MISA Coding Convention

Mục tiêu

- Tạo ra code thống nhất, giúp lập trình viên tập trung vào code thay vì định dạng code
- Giúp lập trình viên hiểu code nhanh hơn khi đã quen với style
- Đơn giản hóa việc thay đổi, bảo trì code

Quy tắc đặt tên

Quy tắc chung

Lựa chọn từ

- Chỉ sử dụng tiếng Anh để đặt tên, không dùng tiếng Việt để đặt tên
- Đặt tên có thể dài, tránh đặt tên viết tắt. VD: thuộc tính `CanScrollHorizontally` thay vì `ScrollableX`
- Không dùng các kí tự gạch ngang, gạch dưới hoặc không phải kí tự (-, _, !, ~...) để đặt tên (trừ trường hợp đặt tên cho internal, private field ở bên dưới)
- Không dùng kí pháp Hungarian để đặt tên vì sẽ làm cho code rất khó đọc. VD:

```
//Code như này khó đọc và không cần thiết vì Visual Studio đã hỗ trợ rất tốt
int intAccountNumber;
bool bReadLine;
string strName;

//Cách viết mới
int AccountNumber;
bool HasReadLine;
string Name;
```

Đặt tên cho phiên bản mới của API đã tồn tại

- Dùng tên giống API cũ khi tạo phiên bản API mới để nêu bật mối quan hệ giữa chúng. VD: `AppDomain` và `AppDomainSetup`
- Thêm suffix chứ không thêm prefix vào các phiên bản API mới. VD: API cũ `ReaderWriterLock`, API mới `ReaderWriterLockSlim`
- Dùng suffix số cho các phiên bản API mới, áp dụng cho các tên chuẩn không có suffix nào phù hợp. VD API cũ: `X509Certificate`, API mới: `X509Certificate2`

Quy tắc viết hoa thường

- Dùng **PascalCasing**: viết hoa chữ cái đầu tiên của mỗi từ. VD: `CustomerName`, `CloseDialog`, `HtmlTag`, `IOStream`, `TextColor...` cho tên namespace, class, member (Constant, Field, Method, Property, Event, Constructor, Type...)
- Dùng **camelCasing**: viết hoa chữ cái đầu tiên của mỗi từ trừ từ đầu tiên. VD: `customerName`, `closeDialog`, `ioStream`, `htmlTag...` cho tên các tham số của 1 hàm

Loại	Quy tắc	Ví dụ
Namespace	Pascal	namespace System.Security {...}
Class	Pascal	public class StreamReader {...}
Interface	Pascal	public interface IEnumerable {..}
Method	Pascal	public virtual string ToString();
Property	Pascal	public int Length { get; }
Event	Pascal	public event EvenHandler Exited;
Field	Pascal	public static readonly Timespan InfiniteTimeout;
Enum	Pascal	Append
Parameter	Camel	public static int ToInt32(string value);
Variable (inline)	Camel	var manyPhrases = new StringBuilder();

Quy tắc đặt tên Solution, Project, Namespace, Assembly

- Đặt tên Solution, Project, Assembly, Namespace theo mẫu quy định dưới đây
- Dùng **PascalCase**, bắt đầu bằng MISA, phân cách các thành phần của Namespace bởi dấu `.`
- Không sử dụng tên Namespace trùng với tên Class

Đối tượng	Cách đặt tên	Ví dụ
Tên solution	MISA.Tên sản phẩm.sln	MISA.SME2017.sln
Tên Assembly	MISA.(tên sản phẩm).(tên tắt project).dll/exe	MISA.SME.DL.dll/MISA.SME2017.exe
Tên Namespace	MISA.(tên sản phẩm).(tên tắt project)	MISA.SME.DL

Quy tắc đặt tên Class, Interface, Struct

- Đặt tên class, struct dùng danh từ, **PascalCase**, không có prefix để phân biệt với method dùng động từ.

```
public class SAInvoice
{
    //...
}
```

- Các class kiểu Collection/Exception/Attribute thì thêm các từ này tương ứng vào cuối tên class. VD:

```
public class WidgetCollection
public class InvalidTransactionException
public class JsonIgnoreAttribute
```

