## Tabla de Contenidos


1. Introducción ..... 4
   1.1 Propósito ..... 4
   1.2 Alcance ..... 5
2 Organización del Código y Estilo ..... 5
   2.1 Archivos ..... 5
   2.2 Orden en la definición de Clases ..... 5
   2.3 Estructura Visual del Código ..... 6
      2.3.1 Espacios en listas ..... 6
      2.3.2 Bloques de código ..... 6
      2.3.3 Uso de brackets: { y } ..... 6
      2.3.4 Indentación de código ..... 7
      2.3.5 Largo de las líneas de código ..... 7
      2.3.6 Tamaño de las funciones ..... 7
   2.4 Comentarios ..... 7
      2.4.1 Código comentado ..... 7
      2.4.2 Comentarios para generación automática de documentación ..... 8
      2.4.3 Comentarios de tareas pendientes ..... 8
      2.4.4 Comentarios en código no C# ..... 8
3 Nomenclatura ..... 9
   3.1 Nombres descriptivos de clases y objetos ..... 9
   3.2 Terminología aplicable al dominio del problema ..... 10
   3.3 Idioma ..... 10
   3.4 Capitalización ..... 11
      3.4.1 Tipos ..... 11
      3.4.2 Normativa de uso ..... 11

3.4.3 Evite nombres que sean similares o diferentes solo en mayúsculas ..... 12
3.4.4 Sensibilidad a Mayúsculas y Minúsculas ..... 12
3.5 Espacios de Nombres - Namespaces ..... 13
3.6 Clases e interfaces ..... 13
3.7 Enumeraciones ..... 14
4 Expresiones e Instrucciones ..... 15
4.1 Sobrecarga de Operadores ..... 15
4.2 Casts (Conversión de Tipos) ..... 15
5 Manejo de Errores y Excepciones ..... 15
6 Manejo de dependencias ..... 18
6.1 Referencias locales ..... 19
6.2 Dependencia de instalaciones locales ..... 19
6.3 Paquetes de terceros ..... 19
6.4 Referencias a otros proyectos ..... 19
6.5 Manejo de NUGET ..... 20
7 Lineamientos del lenguaje ..... 20
7.1 Tipo string ..... 20
7.2 Variables de tipo implícito ..... 21
7.3 Tipos sin signo ..... 22
7.4 Arreglos ..... 22
7.5 Uso de try-catch para manejar excepciones ..... 22
7.6 Uso de la sentencia using ..... 22
7.7 Operadores && y || ..... 23
7.8 El operador “new” ..... 24
7.9 Manejo de eventos ..... 24
7.10 Consultas LINQ ..... 24
8 Principios de diseño ..... 25
8.1 Principios SOLID ..... 26

8.1.1 S-Responsabilidad simple (*Single responsibility*) ..... 26
8.1.2 O-Abierto/Cerrado (Open/Closed) ..... 26
8.1.3 L-Sustitución Liskov (Liskov substitution) ..... 27
8.1.4 I-Segregación del interface (Interface segregation) ..... 27
8.1.5 D-Inversión de dependencias (Dependency inversion) ..... 27
8.2 Principio DRY (Don't Repeat Yourself) ..... 27
8.3 Principio YAGNI (You Ain't Gonna Need It) ..... 28
8.4 Principio KISS (Keep It Simple Stupid) ..... 28
9 Referencias ..... 28

## 1. INTRODUCCIÓN

"Cualquiera puede escribir código que la computadora pueda entender. Los buenos programadores escriben código que los humanos pueden entender". Martin Fowler.

Los estándares de codificación buscan consistencia en el código producido por cada miembro del equipo. Esta consistencia genera código más fácil de entender, que en consecuencia significa que es más sencillo para desarrollar y mantener. Esto reduce el costo total de las aplicaciones.

El código existirá durante mucho tiempo, aún después de que el desarrollador o equipo de desarrollo se haya movido a otros proyectos. Una meta importante durante el desarrollo es el asegurar que la responsabilidad de mantener y evolucionar el código se pueda transferir a otro desarrollador, o equipo de desarrolladores sin tener que invertir un esfuerzo desmesurado para entender el código existente. El código que es difícil de entender corre el riesgo de ser desechado y reescrito, generando costos frecuentemente inmensos que pudieron haberse evitado.

Un segundo objetivo no menos importante de estos estándares es introducir buenas prácticas que implementan o articulan satisfacer atributos de calidad tales como seguridad, escalabilidad y mantenibilidad entre otros. Se busca prevenir código que aunque funcionalmente sea correcto falle en su entrega al usuario por aspectos no funcionales que pudieron prevenirse aplicando buenas prácticas.

### 1.1 Propósito

Las reglas generales contenidas en este documento buscan estandarizar el desarrollo en diversos tipos de aplicaciones. Al aplicarlas se logran beneficios en los siguientes atributos de calidad:

- Legibilidad del código
- Reutilización del código
- Homogeneidad del código
- Confiabilidad

Este documento considera principios de la orientación a objetos, la estructuración y la arquitectura de objetos en múltiples capas y de las pautas del diseño del Framework de .NET.

### 1.2 Alcance

Las reglas presentadas en éste documento se pueden aplicar a cualquier tipo de aplicación desarrollada en Visual C# .NET.
Esta guía no trata temas tales como separación de una aplicación en componentes, el uso de plantillas de aplicaciones, métodos de acceso a los datos, opciones de compilación, etc.
Esta guía no pretende ser un entrenamiento en C#. Incluye solamente un sistema de lineamientos (de carácter obligatorio) y recomendaciones que se concentran en la clarificación del desarrollo.

