VOTECAL
PROGRAMMING STANDARDS

February 26, 2010

CATALYST
CONSULTING GROUP, INC.

211 West Wacker Drive, Suite 450
Chicago, IL 60606
Phone: 312.629.0750
Fax: 312.629.0751
www.catconsult.com

Table of Contents

Section 1. - Identifier Naming

1.1. Descriptive Identifiers

1.1.1. Use descriptive names for all identifiers such that their name, combined with context, are enough to convey meaning without the need for separate comments.

1.1.2. In general, use names that describe a parameter's meaning rather than names that describe a parameter's type.

1.1.3. Use a noun or noun phrase to name classes, interfaces, and properties.

- Example: the interface name IComponent uses a descriptive noun.
- Example: The interface name ICustomAttributeProvider uses a noun phrase.

1.1.4. It is also acceptable to name interfaces with adjectives that describe behavior.

- Example: The interface name IPersistable uses an adjective.

1.1.5. Use verbs or verb phrases to name methods and events.

- Example: Correctly named event names based on verbs include Clicked, Painting, and DroppedDown.

1.1.6. Use a gerund (the "ing" form of a verb) to create an event name that expresses the concept of pre-event, and a past-tense verb to represent post-event.

- Example, a Close event that can be canceled should have a Closing event and a Closed event.
- Do not use the BeforeXxx/AfterXxx naming pattern.

1.1.7. Use abbreviations sparingly.

1.1.8. Do not use abbreviations or contractions as parts of identifier names. For example, use GetWindow instead of GetWin.

1.1.9. Where appropriate, use well-known acronyms to replace lengthy phrase names.

- Example: Use UI for User Interface and OLAP for On-line Analytical Processing.

1.1.10. Where appropriate, use a compound word to name a derived class. The second part of the derived class's name should be the name of the base class.

- Example: ApplicationException is an appropriate name for a class derived from a base class named Exception, because ApplicationException is a kind of Exception.
- Use reasonable judgment when applying this rule. For example, Button is an appropriate name for a class derived from Control. Although a button is a kind of control, making Control a part of the class name would lengthen the name unnecessarily.

1.1.11. Use a singular name for most Enum types, but use a plural name for Enum types that are bit fields.

1.1.12. Namespace names should follow the standard pattern

```
<company name>.<product name>.<top level module>.<bottom level module>
```

1.1.13. File names should match with class names. i.e. For the class HelloWorld, the file name should be helloworld.cs or helloworld.vb. Variables used for iterations in loops are an allowed exception to the rule that meaningful descriptive words must be used to name variables.

```
for ( int i = 0; i < count; i++ )
```

```
{
        ...
}
```

1.2. Case insensitivity

1.2.1. Never use identifier names that require case sensitivity. Names of Methods, Properties, Parameters, and Types must all differ by more than capitalization. In the following example, calculate() and Calculate() are inappropriate method names because they differ only by case.

```
void calculate()
void Calculate()
```

1.3. Prefixes and Suffixes

1.3.1. Do not use Hungarian notation. Hungarian notation is the name given to the outdated practice of prefixing a type abbreviation onto variables names.

- Example: use Char MiddleInitial instead of Char cMiddleInitial

1.3.2. Do not use a type prefix, such as C for class names or m_ for member variables.

- Example, use the class name FileStream rather than CFileStream.

1.3.3. Do prefix interface names with the letter I, to indicate that the type is an interface.

1.3.4. Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter I prefix on the interface name.

- Example:

```
[C#]
public interface IComponent
{
        // Implementation code goes here.
}
public class Component: IComponent
{
        // Implementation code goes here.
}
```

1.3.5. Occasionally, it is necessary to provide a class name that begins with the letter I, even though the class is not an interface. This is appropriate as long as I is the first letter of an entire word that is a part of the class name. For example, the class name IdentityStore is appropriate.

