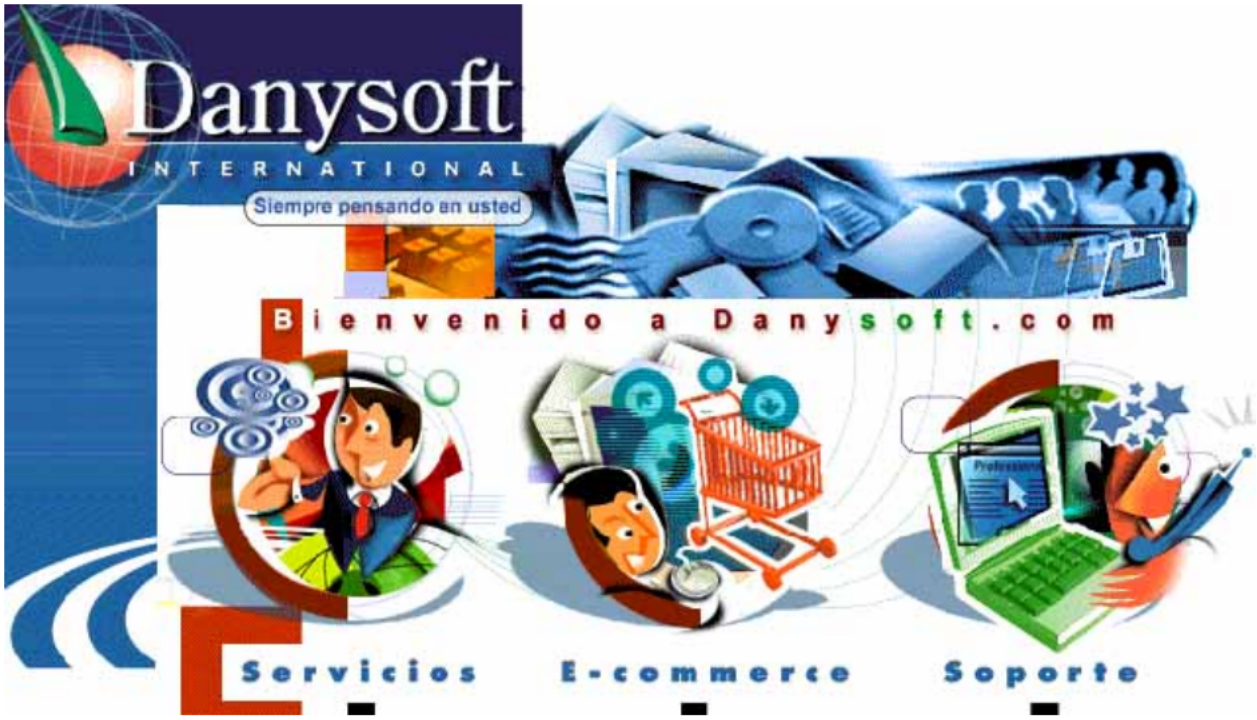
Guías técnicas Grupo Danysoft:

Reglas para codificación en C#



Leonardo Diez
Equipo Grupo Danysoft
septiembre de 2003 - (902) 123146
www.danysoft.com

Este documento se ha realizado utilizando *Doc-To-Help*®, distribuido por :

Danysoft Internacional
Avda de España 17
28100 Alcobendas – Madrid
Tfno. 902.123146
Fax. 902.123145
http://www.danysoft.com
http://www.danyshop.com
danysoft@danysoft.com

Reglas para codificación en C#

1. Introducción

La plataforma .NET ofrece numerosas ventajas a la hora de desarrollar aplicaciones. Entre otras cosas, permite que el código generado sea mucho más claro y fácil de mantener. Sin embargo, para alcanzar este objetivo es necesario establecer reglas que definan un formato común para todo el código que generamos. Esto permitirá compartir código entre los desarrolladores sin implicar costes adicionales.

Dado que son muchos los que hoy día comienzan a desarrollar nuevos productos con C# o VB.NET, os brindamos de forma gratuita este documento, que evitará dedicar tiempo a crear algo que seguramente ya alguien ha tenido que hacer. Esta guía es una base que puede ser utilizada tal como está, modificada según las preferencias de los programadores que vayan a utilizarla o extendida con nuevas reglas; lo importante es que sea respetada por todos los miembros del equipo de desarrollo que la utilice.