## 2 ORGANIZACIÓN DEL CÓDIGO Y ESTILO

### 2.1 Archivos

Sólo debe existir una clase pública por archivo.
El nombre del archivo es derivado del nombre de la clase.

Ejemplo: Clase: Observer
Archivo: Observer.cs

Los archivos C# deben tener su contenido estructurado en el siguiente orden:

- Encabezado de comentario del archivo (Opcional. Más sobre comentarios en la sección de comentarios de este documento)
- Sentencias Using
- Definición del Namespace
- Definiciones de tipos definidas por el usuario (enums y structs)
- Definición de clases

### 2.2 Orden en la definición de Clases

Los miembros de clase deben ser ordenados por categoría, y posteriormente por alcance. De menos alcance restricto (public) a más restrictivo (private).
La definición de clase contiene los miembros de clase en el siguiente orden:

- Tipos anidados
- Campos Miembro
- Funciones Miembros
  - Constructores
  - Finalizadores

- Métodos (propiedades, eventos, operaciones, sobrecargables, estáticos)

- Tipos anidados privados

### 2.3 Estructura Visual del Código

Para mejorar la legibilidad del código el ambiente de desarrollo (Visual Studio) brinda una serie de comportamientos predefinidos en cuanto a espacios y tabulaciones. Como complemento, extensiones como ReSharper o productos similares ayudan al desarrollador a aplicar estos comportamientos tanto en código nuevo como en código existente. Los comportamientos esperados tienen que ver con:

- Espacios en listas
- Bloques
- Indentación del código
- Longitud de la línea
- Tamaño de las funciones

#### 2.3.1 Espacios en listas

Para mejorar la legibilidad y claridad en las listas, después de un separador (punto y coma o coma) se escribirá un espacio en blanco.

```
For (Count = 0, Flags = 0; Count < 5; Count++)
{
        …
}
```

#### 2.3.2 Bloques de código

Un bloque de la declaración compuesta o bloque de código se encapsula entre brackets como sigue:

```
For(…) // bracket must be set on next line
{
        …
}
```

#### 2.3.3 Uso de brackets: { y }

Las llaves {} se utilizan siempre, aunque solamente se trate de una sentencia.

```
If (bFirstTime)
{
        bFirsTime = false;
}
```

#### 2.3.4 Indentación de código

Cada nuevo bloque del código se debe indentar con tabulaciones. No se deben utilizar espacios en vez de tabuladores.

#### 2.3.5 Largo de las líneas de código

Las líneas de código nunca debería sobrepasar el tamaño de la pantalla en la que se está desarrollando. Esto facilita la lectura y edición del código.

#### 2.3.6 Tamaño de las funciones

Lo métodos deben ser cortos por diseño y no deben superar nunca las 100 líneas de código. Es recomendable volver a diseñar los métodos largos en varios métodos cortos. Una buena medida para este aspecto es que la totalidad del código de la función debe caber en la pantalla sin hacer scroll vertical u horizontal; si no es así es hora de hacer "refactor".

### 2.4 Comentarios

Antiguamente se solía dar gran relevancia y valor a los comentarios en el código, de hecho muchas guías como esta dedican numeral tras numeral con reglas obligatorias respecto a comentarios.
Si bien hace años esto fue cierto cuando las limitaciones del lenguaje de programación o de los sistemas hacían difícil crear código expresivo y legible hoy en día esto ya no es el caso.

Hoy en día un comentario es una excusa. Una excusa por no elegir un nombre claro, o un conjunto razonable de parámetros, o por fallar diseñando interfaces intuitivas y lógicas. Son excusas por crear código difícil de mantener, excusas por no utilizar algoritmos y patrones conocidos, por no tener un sistema de control de versiones y por dejar vulnerabilidades o fallas sin corregir.

La recomendación en este punto es, si el desarrollador siente que necesita escribir un comentario para explicar el código que acaba de producir, en vez de escribir el comentario debe hacer "refactor" de su código para que no necesite un comentario.

#### 2.4.1 Código comentado

Frecuentemente los desarrolladores se ven inclinados a comentar secciones de código para reemplazarlos por nuevas implementaciones dejando el código anterior comentado a manera de memoria:

```
// Esta seccion no funcionaba asi que la cambie
//AreaRegistration.RegisterAllAreas();
```

```
//WebApiConfig.Register(GlobalConfiguration.Configuration);
//FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
//RouteConfig.RegisterRoutes(RouteTable.Routes);
//BundleConfig.RegisterBundles(BundleTable.Bundles);
//AuthConfig.RegisterAuth();
```

Esta práctica no es recomendable y puede considerarse como prohibida. Para cambios en la implementación y alternativas de implementación del mismo código el desarrollador debe soportarse en las facilidades del control de versiones. De esa forma es posible ver versión tras versión realmente que fue lo que cambió sin contaminar el código con líneas inútiles y comentarios que frecuentemente están alejados de la realidad.

#### 2.4.2 Comentarios para generación automática de documentación

Este tipo de comentarios se utilizan para generar la documentación de manera automática. Visual Studio proporciona las etiquetas necesarias para crear los comentarios con una sintaxis correcta al escribir las tres barras "///" en la cabecera de una clase, método, etc.
Estos comentarios solo tienen valor cuando se están construyendo APIs que serán expuestas a terceros o al público en general.

