**Section 1 - Scope**

**Scope**

The scope of this document is to identify .NET coding standards and the code review process that will be followed for all .NET applications developed or maintained by staff or consultants employed by the Office of Information Systems.

Best practices are also included in this document for developers to incorporate in their code.

Updates to this document are made on an as-needed basis.

**SECTION 2 - THE REVIEW PROCESS**

**What is a Code Review and Who Performs It?**

A code review is a methodical review of .NET code against the established OIS .NET Coding Standards (located in Section 3 of this document). Two types of .NET code reviews are referenced in this document: Informal and Formal.

The goal of informal code review(s) is to locate and expedite any concerns or questions to the entire project team as early in the coding process as possible. This enables a pro-active approach to resolving potential problems as well as providing knowledge sharing. The informal code review is conducted internally by .NET skilled developers able to provide an objective review of the code. In addition, informal code review(s) must include a review of the SQL by the database technician assigned to the project. The informal reviewer(s) are coordinated by the project manager.

The goal of the formal code review is to provide a final check of the code prior to production implementation to ensure that violation of standards has not been overlooked. The formal code review is conducted by a team of experienced .NET developers provided by Verbat . The review is documented and results stored in the Code Review Library. Project staff are notified once the review is complete.

**wHY IS A CODE REVIEW PERFORMED?**

A code review is performed to protect the shared runtime environment from inefficient and unsafe code execution. It provides an improvement in the overall quality and consistency of coding practices as well as development of best coding practices whereby others can benefit. Reviewing the application functionality, behavior, workflow, user interface, or any other application design issues is not part of the review.

**WHICH PROJECTS REQUIRE A CODE REVIEW?**

All new development requires a .NET Code Review. Additionally, any application that has a maintenance or enhancement release requires a .NET Code Review.

**WHEN ARE CODE REVIEWS PERFORMED?**

Informal code review(s) must be done periodically during the application life cycles. They should be performed early enough so that feedback provided to the programmer can be incorporated in their future assignments. Informal code reviews should be an iterative process, not fixed to a specific point in the development or maintenance lifecycle.

The Formal Code Review is conducted and documented at the end of the construction / maintenance phases and before final user acceptance.

The project team must build time into the project schedule for the informal and formal .NET code reviews. Formal Code Reviews should allow for a 5 – 10 business day turnaround of the results. All projects must be aware of the time limitations for the Formal Code Review and provide advance notification to allow for its scheduling. Items uncovered in the Formal Code Review must be resolved before production implementation.

**The Formal Code Review**

1. The request for a Formal Code Review must be made two weeks prior to when the code will be available for review (at code freeze). The Code Review process takes 5 – 10 days to complete.
2. An application cannot move to production if there is a pending code review.
3. Violations identified during a code review must be addressed prior to production implementation. The following must be provided to the review team for re-review.
   1. Source Code Location and version information, instructions and all necessary software and details to create a buildable solution.
4. It is the responsibility of the Project Manager to notify the Review Team of any changes made after the Code Review has begun, including a summary of any significant changes. The Review Team will identify if an additional Code Review is required.