El hecho de utilizar esta guía tampoco garantiza una total independencia del código de su creador; es importante complementar su uso con la correspondiente documentación.

2. Organización de los ficheros

a) Ficheros de código fuente

- Mantener las clases y los ficheros cortos, con no más de **2.000** líneas de código y que estén claramente divididas en estructuras.
- Crear un fichero para cada clase, con el nombre de la clase como nombre del fichero y la extensión correspondiente. Esta regla puede ser ignorada en los casos en que una clase sea muy dependiente de otra, en cuyo caso podría ser definida en el fichero de la clase importante, o incluso como una clase interna de aquélla.

b) Estructura de directorios

- Crear un directorio para cada nombre de espacio. No utilizar puntos en el nombre de los directorios. Esto hará más fácil la asociación entre directorios y espacios de nombres. Ejemplo: usar **MiEmpresa/MiProyecto/CapaDatos** para el espacio de nombres **MiEmpresa.MiProyecto.CapaDatos**, y no **MiEmpresa/MiProyecto.CapaDatos**.

3. Indentación

a) Espacios en blanco

- Utilizar **Tab** para indentar el código. Nunca utilizar espacios.

b) Ajuste de línea

- Cuando una expresión no quepa en una sola línea de código, dividirla de acuerdo a estos principios:

    - Nueva línea después de una coma.
    - Nueva línea después de un operador aritmético.
    - Buscar operadores con la prioridad más alta posible.
    - Alinear la nueva línea con el comienzo de la sección en la que se encuentra el código.

  Ejemplos:
  - Luego de una coma:

```
llamadaLarga(expr1, expr2,
                expr3, expr4, expr5);
```

  - Luego de un operador aritmético:

```
var = a * b / (c - g + f) +
        4 * z;
```

  - Evitar esto:

```
var = a * b / (c - g +
        f) + 4 * z;
```

  Evitar el último caso, ya que la división ocurre dentro del paréntesis y esto puede dar lugar a confusión.

- Para mantener las líneas alineadas usar **Tab** y complementar con **espacios**. Este es el único caso donde se permite el uso de espacios para indentar.

4. Comentarios

a) Comentarios de bloque

- Los comentarios de bloque deben ser evitados. Para descripciones de clases y sus miembros, utilice los comentarios con /// para generar documentación. Si en algún caso se deben utilizar, use el siguiente formato:

```
/* Línea 1
 * Línea 2
 * Línea 3
*/
```

  Este formato hace que el bloque sea más legible. Igualmente, los comentarios de bloque raramente son útiles. Básicamente, la utilidad que tienen es que permiten comentar temporalmente grandes bloques de código.

b) Comentarios de línea

- Los comentarios de línea se utilizan para explicar línea a línea el código fuente. También se utilizan para comentar líneas de código temporalmente.
- Estos comentarios deben tener el mismo nivel de indentación que el código que describen.

- La longitud de un comentario no debe exceder la del código que explica. Si esto ocurre, probablemente el código es demasiado complejo, lo que implica un riesgo de errores.

c) Comentarios de documentación

- En la plataforma .NET, Microsoft ha introducido un sistema de generación de documentación basado en comentarios XML. Se deben utilizar comentarios de C# con etiquetas XML internas, y seguir las siguientes directivas:
  - Las líneas deben ser precedidas por tres barras para ser aceptadas como comentarios XML.
  - Las etiquetas pueden ser de dos tipos:
    - Elementos de documentación: Etiquetas como **<summary>**, **<param>** o **<exception>**. Estas se relacionan con los elementos de la API del programa que se documenta. Estas etiquetas suelen tener atributos como **name** o **cref**, que son revisados por el compilador, por lo que deben tener valores válidos.
    - Formato / referencia: Etiquetas como **<code>**, **<list>** o **<para>**. Estas permiten definir el formato que la documentación debe tener, o crear referencias a elementos del código.
- Ejemplo sencillo:

```
/// <summary>
/// Esta clase es la responsable de ...
/// </summary>
```