En el caso específico de implementaciones de capas de servicios expuestas al público mediante ODATA o similar estos comentarios deben obligatoriamente incluirse para garantizar que la herramienta genere la documentación pertinente.

#### 2.4.3 Comentarios de tareas pendientes

Los desarrolladores pueden utilizar palabras reservadas como "///TODO:" o "///HACK:" para incluir comentarios sobre el código o lógica que necesita ser completada. Los comentarios colocados después de estas palabras son mostrados en el panel de lista de tareas de Visual Studio.

La recomendación para este tipo de comentarios es evitarlos. Si bien pueden brindar algo de valor para desarrolladores individuales en equipos de trabajo soportados por herramientas de ALM el seguimiento de pendientes debe realizarse principalmente a través de la herramienta de ALM, en este caso Team Foundation Server.

#### 2.4.4 Comentarios en código no C#

En el caso de lenguajes que carezcan de las facilidades de C# para construir código legible se considera aceptable el uso de comentarios que expliquen la intención del código. Ejemplos de estos lenguajes incluyen T-SQL, PowerShell y batch, entre otros.

## 3 NOMENCLATURA

La nomenclatura es uno de los elementos más importantes para obtener una buena legibilidad del código. A continuación se muestran algunas de las reglas que se consideran importantes para una correcta nomenclatura del código:

- Emplear una correcta capitalización en cada caso.
- Usar nombres con significado para facilitar la legibilidad del código.
- Usar nombres para las clases, verbos para los métodos, etc.
- Evitar usar abreviaturas salvo que el nombre completo resulte excesivamente largo.
- Todas las abreviaturas deben ser conocidas y aceptadas universalmente.
- No usar palabras reservadas por el lenguaje.
- Evitar conflictos de nombres con los namespaces o tipos del Framework de .NET
- Usar un diccionario de palabras que permitan identificar correctamente los elementos que se están usando. Los nombres de las variables deben ser los suficientemente significativos para que su literal sea representativo por sí mismo
- La terminología aplicada tiene que estar relacionada con el diccionario de términos usado por el Ministerio de Agricultura y Desarrollo Rural.

Estas recomendaciones se basan enteramente en las convenciones de codificación para .NET disponibles en: https://msdn.microsoft.com/en-us/library/ms229045%28v=vs.110%29.aspx
Al igual que con los aspectos visuales, todos o muchos de los aspectos de nomenclatura o nombramiento los realiza, sugiere o corrige el propio ambiente de desarrollo Visual Studio por sí mismo o con ayuda de extensiones como ReSharper o similares.

### 3.1 Nombres descriptivos de clases y objetos

Las clases y objetos deben nombrarse de forma correcta y descriptiva. El nombre debe indicar lo que realmente es y describir de qué se trata.

Por ejemplo, utilizar nombres como primerNombre, granTotal, estacionPolicial. Aunque los nombres como x1, y1 o fn son fáciles de teclear debido a que son cortos, no hay indicación de qué representan y el resultado en código puede ser más difícil de

entender, mantener o mejorar.

Debe preferirse la legibilidad versus la brevedad. Por ejemplo en vez de calcBenSis utilice calcularBeneficioCiudadanosEnSisben.

### 3.2 Terminología aplicable al dominio del problema

Muchos desarrolladores cometen el error de crear términos genéricos para conceptos cuando nombres perfectamente correctos ya forman parte del diccionario de la materia. En el caso del Ministerio de Agricultura y Desarrollo Rural este lineamiento es de carácter obligatorio pues muchos términos son propios del tema abordado por el ministerio, fuera del ministerio pueden tener otro significado, y otros términos introducidos por el equipo de desarrollo van a resultar confusos.

### 3.3 Idioma

C# así como la inmensa mayoría de lenguajes de programación es enteramente en Inglés. En general suele ser más legible un código fuente escrito 100% en Ingles, facilitando además su entendimiento por ingenieros que hablen cualquier idioma pues en esta industria el inglés se considera universal.

En este caso sin embargo es muy importante aplicar los lineamientos anteriores pues el lenguaje de dominio es específico y va a ser valioso para el equipo de desarrollo encontrar los mismos términos de negocio en el código. Es por eso que la recomendación es utilizar una combinación entre inglés y español que permita aplicar los lineamientos aquí expresados y a la vez armonizar con el resto del lenguaje de programación.

La recomendación es:

- Todos los sustantivos, en especial los que pertenezcan al dominio de negocio deben permanecer en español.
- Todos los verbos deben estar en inglés.
- Prefijos y sufijos de negocio en español.
- Prefijos y sufijos del lenguaje de programación como los descritos en el numeral 3.6 en inglés.

Ejemplos:

```
var agricultores=db.getAgricultores();
foreach(var agricultor in agricultores) {
        agricultor.fetchPredio();
        agricultor.queryProductos();
        agricultor.refreshBeneficio();
}
```

### 3.4 Capitalización

#### 3.4.1 Tipos

- **Pascal:** se capitaliza el primer carácter de todas las palabras.

```
- BlackColor
```

- **Camel:** se capitaliza el primer carácter de todas las palabras excepto el primer carácter.

```
blackColor
```