**Section 3 - .NET coding standards**

|  |
| --- |
| 1. **Calling the Garbage Collector is not allowed.** |
| **SUPPORTING DETAILS:** Calling the garbage collector explicitly is a very resource intensive procedure. It does not need to be called explicitly because it is run by the Common Language Runtime (CLR) during periods of low activity and memory pressure. |
| 1. **All objects that implement the IDisposable interface or inherits from a class that implements this interface must have their dispose method called either in a try finally block or by using a 'using' block when the dispose method is not automatically called and where NOT calling Dispose results in a Resource Leak.** |
| **SUPPORTING DETAILS:** Objects that implement IDisposable typically use unmanaged resources that may not be appropriately cleaned up when the object goes out of scope resulting in a memory leak. This is particularly important for database connections, file streams and socket connections because you could end up locking the resource.Certain objects such as the Data Table do not leak resources if not disposed and do not need to call their Dispose method. If you are in doubt as to whether a particular object needs to be disposed, it is better to err on the side of caution and call the dispose method. Inspecting the disassembled source through Reflector allows verification of the effect of not calling the Dispose.  The easiest way to tell is to wrap the object in question in a *using()* call; Visual Studio will let you know if it doesn’t implement IDisposable. |
| 1. **Overriding default settings for the .NET Request.Form validation is not allowed.** |
| **SUPPORTING DETAILS:** Disabling ASP.NET Request Validation allows potential cross site scripting attacks. IIS will not validate the content of the http request header for any potentially malicious tags or scripts. |
| 1. **SQL statements must be implemented using ADO .Net parameterized query objects.** |
| **SUPPORTING DETAILS:** Parameterized SQL statements are important for both security and application performance. The use of ADO.Net parameterized queries is the standard. Adherence is required whether using handcrafted queries or an ORM tool.  **Security:**  Parameterized queries are the best defense against SQL injection attacks because they prevent the context of the statement from being changed. In typical SQL injection attacks, string terminators and SQL keywords are entered which if concatenated with a dynamic SQL query cause the execution of SQL code not intended by the developer.  **Performance:**  Parameterized queries give a significant performance boost to most database operations because compilation by the DBMS is on first execution only. All subsequent calls use the cached execution path until it is removed from the cache. This works because the SQL statement itself is not changing compared to a completely dynamic statement with embedded values. |
| 1. **General Shared Resource Addresses should not be hard coded.** |
| **SUPPORTING DETAILS:** Hard coding shared resource addresses requires the code to be changed when promoting from Unit Test, through System Test and on to Production. Additionally, system configuration changes cannot be implemented seamlessly when hard coded references are used.  Examples of Shared Resources include, but are not limited to:  SMTP  DB2Creator - available through the connection value or the environment variables  Absolute URLs, use the relative path instead.  IP addresses  Server Names |
| 1. **Turn Option Strict On for VB.NET (disallows late binding).**     1. **For Projects this must be enabled at the Project Level.**    2. **For Websites this must be set in the root config file on the website.** |
| **SUPPORTING DETAILS:** Visual Basic allows conversions of many data types to other data types. Data loss can occur when the value of one data type is converted to a data type with less precision or smaller capacity. A run-time error occurs if such a narrowing conversion fails. Option Strict ensures compile-time notification of these narrowing conversions so they can be avoided. In addition to disallowing implicit narrowing conversions, Option Strict generates an error for late binding. An object is late bound when it is assigned to a variable that is declared to be of type Object. Because Option Strict On provides strong typing, prevents unintended type conversions with data loss, disallows late binding, and improves performance, its use is required.  Source: <http://msdn2.microsoft.com/en-us/library/zcd4xwzs.aspx> |
| 1. **The VB.NET Methods and Keywords listed below must not be used because they are redundant, deprecated or potential performance degraders. Use one of the listed alternate constructs instead.** |
| **SUPPORTING DETAILS:** The use of CLR methods and constructs will prevent the use of VB deprecated code which will not be/is not supported in the future and better supports object-oriented programming (OOP). For more information see <http://msdn.microsoft.com/en-us/library/skw8dhdd.aspx>.   | **Outdated VB6 Keyword/Method** | **VB.NET Preferred Alternate** | | --- | --- | | Call | Method calls no longer require the use of the Call keyword. | | Err | Use the Exception object or a derivative | | Iif | This causes unnecessary overhead due to boxing/un-boxing. Use standard If…Then…Else blocks with the short-circuited Boolean operators (OrElse and AndAlso) | | GoTo | Refactor your code to use one of the more standard control flow statements (if, select case etc.) | | On Error GoTo / Resume | Use Try…Catch…Finally blocks | | Randomize | Random Object is automatically randomized when declared | | EndIf | Use ‘End If’ to terminate the ‘If Then Else’ Block | | Variant | Using the ‘variant’ equivalent, ‘object’ type is not recommended as it carries a performance cost while boxing and unboxing. Either use type specific methods or use generics. | | GoSub | Call procedures with the Call statement, and the GoSub statement is not supported.  Note: As mentioned in the ‘Call’ section, you can directly call the method with the name without the keyword | | Wend | Use While ... End. | | Let | No alternative. The meaning of the Let keyword has changed. Let is now used in LINQ queries.  ‘Let’ Computes a value and assigns it to a new variable within the query. | |