1.3.6. Use EventHandler as a suffix on event handler names.

1.3.7. Name an event argument class with the EventArgs suffix.

1.4. Special Rules relating to events:

1.4.1. For events, specify two parameters named sender and e. The sender parameter represents the object that raised the event. The sender parameter is always of type object, even if it is possible to use a more specific type. The state associated with the event is encapsulated in an instance of an event class named e. Use an appropriate and specific event class for the e parameter type.

1.4.2. Name an event argument class with the EventArgs suffix.

1.4.3. In general, you should provide a protected method called OnXxx on types with events that can be overridden in a derived class. This method should only have the event parameter e, because the sender is always the instance of the type.

1.4.4. The following example illustrates an event handler with an appropriate name and parameters.

```
[Visual Basic]
Public Delegate Sub MouseEventHandler(sender As Object, e As MouseEventArgs)

[C#]
public delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

1.5. Capitalization of Identifiers

Identifier	Case	Example / Notes
Class	Pascal	AppDomain
Enum type	Pascal	ErrorLevel
Enum values	Pascal	FatalError
Event	Pascal	ValueChange
Exception class	Pascal	WebException Note Always ends with the suffix Exception.
Read-only Static field	Pascal	RedValue
Interface	Pascal	IDisposable Note Always begins with the prefix I.
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
Property	Pascal	BackColor
Protected instance field	Camel	redValue Note Rarely used. A property is preferable to using a protected instance field.
Public instance field	Pascal	RedValue Note Rarely used. A property is preferable to using a public instance field.

Capitalization Rules

Pascal Case	The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. For example: BackColor
Camel Case	The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example: backColor
Uppercase	All letters in the identifier are capitalized. Conactinated words may be separated by an underscore for readability. This should almost never be used except for identifiers that consist of two letters. For example: System.IO ; System.Web.UI This is also seen for keywords in SQL statements.

Capitalization Definitions

1.6. Database Naming Conventions

1.6.1. Database table, view and field names should use Pascal case. Underscores should be avoided. The field name may also include a suffix that indicates a particular usage, such as Code or Flag.

1.6.2. Names for other database objects will be prefixed according to the following table:

Other	Data Type
FK_	Foreign Key
PK_	Primary Key
IDX_	Index
TRIGGER_	Trigger (Ex: TRIGGER_TableName_Insert, TRIGGER_TableName_Update, TRIGGER_TableName_Delete)
#tmp	Temporary Table

1.6.3. Stored Procedures

Name stored procedures with a designator indicating the action performed by the procedure, or with the object being affected followed by the action being performed.

1.6.4. Use parenthesis on all conditional expressions for readability as in the following example:

```
SELECT FirstName,
LastName
FROM  Student st JOIN Class cl ON (st.UserID = cl.UserID)
WHERE  (st.vc_Name = 'Joe')
  AND  (stvc_StatusTypeProperty = 'OK')
  AND  (stin_Status = 1)
  AND  (cl.ClassId = 5)
```

1.6.5. Put ANDs and ORs to the left of the condition so that it is more readable (as above)

1.6.6. Put each major SQL clause on a separate line to make them easier to read. For simple queries put each field on its own line to make it easier to read (as above)

1.6.7. Uppercase all SQL reserved words to make them easy to distinguish. (as above)

1.6.8. Avoid use of dynamic SQL whenever possible to increase performance and security. Exceptions may exist where maintenance is greatly facilitated and performance will never be affected

1.6.9. Every procedure must anticipate failure points, generally after each query whether read-only or data modifying, and produce meaningful error messages and return codes to the calling routine. Meaningful messages are dependent on the audience receiving the message, but generally include the step in the procedure being performed and any keys to the data that have failed to produce desired results. In complex applications, more than one return code may need to be defined. Generally, a return code of zero indicates success and a negative number indicates some form of failure. If the need exists to represent various levels of errors or statuses, then develop a standard approach, document it in the calling and called code.

Section 2. - Programming Practices

2.1. Database Interaction

2.1.1. Choosing which method to execute a SQL statement

- When retrieving a single value use the ExecuteScalar method.
- When inserting, updating, or deleting data execute statements use the ExecuteNonQuery method.

2.1.2. DataReader versus DataSet

- The DataReader is a better choice for applications that require optimized read-only and forward-only access to data. The sooner the data is off loaded from the DataReader and the connection is closed the better the performance. When the number of concurrent users reaches the point where contentions for database connections occurs (default connection pool size is 100), the performance of the DataReader will start to degrade.
- The DataSet is a better choice for applications that will not off load the query result immediately, or there is extensive processing involved between data accesses.

2.1.3. Selecting Columns from Results

- Always be explicit in the columns to retrieve in a stored procedure and only retrieve the necessary columns. Never use SELECT *.
- Always refer to fields by their field name and not the ordinal number location in the record set because the ordinal number location is more likely to change than the name.

2.2. Choosing Between Server Controls and Client (HTML) Controls

2.2.1. The use of client-side controls will be minimized in favor of server-side controls.

2.3. Choosing Between Web User Controls and Web Custom Controls

2.3.1. If none of the existing ASP.NET server controls meet the specific requirements of your applications, you can create your own control that encapsulates the functionality required.

2.3.2. Web user control

- Easier to create because they can be created in the visual designer
- Limited visual design support for consumers
- A separate copy of the control is required in each application
- Cannot be added to the toolbox in Visual Studio
- Good for static layout

2.3.3. Web custom control

- Harder to create because they are authored solely in code
- Full visual design support for consumers
- Only a single copy of the control is required – located in the global assembly cache
- Can be added to the toolbox in Visual Studio
- Good for dynamic layout

2.4. State Variables

2.4.1. Declare page-level variables for each Application and State variable that you need. Retrieve the Application or Session state value in the Page_Load event procedure and save the page level variables back in the Page_Unload event procedure. Using Application and Session state variables incorrectly makes it is easy to introduce basic errors in your code that will not be obvious.

2.4.2. In C# it is necessary to test whether a state variable is null before invoking any of its methods, such as ToString(), or you will receive a runtime error if the variable does not contain a value.

2.4.3. Disable session state when it is not being used. Not all applications or pages require per-user session state, and it should be disabled for any that do not. If a page requires access to session variables, but will not modify them, set the EnableSessionState attribute in the page directive to ReadOnly.

2.5. Data Validation

2.5.1. Client-side validation should be used to ensure data is in the proper format. Examples include forcing a field to be filled in, ensuring data is in a particular format, and comparing two fields on the form. This will help decrease the server load and amount of network traffic by decreasing the number of trips to the server. It also allows the user to be immediately notified in the event of an error or miss entry versus having to wait for the data to be posted and returned.

2.5.2. Client-side validation logic that is common across pages should be placed into a JavaScript library (file with a .js extension) and can then be included in multiple pages. This increases maintainability by allowing the code to exist in a single source.

2.5.3. Business logic validation should not be placed in client-side validation unless the validation has been encrypted or hidden in some way, such as a custom control, so that a hacker cannot read it. It is possible for a user to access and save the validation code. It would then be possible to study the code to expose weaknesses or find out enough information to do harm to your application.