- **Upper:** Todas las abreviaturas van en mayúsculas.

```
System.IO
```

#### 3.4.2 Normativa de uso

Reglas para aplicar los tipos de capitalización presentados en el punto anterior dentro del código de las aplicaciones:

| Tipo | Capitalización | Ejemplo |
|---|---|---|
| Clase | Pascal | AppDomain |
| Tipo de enumeración | Pascal | ErrorLevel |
| Valores de enumeración | Pascal | FatalError |
| Evento | Pascal | ValueChanged |
| Clase de excepción | Pascal | WebException |
| Campo estático de sólo lectura | Pascal | RedValue |
| Variable miembro pública | Pascal | public bool **Enabled;** |
| Variable miembro protegida | Camel | protected object **protectedItem;** |
| Variable miembro privada | Camel | private int privateCount; |
| Interfaz | Pascal | IDisposable |
| Método | Pascal | ToString |
| Espacio de nombres (Namespace) | Pascal | System.Drawing |
| Parámetro | Camel | typeName |
| Propiedad | Pascal | BackColor |

Tabla 1: Normativa de capitalización en función del tipo de objeto

#### 3.4.3 Evite nombres que sean similares o diferentes solo en mayúsculas

En pro de la legibilidad y mantenibilidad debe procurarse evitar cambios sutiles en nombres que terminen generando confusión. Por ejemplo, los nombres de variable objeto Persistente y objetos Persistentes no deben utilizarse juntos.

#### 3.4.4 Sensibilidad a Mayúsculas y Minúsculas

No deben utilizarse nombres que requieran forzar la capitalización pues se prestan a confusiones y degradan la mantenibilidad del código.

Ejemplos específicos de lo que no se debe hacer:

Namespaces cuyo nombre difiera solamente en la capitalización.

```
namespace kim.ralls;
namespace Kim.Ralls;
```

Funciones con dos parámetros cuyos nombres difieran solo en la capitalización.

```
void boo(string a, string A)
```

Namespaces con dos tipos cuyo nombre solo difiera en la capitalización.

```
System.WinForms.Point p;
System.WinForms.POINT pp;
```

Tipos con dos propiedades cuyo nombre solo difiera en la capitalización.

```
int Boo {get, set};
int BOO {get, set}
```

Un tipo con dos métodos cuyos nombres difieran solo en la capitalización.

```
void boo();
void Boo();
```

### 3.5 Espacios de Nombres – Namespaces

La regla general para nombrado de namespaces es:

```
MADR.Productores360.Iniciativa.Aspecto.DetalleAspecto
```

Por ejemplo, si se está codificando un componente de acceso a datos para asistencia técnica su namespace será de esta manera:

```
MADR.Productores360.AsistenciaTecnica.AccesoDatos.ServiciosDane
```

No se deben nombrar clases con el mismo nombre del namespace pues el código se vuelve confuso, por ejemplo para el namespace anterior no debería existir una clase llamada ServiciosDane.

### 3.6 Clases e interfaces

- Los nombres de tipo deben ser sustantivos correspondientes al nombre de la entidad representada, por ejemplo Formulario, Predio, Agricultor.
- Se utilizará Pascal Casing.
- No se deben utilizar abreviaturas ni prefijos, la única excepción son las interfaces, que deben llevar el prefijo I.
- En el caso de tipos heredados, se recomienda poner como sufijo el nombre de la clase base, por ejemplo los tipos que hereden de Beneficio deberán llevar el sufijo Beneficio.
- En el caso de tipos que implementan patrones comunes de C#, se recomienda utilizar como sufijo el nombre del patrón:
    - Sufijo Attribute a las clases de atributo personalizadas, por ejemplo ObsoleteAttribute y AttributeUsageAttribute
    - Sufijo EventHandler a los nombres de tipos que se utilizan en eventos (como tipos de valor devueltos de un evento de C#), por ejemplo AssemblyLoadEventHandler
    - Sufijo Callback al nombre de un delegado que no sea un controlador de eventos.
    - No debe agregarse el sufijo Delegate a un delegado.
    - Sufijo EventArgs a las clases que extienden System.EventArgs.
    - Sufijo Exception a los tipos que heredan de System.Exception.

- Sufijo Dictionary a los tipos que implementan System.Collections.IDictionary o System.Collections.Generic.IDictionary<TKey, TValue>.
- Sufijo Collection a tipos que implementan System.Collections.IEnumerable, System.Collections.ICollection, System.Collections.IList, System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.ICollection<T> o System.Collections.Generic.IList<T>.
- Sufijo Stream a los tipos que heredan de System.IO.Stream.
- Sufijo Permission a los tipos que heredan de System.Security.CodeAccessPermission o implementan System.Security.IPermission.

### 3.7 Enumeraciones

- No debe utilizarse un prefijo en los nombres de los valores en la enumeración.

Esto también implica que no debe incluirse el nombre de tipo de enumeración en los nombres de los valores de enumeración. El siguiente ejemplo de código explica la denominación incorrecta de los valores de una enumeración.

```
public  enum Teams
{
   TeamsAlpha,
   TeamsBeta,
   TeamsDelta
}
```

- No debe utilizarse Enum como sufijo para los tipos de enumeración.
- No debe utilizarse Flags como sufijo para los nombres de enumeraciones de marcadores.
- El nombre de la enumeración debe ser singular, a menos que sus valores sean campos de bits.
- Para las enumeraciones que tienen campos de bits como valores, también denominados enumeraciones de marcadores, debe utilizarse el nombre en plural.