| 1. **Include comments in the code.** |
| **SUPPORTING DETAILS:** These comments should explain why the code was written in the manner that it was, and should make it easier for programmers working on maintenance in the future. Use these comments to document anything that you think will help future programmers understand the coding method/logic that was used. |
| 1. **When using HIS and DB2 stored procedures defined with ‘commit on return no’, the .NET code must wrap each call to a procedure (or group of calls to a procedure) in a transaction.** |
| **SUPPORTING DETAILS:** Not wrapping DB2 non-committing stored procedures in a transaction will result in DB2 perpetually locking the resources used in the call. DB2 will eventually close the connection without releasing the lock, and this will lead to the MSDB2 client driver becoming unstable and unrecoverable. This state will force a process re-start to correct. This will allow the transaction to control the unit of work, insuring data integrity, avoiding the risk of partial units of work being committed, and minimizing DB2 locking. |
| 1. **Exceptions must be handled appropriately by the project.**     1. Exceptions must not be swallowed.    2. Only catch expected exceptions for the purpose of handling them.       1. Only mask an exception that is handled within the application.    3. All other exceptions must be allowed to bubble-up to the correct tier. |
| **SUPPORTING DETAILS:** Exception Handling is critical to the project as well as to the environment.   * Swallowing (ignoring) or masking (replacing) unexpected exceptions will result in lost information. Allow the exception information to bubble-up unchanged, so that complete information is available to support staff. * The syntax throw works better than throw ex * Preserve the stack trace information where appropriate. |
| 1. **Suppress the display of exception details to the user.** |
| **SUPPORTING DETAILS**: Suppressing error details in an application is a vital step in implementing a comprehensive strategy to prevent a wide variety of malicious attacks. Error messages that reveal details about the application’s business objects or database structure can be exploited by malicious users to launch an attack against the application. In web applications, suppressing the exception details can be achieved by using custom error pages.   * Customized, user friendly error pages or messages should always be used to hide information that can aide an attacker seeking to exploit vulnerabilities in our applications. * Where possible, application errors and exceptions with their associated stack trace should be emailed to the appropriate groups for troubleshooting the underlying problem. * The web application’s Web.Config file must contain a configuration entry that specifies the use of error pages.   Some options include:   1. Use the error handling of Enterprise Library for web applications. This is the recommended method. 2. Use the <customErrors> sections to configure the error page and set the **mode** property to either **on** or **remote only**. |
| 1. **Applications must be developed such that there is at least logical separation between presentation logic, business logic and persistence logic.**     1. The business logic must encapusulate the business rules, state and interactions inherent to the problem domain.    2. The presentation logic must handle view rendering, input collection and transformation as necessary before passing data to the business object classes for processing.    3. The persistence logic must handle system state persistence and retrieval including transformation necessary to recreate business logic state.    4. Additional layers of separation are permissible for additional separation of concerns as well as for coordination and communication between layers.Exceptions must be handled appropriately by the project. |
|  |
| **SUPPORTING DETAILS**: This standard sets out a bare minimum requirement for separating application concerns to improve maintainability of the system.  Outside of just separating UI/Business logic/persistence logic applications tend to deal with various concerns such as user notifications, document storage and retrieval, workflow management, etc. which are all logically separate and should be identifiable as separate areas of concern for the system. For example, a single logical operation may require a status change on an entity, the storage of a document and an email notification to a user that the action was performed. This should not be implemented as one method that has all the logic to perform these 3 separate tasks but rather there should be a coordination between separate testable areas of functionality within the application to perform the operation.  We strongly discourage code that is not testable in isolation from other dependencies i.e. static classes, code that has side effects within the system, code that is tightly coupled to other code/classes, etc.  We highly recommend developers adhere to the SOLID design principles as much as possible. The following blog article series goes into detail regarding applying the SOLID principles: <http://lostechies.com/jimmybogard/2008/06/17/separation-of-concerns-how-not-to-do-it/> |

