**CureMD Coding Standards and Best Practices**

**Author**

This document is prepared by the CureMD Architecture Team. Most of the information in this document is compiled from the coding standards and best practices published by experienced developers, architects in their articles. Also, we referred to the guidelines published by Microsoft and various other sources.

**Revision History**

Anyone who is editing this document is required to fill the revision history with name and timestamp so that users of this document can easily distinguish the updates.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No** | **Name** | **Description** | **Date** |
| 1 | Shahid Iqbal | Define Coding Standards and Best Practices | 05/15/2015 |
|  |  |  |  |

**Introduction**

This document helps developers to write “Good Code” not “Working Code”, because writing good code is an art and every developer needs to learn and practice it.

Following are the characteristics of good code.

* Reliable
* Maintainable
* Efficient

Most of the developers are inclined towards writing code for higher performance but compromising reliability and maintainability. But considering the long term ROI (Return on Investment); efficiency and performance comes below reliability and maintainability. Because if code is not reliable and maintainable, developers and company will be spending lot of time to identify issues, trying to understand code throughout the life of application.

**Purpose**

To develop reliable and maintainable applications, developers must follow the coding standards and best practices described in this document.

**Follow Coding Standards and Best Practices**

Distribute a copy of this document to all your team members and schedule a coding standards meeting. All members should come to the meeting to discuss the pros and cons of the points in the document. Discuss all points and all members must agree upon the standards you are going to follow. If there is anything that needs to be amended or changed; update in the document, print copies of it and paste it on all the workstations.

After starting the development, code review meetings must be scheduled to ensure that everyone is following the rules.

Following types of code reviews are recommended.

**Peer review** – another team member review the code to ensure that code follows the coding standards and meets the requirements. This level of review can include some unit testing also. Every file in the project must go through this process.

**Architect review** – the architect of the team must review the core modules of the project to ensure that they adhere to the design and there are no big mistakes that can affect the project in the long run.

**Group review** – randomly select one or more files and conduct a group review once in a week. Distribute a printed copy of the files to all team members around 30 minutes before the meeting. Let them read and come up with points for discussion. In the group review, go through every sections of the code and let every member give their suggestions on how could that piece of code can be written in a better way.

*Note: Don’t forget to appreciate the developer for the good work and also make sure he doesn’t get offended by the group attack.*

**Naming Conventions and Standards**

The terms Pascal Casing and Camel Casing are recommended throughout this document.

**Pascal Casing** –First character of all words are upper case and other characters are lower case.

**Example:** ProviderNote

**Camel Casing –** First character of all words, except the first word are upper case and other characters are lower case.

**Example:** providerNote

*Note: Don’t use Hungarian notation to name variables.*

**Example:** string strName; or int iCount;

* Project name should match with assembly or namespace.
* File name should match with class name. For example, for the class PatientDiagnosis the file name should be PatientDiagnosis.cs, and use Pascal Case for file names.
* When defining a namespace, use a product or company as the root and then module name.

**Example:** CureMD.EMR.Diagnosis

* Use Pascal Casing for classes.

public class PatientDiagnosis

{

…

}

* Use Pascal Casing for methods.

public void AddDiagnosis(DiagnosisView diagnosisView)

{

…

}

* Use Camel Casing for variables and method parameters.

int totalCount = 0;

public void AddDiagnosis(DiagnosisView diagnosisView)

{

…

}

* Use the prefix “I” with for interfaces.

**Example:** IDiagnosis

* Use meaningful, descriptive words to name variables. Don’t use abbreviations.

**Good:**

string name;

string address;

**Not Good:**

string nam;

string addr;

* Don’t use single character variable names like i, j or k etc. Use names like index, temp.