- Đặt tên của class con nên kết thúc bằng tên của class cha cho dễ đọc và dễ thể hiện mối quan hệ giữa 2 class. VD: `ReadOnlyCollection` và `Collection`, `ArgumentOutOfRangeException` và `Exception`, `FileStream` và `Stream` ...
- Đặt tên interface dùng tính từ, đôi khi có thể dùng danh từ, prefix với từ `I`. VD: `IComparable`, `IFormattable`, `ICustomAttributeProvider`, `IList`
- Đặt tên tham số kiểu Generic prefix bởi `T`. VD:

```
Nullable<T>
ISessionChannel<TSession>
IDictionary<TKey, TValue>
```

- Đặt tên kiểu enum dùng **PascalCase**, không suffix tên kiểu và không prefix tên các giá trị

```
public enum ImageMode {
    Bitmap = 0,
    Grayscale = 1,
    Indexed = 2,
    Rgb = 3,
}
```

Quy tắc đặt tên Method, Property, Event, Fields

Method

- Đặt tên method dùng động từ và kiểu **PascalCase**, mô tả công việc mà method này thực hiện (**trừ trường hợp là EvenHandler của UI control**). VD:

```
public class String {
    public int CompareTo(...);
    public string[] Split(...);
    public string Trim();
}

//EventHandler của UI Control
private void btnOK_Click(object sender, EventArgs e)
```

Property (Resource áp dụng tương tự)

- Đặt tên property dùng danh từ hoặc tính từ và kiểu **PascalCase**. VD:

```
public int Length { get; }
```

- Với các property kiểu collection đặt tên thêm `s` để mô tả số nhiều thay vì List hay Collection. VD:

```
//Đúng
public ItemCollection Items { get; }

//Sai
public ItemCollectionItemCollection { get; }
```

- Với các property kiểu Boolean đặt tên **dùng các từ khẳng định** hoặc có thể prefix bởi các từ **Is, Can, Has** tùy từng trường hợp cụ thể. VD: `CanSeek` thay vì `CantSeek`

Event

- Đặt tên event dùng động từ mô tả hành động đang xảy ra hoặc đã xảy ra. VD: `Clicked` , `Closing` , `Painting` , `DroppedDown` ...
- Đặt tên các event handler với suffix là `EventHandler` , 2 tham số là `sender` kiểu `object` , `e` kiểu `<EventName>EventArgs` . VD:

```
public delegate void ClickedEventHander(object sender, ClickedEventArgs e);
```

Field

- Đặt tên field (static, constant field) dùng danh từ hoặc tính từ, kiểu **PascalCase**, không prefix. VD:

```
public static readonly string Empty = "";
public const Min = 0;
private const String VersionName = "Version";
```

- Đặt tên field (internal, private field) dùng danh từ hoặc tính từ, kiểu **camelCase**, có prefix _ . VD:

```
private int _freeCount;
private int[] _buckets;
```

Variable (inline)

- Đặt tên các biến cục bộ (inline variable) dùng kiểu **camelCase**. VD:

```
var inputInt = Console.ReadLine();
var manyPhrases = new StringBuilder();
string displayName = nameList[n].LastName + ", " + nameList[n].FirstName;
```

Quy tắc đặt tên Parameter

- Đặt tên parameter dùng **camelCase** mô tả ý nghĩa của parameter thay vì kiểu của parameter. VD:

```
public string Remove(int startIndex, int count)
{
    //...
}
```

Quy tắc đặt tên UI Control

- Đặt tên UI Control **prefix 3 kí tự** + tên Control kiểu **PascalCase**

Data type/Control	Prefix	Example
User Control	ctl	ctlCurrent
Form	frm	frmLogin
Label	lbl	lblHelpMessage
Textbox	txt	txtLastName
Button	btn	btnClose
Checkbox	chk	chkReadOnly
Combobox	cbo	cboProvince
Drop-down	drp	drpAccountNumber
Panel	pnl	pnlGroup
GroupBox	grp	grpNumberFormat
Dialog	dlg	dlgFileOpen
Data grid	grd	grdTitles
Date time picker	dte	dtePublished
StatusBar	stb	stbDateTime
TabControl/TabStripControl	tab	tabOption, tabGeneralInfo
Toolbar	tbr	tbrActions
TreeView	trv	trvOrganization
List box	lst	lstPolicyCodes
ListView	lsv	lsvHeadings
Option button	opt	optGender
ImageList	img	imgAllIcons
Timer	tmr	tmrAlarm
ProgressBar	prg	prgLoadFile
Picture box	pic	picVGA
RichTextBox	rtb	rtbReport
Splitter	spl	splMain
SplitContainer	spc	spcMain
NumericEditor	num	numTaxRate

