

GML Software

# Estándares de codificación para C#

---

## Manual de Procedimientos

Febrero de 2016

Este documento describe los estándares de codificación para C# utilizados en el Departamento de Desarrollo.

Archivo:	EstandaresCodificacion_version1.docx
Páginas:	40
Fecha:	
Autor:	Pedro Arciniegas
Contacto:	
Versión	1

## TABLA DE CONTENIDO

Estándares de codificación para C# .....	1
Tabla de contenido .....	2
Introducción .....	6
Objetivo .....	6
Alcance .....	6
Condiciones de uso de este documento .....	6
Convenciones utilizadas en este documento .....	6
Terminología y definiciones .....	7
Fuentes de información consultadas .....	8
Guía Rápida.....	9
Convenciones de nomenclatura .....	9
Estilos de codificación .....	10
Convenciones de nomenclatura .....	12
Guías genéricas de nomenclatura .....	12
1. REC – Utilizar nombres en inglés para todos los elementos que componen una solución: carpetas, archivos e identificadores.....	12
2. OBL – Utilizar siempre la convención Pascal o Camel .....	12
3. OBL – No utilizar mayúsculas y minúsculas para diferenciar identificadores.....	12
4. OBL - No utilizar nomenclatura Húngara (iCount, strName). .....	12
5. REC – Utilizar abreviaturas sólo cuando el nombre completo sea demasiado extenso. ....	12
6. REC – No utilizar guion bajo (underscore) en los identificadores. ....	12
7. REC – Nombrar los identificadores de acuerdo a su significado y no a su tipo.....	12
8. REC – Utilizar “Can”, “Has” o “Is” para nombrar variables booleanas.....	13
9. REC – Utilizar nombres de operaciones aritméticas para nombrar variables que las implementan. ....	13
10. REC - Nombrar los namespaces de acuerdo a un patrón bien definido.....	13
11. REC – Utilizar sustantivos o frases sustantivos para el nombre de clases o struct.....	13
12. REC – El nombre de las interfaces debe tener el prefijo “I”. ....	13
13. REC – Utilizar nombres similares para la implementación predeterminada de una interface.....	13
14. REG – Utilizar nombres en singular para tipos enumerados. ....	14
15. REG – Utilizar nombres en plural para enumeraciones que representan campos de datos.....	14
16. REC – Evitar el uso de las letras I O y los números 1 y 0 juntos. ....	14
17. OBL – Agregar “EventHandler” para delegados relacionadas a eventos. ....	14
18. OBL – Agregar “CallBack” para delegados relacionadas a métodos callback.....	15
19. REC – No agregar “CallBack” a los métodos callback.....	15
20. REC – Utilizar verbos para nombrar eventos. ....	15
21. OBL – No agregar el texto “Event” a los nombres de los eventos.....	15
22. OBL – Utilizar sufijos “ing”y “ed” para indicar pre-eventos y post-eventos. ....	15
23. OBL – Agregar el prefijo “On” para los Event Handlers. ....	15
24. OBL – Nombrar las exception classes utilizando el sufijo “Exception”. ....	15
25. OBL – No incluir el nombre de la clase dentro del nombre de las propiedades. ....	15
26. OBL – Nombrar las assemblies DLL luego del namespace que las contiene. ....	16

27.	OBL – Utilizar notación Pascal para nombrar los archivos de código fuente.	16
28.	OBL – Nombrar el archivo de código fuente igual a la clase principal que implementa.	16
Comentarios y Documentación Embebida		17
Guías para la escritura de comentarios		17
29.	OBL – Cada archivo debe contener un cabezal identificadorio.	17
30.	OBL – Utilizar “//” para los bloques de comentarios.	17
31.	REC – Los comentarios serán escritos en español si no hay una petición expresa del cliente para hacerlo en inglés.	17
Guías para la escritura de documentación embebida		17
32.	OBL – Utilizar tags XML para documentar tipos y miembros de clase.	17
33.	REC – Utilizar #region para agrupar miembros no públicos.	17
Ciclo de vida de Los objetos		18
Guías para la declaración e inicialización de variables		18
34.	REC – Declarar e inicializar las variables próximo al lugar donde son utilizadas.	18
35.	REC – Inicializar las variables en el punto de su declaración.	18
36.	OBL – Declarar cada variable en una sentencia diferente.	18
37.	OBL – Utilizar const para la declaración de constantes.	18
38.	REC – Utilizar public static readonly para definir instancias predeterminadas de objetos.	18
Guías para la utilización de destructores de clases		19
39.	REC – Asignar el valor null a los campos de referencia cuando no se necesiten.	19
40.	REC – Evitar la implementación de métodos destructores.	19
41.	OBL – Si es necesaria la implementación de un destructor, debe utilizarse GC.SuppressFinalize.	19
42.	OBL – Implementar IDisposable si una clase utiliza recursos no administrados.	20
43.	OBL – No accede a ningún miembro tipo referencia en el destructor.	21
Control de Flujo		22
Guías para el uso de variables dentro de sentencias loop		22
44.	OBL – No cambiar una variable del loop dentro de un bloque for.	22
Guías para el uso de sentencias de control		22
45.	OBL – Todas las primitivas de control de flujo (if, else, while, for, do, foreach, switch) deben codificarse con bloque, aun cuando esté vacío.	22
46.	OBL – Todas las sentencias switch deben poseer una etiqueta default en el último case.	22
47.	REC – Evitar el uso de sentencias return condicionales o múltiples.	22
48.	REC – No realizar comparaciones explícitas true o false.	22
Guías para la Programación de Clases		23
Acerca del uso de propiedades y métodos		23
49.	REC - Utilizar una propiedad cuando el miembro es un miembro lógico de la clase.	23
50.	Utilizar un método cuando la obtención del valor de una propiedad puede tener un efecto colateral observable.	24
51.	Utilizar un método cuando las llamadas sucesivas al miembro de la clase producen resultados diferentes.	24
52.	Utilizar un método cuando el orden de ejecución es importante.	24
53.	Utilizar un método cuando el miembro es estático, pero retorna un valor que puede ser cambiado.	24
54.	Utilizar un método cuando el miembro retorna un array.	24
55.	OBL – Si un miembro retorna una copia de un tipo de referencia o un array, es obligatoria su documentación.	24

