro usuario.
    3. Utilizar HtmlEncode y UrlEncode para mostrar cualquier entrada del usuario en una página Web para evitar ataques al site.
    4. Si no se requiere respuesta, agregar el atributo OneWay a los WebMethod de un WebService:

|  |
| --- |
| **[SoapDocumentMethod(OneWay=true)]**  **[WebMethod(Description="Returns control immediately")]**  **public void SomeMethod()**  **{…}** |

* + 1. Evitar DataBind de página y usar el del control específico.
    2. Evitar capas remotas intermedias en lo posible. Esto disminuye el rendimiento en gran medida.
    3. Evitar el uso de DataBinder.Eval. Utilizar en lo posible un casting al tipo correcto.

|  |
| --- |
| **<ItemTemplate>**  **<tr>**  **<td><%# ((DataRowView)Container.DataItem)["field1"] %></td>**  **<td><%# ((DataRowView)Container.DataItem)["field2"] %></td>**  **</tr>**  **</ItemTemplate>**  **También se puede utilizar el evento OnItemDataBound para evitar el uso del DataBinder.Eval**  **Ejemplo:**  **public void Page\_Init(object sender, System.EventArgs e)**  **{**  **rptr.ItemDataBound += new RepeaterItemEventHandler(rptr\_OnItemDataBound);**  **}**  **public void Page\_Load(object sender, System.EventArgs e)**  **{**  **// TODO: Retrieve data from a database,**  **// and bind the data to a list control.**  **}**  **protected void rptr\_OnItemDataBound(Object sender, RepeaterItemEventArgs e)**  **{**  **DataRowView drv = (DataRowView)e.Item.DataItem;**  **Response.Write(string.Format("<td>{0}</td>",drv["field1"]));**  **Response.Write(string.Format("<td>{0}</td>",drv["field2"]));**  **Response.Write(string.Format("<td>{0}</td>",drv["field3"]));**  **Response.Write(string.Format("<td>{0}</td>",drv["field4"]));**  **}** |

* + 1. Identificar partes estáticas de las páginas y crear user control, para que sea puesta en cache y evitar regenerarlas constantemente.

|  |
| --- |
| **<%@ OutputCache Duration="120" VaryByParam="none" %>**  **// Partial caching for 120 seconds**  **[System.Web.UI.PartialCaching(120)]**  **public class WebUserControl : System.Web.UI.UserControl**  **{**  **// Your Web control code**  **}** |

Ejemplo:

|  |
| --- |
| **[main.aspx]**  **<%@ Register TagPrefix="app" TagName="header" src="header.ascx" %>**  **<%@ Register TagPrefix="app" TagName="menu" src="menu.ascx" %>**  **<%@ Register TagPrefix="app" TagName="advertisements"**  **src="advertisements.ascx" %>**  **<%@ Register TagPrefix="app" TagName="footer" src="footer.ascx" %>**  **<html>**  **<body>**  **<table>**  **<tr><td colspan=3><app:header runat=server /></td></tr>**  **<tr><td><app:menu runat=server /></td><td>Dynamic Content</td><td><app:advertisements runat=server /></td></tr>**  **<tr><td colspan=3><app:footer runat=server /></td></tr>**  **</table>**  **</html>**  **[header.ascx]**  **<%@Control %>**  **Application Header – Welcome <% GetName() %>**    **[menu.ascx]**  **<%@Control %>**  **<%@ OutputCache Duration="30" VaryByParam="none" %>**  **Menu**  **[advertisements.ascx]**  **<%@Control %>**  **<%@ OutputCache Duration="30" VaryByParam="none" %>**  **Advertisements**  **[footer.ascx]**  **<%@Control %>**  **<%@ OutputCache Duration="60" VaryByParam="none" %>**  **Footer** |