**Section 4 - best practices**

**General:**

1. Use strong types instead of Object whenever possible.
2. Use parenthesis to explicitly define the order of operations in expressions.
3. For classes to be instantiated locally, allow access to fields from outside a class through properties only.
4. For classes to be instantiated remotely, avoid properties and only implement methods.
5. For Code Maintenance and readability, do not use abbreviations unless there is an industry standard abbreviation already in existence. (int for Integer, bln for Boolean)
6. Reuse code wherever possible in order to avoid excessive repeated or duplicated code blocks.
7. Execute an Application Center load test.
8. Use lightest object possible (e.g. data table vs. dataset)
9. Make the code as structured and organized as possible
10. Use StringBuilder for complex string manipulations and when you need to concatenate strings multiple times.
11. Eliminate compiler warnings prior to .NET code review.
12. Utilize test fixtures or automated test cases.
13. Use the XML comment format to automatically insert the comment tags. (Ex: ///) Using this format will allow teams to export comments into a separate document.
14. Set the .NET framework to the same version for all projects in the solution.
15. Organize code in such a way that the solution file is in a parent folder and projects in subfolders beneath the solution folder and avoid the dependencies on specific paths. Use project references instead of assembly references when referencing another project in the solution.
16. Avoid excessive or inappropriate use of caching.
17. Avoid excessive use of SQL command execution per request.
18. Avoid excessive use of session.
19. Constant values and optional parameters should be avoided in public interfaces where possible.  The recommendation is to use static readonly instead of const, or overloads instead of optional parameters in the case of methods and constructors.
    * Supporting Details: During compilation, constant values are inserted into the generated IL.  Optional parameters (for a method, constructor, delegate, or indexer), when not specified by the caller, are also seen as constants.  Any assembly with code built against the constant value will emit IL that contains the value of that constant at the time of compilation.  If the constant is subsequently changed and its containing assembly rebuilt, any assemblies with code using those constants also need to be rebuilt, as they contain the original value.  For non-public constant values, this is not a problem as the constant should only be baked into the assembly that contains it, which is obviously rebuilt with the new constant values.

**Code Structure:**

1. Avoid the use of Literals within the code. Use constants wherever possible.
2. Nested logic should use parenthesis to explicitly set the order of evaluation.
3. Use verbose function and variable names.
4. Declare variables as close as possible to their usage.
5. Use Compiler Directives to separate test code from release code.

**SECTION 5 - Requesting an exception or change to the Code Review standards.**

**Requesting an exception or change to the standards.**

1. Project Teams may request exceptions or changes to the standard.
2. The exception or change requests must be provided, in writing, to the Verbat Quality Assurance Specialist. The request must include:
   1. Standard(s) for which they are requesting the exception or change.
   2. Business case justifying why the exception or change is needed.
   3. Technical details of the non-standard implementation or the change being proposed.
   4. Impact to the Department for the exception or change.
   5. List alternatives considered with pros and cons of each alternative.
   6. Provide justification of why requested exception was the chosen alternative.
3. The request for exception or change will be reviewed by a team assembled by Verbat Quality Assurance Specialist.
4. The review team will provide a written recommendation to the client Manager.
5. Final decision will be determined by the client Manager.