- Ejemplo de varias líneas:

```
/// <exception cref="EmailException”>
/// Esta excepción es lanzada cuando ocurre
/// un error durante el envío de un e-mail.
/// </exception>
```

5. Declaraciones

a) Declaraciones de variables locales

- Se recomienda realizar sólo una declaración por línea, ya que esto permite añadir un comentario explicativo a dicha declaración. Ejemplo:

```
int nivel; // nivel de indentación
int tamaño; // tamaño de la tabla
```

- Sin embargo, el uso de nombres claros para las variables puede evitar la necesidad de dichos comentarios explicativos. En este caso, sólo se permite definir dos o más variables en la misma línea cuando todas estas son del mismo tipo de datos. Ejemplo:

```
int nivelIndentacion, tamañoTabla;
```

b) Declaraciones de miembros de clases e interfaces

- Cuando se codifican clases e interfaces con C#, se debe seguir las siguientes reglas:
  - No incluir espacios entre el nombre de un método y los paréntesis donde se encuentran los parámetros del método.
  - La llave de apertura debe aparecer en la línea siguiente a la declaración.
  - La llave de clausura debe comenzar una línea, alineada verticalmente con su llave de apertura.

- Ejemplo:

```
class MiEjemplo : MiClase, IMiInterface
{
	int miInt;
	public MiEjemplo(int miInt)
	{
		this.miInt = miInt;
	}
	void Incrementar()
	{
		++miInt;
	}
	void MetodoVacio()
	{
	}
}
```

c) Inicializaciones

- Inicializar las variables locales lo antes posible; si se puede, durante la declaración. Ejemplo:

```
string nombre = miObjeto.Nombre;
int valor = fecha.Hours;
```

Nota: Utilizar la sentencia **using** cuando se inicializa un cuadro de diálogo. Ejemplo:

```
using (OpenFileDialog openFileDialog = new
OpenFileDialog()) {
...
}
```

6. Sentencias

a) Sentencias simples

- Cada línea debe contener sólo una sentencia.

b) Sentencias de retorno

- Una sentencia de retorno no debe utilizar paréntesis para encerrar el valor de retorno.

No usar: `return (n * (n + 1) / 2);`
Usar: `return n * (n + 1) / 2;`

c) Sentencias if, if-else, if else-if else

- Las sentencias **if**, **if-else** e **if-else if-else** deben tener la siguiente apariencia:

```
if (condicion) {
	// acciones
}

if (condicion) {
```

```
        // acciones
} else {
        // otras acciones
}

if (condicion) {
        // acciones
} else if (condicion) {
        // otras acciones
} else {
        // otras acciones más
}
```

d) Sentencias for / foreach

- Una sentencia **for** debe tener la siguiente forma:

```
for (int i = 0; i < 5; ++i) {
        // acciones
}
```

  o de una sola línea (considerar el uso de la sentencia **while** en estos casos):

```
for (initializacion; condicion; cambio) ;
```

- Una sentencia **foreach** debe ser así:

```
foreach (int i in ListaEnteros) {
        // acciones
}
```

  Nota: Utilizar llaves incluso cuando haya una sola sentencia en el bucle.

e) Sentencias while/do-while

- Una sentencia **while** debe ser escrita de esta forma:

```
while (condicion) {
        // acciones
}
```

- Una sentencia **while vacía** debe ser así:

```
while (condicion) ;
```

- Una sentencia **do while** debe tener la siguiente forma:

```
do {
        // acciones
} while (condicion);
```

f) Sentencias switch

- Una sentencia **switch** debe ser de la siguiente forma:

```
switch (condicion) {
case A:
```

```
        // ...
        break;
case B:
        // ...
        break;
default:
        ...
        break;
}
```

g) Sentencias try-catch

- Una sentencia **try catch** debe tener uno de los siguientes formatos:

```
try {
        // ...
} catch (Exception) {}

try {
        // ...
} catch (Exception e) {
...
}

try {
        // ...
} catch (Exception e) {
        //...
} finally {
        //...
}
```