Exception in this case would be variable used for iterations in loops.

for(int i=0; i<10; i++)

{

…

}

* Don’t use underscore (\_) for local variables.
* All member variables must be prefixed with underscore (\_) so that they can be identified from other local variables.
* Don’t use variable names that resemble keywords.
* Use prefix Can, Is or Has with Boolean variables and properties.
* Avoid adding redundant or meaningless prefixes and suffixes to identifiers.

**Bad Example:**

public enum ColorEnum{...}

public class CPatient{…}

public struct VitalsStruct{…}

* Don’t use parent class name with property name.

**Example:** Patient.FirstName Not Patient.PatientFirstName

* Don’t use names that begin with a number.
* Use a noun or noun phrase for class name, and add appropriate class suffix when sub-classing another type when possible.

**Example:**

public class SegmentCollection : CollectionBase {…}

public class SegmentAttribute: Attribute {…}

private class ApplicationSettings {...}

* Use a verb or verb-object pair for method names.

**Example:**

public void Execute() {…}

public string GetAssemblyVersion(Assembly target) {…}

* Property name should represent the entity it returns. Never prefix property names with Get or Set.

**Example:**

public string FirstName

{

get{…}

set{…}

}

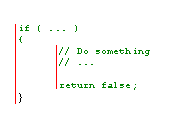
* Use appropriate prefix for the UI elements so that they can be identified from the rest of the variables.

|  |  |
| --- | --- |
| **Control** | **Prefix** |
| Label | lbl |
| TextBox | txt |
| DataGrid | dtg |
| Button | btn |
| ImageButton | imb |
| Hyperlink | hlk |
| DropDownList | ddl |
| ListBox | lst |
| DataList | dtl |
| Repeater | rep |
| Checkbox | chk |
| CheckBoxList | cbl |
| RadioButton | rdo |
| RadioButtonList | rbl |
| Image | img |
| Panel | pnl |
| PlaceHolder | phd |
| Table | tbl |
| Validators | val |

**Code Formatting and Commenting**

Consistent layout, format and organization are key to create maintainable code.

* Never declare more than 1 namespaces per file.
* Avoid putting multiple classes in a single file.
* Group internal class implementation by type in the following order.
  + Member variables
  + Constructors
  + Nested Enums, Structs and Classes
  + Properties
  + Methods
* Sequence declarations within type groups based upon access modifiers and visibility.
  + Public
  + Protected
  + Internal
  + Private
* Use Tab for indentation instead of Spaces, and Tab size must be 4.
* Curly braces ({}) should be in the same level as the code outside braces.



* Comments should be in the same level as the code (Use same level of indentation).
* Use one blank line to separate logical groups of code.
* There must be one blank line between each method inside the class.
* Use a single space before and after each operator and brackets.

**Good:**

if ( showResult == true )

{

…

}

**Not Good:**

if (showResult==true)

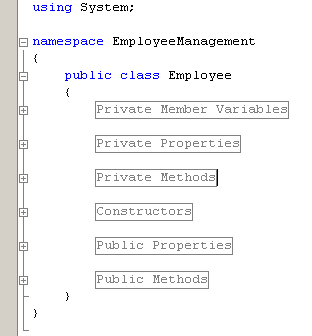
{

…

}

* Use #region to group related pieces of code together.

**Example:**



* All comments should be written in the same language, be grammatically correct, and contain appropriate punctuations.
* Place the comment on a separate line, not at the end of a line of code.
* Begin comment text with an uppercase letter.
* End comment text with a period.
* Insert one space between the comment delimiter (//) and the comment text.
* Use // or /// instead of /\*….\*/.
* Don’t use flowerbox comment blocks.

**Example:**

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Comment block

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* Use inline comments to explain assumptions, known issues, and algorithm insights.
* Don’t use inline comments to explain obvious code. Well written code is self-explanatory.
* Only use comments for bad code to say “fix this code” – otherwise remove or re-write the code.
* Include comments using Task List keyword flags to allow comment-filtering.