Guías para el uso de Propiedades.....	24
56. OBL - Nombrar las propiedades de acuerdo a las convenciones de nomenclatura indicadas en la sección [Convenciones de nomenclatura] de este documento. ....	24
57. REC - Preservar el valor de la propiedad antes de asignar un nuevo valor con Set. ....	24
58. OBL - La asignación de propiedades debe poder realizarse en cualquier orden. ....	25
59. REC - Utilizar métodos que comuniquen el cambio de una propiedad ....	25
60. OBL - Deben utilizarse propiedades Read-Only cuando el usuario no puede cambiar el valor del miembro lógico de la clase. 25	
61. OBL - No deben utilizarse propiedades Write-Only. ....	25
Guías para el uso de Eventos.....	25
62. OBL - Nombrar los Eventos de acuerdo a las convenciones de nomenclatura indicadas en la sección [Convenciones de nomenclatura] de este documento.....	25
63. OBL - Las clases de eventos (Event classes) deben extender System.EventArgs Class ....	25
64. REC - Utilizar un método <code>protected virtual</code> para levantar eventos.....	25
65. OBL - Debe asumirse que un event handler puede contener cualquier código.....	26
66. REC - Utilizar o extender la clase System.ComponentModel.CancelEventArgs Class para permitir al programador controlar los eventos de un objeto. ....	27
Guías para el uso de Métodos.....	28
67. OBL - Nombrar los Métodos de acuerdo a las convenciones de nomenclatura indicadas en la sección [Convenciones de nomenclatura] de este documento.....	28
68. OBL - No utilizar nomenclatura Húngara (hungarian notation). ....	28
69. REC - En forma predeterminada, los métodos no son <code>virtual</code> .....	28
70. REC - Uso de sobrecargas. ....	28
71. REC - Uso de métodos con cantidad variable de argumentos.....	30
Guías para el uso de Constructores .....	30
72. REC - Proveer un constructor privado predeterminado si solo existen métodos estáticos o propiedades en la clase. .30	
73. REC - Minimizar el trabajo que realice el constructor .....	31
74. REC - Proveer un <code>constructor</code> para todas las clases.....	31
75. REC - Proveer un constructor <code>protected</code> que pueda ser utilizado por tipos en una clase derivada. ....	31
76. REC - NO debe proveerse un constructor sin parámetros para un <code>struct</code> . ....	31
77. REC - Utilizar parámetros en constructores como forma rápida de asignar propiedades.....	31
Guías para el uso de Atributos .....	31
78. REC - No utilizar atributos <code>public</code> o <code>protected</code> . ....	31
79. REC - Utilizar una <code>protected property</code> que retorne el valor del atributo para exponerlo a una clase derivada. ....	33
80. REC - Utilizar atributos <code>public static read-only</code> para instancias predefinidas de los objetos de la clase.....	33
Guías para el uso de Parámetros .....	34
81. REC - Verificar la validez de los parámetros que se pasan como argumentos. ....	34
82. Diferencias entre argumentos pasados por valor o por referencia.....	35
Tipos de Datos.....	36
83. REC - Utilización de tipos <code>struct</code> .....	36
84. OBL - No debe proveerse un constructor para un tipo <code>struct</code> . ....	37
85. REC - Recomendaciones para el uso de <code>enums</code> .....	37
86. No debe utilizarse el sufijo "enum" para los tipos definidos como <code>enum</code> . ....	37

87.	REC – Utilizar tipos enum en lugar de constantes estáticas.....	37
88.	No utilizar tipos enum para conjuntos de valores abiertos.....	37
Estilo de codificación .....		38
89.	REC – Las secciones public, protected y private de una clase, deben ser declaradas en ese orden.....	38
90.	OBL – Uso de operadores .....	38
91.	REC – Utilizar Tabs y expandir a 4 espacios .....	38
92.	REC – No crear líneas de código de más de 100 caracteres .....	38
Apéndice 1 – Tabla de Palabras Reservadas.....		39

## INTRODUCCIÓN

Este documento describe reglas y recomendaciones para el desarrollo de aplicaciones utilizando el lenguaje C# en el Departamento de Desarrollo de GML Software.

La información aquí contenida proviene de varias fuentes, las cuales se listan en el Apéndice 1: Bibliografía consultada.

Específicamente, este documento contiene:

- Convenciones de nomenclatura
- Estilo de codificación
- Uso del lenguaje
- Diseño de modelos de objetos.

---

### 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 software desarrollados en el Departamento de Desarrollo.

La utilización de estándares debe mostrar efectos positivos sobre:

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

---

### Alcance

Este documento aplica únicamente al lenguaje C# y .Net Framework Common Type Systems (CTS). No se incluyen referencias o prácticas relacionadas al uso de las bibliotecas de clases de .Net Framework. Se incluyen algunos comentarios acerca de prácticas comunes y problemas conocidos de C#, las cuales se describen en forma breve.

Dado que las prácticas recomendadas en este manual se aplicarán en forma institucional, se han incluido normas que describen el uso de llaves ( { } ) y tabulaciones, para lograr una lectura uniforme del código generado en GML Software.

---

### Condiciones de uso de este documento

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.

---

### Convenciones utilizadas en este documento

Colores y énfasis	Descripción
Azul	Indica una palabra clave de C# o un tipo de .Net.

<b>Negrita</b>	Texto con énfasis adicional que debe ser considerado importante.
Términos claves	Descripción
<i>Siempre</i>	Indica que esta regla DEBE ser respetada, en los términos de este manual.
<i>Nunca</i>	Indica que esta acción NO DEBE ser realizada, en los términos de este manual.
<i>No hacer</i>	Indica que esta acción NO DEBE ser realizada, en los términos de este manual.
<i>Evitar</i>	Indica que esta práctica debe ser evitada siempre que sea posible, pero pueden existir excepciones AUTORIZADAS para su utilización.
<i>Intentar</i>	Indica que esta práctica debe aplicarse siempre que sea posible y apropiado.
<i>Razón</i>	Explica el propósito y las causas que motivan la regla o recomendación.

## Terminología y definiciones

Término	Descripción
Access Modifier	Las palabras claves de C# <code>public</code> , <code>protected</code> , <code>internal</code> y <code>private</code> declaran el alcance del acceso que el código permite a sus tipos y miembros. Aunque el acceso por defecto puede variar, las clases y la mayoría de los demás miembros del código utilizan <code>private</code> en forma predeterminada. Las excepciones de esta regla incluyen las interfaces y los tipos enumerados, que poseen acceso <code>public</code> en forma predeterminada.
Camel Case	Una palabra con la primera letra en minúsculas, y la primera letra de cada una de las palabras subsecuentes en mayúsculas. Ejemplo: <code>customerName</code>
Common type System	El Common Type System de .Net define cómo deben declararse, utilizarse y administrarse los tipos de datos. Todos los tipos nativos de C# están basados en el CTS para asegurar la compatibilidad con otros lenguajes del framework.
Identifier	Es un token definido por el programador, que nombra en forma única un objeto o una instancia de un objeto. Ejemplo: <code>public class MyClassNameIdentifier { ... }</code>
Magic Number	Cualquier literal numérico utilizado dentro de una expresión (o inicialización de variable) que no posea un significado claro. Usualmente este término no aplica a los valores 0 y 1 y cualquier otra expresión numérica equivalente que su evaluación resulte 0.
Pascal Case	Una palabra con la primera letra en mayúsculas, y la primera letra de cada palabra subsecuente también en mayúsculas. Ejemplo: <code>CustomerName</code>

---

## Fuentes de información consultadas

Basado en: Coding Standard C# - Phillips Medical Systems, C# Coding Standards - Lance Hunt, Microsoft Naming Guidelines - Microsoft Corp.



## GUÍA RÁPIDA

En esta sección se incluye un breve resumen de los principales estándares descritos a lo largo de este documento. Estas tablas no son detalladas en sus descripciones, pero brindan una rápida referencia a los elementos.

### Convenciones de nomenclatura

- c Camel case
- P Pascal case
- \_ Prefijo con guion bajo (underscore)
- X No aplica

Identificador	Public	Protected	Internal	Private	Notas
Archivo de proyecto	P	X	X	X	Debe corresponder al Assembly y al Namespace.
Archivo del código fuente	P	X	X	X	Debe corresponder con la clase que contiene.
Namespace	P	X	X	X	Correspondencia parcial Project / Assembly.
Class o Struct	P	P	P	P	Agregar sufijo de la subclase. Ejemplo: AppDomain
Interface	P	P	P	P	Debe tener el prefijo "I". Ejemplo: IDerivable
Exception class	P	P	P	P	Siempre debe tener el sufijo "Exception" luego del nombre. Ejemplo: AppDomainException
Method	P	P	P	P	Utilizar verbo o verbo-objeto. Ejemplo: ToString
Property	P	P	P	P	No utilizar prefijos. No utilizar <code>Get</code> / <code>Set</code> como prefijos. Ejemplo: BackColor
Atributos	P	P	P	_c	Utilizar solo Private fields. No utilizar notación Húngara (Hungarian Notation).

