Estándares de programación

*Pascal-Casing:* los primeros caracteres de cada palabra que compone el nombre esta en mayusculas y todo el resto en minuscula (Ejm: HelloWorld).

*Camel-Casing*: los primeros caracteres de cada palabra que compone el nombre esta en mayusculas, a excepcion de la primera palabra que empieza con minuscula (Ejm: helloWorld).

**Clases**

*Estructura*.

class MyClass

{

#Constantes

#Atributos privados

#Atributos protegidos

#Propiedades publicas

#Propiedades protegidas

#Comandos (Si es MVVM)

#Events

#Constructores

#Metodos Publicos

#Metodos Protegidos

#Metodos Privados

#Event Handlers (tienen que ser privados)

#Metodos estaticos publicos

#Metodos estatidos privados

}

La clase debe estar ordenada lo más parecido posible a la estructura anteriormente mostrada.

No usar regiones (ejm: #region Attributes), a menos que sea para agrupar metodos que cumplan una misma función.

*Constantes*

* son valores que no cambiaran en ningun momento de la ejecución del sistema. Utilizarlos para evitar “valores mágicos”.
* Las constantes se nombran con solo mayusculas y se utiliza barra-baja para separar entre palabras.

double a = 2 \* 2.71828 \* radio - 1; (BAD!)

public const double EULER = 2.71828;

public const double FUNCTION\_ERROR = 1;

double a = 2 \* EULER \* radio – FUNCTION\_ERROR; (GOOD!)

*Atributos*.-

* Son las caracteristicas de la clase
* Los atributos son privados
* Los atributos se nombran en Camel-Casing
* Los atributos que poseen una propiedad para el acceso empiezan con una sola barra-baja como prefijo.
* Los atributos que no tienen una propiedad asociada empiezan con doble barra-baja como prefijo.

private string name;

private int bankAccount; (BAD!)

private string \_name;

private int \_\_bankAccount; (GOOD!)

*Propiedades*

* Accesos a ciertas características de la clase.
* Las propiedades son publicas o protegidas.
* Las propiedades se nombran en Pascal-Casing

public string name {get; set;}

public int bankAccount {get; set;} (BAD!)

public string Name{get; set;}

public int BankAccount {get; set;} (GOOD!)

*Nota: no existen saltos de linea o espaciados entre constantes, atributos y propiedades.*

*Métodos*

* Son servicios y comportamientos que ofrece la clase.
* Nombrar los métodos de la forma más descriptiva possible.
* El nombre del metodo debe estar en notacion Pascal-Casing.
* Los parametros del metodo utilizan notacion Camel-Casing.
* Cualquier método debería tener solo un objetivo.
* Entre metodos de la clase (incluyendo constructores) existen solo un salto de linea de espaciado.

*Palabra reservada “this”*

* Utilizar la palabra reservada ‘this’ al asignar propiedades o llamar métodos de la clase o de la clase padre.
* No utilizar con atributos de la clase, ya que estos son identificados con la barra-baja como prefijo.

public void MethodA()

{

//Some code

PropiedadA = 2; (BAD!)

}

public void MethodA()

{

//Some code

this.PropiedadA = 2; (GOOD!)

}

public void MethodA()

{

//Some code

this.\_\_atributoA = 2; (BAD!)

}

public void MethodA()

{

//Some code

\_\_atributoA = 2; (GOOD!)

}

*Mover código a la izquierda*

* Utilizar lógica negada para evitar la ejecución de código condicional

Public void SomeMethod(object varA)

{

if(varA!=null) (BAD!)

{

//Do Something

}

}

Public void SomeMethod(object varA)

{

if(varA==null)

return; (GOOD!)

//Do Something

}

*Espaciado de estructuras de control*

* Colocar una línea de espaciado antes y despues de estructuras de control como ser: for, if-else, while, foreach, etc.
* Solo debe existir espaciado si es que existe código previo o posterior a la estructura de control.

(BAD!)

public void SomeMethod(object varA)

{

//Some Previous code

for(int i = 0; i < 12 ; i++)

{

//Do something

}

//Some after code

}

(GOOD!)

public void SomeMethod(object varA)

{

//Some Previous code

for(int i = 0; i < 12 ; i++)

{

//Do something

}

//Some after code

}

*Estructuras de control con una sola línea*

* Cuando una estructura de control tiene solo una línea de contenido no se deben colocar llaves.
* Solo se aplica si el contenido es UNA LINEA (no si es otra estructura de control)

(BAD!)

public void SomeMethod(object varA)

{

if([condition])

{

this.PropiedadA = varA;

}

}

(GOOD!)

public void SomeMethod(object varA)

{

if([condition])

this.PropiedadA = varA;

}

(BAD!)

public void SomeMethod(object varA)

{

if([condition])

for(int i = 0; i < 12 ; i++)

this.ProcessData(i);

}

(GOOD!)

public void SomeMethod(object varA)

{

if([condition])

{

for(int i = 0; i < 12 ; i++)

this.ProcessData(i);

}

}

*Asignacion condicional*

* Cuando la asignación de una variable depende de alguna condición reducirlo a una sola línea.