**Example:**

// TODO: Place Database Code Here

// UNDONE: Removed P\Invoke Call due to errors

// HACK: Temporary fix until able to refactor

* Always apply C# comment-blocks /// to public, protected and internal declarations.
* Only use C# comment-blocks for documenting the APIs.
* Always include <summary>, <param>, <return> and <exception> comment sections wherever applicable.

**Language Usage**

* Don’t omit access modifiers. Explicitly declare all identifiers with the appropriate access modifier instead of allowing the default.

**Example:**

**Bad:**

void WriteEvent(string message) {…}

**Good:**

private void WriteEvent(string message) {…}

* Don’t use the default (“1.0”) versioning scheme. Increment the AssemblyVersionAttribute value manually.
* Always set the ComVisibleAttribute to false for all assemblies until it needs to be enabled.
* Consider factoring classes containing unsafe code blocks into a separate assembly.
* Always use the built-in C# data type aliases, not the .Net common type system (CTS).

**Example:**

short NOT System.Int16

string NOT System.String

* Do not use var when the type is not apparent from the right side of the assignment.
* Try to use int for any non-fractional numeric values that will fit the int data type.
* Only use long for variables potentially containing values too large for an int.
* Try to use double for fractional numbers to ensure decimal precision in calculations.
* Only use float for fractional numbers that will not fit double or decimal.
* Avoid using float unless you fully understand the implications upon calculations.
* Try to use decimal when fractional numbers must be rounded to a fixed precision for calculations. Typically, this will involve money.
* Avoid using sbyte, uint and ulong unless it is for interop (P\Invoke) with native libraries.

Avoid specifying the type for an enum; use the default of int unless you have an explicit need for long (very uncommon).

* Declare readonly or static readonly variables instead of constants for complex types.
* Only declare constants for simple types.
* Avoid direct cast. Instead, use the “as” operator and check for null.

**Example:**

object dataObject = LoadData();

DataSet ds = dataObject as DataSet;

if( ds != null ) {…}

* Avoid boxing and unboxing value types.

**Example:**

int count = 1;

object refCount = count; // Implicitly boxed.

int newCount = (int)refCount; // Explicitly unboxed.

* Try to use “@” prefix for string literals instead of escape strings.
* Prefer string.Format() or StringBuilder over string concatenation.
* Don’t compare strings to String.Empty or “” to check for empty strings. Instead, use String.IsNullOrEmpty.
* Use String.Equals() for string equality comparisons.

**Best Coding Practices**

* Avoid writing very long methods. A method should typically have 1 to 25 lines of code. If a method has more than 25 lines of code, you must consider re-factoring into separate methods.
* A method should do only ‘one job’. Don’t combine more than one job in a single method, even if those jobs are very small.
* Always watch for unexpected values. For example, if you are using a parameter with 2 possible values, never assume that if one is not matching then the only possibility is the other value.
* Do not hardcode numbers. Use constants instead. Declare constant in the top of the file and use it in your code.

*Note: However, using constants are also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed.*