# 4 EXPRESIONES E INSTRUCCIONES

## 4.1 Sobrecarga de Operadores

La sobrecarga de operadores es útil cuando es inmediatamente obvio cual será el resultado de la operación. Por ejemplo, hace mucho sentido restar un valor Time de otro y obtener un TimeSpan.

```
class Time
{
  TimeSpan operator -(Time t1, Time t2) { }
  TimeSpan Difference(Time t1, Time t2) { }
}
```

Antes de tomar ventaja de estas facilidades del lenguaje debe considerarse su conveniencia desde el punto de vista de legibilidad y mantenibilidad del código. Si al introducir el operador se está fortaleciendo esos aspectos en vez de debilitarlos.

## 4.2 Casts (Conversión de Tipos)

Debe evitarse perder precisión en un cast implícito. Por ejemplo, no debería existir un cast implícito de un Double a Int32, pero puede existir uno de Int32 a Int64. En los casos donde por razones de peso sea absolutamente necesario perder precisión o hacer truncamientos es necesario o bien notificar al usuario interactivamente si se trata de una operación de este tipo, o si son procesos en batch o sin interacción de usuario dejar al menos una alerta en el registro de la aplicación sobre el truncamiento.
No deben generarse excepciones de un cast implícito porque será muy difícil para el desarrollador entender que está sucediendo.

# 5 MANEJO DE ERRORES Y EXCEPCIONES

> **Recomendación:** Utilice al menos estos constructores comunes.

```
public class Xxx Exception : Exception
{
  XxxException() { }
  XxxException(string message) { }
  XxxException(string message, Exception inner) { }
}
```

> **Recomendación:** Utilice tipos de excepción predefinidos. Solo defina nuevas excepciones para escenarios programáticos.

Agregue una nueva clase de excepción para que un programador pueda tomar una acción diferente en código basado en la clase de la excepción.
No defina una nueva clase de excepción a menos que exista un escenario para un desarrollador que la necesite.

Por ejemplo, hace sentido definir *FileNotFoundException* debido a que un programador puede decidir crear un archive no existente, sin embargo *FileIOException* no es algo que típicamente sea manejado en código.

Asegúrese de que en excepciones, que no tengan un mensaje explicito (no null), la propiedad Message revisa el árbol de excepciones hasta obtener el siguiente texto real.
No derive nuevas excepciones de la clase base Exception. La clase base tendrá múltiples subclases de acuerdo al namespace individual.

Agrupe nuevas excepciones de la clase base Exception por namespace. La clase base tendrá múltiples subclases de acuerdo al namespace individual. Por ejemplo, habrá subclases para XML, IO, Collections, etc. Cada una de estas áreas hereda sus propias excepciones como sea necesario. Cualquier excepción que otras librerías o escritores de librerías desean agregar extenderá la excepción directamente. Un nombre sencillo para todas las excepciones relacionadas debe ser creado, y todas las excepciones relacionadas a una aplicación o librería deben extender desde ella.

Utilice mensajes de error gramáticamente correctos, incluyendo la puntuación.
Cada oración termina con un punto. Generalmente el código que despliega un mensaje de excepción al usuario maneja el caso en el que el desarrollador olvido poner el punto final.

Provea propiedades de la excepción que permitan el acceso programático. Incluya información adicional en una excepción (además de incluir una descripción) solo cuando exista un escenario donde la información adicional sea de utilidad. Solo en raras ocasiones se requiere incluir información adicional para la excepción.

Levante excepciones solamente en casos excepcionales, el flujo normal de la aplicación no debe ser controlado con excepciones. No utilice excepciones para errores normales o esperados

> **Recomendación:** Regrese null solo para casos de error extremadamente comunes. Por ejemplo, File.open regresa un null si el archivo no es encontrado, pero levanta una excepción si el archivo está bloqueado (locked).

Diseñe las clases de forma que nunca debieran levantar una excepción. Por ejemplo, una clase FileStream expone otra forma de determinar si el final del archivo ha sido alcanzado, por lo que evita que la excepción sea levantada si el desarrollador hace una lectura que pasa del fin del archivo. Por ejemplo:

```
class Boo
{
  void Bar()
  {
    FileStream stream = File.Open("myfile.txt");
    byte b;
    // ReadByte returns -1 at EOF
    while ((b = stream.ReadByte()) > = 0)
    {
      // do something
    }
  }
}
```

Levante un *InvalidOperationException* si está el estado correcto.

Se supone que una excepción *System.InvalidOperationException* debería ser levantada si la propiedad o la llamada al método no es la correcta, dado el estado actual del objeto.

Levante una excepción *ArgumentException* o una especialización de esta si los argumentos de entrada son incorrectos. Cuando se detecta un parámetro incorrecto, levantar un *System.ArgumentException* o una especialización de esta.

> **Recomendación:** Utilice métodos de construcción de excepciones.

Es común en una clase, levantar la misma excepción desde diferentes lugares en su implementación. Para evitar repetir código, utilice métodos de ayuda para crear una nueva excepción y regresarlo. Por ejemplo:

```
class File
{
  string fileName;

  public byte[] Read(int bytes)
  {
```

```
    if (!ReadFile(handle, bytes))
      throw NewFileIOException();
  }

  FileException NewFileIOException()
  {
    string description = // build localized string, including fileName
    return new FileException(description);
   }
}
```

Otra alternativa es utilizar el constructor de la excepción para construirla. Esto es mejor para clases de excepciones globales como *ArgumentException*.

Levantar excepciones es preferible a regresar un código de error (o HResult).
Levante la excepción más específica.

Se debe favorecer el uso de excepciones existentes a crear nuevas.
Asigne todos los campos de las excepciones que utilice.

