EMSys
Estándar de Implementación
Versión 2.2

Historia de revisiones

Fecha	Versión	Descripción	Autor
22/08/2016	2.0	Estándar básico	Andres Veiro
25/08/2016	2.1	Agrego las convenciones proporcionadas por Sonda	Andres Veiro
28/08/2016	2.2	Revisión y correcciones de estilo y contenido	Marccio Silva

Contenido

1.Objetivo

El objetivo de este documento es que todos los implementadores usen las mismas convenciones al momento de codificar, permitiendo mejorar la legibilidad, facilitar el mantenimiento, incluyendo recomendaciones de estilo para la codificación en *C#*.

Cabe destacar que las convenciones aquí presentadas se basan en las convenciones de codificación de *Microsoft*.

2.Convenciones Generales

Se utiliza la configuración predeterminada de *Visual Studio*: sangría automática, sangrías de cuatro caracteres, tabuladores guardados como espacios. Para hacer esto de forma automática, se debe usar Ctrl+K seguido de Ctrl+D.

Se debe escribir una sola declaración por línea. Entonces, el siguiente ejemplo no será admitido:

```
int a; int b;
```

La forma correcta será:

```
int a;
int b;
```

Se deberá escribir una única instrucción por línea. Entonces, el siguiente ejemplo no será admitido:

```
BuscarMenor(); BuscarMayor();
```

La forma correcta será:

```
BuscarMenor();
BuscarMayor();
```

En la definición de clases entre los métodos y las propiedades se deberá dejar una línea en blanco.

Se deberán usar paréntesis en las cláusulas de una expresión. Por ejemplo:

```
If ((var1 > var2) && ( var1 > var3))
{
    //código
}
```

El operador “+” se deberá utilizar para concatenar cadenas de caracteres cortas. El siguiente ejemplo no será admitido:

```
String variableParaConcatenar = "ejemplo";
String variableConcat;
String prueba = "esto es un string" + variableContact + variableParaConcatenar
```

La forma correcta es:

```
String.Format("esto es un string {0} {1}", variableContact, variableParaConcatenar)
```

En caso de utilizar variables de tipo entero (*int*), será preferible utilizar enteros con signo (no *unsigned*).

Se deberán de utilizar bloques *try-catch* para controlar la mayor parte de excepciones.

Se deberán de utilizar nombres descriptivos tanto para variables como funciones.

El código deberá ser escrito en Inglés para facilitar la codificación de métodos como los *get*, *set*. Además, se deberá evitar el uso de caracteres como “ñ” y tildes.

3.Convenciones Específicas

3.1.Comentarios

- Los comentarios se deberán realizar utilizando “//”
- Se deberán situar en una línea independiente al código, preferiblemente en la previa
- Deberán comenzar con letra mayúscula
- Entre “//” y el comentario se deberá agregar un espacio
- Si el comentario ocupa varias líneas no se deberán crear bloques “/**/”

El siguiente ejemplo no será admitido:

```
/*esto es un comentario de varias filas,
o sea incorrecto*/
```

La forma correcta será:

```
// Esto es un comentario correcto
//que tiene dos líneas
```

Todas las funciones deben tener un encabezado que tenga una descripción de lo que hace, parámetros utilizados y significado de su retorno. Esto se podrá autogenerar en *Visual Studio* utilizando la secuencia de caracteres “///”; de esta manera la herramienta Intellisense podrá

mostrar al implementador esta descripción al escribir el nombre del método, facilitando su utilización y entendimiento.

3.2.Variables

Las variables privadas deberán comenzar con "_" y continuar con minúscula. Por ejemplo:

```
private string _ejemplo;
```

Las variables públicas empiezan con minúscula. Por ejemplo:

```
public string ejemplo;
```

Los nombres de variables que incluyan más de una palabra deberán comenzar con una letra minúscula, y cada nueva palabra deberá comenzar con una letra mayúscula. Por ejemplo:

```
private string _ejemploPalabra
```

3.3.Funciones

El nombre de las funciones se utiliza el estilo *Camel* (primera letra de cada palabra en mayúscula y palabras unidas).

4.Manejo de excepciones

Se manejan excepciones propias y genéricas, además esta sección indica qué logs de reporte se deberán agregar.