* Avoid declaring string literals inline. Instead use Resources, Constants, Configuration Files, Registry or other data sources.
* Convert strings to lowercase or upper case before comparing. This will ensure the string will match even if the string being compared has a different case.
* Never concatenate strings inside a loop.
* Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.
* Use enum wherever required. Do not use numbers or strings to indicate discrete values.
* Do not make the member variables public or protected. Keep them private and expose public and protected Properties.
* Avoid invoking methods within a conditional expression.
* Keep the lifetime of variables as short as possible when the variables represent a finite resource for which there may be contention, such as a database connection.
* Keep the scope of variables as small as possible to avoid confusion and to ensure maintainability. Also, when maintaining legacy source code, the potential for inadvertently breaking other parts of the code can be minimized if variable scope is limited.
* The event handler should not contain the code to perform the required action. Rather call another method from the event handler.
* Do not programmatically click a button to execute the same action you have written in the button click event. Rather, call the same method which is called by the button click event handler.
* Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.
* Never assume that your code will run from drive "C:". You may never know, some users may run it from network or from a "Z:".
* 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 startup, if required. Give a friendly message to the user in case of any problems.
* If the required configuration file is not found, application should be able to create one with default values.
* 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.
* 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."
* 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.
* Avoid having very large files. If a single file has more than 1000 lines of code, it is a good candidate for refactoring. Split them logically into two or more classes.
* Avoid public methods and properties, unless they really need to be accessed from outside the class. Use “internal” if they are accessed only within the same assembly.
* Avoid passing too many parameters to a method. If you have more than 4 to 5 parameters, it is a good candidate to define a class or structure.
* If you have a method returning a collection, return an empty collection instead of null, if you have no data to return. For example, if you have a method returning an ArrayList, always return a valid ArrayList. If you have no items to return, then return a valid ArrayList with 0 items. This will make it easy for the calling application to just check for the “count” rather than doing an additional check for “null”.
* Logically organize all your files within appropriate folders. Use 2 level folder hierarchies. You can have up to 10 folders in the root folder and each folder can have up to 5 sub folders. If you have too many folders than cannot be accommodated with the above mentioned 2 level hierarchy, you may need re factoring into multiple assemblies.
* Make sure you have a good logging class which can be configured to log errors, warning or traces. If you configure to log errors, it should only log errors. But if you configure to log traces, it should record all (errors, warnings and trace). Your log class should be written such a way that in future you can change it easily to log to Windows Event Log, SQL Server, or Email to administrator or to a File etc without any change in any other part of the application. Use the log class extensively throughout the code to record errors, warning and even trace messages that can help you trouble shoot a problem.
* If you are opening database connections, sockets, file stream etc, always close them in the finally block. This will ensure that even if an exception occurs after opening the connection, it will be safely closed in the finally block.
* Declare variables as close as possible to where it is first used. Use one variable declaration per line.
* Use StringBuilder class instead of String when you have to manipulate string objects in a loop. The String object works in weird way in .NET. Each time you append a string, it is actually discarding the old string object and recreating a new object, which is a relatively expensive operations.
* Do not use the “new” keyword on method and property declarations to hide members of a derived type.
* Only use the “base” keyword when invoking a base class constructor or base implementation within an override.
* Consider using method overloading instead of the params attribute.
* Always use the “lock” keyword instead of the Monitor type.
* Avoid locking on a Type.

**Example:**

lock( typeof( MyClass ) )

* Avoid locking on the current object instance.

**Example:**

lock( this )

* Avoid explicit Boolean tests in conditionals.

**Bad:**

if(IsValid == true)

{…}

**Good:**

if(IsValid)

{…}

* Avoid compound conditional expressions – use Boolean variables to split parts into multiple manageable expressions.

**Bad:**

if (((value > \_highScore) && (value != \_highScore)) && (value < \_maxScore))

{…}

**Good:**

isHighScore = (value >= \_highScore);

isTiedHigh = (value == \_highScore);

isValid = (value < \_maxValue);

if ((isHighScore && ! isTiedHigh) && isValid)