Identificador	Public	Protected	Internal	Private	Notas
					Ejemplo: <code>_listItem</code>
Constant	P	P	P	_c	Ejemplo: <code>_maximunItems</code>
Static field Read-only	P	P	P	_c	Utilizar sólo atributos Private. Ejemplo: <code>_redValue</code>
Enum	P	P	P	P	Utilizar PascalCase también para las opciones. Ejemplo: <code>ErrorLevel</code>
Delegate	P	P	P	P	
Event	P	P	P	P	Ejemplo: <code>ValueChanged</code>
Inline variable	X	X	X	c	Evitar caracteres únicos y tipos enumerados.
Parameter	X	X	X	c	Ejemplo: <code>typeName</code>

## Estilos de codificación

Código	Estilo
Archivo del código fuente (Source code)	Un archivo por Namespace y un archivo por clase.
Llaves {}	Siempre en una nueva línea. Utilizar paréntesis cuando sea opcional.
Tabulación (Indent)	Utilizar siempre tabs de 4 caracteres, expandir a espacios.
Comentarios (Comments)	Utilizar <code>/*</code> . No utilizar <code>/*...*/</code> .
Variables	Una variable por cada declaración.
Tipos de datos nativos (Native Data Types)	Priorizar el uso de tipos nativos de C# ante tipos del CTS. Ejemplo: <code>int</code> en lugar de <code>Int32</code> .
Enumerados (Enums)	Evitar cambiar el tipo de dato predeterminado.
Tipos genéricos (Generic types)	Utilizar preferiblemente tipos genéricos ante tipos estándar o strong-typed classes.
Propiedades (Properties)	No utilizar prefijos <code>Get</code> / <code>Set</code> .

Código	Estilo
Métodos (Methods)	Utilizar un máximo de 7 parámetros.
Base / This	Utilizar solo en <b>constructors</b> o dentro de <b>override</b> .
Condiciones ternarias (Ternary conditions)	
foreach	No modificar tipos enumerados dentro de una sentencia foreach.
Condicionales (Conditionals)	Evitar la evaluación de condiciones booleanas contra valores <code>true</code> o <code>false</code> . No realizar asignaciones embebidas. No utilizar invocación de métodos embebidos.
Manejos de excepciones (Exception Handling)	No utilizar excepciones para el control de flujo. Utilizar <code>throw</code> ; NO <code>throw e</code> ; cuando se relanza la excepción. Sólo utilizar <b>catch</b> cuando es posible manejar la excepción. Utilizar validaciones para evitar la ocurrencia de excepciones.
Eventos (Events)	Siempre debe validarse el valor <b>null</b> antes de la invocación.
Bloqueo (Locking)	Utilizar <code>lock()</code> en lugar de <code>Monitor.Enter()</code> No utilizar lock en un tipo, por ejemplo: <code>lock(typeof(MyType))</code> ; Evitar el uso de lock ante un this, por ejemplo: <code>lock(this)</code> ; Preferiblemente, utilizar lock en objetos declarados como private, por ejemplo: <code>lock(myVariable)</code> ;
Dispose() / Close()	Utilizarlo siempre que sea posible.
Finalizers	Evitar la implementación de estos métodos. Utilizar la sintaxis del destructor de C#. Ejemplo: <code>~MyClass() { ... }</code> Nunca debe definirse un método llamado <code>Finalize()</code> ;
AssemblyVersion	Incrementar en forma manual.
ComVisibleAttribute	Debe estar con el valor <code>false</code> para todos los assemblies, a menos que se requiera que sea consumido por COM.

## CONVENCIONES DE NOMENCLATURA

La consistencia del código es la clave para su fácil mantenimiento. Esta aseveración aplica a la nomenclatura utilizada para los nombres de carpetas, archivos y todos los identificadores contenidos en una aplicación.

---

### Guías genéricas de nomenclatura

1. **REC – Utilizar nombres en inglés para todos los elementos que componen una solución: carpetas, archivos e identificadores.**

2. **OBL – Utilizar siempre la convención Pascal o Camel**

- 2.1. OBL - Utilizar nomenclatura Pascal y Camel para la nomenclatura de todos los elementos que componen una solución: carpetas, archivos e identificadores.

- 2.2. OBL - No utilizar nombres que comiencen con un carácter numérico.

3. **OBL – No utilizar mayúsculas y minúsculas para diferenciar identificadores.**

No definir namespaces, clases, métodos, propiedades, campos o parámetros con igual nombre, pero que difieren en la utilización de las mayúsculas y minúsculas.

4. **OBL - No utilizar nomenclatura Húngara (iCount, strName).**

5. **REC – Utilizar abreviaturas sólo cuando el nombre completo sea demasiado extenso.**

- 5.1. REC - Evitar el uso de contracciones de palabras en los identificadores.

- 5.2. REC - Evitar el uso de abreviaturas no estándar, por ejemplo: no utilizar **GetWin** si es posible utilizar **GetWindow**.

- 5.3. REC - Cuando sea pertinente, utilizar abreviaturas estándar, por ejemplo: puede utilizarse **UI** en lugar de **UserInterface**.

- 5.4. REC - Utilizar mayúsculas para las abreviaturas de dos caracteres, y notación Pascal para abreviaturas más extensas.

6. **REC – No utilizar guion bajo (underscore) en los identificadores.**

7. **REC – Nombrar los identificadores de acuerdo a su significado y no a su tipo.**

Evitar el uso de terminología dependiente del lenguaje en los identificadores.

Como un ejemplo, supongamos que tenemos varios métodos que sobrecargados que escriben datos en un stream.

No debemos utilizar definiciones como:

```
void Write (double doubleValue);
```

```
void Write (long longValue);
```


La forma correcta, sería:

```
void Write (double value);
```

```
void Write (long value);
```

#### 8. REC – Utilizar “Can”, “Has” o “Is” para nombrar variables booleanas.

Siempre que sea posible es recomendable utilizar los prefijos “Can”, “Has” o “Is” delante del nombre de las variables de tipo Boolean.

 Ejemplo:

```
IsIncludedInList();  
CanInsert();
```

#### 9. REC – Utilizar nombres de operaciones aritméticas para nombrar variables que las implementan.

Siempre que sea posible es recomendable utilizar los nombres de operaciones aritméticas estándar cuando su resultado se almacene en variables definidas para esos efectos.

Ejemplos de estas operaciones son: Sum, Average, Count, Min, Max.

En la tabla siguiente se muestran los nombres y abreviaturas sugeridos para identificar estas operaciones. Las abreviaturas son iguales a las utilizadas por MS-Excel.

 Ejemplo:

```
AverageSalary(long year);  
CountRows();
```


#### 10. REC - Nombrar los namespaces de acuerdo a un patrón bien definido.

Los namespaces deben ser nombrados en Pascal, de acuerdo al siguiente patrón:

```
<company>.<project>.<layer>.<component>
```

#### 11. REC – Utilizar sustantivos o frases sustantivos para el nombre de clases o struct.


Si la clase desarrollada, es una clase derivada, es una buena práctica utilizar un nombre compuesto.