2.5.4. Data validation should be performed at both the client and the server. The server-side validation is required to avoid a security weakness exposed by client-side validation. It is possible for a user to access and save your source code, modify your validation scripts (or simply remove them), and submit the form to your server with inappropriate data. The only way to be absolutely sure that validation has been performed is to perform validation on the server as well.

2.5.5. Server-side validation should include the use the ServerVariables collection to detect how the user reached your page. By comparing it with the server and individual page from which you expected the submission, you can catch any malicious user's attempting to post to your form from another location

2.6. Constant Values

2.6.1. Do not hardcode numbers. Use constants instead.

2.6.2. Do not hardcode strings. Use resource files.

2.6.3. Use enum wherever required. Do not use numbers or strings to indicate discrete values.

2.7. Miscellaneous

2.7.1. Do not make the member variables public or protected. Keep them private and expose public/protected Properties.

2.7.2. Avoid using too many member variables. Declare local variables and pass them to methods instead of sharing member variables between methods. Otherwise, it will be difficult to track which method changed the value and when.

2.7.3. When displaying error messages, in addition to telling what is wrong, the message should also tell what should the user do to solve the problem. Instead of message like "Failed to update database", suggest what should the user do: "Failed to update database. Please make sure the login id and password are correct."

2.7.4. Show short and friendly message to the user. But log the actual error with all possible information. This will help a lot in diagnosing problems.

2.8. Local Persistent Data

2.8.1. Never hardcode a path or drive name in code. Use configuration files or get the application path programmatically and use relative path.

2.8.2. In the application start up, do some kind of "self check" and ensure all required files and dependencies are available in the expected locations. Check for database connection in start up, if required. Give a friendly message to the user in case of any problems.

2.8.3. If the required configuration file is not found, application should be able to create one with default values.

2.8.4. If a wrong value found in the configuration file, application should throw an error or give a message and also should tell the user what are the correct values.

2.8.5. Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."

2.9. Exception Handling

2.9.1. Never do a 'catch exception and do nothing'. If you hide an exception, you will never know if the exception happened or not.

2.9.2. Always catch only the specific exception, not generic exception.

2.9.3. No need to catch the general exception in all your methods. Leave it open and let the application crash. This will help you find most of the errors during development cycle.

2.9.4. You can have an application level (thread level) error handler where you can handle all general exceptions. In case of an 'unexpected general error', this error handler should catch the exception and should log the error in addition to giving a friendly message to the user before closing the application, or allowing the user to 'ignore and proceed'.

2.9.5. Do not write try-catch in all your methods. Use it only if there is a possibility that a a specific exception may occur. For example, if you are writing into a file, handle only FileIOException.

2.9.6. Do not write very large try-catch blocks. If required, write separate try-catch for each task you perform and enclose only the specific piece of code inside the try-catch. This will help you find which piece of code generated the exception and you can give specific error message to the user.

2.9.7. You may write your own custom exception classes, if required in your application. Do not derive your custom exceptions from the base class SystemException. Instead, inherit from ApplicationException.

2.10. Development and Production Environments

2.10.1. All compiler warnings should be addressed as if they were fatal compilation errors. Even though warnings do not stop compiling, they are often indicators of larger overall issues and should not be ignored. No compilation warnings should exist in production code.

2.10.2. To ensure warnings are treated the same as error messages; the “Treat Warnings as Errors” option should be enabled for each project.

2.10.3. The web site folder structure should match the namespace used in the development of the pages and controls. This makes it easier to locate the source code for a particular item.

2.10.4. Design the file structure of the web site along with the supporting namespace prior to doing actual development. It is easier to do this design up front than to go back through all of the content and change the namespace locations.

Section 3. - Commenting

3.1. General Guidelines

3.1.1. If the above rules for the naming of identifiers are followed well enough, then the reader should be able to understand the code's functionality with no comments at all.

3.1.2. Comments should describe the rationale for design choices.

3.1.3. Comment as you code, because most likely you will not have time to return and do it later.

3.1.4. Comments should describe what the code does and why it does it. We are not interested in how it does it because that is typically obvious by looking at the code.

3.1.5. All of the dead or old lines of code should be deleted and not just commented out. This increases the readability of the source code.

3.1.6. All sections of code within the body of a method that require an explanation are commented using // comments. Multi-line /* */ comments can then be used during debugging to easily comment out blocks of code without having other comments interfere.

3.1.7. All properties and methods should be documented with appropriate information about the method, its functionality, and history information.

3.2. Debug Asserts

3.2.1. Use Debug Assert statements instead of comments to instruct callers on input restrictions for functions.

- Correct:

```
Void CalculateAverage ( Int16 elementCount )
{
       Assert ( elementCount > 0 );
       // Do domething…
}
```

- Incorrect:

```
// elementCount must be > 0
Void CalculateAverage ( Int16 elementCount )
{
       // Do domething…
}
```

3.3. Optional XML based comments

3.3.1. Use XML documentation for publicly accessible properties and methods. This will allow for automatic documentation to be produced for all public properties and methods.

3.3.2. Todo: Insert instructions & examples

3.4. Database Stored Procedure Comments

3.4.1. Document all database stored procedures and functions with a leading comment block containing information about the procedure or function.

```
/*============================================================

DESCRIPTION:

Used to retrieve site dictionary entries by family id and display
indicator.


TESTING STRINGS:

      SqlCommandText="p_getSiteDictionaryByFamilyID @in_FamilyId=1"

      1.  EXEC p_getSiteDictionaryByFamilyID 1

      2.  EXEC p_getSiteDictionaryByFamilyID 1, '2002'


CHANGE CONTROL:

      Initials     Date         Description

      --------     ----------   --------------------------

      MAS          04-28-2002   Created

============================================================*/
```

Section 4. - Code Formatting

4.1. Source Layout

4.1.1. Layout allows the person reviewing the code to easily locate important items within the code. While this is not as high of a priority due to the capabilities built into many editors, it is still important to have consistency across all source code.

4.1.2. The layout of each source file should follow the following sequence

- Copyright Notice
- File Description
- References
- Namespace Declaration
- Constants and Enumerations
- Structures and Interfaces
- Class Definitions

4.1.3. The layout of the actual class should follow the following sequence

- Enumerated Types
- Constants
- Fields and properties
- Constructors
- Destructors
- Static Methods
- Public Methods
- Protected Methods
- Private Methods

4.1.4. Regions should be used to group the above blocks of code into collapsible sections that will make the code easier to read

4.2. Indentation and Spacing

4.2.1. Use TAB for indentation. Never use SPACES.

4.2.2. Comments should be in the same level as the code.

4.2.3. Curly braces ({}) should be in the same level as the code outside the braces.

4.2.4. Use one blank line to separate logical groups of code.

```
bool SayHello ( string name )
{
       string fullMessage = "Hello " + name;
       DateTime currentTime = DateTime.Now;

       string message = fullMessage + ", the time is : " +
       currentTime.ToShortTimeString();
```

```
        MessageBox.Show ( message );

        if ( ... )
        {
                // Do something
                // ...

                return false;
        }

        return true;
}
```

4.2.5. There should be a single blank line between each method inside the class.

4.2.6. The curly braces should be on a separate line and not in the same line as the preceeding conditional expression (if, for, while etc.)

- Correct:

```
if ( ... )
{
        // Do something
}
```

- Incorrect:

```
if ( ... )   {
        // Do something
}
```

4.2.7. Use a single space before and after each operator and brackets.

- Correct:

```
if ( showResult == true )
{
        for ( int i = 0; i < 10; i++ )
        {
                //
        }
}
```

- Incorrect:

```
if(showResult==true)
{
        for(int      i= 0;i<10;i++)
        {
                //
        }
}
```

4.3. Control Structures

4.3.1. A switch-case statement must always contain a default branch and break statement. This allows for a catchall processing routine to be written, in order to ensure that assumptions made in the switch-case statement are true.

4.3.2. Use parenthesis to clarify the order for operators in expressions. There are a number of common pitfalls that are related to the order of evaluation for operators.

4.3.3. Always enclose statements following if, if-else, and if else-if conditions with braces.

```
If ( Name.Equals(“John”) )
{
      // Processing here…
}
else if ( Name.Equals(“Jane”) )
{
      // Processing here…
}
```

4.3.4. Align open and close braces vertically where brace pairs align and indent the code contained in the braces.

```
for( int i = 0; i < MAX_NUMBER; i++)
{
      // Statements here
}
```