{…}

* Never do a 'catch exception and do nothing'. If you hide an exception, you will never know if the exception happened or not. Lot of developers uses this handy method to ignore non-significant errors. You should always try to avoid exceptions by checking all the error conditions programmatically. In any case, catching an exception and doing nothing is not allowed. In the worst case, you should log the exception and proceed.
* In case of exceptions, give a friendly message to the user, but log the actual error with all possible details about the error, including the time it occurred, method and class name etc.
* Always catch only the specific exception, not generic exception.
* 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. 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'.
* When you re throw an exception, use the throw statement without specifying the original exception. This way, the original call stack is preserved.
* Do not write try-catch in all your methods. Use it only if there is a possibility that a specific exception may occur and it cannot be prevented by any other means. For example, if you want to insert a record if it does not already exists in database, you should try to select record using the key. Some developers try to insert a record without checking if it already exists. If an exception occurs, they will assume that the record already exists. This is strictly not allowed. You should always explicitly check for errors rather than waiting for exceptions to occur. On the other hand, you should always use exception handlers while you communicate with external systems like network, hardware devices etc. Such systems are subject to failure anytime and error checking is not usually reliable. In those cases, you should use exception handlers and try to recover from error.
* 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.
* Write your own custom exception classes if required in your application. Derive your custom exceptions from the base class System.Exception.
* Avoid nesting a try/catch within a catch block.
* Always use validation to avoid exceptions.
* Create objects as late as possible, and destroy them as early as possible to free resources.
* Be wary of using ASP Session variables in a Web farm environment. At a minimum, do not place objects in ASP Session variables because session state is stored on a single machine. Consider storing session state in a database instead.
* Stateless components are preferred when scalability or performance are important. Design the components to accept all the needed values as input parameters instead of relying upon object properties when calling methods. Doing so eliminates the need to preserve object state between method calls. When it is necessary to maintain state, consider using alternative methods, such as maintaining state in a database.
* Do not use session variables throughout the code. Use session variables only within the classes and expose methods to access the value stored in the session variables.
* Do not store large objects in session. Storing large objects in session may consume lot of server memory depending on the number of users.
* Always use style sheet to control the look and feel of the pages. Never specify font name and font size in any of the pages. Use appropriate style class. This will help you to change the UI of your application easily in future. Also, if you like to support customizing the UI for each customer, it is just a matter of developing another style sheet for them.
* Try to minimize the scope and duration of transactions.
* Do not open data connections using a specific user's credentials. Connections that have been opened using such credentials cannot be pooled and reused, thus losing the benefits of connection pooling.
* Never use SELECT \*. Always be explicit in which columns to retrieve and retrieve only the columns that are required.
* Use stored procedures in lieu of SQL statements in source code to leverage the performance gains they provide.
* Use a stored procedure with output parameters instead of single-record SELECT statements when retrieving one row of data.
* Verify the row count when performing DELETE operations.
* Perform data validation at the client during data entry. Doing so avoids unnecessary round trips to the database with invalid data.
* Avoid using functions in WHERE clauses.
* If possible, specify the primary key in the WHERE clause when updating a single row.
* When using LIKE, do not begin the string with a wildcard character because SQL Server will not be able to use indexes to search for matching values.
* Use WITH RECOMPILE in CREATE PROC when a wide variety of arguments are passed, because the plan stored for the procedure might not be optimal for a given set of parameters.
* Stored procedure execution is faster when you pass parameters by position (the order in which the parameters are declared in the stored procedure) rather than by name.
* Use triggers only for data integrity enforcement and business rule processing and not to return information.
* After each data modification statement inside a transaction, check for an error by testing the global variable @@ERROR.
* Use forward-only/read-only recordsets. To update data, use SQL INSERT and UPDATE statements.
* Never hold locks pending user input.
* Use uncorrelated subqueries instead of correlated subqueries. Uncorrelated subqueries are those where the inner SELECT statement does not rely on the outer SELECT statement for information. In uncorrelated subqueries, the inner query is run once instead of being run for each row returned by the outer query.
* Always separate presentation layer from business logic. Never access database from the UI pages. Always have a data layer class which performs all the database related tasks. This will help you support or migrate to another database back end easily.
* Use try-catch in your data layer to catch all database exceptions. This exception handler should record all exceptions from the database. The details recorded should include the name of the command being executed, stored proc name, parameters, connection string used etc. After recording the exception, it could be re thrown so that another layer in the application can catch it and take appropriate action.
* Separate your application into multiple assemblies. Group all independent utility classes into a separate class library. All your database related files can be in another class library.
* Always prefer interfaces over abstract classes.
* Only make members virtual if they are designed and tested for extensibility.