**SUPPORTING DETAILS – Requesting Exceptions or Changes**

The Application Development arena is constantly changing. The .Net Coding Web Standards must also change to meet the needs of our growing Application Development community. Project Teams requesting changes/exceptions are required to provide the listed documentation to assist in the research and understanding of their particular situation.

**Naming Conventions and Standards**

|  |
| --- |
| Note :  The terms Pascal Casing and Camel Casing are used throughout this document.  **Pascal Casing** - First character of all words are Upper Case and other characters are lower case.  Example: BackColor  **Camel Casing -** First character of all words, except the first word are Upper Case and other characters are lower case.  Example: backColor |

1. Use Pascal casing for Class names

public class **HelloWorld**

{

...

}

1. Use Pascal casing for Method names

void **SayHello**(string name)

{

...

}

1. Use Camel casing for variables and method parameters

int **totalCount** = 0;

void SayHello(string name)

{

string **fullMessage** = "Hello " + name;

...

}

1. Use the prefix “I” with Camel Casing for interfaces ( Example: **IEntity** )
2. Do not use Hungarian notation to name variables.

In earlier days most of the programmers liked it - having the data type as a prefix for the variable name and using m\_ as prefix for member variables. Eg:

string m\_sName;

int nAge;

However, in .NET coding standards, this is not recommended. Usage of data type and m\_ to represent member variables should not be used. All variables should use camel casing.

|  |
| --- |
| Some programmers still prefer to use the prefix **m\_** to represent member variables, since there is no other easy way to identify a member variable. |

1. Use Meaningful, descriptive words to name variables. Do not use abbreviations.

Good:

string address

int salary

Not Good:

string nam

string addr

int sal

1. Do not use single character variable names like i, n, s etc. Use names like index, temp

One exception in this case would be variables used for iterations in loops:

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

{

...

}

If the variable is used only as a counter for iteration and is not used anywhere else in the loop, many people still like to use a single char variable (i) instead of inventing a different suitable name.

1. Do not use underscores (\_) for local variable names.
2. All member variables must be prefixed with underscore (\_) so that they can be identified from other local variables.
3. Do not use variable names that resemble keywords.
4. Prefix boolean variables, properties and methods with “is” or similar prefixes.

Ex: private bool \_isFinished

1. Namespace names should follow the standard pattern

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

1. Use appropriate prefix for the UI elements so that you can identify them from the rest of the variables.

There are 2 different approaches recommended here.

* 1. Use a common prefix ( ui\_ ) for all UI elements. This will help you group all of the UI elements together and easy to access all of them from the intellisense.
  2. Use appropriate prefix for each of the ui element. A brief list is given below. Since .NET has given several controls, you may have to arrive at a complete list of standard prefixes for each of the controls (including third party controls) you are using.

|  |  |
| --- | --- |
| **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 |

1. File name should match with class name.

For example, for the class HelloWorld, the file name should be helloworld.cs (or, helloworld.vb)

1. Use Pascal Case for file names.

**Indentation and Spacing**

1. Use TAB for indentation. Do not use SPACES. Define the Tab size as 4.
2. Comments should be in the same level as the code (use the same level of indentation).

Good:

// Format a message and display

string fullMessage = "Hello " + name;

DateTime currentTime = DateTime.Now;

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

MessageBox.Show ( message );

Not Good:

// Format a message and display

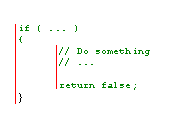
string fullMessage = "Hello " + name;

DateTime currentTime = DateTime.Now;

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

MessageBox.Show ( message );

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



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

Good:

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;

}

Not Good:

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;

}

1. There should be one and only one single blank line between each method inside the class.
2. The curly braces should be on a separate line and not in the same line as if, for etc.