* + 1. Utilizar DataReader para obtener un gran conjunto de datos, como el usado en exportación de archivos. Jamás pretender usar un DataSet, ya que el uso de esta funcionalidad puede aumentar el uso de la memoria por encima del límite usado por ASP.NET para generar un reciclaje del pool de IIS.
    2. Utilizar url completas para ir entra HTTP y HTTPS. De lo contrario, la elección del protocolo se realiza tarde, significando un degrade de rendimiento.
    3. Utilizar Kernel Cache en Windows Server 2003. Esto mejora el rendimiento.

|  |
| --- |
| **<system.web>**  **…**  **<httpRuntime enableKernelOutputCache="true" …./>**  **…**  **<system.web>** |

* + 1. Preferir llamar en forma repetida a Response.Write a una concatenación de una cadena para luego imprimirla con Response.Write. Usar un objeto inmutable siempre tiene penalidad.
    2. Utilizar seguridad declarativa para verificar por credenciales. Ejemplo:

|  |
| --- |
| **<[PrincipalPermissionAttribute(SecurityAction.Demand, Name = "MyUser", Role = "User")]**  **public static void PrivateInfo()**  **{**  **//Print secret data.**  **Console.WriteLine("\n\nYou have access to the private data!");**  **}** |

Esto impide que se ejecute código bajo una credencial que no tiene permiso, y tiene la ventaja que es verificado al cargarse el assembly, por lo que no se ejecuta si no se tienen los permisos correspondientes.

* + 1. Hacer requerimientos de permisos vía declaraciones de seguridad:

|  |
| --- |
| **[FileIOPermissionAttribute(SecurityAction.PermitOnly, All = @"C:\ ")]**  **[FileIOPermissionAttribute(SecurityAction.Deny, All = @"C:\MyLog.log")]** |

Esto haría que no se pueda acceder a C:\ de otra forma como [\\MyServer\C$\](file:///\\MyServer\C$\) y luego elimina permiso para acceder a un recurso. La importancia de esto, es que si por un error del Framework o de nuestro software, como un buffer overflow, alguien tiene posibilidad de ejecutar un código dentro de nuestro componente, será denegado el permiso de hacer lo que se supone nuestro componente no debe hacer. Si nuestro componente no accede a disco, entonces decirlo explícitamente.

* + 1. Los ASPX deben tener el mínimo posible de cambios de contexto.Un cambio de contexto se refiere a la inclusión de bloques <% %> o bloques de script. Mientras menos sean, es mejor el rendimiento ya que no hay tantos cambios entre diferentes engine para procesar la página.
    2. El identado debe ser uniforme: o siempre es tab o siempre es espacio, pero no una mezcla de ambos, pero en aspx siempre debe ser un tabulador y la separación de parámetros etc. deben ser siempre un espacio. Se recomienda 4 espacios en código.

|  |
| --- |
| **<tab>while(Count<spc>!=<spc>0)**  **<tab>{**  **<tab><tab>j<spc>=<spc>Count;** |

* + 1. No debe dejarse caracteres de espacios o tabulaciones "basura" en el código:

|  |
| --- |
| **<tab><tab><tab>try**  **<tab><tab><tab>{<tab><tab><tab><tab>**  **<tab><tab><tab><tab>XmlNodeList<spc>nodo<spc>=<spc>xmlData.SelectNode(sNombre).Item(0).ChildNodes;<spc><spc><spc><spc>**  **En este caso los <Tab> y los <spc> de más deben eliminarse:**  **<tab><tab><tab>try**  **<tab><tab><tab>{**  **<tab><tab><tab><tab>XmlNodeList<spc>nodo<spc>=<spc>xmlData.SelectNode(sNombre).Item(0).ChildNodes;** |

Nota: En Visual Studio se hacen visible al presionar la secuencia <CTRL>+R,<CTRL>+W o al seleccionar Edit->Advanced->View white space

* + 1. No dejar activado el viewstate en páginas que no lo necesiten.
    2. Los <tr> y <td> deben escribirse en una línea cuando son cortos:

|  |
| --- |
| **<tr><td>&nbsp;</td></tr>**  **Es preferible a:**  **<tr>**  **<tab><td>&nbsp;</td>**  **</tr>** |