Data type/Control	Prefix	Example
CurrencyEditor	cur	curDiscountAmount
ListBar	lbr	lbrStockList
ExplorerBar	exp	expModule
DropDownButton	drpb	drpbJobPhase
ScrollBar	scb	scbMyScrollBar
Popup Menu	pop	popFile, popSystem
Menu Item/Button	mnu	mnuFileLogin, mnuFileExit, mnuSystemOption

Quy tắc định dạng code

- Sử dụng thiết lập mặc định của Code Editor Visual Studio, ngoặc nhọn được sắp theo hàng dọc. VD:

```
class Program
{
    static void Main(string[] args)
    {
    }
}
```

- Chỉ viết một câu lệnh, 1 khai báo trên mỗi dòng cho code trực quan, dễ nhìn

```
//Đúng
private Customer MasterObject, DetailObject;
private List<Customer> Customers;

//Sai
private Customer MasterObject, DetailObject; private List<Customer> Customers;
```

- Thêm ít nhất 1 dòng trắng giữa các **định nghĩa phương thức và định nghĩa thuộc tính** để cho code dễ nhìn

```
//Đúng
public int SortOrder { get; set; }

public bool Inactive { get; set; }

public void SetLocalization() {...}

public void ShowResult() {...}

//Sai
public int SortOrder { get; set; }
public bool Inactive { get; set; }

public void SetLocalization() {...}
public void ShowResult() {...}
```

- Sử dụng dấu ngoặc đơn để tạo mệnh đề trong một biểu thức:

```
if ((val1 > val2) && (val1 > val3))
{
  //code
}
```

Quy tắc bẫy lỗi

- Bắt buộc bẫy lỗi (sử dụng **try-catch**) trong tất cả các Event của Form/Control trên Form, code multi-thread

```
private void frmSAInvoice_KeyDown(object sender, KeyEventArgs e)
{
    try
    {
        this.uctDetail.MISASAInvoiceDetail_KeyDown(sender, e);
    }
    catch (Exception ex)
    {
        MISAMessageBox.ShowExceptionMessage(ex);
    }
}

Task.Factory.StartNew(() =>
{
    try
    {
        ...
    }
    catch (Exception ex)
    {
        CommonFunctionLogging.ErrorLogging(ex.ToString());
    }
});
```

- Không dùng **Exception** để điều khiển luồng logic ứng dụng bởi vì **throw exception là một câu lệnh rất tốn kém tài nguyên**

```
//Sai
//...search for Product
if(dr.Read() == 0) // no record found, ask to create
{
    //this is an example of throwing an unnecessary exception
    throw( new Exception("User Account Not found"));
}

//Đúng
if(dr.Read() == 0)
{
     return false;
}
```

- Nghiêm cấm sử dụng cú pháp **try-catch** để che dấu lỗi (không xử lý gì sau từ khóa catch), **bắt buộc phải xử lý lỗi, ít nhất cũng phải ghi log lại**

```
//Sai
private void frmMain_load(object sender, EventArgs e)
{
    try
    {
        ...
        SetColorStyle();
    }
    catch(Exception ex)
    {
    }
}

//Đúng
private void frmMain_load(object sender, EventArgs e)
{
    try
    {
        ...
        SetColorStyle();
    }
    catch(Exception ex)
    {
        MISAMessageBox.ShowExceptionMessage(ex);
    }
}
```

- Dùng **try-catch** cho phần lớn trường hợp bẫy lỗi, ngoài ra có thể dùng **using**. Nếu khối lệnh **try-finally** mà code trong finally chỉ là phương thức **Dispose** thì có thể dùng **using** thay thế.

```
//This try-finally statement only calls Dispose in the finally block
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
      (IDisposable)font1).Dispose();
    }
}

//You can do the same thing with a using statement
using(Font font2 = new Font("Arial", 10.0f))
{
    byte charset = new font2.GdiCharSet;
}
```