Good:

if ( ... )

{

// Do something

}

Not Good:

if ( ... ) {

// Do something

}

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

Good:

if ( showResult == true )

{

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

{

//

}

}

Not Good:

if(showResult==true)

{

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

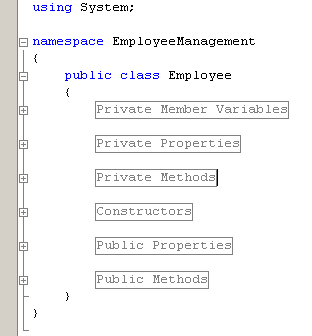
{

//

}

}

1. Use #region to group related pieces of code together. If you use proper grouping using #region, the page should like this when all definitions are collapsed.



1. Keep private member variables, properties and methods in the top of the file and public members in the bottom.

**Good Programming practices**

1. Avoid writing very long methods. A method should typically have 1~25 lines of code. If a method has more than 25 lines of code, you must consider re factoring into separate methods.
2. Method name should tell what it does. Do not use mis-leading names. If the method name is obvious, there is no need of documentation explaining what the method does.

Good:

void SavePhoneNumber ( string phoneNumber )

{

// Save the phone number.

}

Not Good:

// This method will save the phone number.

void SaveDetails ( string phoneNumber )

{

// Save the phone number.

}

1. A method should do only 'one job'. Do not combine more than one job in a single method, even if those jobs are very small.

Good:

// Save the address.

SaveAddress ( address );

// Send an email to the supervisor to inform that the address is updated.

SendEmail ( address, email );

void SaveAddress ( string address )

{

// Save the address.

// ...

}

void SendEmail ( string address, string email )

{

// Send an email to inform the supervisor that the address is changed.

// ...

}

Not Good:

// Save address and send an email to the supervisor to inform that

// the address is updated.

SaveAddress ( address, email );

void SaveAddress ( string address, string email )

{

// Job 1.

// Save the address.

// ...

// Job 2.

// Send an email to inform the supervisor that the address is changed.

// ...

}

1. Use the c# or VB.NET specific types (aliases), rather than the types defined in System namespace.

int age; (not **I**nt**16**)

string name; (not **S**tring)

object contactInfo; (not **O**bject)

|  |
| --- |
| Some developers prefer to use types in Common Type System than language specific aliases. |

1. 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.

Good:

If ( memberType == eMemberTypes.Registered )

{

// Registered user… do something…

}

else if ( memberType == eMemberTypes.Guest )

{

// Guest user... do something…

}

**else**

**{**

**// Un expected user type. Throw an exception**

**throw new Exception (“Un expected value “ + memberType.ToString() + “’.”)**

**// If we introduce a new user type in future, we can easily find**

**// the problem here.**

**}**

Not Good:

If ( memberType == eMemberTypes.Registered )

{

// Registered user… do something…

}

else

{

// Guest user... do something…

// If we introduce another user type in future, this code will

// fail and will not be noticed.

}

1. Do not hardcode numbers. Use constants instead. Declare constant in the top of the file and use it in your code.

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.

1. Do not hardcode strings. Use resource files.
2. 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.

if ( name.ToLower() == “john” )

{

//…

}

1. Use String.Empty instead of “”

Good:

If ( name == String.Empty )

{

// do something

}

Not Good:

If ( name == “” )

{

// do something

}

1. 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.
2. Use enum wherever required. Do not use numbers or strings to indicate discrete values.

Good:

enum MailType

{

Html,

PlainText,

Attachment

}

void SendMail (string message, MailType mailType)

{

switch ( mailType )

{

case MailType.Html:

// Do something

break;

case MailType.PlainText:

// Do something

break;

case MailType.Attachment:

// Do something

break;

default:

// Do something

break;

}

}

Not Good:

void SendMail (string message, string mailType)

{

switch ( mailType )

{

case "Html":

// Do something

break;

case "PlainText":

// Do something

break;

case "Attachment":

// Do something

break;

default:

// Do something

break;