7. Espaciado

a) Líneas en blanco

- Las líneas en blanco mejoran la legibilidad del código. Separan los bloques de código que están relacionados lógicamente.
- Usar dos líneas en blanco entre:
  - Secciones lógicas de un fichero
  - Definiciones de clases o interfaces
- Usar una línea en blanco entre:
  - Métodos
  - Propiedades
  - Sección de variables locales y la primera sentencia de un método
  - Secciones lógicas dentro de un método
- Las líneas en blanco debe ser indentadas como si contuvieran una sentencia, lo que hará más fácil la inserción de código en el futuro.

b) Espacios entre términos

- Debe haber un espacio luego de una coma o un punto y coma. Ejemplos:

```
Prueba(a, b, c);
```

No usar:

```
Prueba(a,b,c);
```

```
Prueba( a, b, c );
```

- Debe haber un espacio alrededor de los operadores (excepto los unarios, como el de incremento o la negación lógica). Ejemplos:

```
a = b; // no usar a=b;
for (int i = 0; i < 10; ++i)
      // acciones
```

No usar:

```
for (int i=0; i<10; ++i)
      // acciones
for(int i=0;i<10;++i)
      // acciones
```

c) Formato de tabla

- Un bloque lógico de líneas debe tener un formato de tabla. Ejemplo:

```
string nombre   = "Mr. Ed";
int miValor     = 5;
Prueba aPrueba  = new Prueba(5, true);
```

- Usar **espacios** para dar el formato de tabla. No utilizar tabulaciones, ya que se puede perder el formato si se cambia la cantidad de espacios por tabulación.

8. Convenios de nombres

a) Mayúsculas / minúsculas

1. Estilo PasCal

Este convenio determina que la primera letra de cada palabra debe ser mayúscula. Ejemplo: `ContadorPrueba`.

2. Estilo caMel

Este convenio determina que la primera letra de cada palabra debe ser mayúscula, exceptuando la primera palabra. Ejemplo: `contadorPrueba`.

3. Mayúsculas

Este convenio determina que toda la palabra va en letras mayúsculas. Sólo utilizar para nombres que representan abreviaturas de uno o dos caracteres. Ejemplos: PI, E.

b) Directivas para asignación de nombres

- Un nombre debe describir la función semántica del elemento, es decir, qué hace, o qué valor representa.
- La notación Húngara es un conjunto de prefijos y sufijos predefinidos que se añaden a los nombres de variables para indicar su tipo. Esto no respeta lo que dice el párrafo anterior, por lo que **no** se debe utilizar esta notación para los nombres. La única excepción a dicha regla es el código de la GUI (Interfaz Gráfica de Usuario). A todos los nombres de campos y variables que contengan elementos de la GUI debe añadirse, como sufijo, su tipo sin abreviar. Ejemplos:

```
System.Windows.Forms.Label emailLabel;
```

```
System.Windows.Forms.TextBox emailTextBox;
```

- El uso de guión bajo está prohibido.

c) Nombres de clases

- Usar el estilo PasCal.
- Utilizar sustantivos o frases en función de sustantivo.
- No añadir prefijos de clase.

d) Nombres de interfaces

- Usar el estilo PasCal.
- Utilizar sustantivos, frases en función de sustantivo o adjetivos que describan comportamiento.
- Añadir el prefijo I, manteniendo la primera letra del nombre en mayúsculas.

e) Nombres de enumeraciones

- Usar el estilo PasCal, tanto para el nombre de la enumeración como para los valores.
- Utilizar nombres en singular para enumeraciones que obligan a escoger sólo un valor. Ejemplo: la enumeración MessageBoxDefaultButton permite determinar cuál de los botones de un cuadro de mensaje es el predeterminado.
- Utilizar nombres en plural para enumeraciones que permiten escoger valores. Ejemplo: La enumeración MessageBoxButtons permite escoger qué botones se incluyen en un cuadro de mensaje.
- No añadir prefijos ni sufijos al nombre del tipo o de los valores.

f) Nombres de campos estáticos, readonly y constantes

- Usar el estilo PasCal.
- Utilizar sustantivos, frases en función de sustantivo o abreviaciones de sustantivos.

g) Nombres de parámetros y campos no constantes

- Usar el estilo caMel.

h) Nombres de variables

- Usar el estilo caMel.
- Utilizar `i, j, k, l, m, n`, etc. para los contadores locales cuando se utilizan para bucles triviales.

i) Nombres de métodos

- Usar el estilo PasCal.
- Utilizar verbos o frases verbales.

j) Nombres de propiedades

- Usar el estilo PasCal.