* + 1. Las sentencias en aspx deben son xml y por tanto deben cerrar como xml

|  |
| --- |
| **<asp:Image id="img1" runat="server"></asp:Image>**  **Debe escribirse como:**  **<asp:Image id="img1" runat="server"/>** |

* + 1. No incluir formato innecesario en HTML:

|  |
| --- |
| **<TABLE id="Table1" cellSpacing="1" cellPadding="1" width="80%" align="center" border="0">**  **debe ser**  **<TABLE id="Table1" cellSpacing="1" cellPadding="1" width="80%" align="center">** |

Ya que border es por defecto cero. Otra razón: el formato debe aplicarse preferiblemente por ccs y no estar en duro en el html. Formato innecesario es el ancho de una columna puesta en todas las filas ya que basta con hacerlo en la primera. (No se vale el argumento de que lo coloca un Designer: el programador es responsable del código que entrega independientemente de con que herramienta haya trabajado).

* + 1. En etiquetas preferir incluir el valor como propiedad:

|  |
| --- |
| **<asp:Label id="Error" runat="server" CssClass="cc1" text="NroError"/>**  **a**  **<asp:Label id="Error" runat="server" CssClass="cc1">NroError</asp:Label>** |

* + 1. No dejar en las páginas aspx comentarios o información no útil como:

|  |
| --- |
| **<meta name="GENERATOR" Content="Microsoft Visual Studio .NET 7.1"><meta content="C#" name="CODE\_LANGUAGE">**  **<span></span>** |

Los comentarios están bien en .cs ya que son compiladas, pero no tiene sentido enviarlas al cliente ya que ocupan ancho de banda y reducen la escalabilidad del site. En el caso descrito arriba, indicar la herramienta con que se diseño no es útil y menos indicar que se trabaja con lenguaje c# cuando la página usa codebehind.

* + 1. El formato de HTML sigue el esquema de capas y esto debe respetarse: No se puede cerrar un bloque que no existe (cada </td> debe tener su <td> correspondiente) y asi sucesivamente y no se puede tener texto como <tr><td><span></td></span></tr> donde el bloque de <span> se superpone a un bloque <td>.
    2. Todos los tag en HTML deben estar en minúsculas, es decir <tr> y no <TR>. Antiguamente era mayúscula, por lo que hay que configurar las herramientas acordemente. Recordar que en Visual Studio esto es configurable.
    3. No puede existir un campo en una pagina ASPX o HTML con longitud abierta: Todos deben estar limitados a la longitud del campo en la base de datos.
    4. Todo campo de ser posible debe poder seleccionarse con el uso de aceleradores. Incluso en una página web no es sano pensar que todos disponen de un Mouse y que esta sea la forma preferida de navegación.
    5. Toda validación de formato, al dar el error debe mencionar el formato correcto. Ejemplo: “Formato de fecha inválido: Introduzca la fecha en formato DD/MM/YYYY”
    6. Todo comentario debe estar en tercera persona.
    7. Evitar llamar a Response.Redirect con excepción. Se debe incluir el segundo parámetro para evitar la excepción de ThreadAbort.
    8. El identado puede mejorarse. Ejemplo:

ChequeEmitidoDetalle.aspx inicialmente tiene un tamaño de 5020 bytes  
Después de:

1. Eliminar espacios innecesarios

2. Eliminar enter innecesarios

3. Llevar todos los tag a minúsculas

4. Identar el código HTML con HTML y el script con script (Son contextos diferentes y por tanto se deben trabajar como tales)

El tamaño final es de 4.711

La carpeta de ChequeDevuelto fue revisada y se aplicaron la mayoría de las recomendaciones. Esta se puede usar como guía.

* + 1. Una ventaja del using sobre llamar directamente al Dispose, es que el orden de llamada se corresponde con el orden inverso de creación de objetos, lo que evita que se pueda llamar Dispose a un objeto cuando otro objeto tiene una referencia a él. Otra ventaja es que resulta más simple de visualizar a que objeto se esta invocando el Dispose y donde. La necesidad del cambio es media, ya que actualmente aunque se esta liberando el recurso, no se esta haciendo en un finally, lo que implica que de ocurrir una excepción, no se estaría llamando al IDispose de una manera controlada.