Quy tắc sử dụng Enumeration

- Khi khai báo các Enum thể hiện giá trị trong Database hoặc các Enum giao tiếp giữa nhiều nền tảng khác nhau (Web, Desktop, Mobile) thì bắt buộc phải gán giá trị xác định cho biến

```
//Giá trị tương ứng với id trong bảng EquipmentChangeCategory
public enum EquipmentChangeCategory
{
    Increasing = 1,
    Revalution = 2,
    TransferFrom = 3,
    ChangeInformation = 4,
    Depreciation = 5,
    Reducing = 6
}

//Enum giao tiếp giữa client và web service
public enum EnumServiceErrorType
{
    Unknown = -1,
    None = 0,
    APIParameterNullOrInvalid = 1,
    SystemIsMaintain = 2,
    TokenInvalid = 3,
    PushDataToQueueFail = 4,
    Duplicate = 5
}
```

- Khuyến cáo: Các dev khi gán giá trị cho biến trong Enum cần kiểm tra xem giá trị đó đã tồn tại trong Enum chưa để tránh trường hợp gán trùng giá trị các biến trong Enum

```
//Giá trị tương ứng với id trong bảng EquipmentChangeCategory
public enum EquipmentChangeCategory
{
    ...
    ChangeInformation = 4,
    Depreciation = 4,
    ...
}
```

- Với các enum không thể hiện giá trị trong Database, không giao tiếp giữa nhiều nền tảng khác nhau thì không cần phải khai báo giá trị

```
public enum FormsProtectionEnum
{
    All,
    None,
    Encryption,
    Validation
}
```

Quy tắc phân nhóm code

- Phải sử dụng **Region** phân nhóm code để tiện cho việc sửa đổi, bảo trì
- Phân nhóm code theo cấu trúc như sau: (theo thứ tự bắt buộc, nhưng không bắt buộc có đủ tất cả các region)

```
public class Account
{
    #region Declaration

    public static string BankName;
    public static decimal Reserves;

    #endregion

    #region Property

    public string Number {get; set;}
    public DateTime DateOpened {get; set;}

    #endregion

    #region Constructor

    public Account()
    {
        // ...
    }

    #region

    #region Method

    public decimal GetAccountBalance(string accountNo){...}

    #endregion

    #region Event

    #endregion
}
```

- Tùy theo yêu cầu của các form, class lập trình viên có thể chia nhỏ các **Region** chính trên thành các **sub-region**. VD: **region** Method có thể chứa các **region** con sau:

```
Method
    Public
    //Trường hợp là base form/class
    Overridable
    //Trường hợp là derive form/class
    Override
    Private
    Other
```

- Trường hợp form hoặc class có sử dụng các component độc lập (**Security, Document, MassEmail** ...) thì phải tạo các **Region** riêng cho từng component, chứa toàn bộ code liên quan đến việc tương tác với các component đó

Quy tắc comment code

- Sử dụng **tiếng Việt có dấu (Unicode)** để viết comment.
- Comment cho Module, Class. Mỗi Module, Class cần có mô tả ngắn về mục đích của Module hay Class đó, nội dung gồm:
  - Mục đích Module hay Class thực hiện những công việc gì + Ngữ cảnh sử dụng khi nào
  - Những biến/hàm quan trọng (không bắt buộc): Liệt kê tên các biến và hàm quan trọng trong Module/Class
  - Người tạo/sửa: Người tạo/sửa Module hay Class (Lưu ý: Không viết trong thẻ để Tool tạo help không sinh ra)

```
/// <summary>
/// Các API thực hiện công việc thao tác với Database sử dụng ở tầng Data Access gồm:
/// <list type="bullet">
/// <item>Các hàm thực hiện thêm/sửa/xóa bản ghi trong Database sử dụng StoreProcedure</item>
/// <item>Các hàm thực hiện load dữ liệu phân trang hoặc tất cả dữ liệu sử dụng StoreProcedure. Dữ liệu trả về dạng List object hoặc DataTable/DataSet</item>
/// <item>Lấy tổng số bản ghi từ một Table/View để phục vụ việc load phân trang dữ liệu</item>
/// </list>
/// </summary>
/// Created by nttung - 2015
public class DALUtil
{
    ...
}
```