 Ejemplo:

Si tenemos una clase llamada `Button`, una clase derivada de este podría nombrarse como `ShadowButton`.

#### 12. REC – El nombre de las interfaces debe tener el prefijo “I”.

Todas las interfaces deben comenzar con el prefijo “I”, y debe utilizarse un sustantivo, una frase con sustantivo o un adjetivo para su nombre.

 Ejemplo:

```
IComponent – sustantivo  
IComponentAttributeProvider – frase con sustantivo  
IPersistable – adjetivo
```

#### 13. REC – Utilizar nombres similares para la implementación predeterminada de una interface.

Si estamos proveyendo una implementación predeterminada (default) para una interface particular, se debe utilizar un nombre similar para la clase que la implementa.


Nota: Observar que esto aplica sólo a las clases que implementan **únicamente** la interfaz referida.

 Ejemplo:

Si tenemos una clase que implementa la interface `IComponent`, puede llamarse `Component` o `ComponentDefault`.

#### 14. REG – Utilizar nombres en singular para tipos enumerados.

Para el nombre de los tipos enumerados, debe utilizarse la palabra en singular.

 Ejemplo:

Si tenemos un tipo enum en el cual almacenaremos un valor de una colección restringida de valores, debe llamarse `Protocol` y no `Protocols`.

```
public enum Protocol
{
    Tcp;
    Udp;
    http;
    Ftp;
}
```

#### 15. REG – Utilizar nombres en plural para enumeraciones que representan campos de datos.

Ejemplo:

```
[flags]
public Enum SearchOptions
{
    CaseInsensitive = 0x01,
    WholeWordOnly = 0x02,
    AllDocuments = 0x04,
}
```

#### 16. REC – Evitar el uso de las letras I O y los números 1 y 0 juntos.


Para generar código más limpio, es conveniente evitar el uso de las letras I O y los números 1 y 0 juntos, ya que puedan generar una confusión entre números y letras.

 Ejemplos:

```
Bool b001 = (10 == 10) ? (I1 == 11) : (101 != 101);
```

#### 17. OBL – Agregar “EventHandler” para delegados relacionadas a eventos.

Los delegados que sean utilizados para definir un event handler para un evento, deben nombrarse agregando el texto “EventHandler” a su nombre (**EventNameEventHandler**).

 Ejemplo:

```
public delegate CloseEventHandler(object sender, EventArgs arguments)
```

#### 18. OBL – Agregar “CallBack” para delegados relacionadas a métodos callback.

Los delegados que sean utilizados para pasar una referencia a un método callback (no a un evento), deben nombrarse agregando el texto “CallBack” a su nombre (**MethodNameCallBack**).



Ejemplo:

```
public delegate AsyncIOFinishedCallBack(IpcClient client, string message)
```

#### 19. REC – No agregar “CallBack” a los métodos callback.

No deben utilizarse sufijos o prefijos “CallBack” o “CB” para indicar los métodos callback

#### 20. REC – Utilizar verbos para nombrar eventos.

Utilizar verbos para nombrar los eventos de una clase.



Ejemplo:

```
Minimize()
```

```
Maximize()
```

#### 21. OBL – No agregar el texto “Event” a los nombres de los eventos.

#### 22. OBL – Utilizar sufijos “ing”y “ed” para indicar pre-eventos y post-eventos.

No utilizar patrones del estilo “BeginXXX” o “EndXXX”. Si es necesario proveer diferentes eventos que expresen instancias previas y posteriores en el tiempo, utilizar para su nombre el sufijo “ing” si indica anterioridad, y el sufijo “ed” si indica posterioridad.



Ejemplo:

```
EntityValidating() – Evento pre-validación
```

```
EntityValidate() – Evento de validación
```

```
EntityValidated() – Evento post-validación
```

#### 23. OBL – Agregar el prefijo “On” para los Event Handlers.

Para nombrar los event handlers, debe utilizarse el texto “On” antepuesto a su nombre.



Ejemplo:

Si tenemos un método que maneja el evento `Close` de una clase, debe nombrarse como `OnClose()`.

#### 24. OBL – Nombrar las exception classes utilizando el sufijo “Exception”.

Las exception classes deben nombrarse agregando el texto “Exception” a su nombre (**ClassNameException**).



Ejemplo:

```
IpcException()
```

#### 25. OBL – No incluir el nombre de la clase dentro del nombre de las propiedades.

El nombre de las propiedades no debe incluir el nombre de la clase.



Ejemplo:

Utilizar `Customer.Name`

No utilizar `Customer.CustomerName`

## **26. OBL – Nombrar las assemblies DLL luego del namespace que las contiene.**

Para permitir el almacenamiento de los assemblies en el GAC, sus nombres deben ser únicos. Además debe utilizarse el nombre del namespace como prefijo del nombre del assembly.

 Ejemplo:

Consideremos un grupo de clases organizadas bajo el namespace `GML.Software.ICWorkFlow.Platform.OSInterface`. En este caso, el assembly generado por estas clases será llamado `GML.Software.ICWorkFlow.Platform.OSInterface.dll`.

Si se generan múltiples assemblies desde el mismo namespace, es posible agregar un prefijo único al nombre del namespace.

## **27. OBL – Utilizar notación Pascal para nombrar los archivos de código fuente.**

No utilizar guion bajo para nombrar los archivos de código fuente.

## **28. OBL – Nombrar el archivo de código fuente igual a la clase principal que implementa.**

El archivo de código fuente de una clase, debe tener el mismo nombre que la clase principal. Adicionalmente no debe incluirse más de una clase principal por archivo.



## COMENTARIOS Y DOCUMENTACIÓN EMBEBIDA

En esta sección se describen las guías para la escritura de comentarios y documentación embebida en el código fuente de las aplicaciones.

---

### Guías para la escritura de comentarios

#### 29. OBL – Cada archivo debe contener un cabezal identificadorio.

El cabezal de cada archivo es un bloque `#region` que contiene el texto de copyright y el nombre del archivo.

Ejemplo:

```
# region Copyright GML Software - 2003
//
// Todos los derechos reservados. La reproducción o transmisión en su
// totalidad o en parte, en cualquier forma o medio electrónico, mecánico
// o similar es prohibida sin autorización expresa y por escrito del
// propietario de este código.
//
// Archivo: PatientAdministration.cs
//
#endregion
```

#### 30. OBL – Utilizar “//” para los bloques de comentarios.

#### 31. REC – Los comentarios serán escritos en español si no hay una petición expresa del cliente para hacerlo en inglés.

---

### Guías para la escritura de documentación embebida

#### 32. OBL – Utilizar tags XML para documentar tipos y miembros de clase.

Todos los tipos privados y públicos, métodos, campos, eventos, etc. deben ser documentados utilizando tags XML. Utilizando estos tags, IntelliSense provee información útil acerca de los tipos de datos, y además es posible utilizar herramientas de documentación automática basada en estos tags.

#### 33. REC – Utilizar `#region` para agrupar miembros no públicos.

Si una clase contiene un largo número de miembros, atributos y/o propiedades, se ubicarán todos los miembros no públicos en un bloque `#region`. Se utilizarán regiones separadas para dividir los miembros `public`, `private`, `protected` e internos de una clase.

## CICLO DE VIDA DE LOS OBJETOS

Esta sección describe las guías de programación y recomendaciones que aplican a la codificación del ciclo de vida de un objeto: declaración e inicialización de variables, e implementación de constructores y destructores de clases.