No utilice excepciones Inner (Inner exceptions - excepciones encadenadas).
Asigne mensajes de texto significativo y dirigido al desarrollador en una excepción.

No tenga métodos que levanten NullReferenceException o *IndexOutOfRangeException*. Realice una revisión de los argumentos en miembros protegidos. Recuerde limpiar el estatus en la documentación si el método protegido no realiza revisión de los argumentos de entrada. A menos que otra cosa sea anotada, asuma que la revisión de argumentos es realizada. Pueden existir mejoras en el rendimiento cuando la revisión de argumentos no se realiza.

Realice limpieza de los resultados intermedios cuando se levanta una excepción. Los invocantes deben ser capaces de asumir que no hay efectos secundarios cuando ocurre una excepción en una función.

> **Ejemplo:** Si Hashtable.insert levanta una excepción, entonces el invocante debe asumir que el elemento no fue agregado al Hashtable.

## 6 MANEJO DE DEPENDENCIAS

Frecuentemente cuando se retoma un proyecto o se agrega un nuevo miembro al equipo de desarrollo pasa un tiempo considerable preparando el ambiente de desarrollo debido a dependencias. A pesar de contar con el código fuente este no compila o corre debido a dependencias fuera del contenedor del proyecto de desarrollo. A continuación

se describen algunas prácticas para minimizar ese tiempo.

### 6.1 Referencias locales

No debe hacerse referencia a recursos locales del computador de desarrollador, con rutas físicas locales a su propio computador. En el caso de referencias a librerías del sistema estas deben referenciarse desde el GAC. En el caso de referencias a librerías de productos de terceros como paquetes de controles o similares el manejo de las referencias debe delegarse en NUGET. Si el producto o librería no está disponible en NUGET y se trata de un binario esta debe agregarse como parte del contenedor del proyecto de desarrollo para que a su vez haga parte de los artefactos en control de código.

### 6.2 Dependencia de instalaciones locales

Comúnmente es inevitable depender de algunos instaladores, entre ellos el SDK de Azure, el SDK de Dynamics, entre otros. La recomendación en este punto es que depender del instalador sea el último recurso luego de haber agotado la disponibilidad en NUGET de la dependencia.

### 6.3 Paquetes de terceros

Los proyectos modernos frecuentemente integran una serie de paquetes de terceros, muchos de ellos sin una empresa detrás de ellos ofreciendo soporte y actualizaciones. Se recomienda antes de incluir un paquete de terceros:

- Evaluar los términos de la licencia del paquete verificando que los términos de eso sean acordes a lo que se hará con el componente.
- Evaluar la continuidad del componente observando cuantos desarrolladores están activamente desarrollando el producto, con qué frecuencia liberan versiones y que tan rápido responden a problemas reportados, especialmente de seguridad.
- Al incluir un paquete de un tercero se asumen una serie de riesgos medidos sobre la seguridad, escalabilidad y confiabilidad de ese componente. Ese riesgo baja al utilizar librerías populares y ampliamente conocidas en el mercado.

### 6.4 Referencias a otros proyectos

Las referencias entre proyectos del mismo contenedor de proyecto deben agregarse como referencias al proyecto, no a la salida binaria del proyecto. Específicamente no se deben referenciar DLLs sino el proyecto que produce la o las DLLs.

### 6.5 Manejo de NUGET

Las dependencias que se agreguen vía NUGET deben agregarse a nivel de la solución, no a nivel de uno o varios proyectos específicos. Al agregarlos a nivel de la solución Visual Studio / NUGET logran un manejo óptimo de las dependencias y sus versiones.
En el caso de proyectos nuevos la plantilla de Visual Studio hace referencia a las dependencias en la versión que existía con esa versión de Visual Studio; es recomendable luego de crear un nuevo proyecto revisar en el panel de NUGET las versiones actualizadas de esas dependencias. En la mayoría de los casos es recomendable actualizar a la última versión.

## 7 LINEAMIENTOS DEL LENGUAJE

### 7.1 Tipo string

- Para concatenar pocas o cortas cadenas es aceptable utilizar el operador “+”.

```
string displayName = nameList[n].LastName + ", " + nameList[n].FirstName;
```

Para concatenar cadenas en bucles, muchas cadenas, o segmentos muy largos de texto debe utilizarse un objeto StringBuilder.

```
 var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
Debug.Print("tra" + manyPhrases);
```

El operador “+” o la sintaxis string.Concat son intercambiables. No hay ninguna diferencia en desempeño puesto que ambas expresiones terminan compiladas en el mismo código. El uso del uno o del otro tiene que ver más con la legibilidad del código.

- Para inicializar o asignar una cadena vacía debe utilizarse string.Empty en vez de “”

```
//Incorrecto
var stringVacio="";
//Correcto
var stringVacio=string.Empty;
```

Para determinar si un string está vacio debe evaluarse mediate string.IsNullOrEmpty en vez de utilizar la comparación contra “”

```
//Incorrecto
if(nombreAgricultor==””) { … }
//Correcto
if(string.IsNullOrEmpty(nombreAgricultor)) { … }
```

### 7.2 Variables de tipo implícito

Se recomienda el uso de tipos implícitos en general, en especial cuando el tipo de la variable es obvio a partir del lado derecho de su asignación, o cuando el tipo realmente no tiene importancia:

```
var var1 = "This is clearly a string.";
var var2 = 27;
var var3 = Convert.ToInt32(Console.ReadLine());
```