- Comment cho Sub/Function và Event. Nội dung cần bao gồm:
  - Mục đích Sub/Function/Event thực hiện những công việc gì + Ngữ cảnh sử dụng khi nào, ví dụ
  - Mô tả rõ các tham số đầu vào
  - Mô tả rõ kết quả đầu ra của Function
  - Người tạo/sửa: Người tạo/sửa Sub/Function/Event (Lưu ý: Không viết trong thẻ để Tool tạo help không sinh ra)

```
/// <summary>
///  Hàm show form bàn phím số
/// <para/>Sử dụng ở những control có nhu cầu nhập số trên máy POS
/// </summary>
/// <param name="parentForm">Form cha</param>
/// <param name="currentValue">Giá trị hiện tại</param>
/// <param name="maxValue">Giá trị lớn nhất có thể nhập vào</param>
/// <param name="minValue">Giá trị nhỏ nhất có thể nhập vào</param>
/// <param name="formText">Tiêu đề của form bàn phím số khi show lên</param>
/// <returns>Giá trị số nhập vào trên form bàn phím số</returns>
/// <remarks></remarks>
/// Created by nnanh - 12/2015
/// Modified by nttung1 - 1/2016: Sửa lỗi/bổ sung ...
public static decimal ShowFormInputNumber(Form parentForm, decimal currentValue,
string formText, decimal minValue, decimal maxValue)
```

```
/// <summary>
/// Raise sự kiện thêm Detail mới để Form List xử lý
/// <para/>Khi muốn đóng Tab Detail và thêm mới 1 Detail, đối tượng List bắt sự
kiện này để đóng Tab Detail hiện tại lại và mở ra 1 Tab Detail mới.
/// Ví dụ: Khi bấm nút Cất và Thêm trên Detail
/// </summary>
/// <param name="master">Đối tượng Master hiện tại</param>
/// <remarks></remarks>
/// Created by nnanh - 12/2015
event CloseAndAddDetailVoucher(Master master)
```

- Comment cho các phần code phức tạp hoặc khó hiểu

```
app.Version = Info.Version;
app.Token = token; //Sau sẽ lấy về và lưu trữ trong DBOption
if (isCompress)
{
    app.Data = Utility.Compress(data);
}
else
{
    app.Data = data;
}

var jsonUpload = syncService.SyncUploadData(CommonFunctionFE.SerializeObject(app));
//Nếu mà có nén dữ liệu thì thực hiện giải nén chuỗi Result
if (isCompress)
{
    jsonUpload = Utility.Decompress(jsonUpload);
}
```

Quy tắc thiết kế dataset

- Tên của **Dataset** khi visual design đặt như sau: "**Dataset**" + Mục đích. VD: DatasetDictionary, DatasetCAPayment, ...

- Số lượng **DataTable** trong một **Dataset**: quy định **từ 10 trở xuống**. Mỗi Dataset chỉ được phép có **tối đa 10 DataTable**. Khi thiết kế phải tách Dataset để đảm bảo nguyên tắc này. Nếu một Dataset có nhiều DataTable thì tốc độ load/save dữ liệu sẽ bị ảnh hưởng nghiêm trọng.

Coding Guidelines

Khởi tạo đối tượng

- Nên khởi tạo đối tượng dùng var cho ngắn gọn, ngoại trừ kiểu nguyên thủy (int, string, double...)

```
var instance1 = new ExampleClass();
//Thay vì khởi tạo như bên dưới
ExampleClass instance2 = new ExampleClass();

//Ngoại trừ
int index = 100;
bool isCompleted;
```

- Sử dụng object initializer để đơn giản hóa việc khởi tạo đối tượng

```
//Object initializer
var instance3 = new ExampleClass {Name = "Desktop", ID = 37314, Location = "Ha Noi", Age = 2.3};

//Cách khởi tạo thông thường
var instance4 = new ExampleClass();
instance4.Name  = "Desktop";
instance4.ID  = 37414;
instance4.Location  = "Ha Noi";
instance4.Age  = 2.3;
```

Xử lý sự kiện

- Tránh việc **AddHandler** mà không có **RemoveHandler** tường mình vì có thể dẫn đến **MemoryLeak** khi đối tượng được **AddHandler** sống lâu hơn đối tượng chứa Handler

```
//Sai
class Form2
{
    Form1 _form1;
    public Form2(Form1 form1)
    {
        _form1 = form1;
        _form1.Load += Form1_Load;
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        ...
    }
}

//Đúng
class Form2
{
    Form1 _form1;
    public Form2(Form1 form1)
    {
        _form1 = form1;
        _form1.Load += Form1_Load;
        this.FormClosed += Form1_FormClosed;
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        ...
    }

    private void Form1_FormClosed(object sender, FormClosedEventArgs e)
    {
        _form1.Load -= Form1_Load;
    }
}
```

- Nếu định nghĩa event handler mà không cần remove thì dùng lambda expression cho đơn giản

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