(BAD!)

public void SomeMethod()

{

if([condition])

this.PropiedadA = 0;

else

this.PropiedadA = 1;

}

(GOOD!)

public void SomeMethod()

{

this.PropiedadA = ([condition])? 0:1;

}

*Los event handlers deben tener contenido mínimo*

* Los métodos que son utilizados como event handlers deben tener poco código de contenido.
* En el caso de que requiera bastante código se debe llevar la lógica requerida a un método auxiliar

Suponiendo que agregamos el eventhandler de la siguiente forma.

public void AddHandlers()

{

ListBox1.SelectionChanged += new EventHandler(ListBox1\_SelectionChange);

}

(BAD!)

private void ListBox1\_SelectionChange(object sender, EventArgs e)

{

//Demasiado codigo (Más de 5 líneas)

}

(GOOD!)

private void ListBox1\_SelectionChange(object sender, EventArgs e)

{

this.OnListBoxSelectionChanged(e);

}

private void OnListBoxSelectionChanged(EventArgs e)

{

//Todo el código que se quiera y requiera.

}

*Remover los EventHandlers que se hayan agregado*

* La mayoria de los eventhandler que se haya agregado debe ser removido en algun momento para evitar “Memory Leaks”
* Siempre tener en cuenta si es posible que el EventHandler cause un memory leak.

Suponiendo que agregamos el eventhandler de la siguiente forma.

public void AddHandlers()

{

ListBox1.SelectionChanged += new EventHandler(ListBox1\_SelectionChange);

}

(GOOD!)

public void RemoveHandlers()

{

ListBox1.SelectionChanged -= new EventHandler(ListBox1\_SelectionChange);

//or

ListBox1.SelectionChanged -= ListBox1\_SelectionChange;

}

*Separar por bloques el cuerpo de un método*

* Separar por bloques “lógicos” el cuerpo de un método para aumentar su legibilidad.
* Los bloques se separan por solo 1 línea de espaciado
* La lógica para separar los bloques puede variar (por etapas de ejecución, por inicialización de variables y procesamiento, etc).

(BAD!)

public void Method()

{

objectA a = new objectA();

objectB b = new objectB(a);

objectC c = new objectC(a,b);

c.BeginSync();

c.ProcessData();

c.EndSync();

b.SaveState();

a.Close();

a.Dispose();

}

(GOOD!)

public void Method()

{

objectA a = new objectA();

objectB b = new objectB(a);

objectC c = new objectC(a,b);

c.BeginSync();

c.ProcessData();

c.EndSync();

b.SaveState();

a.Close();

a.Dispose();

}

*Separar en metodos o clases auxiliares*

* Aquellos métodos cuyo contenido sea extenso (más de 25 lineas) debe ser separado.
* Separar con motivos de refactorización
* Dividir para mantener la esencia del método y aumentar legibilidad.

(BAD!)

public string Method(string data, string parameters)

{

objectA a = new objectA();

objectB b = new object();

//Procesamiento EXTENSO de las variables a, b y los parámetros del método

return result;

}

(GOOD!)

public string Method(string data, string parameters)

{

objectA a = new objectA();

objectB b = new object();

return this.ProcessData(data, parameters, a, b);

return result;

}

private string ProcessData(string data, string parameters, objectA a, objectB b)

{

//Procesamiento EXTENSO de las variables a, b y los parámetros del método

}

*No utilizar strings para identificar estados*

* Los strings son valores que pueden cambiar fácilmente, asi que utilizarlos para identificar estados, etapas (cualquier tipo de valor que tenga un dominio bastante limitado) es incorrecto y riesgoso.
* Utilizar en su lugar enums o constantes.

(BAD!)

public string ProcessData(string data, string state)

{

switch(state)

{

case “Open”:

//Do something.

break;

case “Close”:

//Do something

break;

case “Error”:

//Do something;

break;

}

}

(GOOD! BUT COULD BE BETTER)

public const string OPEN\_STATE = “Open”;

public const string CLOSE\_STATE = “Close”;

public const string ERROR\_STATE = “Error”;

public string ProcessData(string data, string state)

{

switch(state)

{

case OPEN\_STATE:

//Do something.

break;

case CLOSE\_STATE:

//Do something

break;

case ERROR\_STATE:

//Do something;

break;

}

}

Nota: aún si se utilizan constantes existe el problema de que se puede enviar cualquier valor en el parámetro ‘state’. También puede causar confusion al momento de utilizar porque no se sabe que se debe enviar en ese parámetro.

(GOOD!)

public enum State

{

Open,

Close,

Error

}

public string ProcessData(string data, State state)

{

switch(state)

{

case State.Open:

//Do something.

break;

case State.Close:

//Do something

break;

case State.Error:

//Do something;

break;

}

}

Consejos

* El nombrado adecuado de clases, metodos, variables (locales y de la clase) ayuda a la legibilidad.
* Las clases deben siempre tener SOLO UNA RESPONSABILIDAD
* Identificar correctamente cuando algún aspecto de una clase debe ser un atributo, propiedad o comportamiento (método).
* Siempre tomar en cuenta el tener una correcta encapsulación.
* Si se llegan a tener clases con bastante código repetitivo y/o comportamientos similares analizar la opción de utilizar herencia para reunir todos los aspectos comunes.
* No explotar los servicios de una clase, si es posible agregar servicios más adecuados y específicos. (EJM DE INVENTARIO: Si una categoria tiene un servicio para retornar todos sus productos, no explotar este servicio para contar el numero de productos total, obtener los productos que cumplan con algun criterio. La solución correcta es que la categoria pueda ofrecer esos servicios).
* No comentar excesivamente el código. “Código bien hecho no requiere de comentarios”.
* No utilizar el try-catch indiscriminadamente, dejar que ocurran errores en la aplicación para poder identificarlos.
* Siempre lanzar excepciones especializadas y NO del tipo Exception (muy general).
* En el catch siempre manejar excepciones puntuales y especializadas y NO del tipo Exception.

“Cualquiera escribe código que una máquina entiende, el truco es escribir código que un ser humano pueda entender”