---

### Guías para la declaración e inicialización de variables

#### 34. REC – Declarar e inicializar las variables próximo al lugar donde son utilizadas.

#### 35. REC – Inicializar las variables en el punto de su declaración.

Siempre que sea posible, es conveniente inicializar las variables en el mismo punto de su declaración.

Si se está utilizando inicialización de campos, éstos serán inicializados antes de llamar el constructor de la instancia. De la misma forma, los campos estáticos se inicializan cuando el constructor estático es llamado.

El compilador siempre inicializa al valor por defecto cualquier variable que no haya sido inicializada.



Ejemplo:

```
enum Color
{
    Red,
    Green
}

bool fatalSituation = IsFatalSituation();
Color backgroundColor = fatalSituation ? Color.Red : Color.Green;
```

#### 36. OBL – Declarar cada variable en una sentencia diferente.

#### 37. OBL – Utilizar `const` para la declaración de constantes.

La declaración de constantes utilizando `const`, asegura la asignación de memoria una única vez para el ítem que está siendo declarado.



Ejemplo:

```
private const int MaxUsers = 100;
```

#### 38. REC – Utilizar `public static readonly` para definir instancias predeterminadas de objetos.

Para definir instancias predeterminadas de objetos, es conveniente utilizar la declaración `public static readonly`. Por ejemplo, consideremos una clase `Color` que internamente almacena los valores numéricos del rojo, negro y blanco. Esta clase tiene un constructor que toma un valor numérico, de forma que puede exponer varios colores predeterminados.

```
public struct Color
{
    public static readonly Color Red = new Color(0xFF0000);
    public static readonly Color Black = new Color(0x000000);
}
```

```
public static readonly Color White = new Color(0xFFFFFFFF);  
}
```

---

## Guías para la utilización de destructores de clases

### 39. REC – Asignar el valor null a los campos de referencia cuando no se necesiten.


La asignación del valor null a los objetos mejora la utilización de la memoria porque el objeto pierde la referencia, permitiendo a la GC “limpiar” la asignación de memoria en forma temprana.

### 40. REC – Evitar la implementación de métodos destructores.

El uso de destructores en C# no es promovido debido a que introduce un fuerte impacto en el desempeño, por el trabajo que debe realizar el GC.

### 41. OBL – Si es necesaria la implementación de un destructor, debe utilizarse `GC.SuppressFinalize`.

Si es necesaria la implementación de un destructor, es necesario verificar que el programador haya utilizado `GC.SuppressFinalize()` dentro del método `Close()`.

 Ejemplo:

```
public class IpcPeer  
{  
    bool connected = false;  
    public void Connect()  
    {  
        // Do some work and then change the state of this object.  
        connected = true;  
    }  
    public void Close()  
    {  
        // Close the connection, change the state, and instruct  
        // the GC, NOT to call the destructor  
        connected = false;  
        GC.SuppressFinalize(this);  
    }  
    ~IpcPeer()  
    {  
        // If the detructor is called, then Close() was not called.  
        If (connected)  
        {  
            // Warning! User has not called Close().  
            // Notice that you can't call Close() from here because
```

```
// the objects involved may have already been garbage
// collected (ver regla XX)

    }

}

}
```

#### 42. OBL – Implementar `IDisposable` si una clase utiliza recursos no administrados.

Si una clase utiliza recursos no administrados, tales como objetos COM, conexiones, sockets, colas, u objetos que consumen recursos extensamente, éstos deben ser liberados tan pronto como sea posible.

La implementación de la interface `IDisposable` permite a las clases de la aplicación la liberación explícita de los recursos no administrados.



Ejemplo:

```
public class ResourceHolder : IDisposable
{
    ///<summary>
    ///Implementation of the IDisposable interface
    ///</summary>
    public void Dispose()
    {
        // Call internal Dispose(bool)
        Dispose(true);
        // Prevent the destructor from being called
        GC.SuppressFinalize(this);
    }
    ///<summary>
    /// Central method for cleaning up resources
    ///</summary>
    protected virtual void Dispose4(bool explicit)
    {
        // If explicit is true, then this method was called through the
        // public Dispose()
        if (explicit)
        {
            // Release or cleanup managed resources
        }

        // Always release or cleanup (any) unmanaged resources
    }
}
```

```
~ResourceHolder()
{
    // Since other managed objects are disposed automatically, we
    // should not try to dispose any managed resources (see Rule 5@114).
    // We therefore pass false to Dispose()
    Dispose(false);
}
}
```

Si otra clase deriva de esta clase, la primera sólo debe sobre implementar el `override` del método `Dispose(bool)` de la clase base. No debe implementar ella misma la interface `IDisposable`, ni un destructor, pues se dispara automáticamente la llamada al destructor de la clase base.

 Ejemplo:

```
public class DerivedResourceHolder : ResourceHolder
{
    protected override void Dispose(bool explicit)
    {
        if (explicit)
        {
            // Release or cleanup managed resources of this derived
            // class only.
        }
        // Always release or cleanup (any) unmanaged resources.
        // Call Dispose on our base class.
        base.Dispose(explicit);
    }
}
```

#### **43. OBL – No accede a ningún miembro tipo referencia en el destructor.**

## CONTROL DE FLUJO

En esta sección se describen las guías y recomendaciones para la programación de los procesos de control de flujo y el uso del lenguaje en cada caso.

---

### Guías para el uso de variables dentro de sentencias loop

#### 44. OBL – No cambiar una variable del loop dentro de un bloque `for`.

La actualización de una variable loop dentro de un bloque `for` es generalmente considerada confuso, y aún más cuando el valor de la variable se modifica en más de un lugar.

---

### Guías para el uso de sentencias de control

#### 45. OBL – Todas las primitivas de control de flujo (`if`, `else`, `while`, `for`, `do`, `foreach`, `switch`) deben codificarse con bloque, aun cuando esté vacío.

#### 46. OBL – Todas las sentencias `switch` deben poseer una etiqueta `default` en el último `case`.


Es recomendable un comentario como “no action” donde sea explícita esta intención. Si la etiqueta `default` es la última, es más sencillo de ubicar.

#### 47. REC – Evitar el uso de sentencias `return` condicionales o múltiples.

La norma para un flujo simple es: “una entrada, una salida”. En algunos casos tales como validación de precondiciones, puede ser una buena práctica forzar la salida si una de las precondiciones no se cumple.

#### 48. REC – No realizar comparaciones explícitas `true` o `false`.

Usualmente es considerada una mala práctica comparar una expresión booleana contra expresiones `true` o `false`.

 Ejemplo:

```
while [condition == false] // Mala práctica
while [condition != true] // Mala práctica
while (((condition == true) == true) == true) // Mala práctica
while (booleanCondition) // Práctica recomendada
```

## GUÍAS PARA LA PROGRAMACIÓN DE CLASES

Esta sección describe las guías y recomendaciones a seguir en la programación orientada a objetos.

El contenido de esta sección está dividido en:

- Guía para el uso de Propiedades  
Describe las recomendaciones y guías para el uso de propiedades dentro de bibliotecas de clases (class libraries).
- Guía para el uso de Eventos  
Describe las recomendaciones y guías para el uso de eventos dentro de bibliotecas de clases (class libraries).
- Guía para el uso de Métodos  
Describe las recomendaciones y guías para el uso de métodos dentro de bibliotecas de clases (class libraries).
- Guías para el uso de Constructores  
Describe las recomendaciones y guías para el uso de constructores dentro de bibliotecas de clases (class libraries).
- Guía para el uso de variables y campos  
Describe las recomendaciones y guías para el uso de variables y campos dentro de bibliotecas de clases (class libraries).
- Guía para el uso de Parámetros  
Describe las recomendaciones y guías para el uso de parámetros dentro de bibliotecas de clases (class libraries).