Con el cambio, quedaría algo como:

|  |
| --- |
| **private void MostrarImagen()**  **{**  **//System.Drawing.Bitmap imagenAnv = ObtenerImagen("ChequeAnverso.jpg");**  **using(System.Drawing.Bitmap imagenAnv = ObtenerImagen(40069464))**  **using(Bitmap imagenRev = ObtenerImagen("ChequeReverso.jpg"))**  **{**  **Color oColor = Color.White;**  **using(SolidBrush oBrush = new SolidBrush(oColor))**  **{**  **//Proceso la Imagem de Cheque Anverso.**  **using(Graphics oGraphicAnv = Graphics.FromImage(imagenAnv))**  **{**  **//Borro Nro de Cuenta.**  **oGraphicAnv.FillRectangle(oBrush, 0, 0, 90, 30);**  **//Borro Firma.**  **oGraphicAnv.FillRectangle(oBrush, 200, 80, 150, 40);**  **Response.ContentType = "image/jpeg";**  **imagenAnv.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);**  **}**  **}**  **}**  **}** |

* + 1. Aunque se está validando un ListBox, ¿Qué ocurre cuando por error se incluya otra opción y no se codifique?, Si se lee de base de datos ¿Qué ocurre si alguien modifica el valor en la base de datos?, Aunque esto pueda parecer excesivo, a medida que la aplicación se complica, este tipo de medidas permite ahorrar tiempo al depurar, además que permite otro nivel de validación de contenido.
    2. Muchos programadores acostumbran dejar muchos espacios entre las líneas de código para facilitar la lectura: el problema con exagerar es que en muchos casos hace que los métodos ocupen más de lo que se ve en el monitor de una sola vez sin paginar (usar el scroll bar). Esto permite que deje variables declaradas sin utilizar ya que no "ve" el código como una totalidad.
    3. Utilizar el recolector de elementos GarbageCollector para realizar el seguimiento de los objetos asignados en la memoria, de esta forma el recolector periódicamente reclama la memoria asignada a los objetos para los que no existen referencias válidas.

|  |
| --- |
| **Imports System**  **Namespace GCCollectInt\_Example**  **Class MyGCCollectClass**  **Private maxGarbage As Long = 10000**  **Public Shared Sub Main()**  **Dim myGCCol As New MyGCCollectClass**  **'Determine the maximum number of generations the system**  **'garbage collector currently supports.**  **Console.WriteLine("The highest generation is {0}", GC.MaxGeneration)**  **myGCCol.MakeSomeGarbage()**  **'Determine which generation myGCCol object is stored in.**  **Console.WriteLine("Generation: {0}", GC.GetGeneration(myGCCol))**  **'Determine the best available approximation of the number**  **'of bytes currently allocated in managed memory.**  **Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(False))**  **'Perform a collection of generation 0 only.**  **GC.Collect(0)**  **'Determine which generation myGCCol object is stored in.**  **Console.WriteLine("Generation: {0}", GC.GetGeneration(myGCCol))**  **Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(False))**  **'Perform a collection of all generations up to and including 2.**  **GC.Collect(2)**  **'Determine which generation myGCCol object is stored in.**  **Console.WriteLine("Generation: {0}", GC.GetGeneration(myGCCol))**  **Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(False))**  **Console.Read()**  **End Sub**  **Sub MakeSomeGarbage()**  **Dim vt As Version**  **Dim i As Integer**  **For i = 0 To maxGarbage - 1**  **'Create objects and release them to fill up memory**  **'with unused objects.**  **vt = New Version**  **Next i**  **End Sub**  **End Class**  **End Namespace** |

Como es importante en un grupo de trabajo, que todos codifiquen igual ya que esto mejora la facilidad con que un programador corrija, complete o modifique el trabajo de otro, se recomienda que hasta cosas simples como esta sean estandarizadas.

1. CICLO DE REVISIÓN

Anual