}

}

1. Do not make the member variables public or protected. Keep them private and expose public/protected Properties.
2. The event handler should not contain the code to perform the required action. Rather call another method from the event handler.
3. 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.
4. Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.
5. 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:".
6. In the application start up, do some kind of "self check" and ensure all required files and dependancies 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.
7. If the required configuration file is not found, application should be able to create one with default values.
8. 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.
9. 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."
10. 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."
11. 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.
12. Do not have more than one class in a single file.
13. Have your own templates for each of the file types in Visual Studio. You can include your company name, copy right information etc in the template. You can view or edit the Visual Studio file templates in the folder C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\ItemTemplatesCache\CSharp\1033. (This folder has the templates for C#, but you can easily find the corresponding folders or any other language)
14. 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.
15. 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.
16. Avoid passing too many parameters to a method. If you have more than 4~5 parameters, it is a good candidate to define a class or structure.
17. 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”.
18. Use the AssemblyInfo file to fill information like version number, description, company name, copyright notice etc.
19. 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.
20. 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.
21. 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.
22. Declare variables as close as possible to where it is first used. Use one variable declaration per line.
23. 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.

Consider the following example:

public string ComposeMessage (string[] lines)

{

   string message = String.Empty;

   for (int i = 0; i < lines.Length; i++)

   {

      message += lines [i];

   }

   return message;

}

In the above example, it may look like we are just appending to the string object ‘message’. But what is happening in reality is, the string object is discarded in each iteration and recreated and appending the line to it.

If your loop has several iterations, then it is a good idea to use StringBuilder class instead of String object.

See the example where the String object is replaced with StringBuilder.

public string ComposeMessage (string[] lines)

{

    StringBuilder message = new StringBuilder();

    for (int i = 0; i < lines.Length; i++)

    {

      message.Append( lines[i] );

    }

    return message.ToString();

}

**Architecture**

1. Always use multi layer (N-Tier) architecture.
2. 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.
3. 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.
4. 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.

**ASP.NET**

1. 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. A class can access the session using System.Web.HttpCOntext.Current.Session
2. Do not store large objects in session. Storing large objects in session may consume lot of server memory depending on the number of users.
3. 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

**Comments**

Good and meaningful comments make code more maintainable. However,

1. Do not write comments for every line of code and every variable declared.
2. Use **//** or **///** for comments. Avoid using **/\* … \*/**
3. Write comments wherever required. But good readable code will require very less comments. If all variables and method names are meaningful, that would make the code very readable and will not need many comments.
4. Do not write comments if the code is easily understandable without comment. The drawback of having lot of comments is, if you change the code and forget to change the comment, it will lead to more confusion.
5. Fewer lines of comments will make the code more elegant. But if the code is not clean/readable and there are less comments, that is worse.
6. If you have to use some complex or weird logic for any reason, document it very well with sufficient comments.
7. If you initialize a numeric variable to a special number other than 0, -1 etc, document the reason for choosing that value.
8. The bottom line is, write clean, readable code such a way that it doesn't need any comments to understand.
9. Perform spelling check on comments and also make sure proper grammar and punctuation is used.

**Exception Handling**

1. 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.
2. 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.
3. Always catch only the specific exception, not generic exception.

Good:

void ReadFromFile ( string fileName )

{

try

{

// read from file.

}

catch (FileIOException ex)

{

// log error.

// re-throw exception depending on your case.

throw;

}

}

Not Good:

void ReadFromFile ( string fileName )

{

try

{

// read from file.

}

catch (Exception ex)

{

// Catching general exception is bad... we will never know whether

// it was a file error or some other error.

// Here you are hiding an exception.

// In this case no one will ever know that an exception happened.

return "";

}

}

1. 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'.
2. When you re throw an exception, use the throw statement without specifying the original exception. This way, the original call stack is preserved.

Good:

catch

{

// do whatever you want to handle the exception

throw;

}

Not Good:

catch (Exception ex)

{

// do whatever you want to handle the exception

throw ex;

}

1. 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.
2. 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.
3. 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.