---

### Acerca del uso de propiedades y métodos

Generalmente los métodos representan acciones, y las propiedades representan datos de una clase.

Podemos utilizar las siguientes recomendaciones como guía:

#### 49. REC - Utilizar una propiedad cuando el miembro es un miembro lógico de la clase.

 Ejemplo:

En la siguiente declaración, Name es una propiedad porque es un miembro lógico de la clase.

```
public string Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
}
```

**50. Utilizar un método cuando la obtención del valor de una propiedad puede tener un efecto colateral observable.**

**51. Utilizar un método cuando las llamadas sucesivas al miembro de la clase producen resultados diferentes.**


**52. Utilizar un método cuando el orden de ejecución es importante**

Debe considerarse que las propiedades de tipo deben poder asignarse (set) y obtenerse (get) en cualquier orden.

**53. Utilizar un método cuando el miembro es estático, pero retorna un valor que puede ser cambiado.**

**54. Utilizar un método cuando el miembro retorna un array.**

Las propiedades que retornan arrays pueden ser difíciles de entender, y resultar engañosas. Usualmente es necesario retornar una copia del array interno, pues el usuario no puede cambiar el estado interno. Esta consideración, asociada al hecho que el usuario puede fácilmente asumir que es una propiedad indizada, conduce a que el código sea ineficiente.

 Ejemplo:

En este ejemplo cada llamada a la propiedad `Methods` crea una copia del array. Como resultado, tendremos  $2^{n+1}$  copias del array que será creado en el loop.

```
Type type = // Get a type.
for (int i = 0; i < type.Methods.Length; i++)
{
    if (type.Methods[i].Name.Equals ("text"))
    {
        // Perform some operation.
    }
}
```

**55. OBL – Si un miembro retorna una copia de un tipo de referencia o un array, es obligatoria su documentación.**

En forma predeterminada, todos los miembros que necesitan retornar un objeto interno o un array de objetos, retornarán una referencia a este objeto o al array. En estos casos, esta necesidad **siempre** debe documentarse claramente en la especificación.

---

## Guías para el uso de Propiedades

**56. OBL - Nombrar las propiedades de acuerdo a las convenciones de nomenclatura indicadas en la sección [Convenciones de nomenclatura] de este documento.**

**57. REC - Preservar el valor de la propiedad antes de asignar un nuevo valor con Set.**

Cuando se cambia el valor de una propiedad utilizando el método `Set` de la propiedad, debe preservarse el valor anterior antes del cambio. Esto asegura que el dato no se pierde si el método `Set` produce una excepción.




**58. OBL - La asignación de propiedades debe poder realizarse en cualquier orden.**

Las propiedades no deben tener estado respecto a otras propiedades de la clase, es decir no debería haber diferencia en el estado del objeto si se asigna primero la propiedad A o la propiedad B.

**59. REC - Utilizar métodos que comuniquen el cambio de una propiedad**

Los componentes deben disparar eventos que informen el cambio de una propiedad si es necesario notificar a sus clientes cuando se produce un cambio de valor dentro del código.

La convención de nomenclatura para estos eventos, consiste en agregar el sufijo “Changed” al nombre del evento que implementa la notificación.

 Ejemplo:

Un campo de texto puede disparar el evento `TextChanged` cuando su propiedad `text` cambie de valor.

**60. OBL - Deben utilizarse propiedades Read-Only cuando el usuario no puede cambiar el valor del miembro lógico de la clase.**

La implementación de una propiedad Read-Only se realiza programando sólo el método `Get` de la propiedad.

**61. OBL - No deben utilizarse propiedades Write-Only.**


No deben implementarse propiedades que sólo contengan el método `Set`.

---

## Guías para el uso de Eventos

**62. OBL - Nombrar los Eventos de acuerdo a las convenciones de nomenclatura indicadas en la sección [Convenciones de nomenclatura] de este documento.**

**63. OBL - Las clases de eventos (Event classes) deben extender System.EventArgs Class**


 Ejemplo:

```
public class MouseEvent: EventArgs {}
```

**64. REC - Utilizar un método `protected virtual` para levantar eventos.**

Esta técnica no es apropiada para clases “selladas”, porque impide que otras clases deriven de ella. El propósito de este método es proveer una forma para que una clase derivada pueda manejar el evento utilizando `override`. Esta técnica es más natural que el uso de `delegates` en situaciones donde el programador está creando una clase derivada.

El nombre del método agrega la palabra “On” al nombre del evento que será levantado.

 Ejemplo:

```
public class Button
{
    ButtonClickHandler onClickHandler;
    protected virtual void OnClick(EventArgs e)
    {
```

```
// Call the delegate if non-null.  
if (onClickHandler != null)  
    onClickHandler(this, e);  
}  
}
```

#### 65. OBL – Debe asumirse que un event handler puede contener cualquier código.

Las clases deben estar preparadas para que los event handler realicen cualquier operación, y en todos los casos el objeto quede en un estado apropiado luego que el evento sea levantado. Debe considerarse también el uso de un bloque try/finally en el punto del código donde se levanta el evento

Desde que el programador realiza una function callback sobre el objeto que realice otras acciones, no debe asumirse nada acerca del estado del objeto cuando el control retorna al punto desde donde se levantó el evento.

 Ejemplo:

```
public class Button  
{  
    ButtonClickHandler onClickHandler;  
    protected void DoClick()  
    {  
        // Paint button in indented state.  
        PaintDown();  
        try  
        {  
            // Call event handler.  
            OnClick();  
        }  
        finally  
        {  
            // Window might be deleted in event handler.  
            if (windowHandle != null)  
                // Paint button in normal state.  
                PaintUp();  
        }  
    }  
    protected virtual void OnClick(ClickEvent e)  
    {  
        if (onClickHandler != null)  
            onClickHandler(this, e);  
    }  
}
```

```
}  
  
}
```

## 66. REC – Utilizar o extender la clase `System.ComponentModel.CancelEventArgs` Class para permitir al programador controlar los eventos de un objeto.

Por ejemplo, el control `TreeView` levanta el evento `BeforeLabelEdit` cuando el usuario edita un nodo del árbol.

 Ejemplo:

Este ejemplo muestra cómo puede utilizarse este método para prevenir la edición de un nodo.

```
public class Form1: Form  
{  
    TreeView treeView1 = new TreeView();  
  
    void treeView1_BeforeLabelEdit(object source,  
        NodeLabelEditEventArgs e)  
    {  
        e.CancelEdit = true;  
    }  
}
```

En este caso, el usuario no recibe un mensaje de error, sino que la etiqueta no queda habilitada para su edición

Los eventos `Cancel` no son apropiados en los casos donde el programador debe cancelar la operación y devolver una excepción. En estos casos, debe levantarse la excepción dentro del event handler. Por ejemplo, el usuario quizá desee incluir cierta lógica de validación en el control cuando se edita, por ejemplo:

```
public class Form1: Form  
{  
    EditBox edit1 = new EditBox();  
  
    void TextChanging(object source, EventArgs e)  
    {  
        throw new RuntimeException("Invalid edit");  
    }  
}
```

## Guías para el uso de Métodos

**67. OBL - Nombrar los Métodos de acuerdo a las convenciones de nomenclatura indicadas en la sección [Convenciones de nomenclatura] de este documento.**

**68. OBL - No utilizar nomenclatura Húngara (hungarian notation).**

**69. REC - En forma predeterminada, los métodos no son *virtual*.**

Es recomendable mantener esta premisa en todas las situaciones en las cuales no sea expresamente necesario proveer métodos virtuales.