Se debe tener en cuenta que:

1. La primera función que captura (*catch*) la excepción es la que guarda el log.
2. Si esa función es invocada por otra entonces genera una nueva excepción del tipo correspondiente con el StringType adecuado.
3. Si la función de más "arriba" es del tipo interfaz gráfica entonces muestra un mensaje agradable (no destinado a un usuario técnico) e informativo respecto de la excepción.

4.1.Excepciones propias

Con el fin de soportar las restricciones exigidas por Sonda, se deberán utilizar excepciones propias con el fin de facilitar las actividades de implementación y verificación. Los tipos de excepciones sugeridos son:

- **LogicException** – Cuando es una excepción generada en la lógica (*.cs*)
- **InterfaceException** – Cuando es una excepción generada en la UI (*.xaml.cs*)

4.2.Cómo capturar errores

Dependiendo del tipo de función, la captura de errores se puede dar de algunas de las maneras mencionadas en las siguientes subsecciones.

4.2.1.Interfaces de primer nivel

Estas funciones son las que generan acciones, por ejemplo *ClickHandler()*
El siguiente código ilustra cómo se deberían de manejar:

```
try
{
	// Código.
}
// Sólo si desde esta UI se invoca a otro elemento de UI.
catch (InterfaceException ex)
{
	MainWindow.ShowError(ex.Message);
}
// Sólo si desde esta UI se invoca lógica.
catch (LogicException ex)
{
	MainWindow.ShowError(ex.Message);
}
catch (Exception ex)
{
	LogMaker.MakeErrorLog(ActionType.Y, ex, "En NOMBRE_FUNCION", EntityType.Z,
ModuleType.W, (-1|id));
	MainWindow.ShowError(StringsType.X);
}
```

4.2.2.Interfaces de segundo nivel

Estas funciones son invocadas por otras, no son generadas por acciones de usuario.
El siguiente código ilustra cómo se deberían de manejar:

```
try
{
	// Código.
}
// Sólo si desde esta UI se invoca a otro elemento de UI.
catch (InterfaceException)
{
	throw;
}
// Sólo si desde esta UI se invoca lógica.
catch (LogicException)
{
	throw;
}
catch (Exception ex)
{
	LogMaker.MakeErrorLog(ActionType.Y, ex, "En NOMBRE_FUNCION", EntityType.Z,
ModuleType.W, (-1|id));
	throw new InterfaceException(StringsType.X, ex.InnerException);
}
```

4.2.3.Lógica de primer nivel

Estas funciones son la de nivel superior o “más arriba”, pero que no forman parte de la GUI.
El siguiente código ilustra cómo se deberían de manejar:

```
try
{
	// Código.
}
// Sólo si desde esta lógica se invoca otra UI.
catch (InterfaceException ex)
{
	// Según cada caso evalúa que hacer, dado que si bien el componente
	// el componente actual es el de más alto nivel, no es de GUI.
}
// Sólo si desde esta lógica se invoca a otro elemento de lógica.
catch (LogicException)
{
	// Según cada caso evalúa que hacer, dado que si bien el componente
	// el componente actual es el de más alto nivel, no es de GUI.
}
catch (Exception ex)
{
	LogMaker.MakeErrorLog(ActionType.Y, ex, "En NOMBRE_FUNCION", EntityType.Z,
ModuleType.W, (-1|id));
	// Según cada caso evalúa que hacer, dado que si bien el componente
	// el componente actual es el de más alto nivel, no es de GUI.
}
```

4.2.4.Lógica que se invoca desde otro lugar

Estas funciones corresponden a la UI u otros elementos de lógica.
El siguiente código ilustra cómo se deberían de manejar:

```
try
{
	// Código.
}
// Sólo si desde esta lógica se invoca otra UI.
catch (InterfaceException)
{
	throw;
}
// Sólo si desde esta lógica se invoca a otro elemento de lógica.
catch (LogicException)
{
	throw;
}
catch (Exception ex)
{
	LogMaker.MakeErrorLog(ActionType.Y, ex, "En NOMBRE_FUNCION", EntityType.Z,
ModuleType.W, (-1|id));
	throw new LogicException(StringsType.X, ex.InnerException);
}
```

5.Referencias

Se utilizaron como referencia las convenciones de Microsoft:
http://msdn.microsoft.com/es-es/library/ff926074%28v=vs.110%29.aspx

La sección relacionada al manejo de las excepciones fue tomada de la guía proporcionada por Sonda.