Se recomienda evitarlo cuando el tipo no es aparente a partir del lado derecho de la asignación:

```
int var4 = ExampleClass.ResultSoFar();
```

No debe dependerse del nombre de la variable para determinar su tipo, esto no solo dificulta la legibilidad sino que puede ser falso.

```
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- No debe utilizarse “var” en lugar de “dynamic”

- Deben usarse tipos implícitos para bucles tipo for:

```
var syllable = "ha";
var laugh = "";
for (var i = 0; i < 10; i++)
{
    laugh += syllable;
    Console.WriteLine(laugh);
}
```

- Y para bucles foreach

```
foreach (var ch in laugh)
{
    if (ch == 'h')
```

```
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

### 7.3 Tipos sin signo

En general se recomienda utilizar **int** en vez de tipos sin signo. El tipo integer es común en c# y va a interoperar adecuadamente con más sistemas y plataformas que un tipo sin signo.

### 7.4 Arreglos

- Se recomienda utilizar una sintaxis consistente para definir los arreglos. Se recomienda la siguiente sintaxis para definir e inicializar arreglos:

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

### 7.5 Uso de try-catch para manejar excepciones

Cuando se introduzcan bloques try-catch al final del manejo de la excepción en la inmensa mayoría de los casos es recomendable hacer **throw** sin argumentos como se observa a continuación:

```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}", index);
        throw;
    }
}
```

### 7.6 Uso de la sentencia using

En el ciclo de vida de cualquier objeto que implemente **IDisposable** o cualquier rutina con un try-**finally** donde el único código en el bloque **finally** es un llamado al **Dispose** se recomienda simplificar el código con la sentencia **using**. Este código:

```
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}
```

Se simplifica así:

```
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset = font2.GdiCharSet;
}
```

### 7.7 Operadores && y ||

Se recomienda utilizar los operadores && y || en vez de & y | para evitar excepciones en tiempo de ejecución y mejorar el desempeño:

```
Console.Write("Enter a dividend: ");
var dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
var divisor = Convert.ToInt32(Console.ReadLine());
```

Si el divisor es 0 la segunda condición va a causar una excepción en tiempo de ejecución. El operador && hace falsa la evaluación inmediatamente, el operador & permite ambas evaluaciones produciendo un error en tiempo de ejecución cuando el divisor es 0.

```
if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

### 7.8 El operador "new"

- Se recomienda seguir una estructura consistente en la instanciación de nuevos objetos utilizando tipado implícito:

```
var instance1 = new ExampleClass();
```

- Se recomienda utilizar inicializadores de objetos para simplificar su creación

```
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,
   Location = "Redmond", Age = 2.3 };
```

### 7.9 Manejo de eventos

- Se recomienda el uso de expresiones lamda para suscribirse a eventos:

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

### 7.10 Consultas LINQ

- Se recomienda utilizar nombres representativos del resultado que se espera para nombrar las consultas:

```
var seattleCustomers = from cust in customers
                       where cust.City == "Seattle"
                       select cust.Name;
```

- Se recomienda utilizar alias para asegurar los nombre de las propiedades de tipos anónimos:

```
var localDistributors =
   from customer in customers
   join distributor in distributors on customer.City equals distributor.City
```

```
    select new { Customer = customer, Distributor = distributor };
```

- Se recomienda renombrar propiedades cuando los nombres que resulten de la consulta pueden resultar ambiguos:

```
var localDistributors2 =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorID = dist.ID };
```

- Se deben utilizar tipos implícitos:

```
var seattleCustomers = from cust in customers
                       where cust.City == "Seattle"
                       select cust.Name;
```

- Se debe incluir la cláusula **where** antes de cualquier otra cláusula de consulta para procurar que las cláusulas posteriores operen en un conjunto de datos reducido:

```
var seattleCustomers2 = from cust in customers
                        where cust.City == "Seattle"
                        orderby cust.Name
                        select cust;
```

- Se recomienda utilizar varias cláusulas **from** en vez de un **join** para acceder a colecciones anidadas:

```
var scoreQuery = from student in students
                 from score in student.Scores
                 where score > 90
                 select new { Last = student.LastName, score };
```

## 8 PRINCIPIOS DE DISEÑO

Fuera de estilo y reglas básicas es importante aplicar los principios de diseño que como industria se consideran hoy en día básicos en la construcción de código de clase mundial. Esta sección describe los principios fundamentales que se deben aplicar para lograr un código con los atributos de calidad propuestos.

### 8.1 Principios SOLID

Solid es un acrónimo inventado por Robert C.Martin para establecer los cinco principios básicos de la programación orientada a objetos y diseño. Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.

El objetivo de tener un buen diseño de programación es abarcar la fase de mantenimiento de una manera más legible y sencilla así como conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo. Los costos de mantenimiento pueden abarcar el 80% de un proyecto de software por lo que hay que valorar un buen diseño.

#### 8.1.1 S-Responsabilidad simple (*Single responsibility*)

Este principio trata de destinar cada clase a una finalidad sencilla y concreta. A veces se incluyen métodos reutilizables que no tienen nada que ver con la clase simplemente porque esta lo utiliza.

El problema surge cuando se quiere reutilizar ese mismo método desde otra clase. Si no se re factoriza en ese momento y se crea una clase destinada para la finalidad del método a largo plazo el código termina plagado de clases que realizan tareas que no son su responsabilidad.
Esto conlleva a métodos difíciles de detectar, código difícil de mantener, reutilizar y probar.

#### 8.1.2 O-Abierto/Cerrado (Open/Closed)

Principio atribuido a Bertrand Meyer que habla de crear clases extensibles sin necesidad de entrar al código fuente a modificarlo. Es decir, el diseño debe ser abierto para poderse extender pero cerrado para poderse modificar. Para conseguir este principio hay que tener muy claro cómo va a funcionar la aplicación, por donde se puede extender y cómo van a interactuar las clases.

El uso más común de extensión es mediante la herencia y la re implementación de métodos. Existe otra alternativa que consiste en utilizar métodos que acepten una interface de manera que podemos ejecutar cualquier clase que implemente ese interface. En todos los casos, el comportamiento de la clase cambia sin necesidad de tocar código interno.

#### 8.1.3 L-Sustitución Liskov (Liskov substitution)

Este principio fue creado por Bárbara Liskov y habla de la importancia de crear todas las clases derivadas para que también puedan ser tratadas como la propia clase base. Cuando se crean clases derivadas no se deben re implementar métodos que hagan que los métodos de la clase base no funcionen si se tratan como un objeto de esa clase base.

#### 8.1.4 I-Segregación del interface (Interface segregation)

Este principio fue formulado por Robert C. Martin y trata de algo parecido al primer principio. Cuando se definen interfaces estos deben ser específicos a una finalidad concreta. Por ello, si se definen una serie de métodos abstractos que debe utilizar una clase a través de interfaces, es preferible tener muchos interfaces que definan pocos métodos que tener un interface con muchos métodos.

El objetivo de este principio es principalmente poder reaprovechar los interfaces en otras clases. Si existe una interface que compara y clona en la misma interface, de manera más complicada se podrá utilizar en una clase que solo debe comparar o en otra que solo debe clonar.

#### 8.1.5 D-Inversión de dependencias (Dependency inversion)

También fue definido por Robert C. Martin. El objetivo de este principio conseguir desacoplar las clases. En todo diseño siempre podrá existir algún acoplamiento pero hay que evitarlo en la medida de lo posible. Un sistema no acoplado no hace nada pero un sistema altamente acoplado es muy difícil de mantener.
El objetivo de este principio es el uso de abstracciones para conseguir que una clase interactúe con otras clases sin que las conozca directamente. Es decir, las clases de nivel superior no deben conocer las clases de nivel inferior. Dicho de otro modo, no debe conocer los detalles. Existen diferentes patrones como la inyección de dependencias o service locator que permiten invertir el control.

### 8.2 Principio DRY (Don't Repeat Yourself)

Este principio fue formulado por Andy Hunt y Dave Thomas en "The Pragmatic Programmer". La interpretación popular de este principio es algo así como:
"No repitas código, si hay código duplicado refactorizas y reutilizas el código en lugar de duplicarlo".

El código duplicado es un problema entre otras cosas porque al hacer cambios deben hacerse en varios sitios con el riesgo de olvidar hacer esos cambios en algún sitio y dejar el código inconsistente. Incluso aunque no falte ningún el propio hecho de tener

que cambiar en varios sitios obliga un esfuerzo extra de mantenimiento.
El principio DRY tiene además unas pretensiones mucho más amplias. Su formulación original es "Cada pieza de conocimiento dentro de un sistema debe tener una representación única, clara y acreditada" -Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.-

### 8.3 Principio YAGNI (You Ain't Gonna Need It)

YAGNI es un principio de Extreme Programming (XP) que establece que no se debe implementar funcionalidad hasta que sea necesario. En palabras de Ron Jeffries uno de los fundadores de XP "Siempre implementa las cosas que necesitas actualmente, nunca las que preveas que vas a necesitar".

Incluso si existe total seguridad de que hay algo que se necesitará en el futuro es mejor no implementarlo aún por varios motivos. Primero puede que después de todo no se necesite, y segundo porque puede que lo que se necesite en el futuro sea bastante diferente de lo que en el momento se cree que se va a necesitar.

El sentido de este principio no es que no se sobredimensione algo basándose en las cosas que se asumen sin tener información completa, y, evitar la creación de código u otra clase de artefactos que en realidad no se necesitarán.

### 8.4 Principio KISS (Keep It Simple Stupid)

Este principio establece que "la mayoría de los sistemas funcionan mejor si se mantienen simples en lugar de hacerlos complejos". La frase se le atribuye al ingeniero aeronáutico Kelly Johnson. No es un principio específico del diseño de software sino más bien de la ingeniería en general, pero su aplicación es muy recomendable en el desarrollo de sistemas.
En cada iteración del desarrollo se debe crear el diseño más simple que podría funcionar. Cada decisión de diseño debe tomar en cuenta el criterio de la forma más simple de lograr lo propuesto.

## 9 REFERENCIAS

A continuación los libros y artículos de referencia en los que se basa el contenido de este documento y que son base para desarrollos C# de clase mundial.

- Clean Code: A Handbook of Agile Software Craftsmanship
- Refactoring: Improving the Design of Existing Code
- The Pragmatic Programmer: From Journeyman to Master

- The Clean Coder: A Code of Conduct for Professional Programmers
- Application Architecture for .NET: Designing Applications and Services
- Naming Guidelines from Microsoft for .Net
- Handling Exceptions
- Code Complete Second Edition / http://cc2e.com/
- 10 ways to make your .NET projects play nice with others

## 10. Historial de Cambios

| Fecha | Versión | Descripción |
| --- | --- | --- |
| 25 junio 2015 | 1 | Inicial |