**70. REC – Uso de sobrecargas.**

La sobrecarga ocurre cuando una clase contiene dos métodos con el mismo nombre, pero diferentes firmas. En esta guía incluimos las recomendaciones más importantes para su uso:

Utilizar sobrecargas para proveer diferentes métodos que hacen semánticamente lo mismo.

*70.1. REC - Utilizar sobrecarga en lugar de default arguments.*

Los argumentos predeterminados (default arguments) no son totalmente compatibles en esta versión del lenguaje y no están permitidos en el Common Language Specification (CLS).

 Ejemplo:


El siguiente ejemplo muestra el uso del método `String.IndexOf` con overloading.

```
int String.IndexOf (String name);  
int String.IndexOf (String name, int startIndex);
```

Los argumentos predeterminados (default arguments) no son totalmente compatibles en esta versión del lenguaje y no están permitidos en el Common Language Specification (CLS).

*70.2. REC - Los valores predeterminados deben utilizarse correctamente.*

En una familia de métodos sobrecargados, el método complejo debe utilizar nombres de parámetros que indiquen el cambio del estado asumido por defecto en el método simple. Por ejemplo, en el siguiente código, el primer método asume que la búsqueda no es case-sensitive. El segundo método utiliza el nombre `ignoreCase` en lugar de `caseSensitive` para indicar como cambia el comportamiento predeterminado.

 Ejemplo

```
// Method #1: ignoreCase = false.  
MethodInfo Type.GetMethod(String name);  
  
// Method #2: Indicates how the default behavior of method #1 is being  
// changed.  
MethodInfo Type.GetMethod (String name, Boolean ignoreCase);
```

*70.3. Utilizar un orden y una nomenclatura consistente para los parámetros.*

Es común proveer un conjunto de métodos sobrecargados con una cantidad incremental de parámetros para permitir al programador especificar el nivel de información que necesite según las circunstancias. Cuantos más parámetros se especifiquen, mayor cantidad de opciones poseerá el programador.

En el siguiente ejemplo, el método Execute está sobrecargado y posee un orden y una nomenclatura consistente en sus diferentes opciones. Cada variante del método Execute utiliza la misma semántica para el conjunto compartido de parámetros.

```
public class SampleClass
{
    readonly string defaultForA = "default value for a";
    readonly int defaultForB = "42";
    readonly double defaultForC = "68.90";
    public void Execute()
    {
        Execute(defaultForA, defaultForB, defaultForC);
    }
    public void Execute (string a)
    {
        Execute(a, defaultForB, defaultForC);
    }
    public void Execute (string a, int b)
    {
        Execute (a, b, defaultForC);
    }
    public void Execute (string a, int b, double c)
    {
        Console.WriteLine(a);
        Console.WriteLine(b);
        Console.WriteLine(c);
        Console.WriteLine();
    }
}
```

Nota: Observar que el único método en el grupo que debe ser virtual es el que posee la mayor cantidad de parámetros y solo cuando lo que se necesita es extensibilidad.

**70.4. REC** – Si un método debe poseer la posibilidad de ser sobrescrito (override), sólo debe implementarse la versión más completa como virtual, y definir las demás operaciones en función de esa implementación.




Ejemplo:

```
public class SampleClass
{
    private string myString;
    public MyClass(string str)
```

```
{
    this.myString = str;
}
public int IndexOf(string s)
{
    return IndexOf (s, 0);
}
public int IndexOf(string s, int startIndex)
{
    return IndexOf(s, startIndex, myString.Length - startIndex );
}
public virtual int IndexOf(string s, int startIndex, int count)
{
    return myString.IndexOf(s, startIndex, count);
}
}
```

#### 71. REC – Uso de métodos con cantidad variable de argumentos

Algunas veces es deseable exponer un método que tome una cantidad variable de argumentos. Un ejemplo clásico puede ser el método print de C.

 Ejemplo:

```
void Format(string formatString, params object [] args)
```

Nota: **No** debe utilizarse `VarArgs` o paréntesis (...), pues el Common Language Specification no lo soporta.

---

### Guías para el uso de Constructores

#### 72. REC – Proveer un constructor privado predeterminado si solo existen métodos estáticos o propiedades en la clase.

En el siguiente ejemplo, el constructor privado impide que la clase sea creada desde el exterior.

```
public sealed class Environment
{
    // Private constructor prevents the class from being created.
    private Environment()
    {
        // Code for the constructor goes here.
    }
}
```

### 73. REC – Minimizar el trabajo que realice el constructor

Los constructores no deben hacer más que capturar los parámetros necesarios. Esto disminuye el costo de performance en las siguientes operaciones hasta que el usuario utilice una instancia específica de la clase.

### 74. REC – Proveer un constructor `private` para todas las clases.

En el caso de un tipo, utilizar un constructor `private`. Si no se especifica un constructor, muchos lenguajes de programación (tales como C#) implícitamente agregan un constructor público, y si es una clase abstracta agregan en constructor `protected`.

Debe tomarse cuidado si se agrega un constructor no predefinido a una clase en una versión posterior, pues será eliminado el constructor implícito y esto causará un error en los clientes de esa clase. Por esto mismo, la mejor práctica es proveer siempre explícitamente el constructor de la clase, aun cuando sea el predefinido y sea público.


### 75. REC – Proveer un constructor `protected` que pueda ser utilizado por tipos en una clase derivada.

### 76. REC – NO debe proveerse un constructor sin parámetros para un `struct`.

Muchos compiladores no permiten que un tipo `struct` posea constructor sin parámetros. En este caso, si el programador lo implementa, el runtime inicializa todos los campos del `struct` en cero.

### 77. REC – Utilizar parámetros en constructores como forma rápida de asignar propiedades.

No hay diferencia alguna en invocar primero el constructor y luego asignar las propiedades, o invocar al constructor pasando los valores directamente.

 Ejemplo:

Los tres ejemplos a continuación son equivalentes:

```
// Example #1.  
Class SampleClass = new Class();  
SampleClass.A = "a";  
SampleClass.B = "b";  
  
// Example #2.  
Class SampleClass = new Class("a");  
SampleClass.B = "b";  
  
// Example #3.  
Class SampleClass = new Class ("a", "b");
```

---

## Guías para el uso de Atributos

### 78. REC – No utilizar atributos `public` o `protected`.

Evitando el uso de atributos expuestos directamente al programador, las clases pueden ser actualizadas con más facilidad pues el atributo no debe ser transformado en propiedad, manteniendo así la compatibilidad binaria.

La recomendación es que siempre se implementen los métodos Get y Set en lugar de declarar los atributos como públicos. El siguiente ejemplo muestra el uso correcto de atributos privados y propiedades con métodos Get y Set.