- Utilizar sustantivos o frases en función de sustantivo.
- Para las propiedades normales, utilizar el mismo nombre que el campo que almacena su valor.

k) Nombres de eventos

- Usar el estilo PasCal.
- Utilizar verbos.
- Utilizar tiempo presente para eventos generados antes de que algo suceda y pasado para eventos generados luego que algo sucedió.
- Utilizar el sufijo **EventHandler** para el delegado que define la firma de los manejadores del evento.
- Llamar a los parámetros del evento **sender** y **e**.
- Utilizar el sufijo **EventArgs** para las clases de argumentos del evento.

** Resumen **

Tipo	*Estilo*	*Notas*
Clase/ estructura	PasCal	
Interface	PasCal	Comienza con I
Enumeración nombre	PasCal	
Enumeración valores	PasCal	
Clases de excepción	PasCal	Finaliza con Exception
Campos públicos	PasCal	
Métodos	PasCal	
Nombres de espacios	PasCal	
Propiedades	PasCal	
Campos protegidos/privados	caMel	
Parámetros	caMel	

9. Prácticas de programación

a) Visibilidad

- No definir campos públicos; hacerlos privados. Para éstos, no añadir la palabra clave **private**, ya que éste es el valor por defecto. Utilizar propiedades para hacer visibles los valores de dichos campos. La excepción a esta regla son los campos estáticos y constantes.

b) No utilizar números 'mágicos'

- No utilizar valores númericos constantes directamente en el código fuente. Reemplazarlos luego puede ser un proceso muy propenso a errores e inproductivo. Utilizar constantes en su lugar. Ejemplo:

```
public class Matematicas
{
      public const double PI = 3.1415926583;
}
```

10. Ejemplos de código

a) Posicionamiento de las llaves

```
namespace EjemploDeLlaves
{
	public enum TipoPrueba {
		Probarme,
		Probarte
	}

	public class Prueba
	{
		TipoPrueba tipoPrueba;
		public TipoPrueba Prueba {
			get {
				return tipoPrueba;
			}
			set {
				tipoPrueba = value;
			}
		}

		void HacerAlgo()
		{
			if (prueba == TipoPrueba.Probarme) {
				//...se hacen cosas
			} else {
				//...se hacen otras cosas
			}
		}
	}
}
```

- Las llaves deben comenzar una nueva línea solo después de:
  - Declaraciones de nombres de espacios
  - Declaraciones de clases, interfaces o estructuras.
  - Declaraciones de métodos.

b) Nombres de variables

- En lugar de:

```
for (int i = 1; i < numero; ++i) {
	cumpleCriterio[i] = true;
}
for (int i = 2; i < numero / 2; ++i) {
	int j = i + i;
	while (j <= numero) {
		cumpleCriterio[j] = false;
		j += i;
	}
}
for (int i = 0; i < numero; ++i) {
	if (cumpleCriterio[i]) {
		Console.WriteLine(i + " cumple el criterio");
	}
}
```

utilizar un sistema más inteligente:

```
for (int candidatoPrimo = 1; candidatoPrimo < numero;
     ++candidatoPrimo)
{
      esPrimo[candidatoPrimo] = true;
}
for (int factor = 2; factor < numero / 2; ++factor) {
      int numeroFactorizable = factor + factor;
      while (numeroFactorizable <= numero) {
            esPrimo[numeroFactorizable] = false;
            factorableNumber += factor;
      }
}
for (int candidatoPrimo = 0; candidatoPrimo < numero;
     ++candidatoPrimo)
{
      if (esPrimo[candidatoPrimo]) {
            Console.WriteLine(candidatoPrimo  + " es
      primo.");
      }
}
```

- Nota: Variables contadoras o utilizadas para acceder a propiedades indexadas generalmente deben ser llamadas i, j, k, etc. Pero en casos como éste tiene sentido reconsiderar esa regla. En general, cuando las variables de contador o de acceso a propiedades indexadas se reutilizan, es preferible darles un nombre significativo.