Ejemplo:

```
public struct Point
{
    private int xValue;
    private int yValue;

    public Point(int x, int y)
    {
        this.xValue = x;
        this.yValue = y;
    }

    public int X
    {
        get
        {
            return xValue;
        }
        set
        {
            xValue = value;
        }
    }

    public int Y
    {
        get
        {
            return yValue;
        }
        set
        {
            yValue = value;
        }
    }
}
```



**79. REC - Utilizar una `protected` property que retorne el valor del atributo para exponerlo a una clase derivada.**

 Ejemplo:

```
public class Control: Component
{
    private int handle;
    protected int Handle
    {
        get
        {
            return handle;
        }
    }
}
```

**80. REC - Utilizar atributos `public static read-only` para instancias predefinidas de los objetos de la clase.**

Debe utilizarse nomenclatura Pascal, pues los atributos son públicos. El siguiente ejemplo muestra el uso correcto de los atributos declarados como `public static read-only`.

```
public struct Color
{
    public static readonly Color Red = new Color(0x0000FF);
    public static readonly Color Green = new Color(0x00FF00);
    public static readonly Color Blue = new Color(0xFF0000);
    public static readonly Color Black = new Color(0x000000);
    public static readonly Color White = new Color(0xFFFFFFFF);

    public Color(int rgb)
    { // Insert code here.}

    public Color(byte r, byte g, byte b)
    { // Insert code here.}

    public byte RedValue
    {
        get
        {
            return Color;
        }
    }
}
```

```

    }
    public byte GreenValue
    {
        get
        {
            return Color;
        }
    }
    public byte BlueValue
    {
        get
        {
            return Color;
        }
    }
}

```


---

## Guías para el uso de Parámetros

### 81. REC - Verificar la validez de los parámetros que se pasan como argumentos.

Se recomienda utilizar la clase `System.ArgumentException` Class, o una clase derivada de `System.ArgumentException`.

Es importante destacar que la validación que se realiza en este ejemplo, no necesariamente tiene que ocurrir en los métodos public o protected en sí mismos, sino que puede realizarse en un nivel más bajo que incluya procedimientos privados. El punto clave es que la interfaz de trabajo que se exponga al programador verifique la validez de los argumentos.

 Ejemplo:

```

class SampleClass
{
    public int Count
    {
        get
        {
            return count;
        }
        set
        {
            // Check for valid parameter.
            if (count < 0 || count >= MaxValue)

```

```

        throw new ArgumentOutOfRangeException(
            Sys.GetString(
                "InvalidArgument", "value", count.ToString()));
    }
}

public void Select(int start, int end)
{
    // Check for valid parameter.
    if (start < 0)
        throw new ArgumentException(
            Sys.GetString("InvalidArgument", "start", start.ToString()));
    // Check for valid parameter.
    if (end < start)
        throw new ArgumentException(
            Sys.GetString("InvalidArgument", "end", end.ToString()));
}
}

```


## 82. Diferencias entre argumentos pasados por valor o por referencia.

Cuando se pasan parámetros por valor, sólo se realiza una copia del valor, y por lo tanto no tiene efecto sobre el valor original.

 Ejemplo:


```
public void Add(object value){}
```

Cuando un parámetro es pasado por referencia, lo que se está transmitiendo es la ubicación en memoria del valor, por lo tanto el valor original es alterado.

 Ejemplo:

```
public static int Exchange(ref int location, int value){}
```

Un parámetro de salida (output parameter) representa la misma ubicación en memoria que la variable especificada en como argumento en la invocación del método, como resultado los cambios se realizarán sólo en el parámetro de salida.

 Ejemplo:

```

[DllImport("Kernel32.dll")]
public static extern bool QueryPerformanceCounter(out long value)

```

## TIPOS DE DATOS

Esta sección describe el uso de tipos para la definición de variables y atributos. En esta primera versión sólo se describe el uso de los tipos `struct` y `enum`.

### 83. REC – Utilización de tipos struct

Es recomendable la utilización de tipos struct cuando el tipo de dato cumple alguno de los siguientes criterios:

- Actúa como un tipo primitivo
- Tiene un tamaño de instancia menor a 16 bytes.
- Es estático.
- Es deseable que los valores conserven una determinada semántica.



Ejemplo:

```
public struct Int32: IComparable, IFormattable
{
    public const int MinValue = -2147483648;
    public const int MaxValue = 2147483647;
    public static string ToString(int i)
    {
        // Insert code here.
    }
    public string ToString(string format, IFormatProvider formatProvider)
    {
        // Insert code here.
    }
    public override string ToString()
    {
        // Insert code here.
    }
    public static int Parse(string s)
    {
        // Insert code here.
        return 0;
    }
    public override int GetHashCode()
    {
        // Insert code here.
        return 0;
    }
}
```

```

    }

    public override bool Equals(object obj)
    {
        // Insert code here.

        return false;
    }

    public int CompareTo(object obj)
    {
        // Insert code here.

        return 0;
    }
}

```


#### **84. OBL – No debe proveerse un constructor para un tipo struct.**

C# no permite que un tipo struct posea un constructor predeterminado. El runtime inserta un constructor que inicializa todos los valores a estado “cero”. Esto permite que los arrays de structs se creen sin necesidad de ejecutar el constructor en cada instancia.

#### **85. REC – Recomendaciones para el uso de enums**

##### **86. No debe utilizarse el sufijo “enum” para los tipos definidos como enum.**

Es recomendable utilizar tipos enum para parámetros, propiedades y valores de retorno que debe ser fuertemente tipados. Siempre deben definirse valores enumerados utilizando enum si son usados en un parámetro o propiedad. Esto permite que las herramientas de desarrollo conocer los posibles valores para las propiedades o parámetros.

 Ejemplo:

```

public enum FileMode
{
    Append,
    Create,
    CreateNew,
    Open,
    OpenOrCreate,
    Truncate
}

```

#### **87. REC – Utilizar tipos enum en lugar de constantes estáticas**

##### **88. No utilizar tipos enum para conjuntos de valores abiertos.**

## ESTILO DE CODIFICACIÓN

Las preferencias en el estilo de codificación causan generalmente controversias entre los desarrolladores, pues muchas veces se enfrentan preferencias personales. Sin embargo, establecer un estilo uniforme de codificación dentro de un centro de desarrollo aporta claridad y genera bibliotecas de código entendible y de fácil mantenimiento.

Esta sección describe el estilo de codificación que se aplica dentro del departamento de Desarrollo de GML Software, con el objetivo de crear aplicaciones cuyo código sea claro, consistente y fácil de entender por todos los programadores.

### 89. REC – Las secciones **public**, **protected** y **private** de una clase, deben ser declaradas en ese orden.

Si bien C# no posee el mismo concepto que C++ en cuanto al acceso a las regiones de código, será entendido como un estándar la definición de las secciones en el orden que se indica.

Las variables privadas deben ser declaradas al principio de la clase (preferiblemente cerca su propia `#region`).

Las clases anidadas deben definirse al final de la clase.

### 90. OBL – Uso de operadores

Esta regla abarca el uso de los siguientes operadores:

Unarios: `&` `*` `+` `-` `~` `!`


Incrementales y decrementales: `--` `++`

Llamadas a funciones y operaciones embebidas: `()` `[]`

Acceso: `.`

No es permitido agregar espacios en blanco entre estos operadores y sus operandos.

No es permitido separar los operadores unitarios de sus operandos con una nueva línea.

 Ejemplo:

```
a = -- b; // mal
a = --c; // bien
a = -b - c; // bien
```

### 91. REC – Utilizar Tabs y expandir a 4 espacios

Esta medida puede personalizarse en Visual Studio.

### 92. REC – No crear líneas de código de más de 100 caracteres

## APÉNDICE 1 – TABLA DE PALABRAS RESERVADAS

Esta tabla resume las palabras reservadas de C#.

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		