Coding Standards

By:-   Rajeev Gupta

Introduction

Anybody can write code. With a few months of programming experience, we can write 'working applications'. Making it work is easy, but doing it the right way requires more work, than just making it work.

Believe it, majority of the programmers write 'working code', but not 'good code'. Writing 'good code' is an art and you must learn and practice it.

Everyone may have different definitions for the term 'good code'. In our definition, the following are the characteristics of good code.

Reliable

Maintainable

Efficient

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

Software development phase
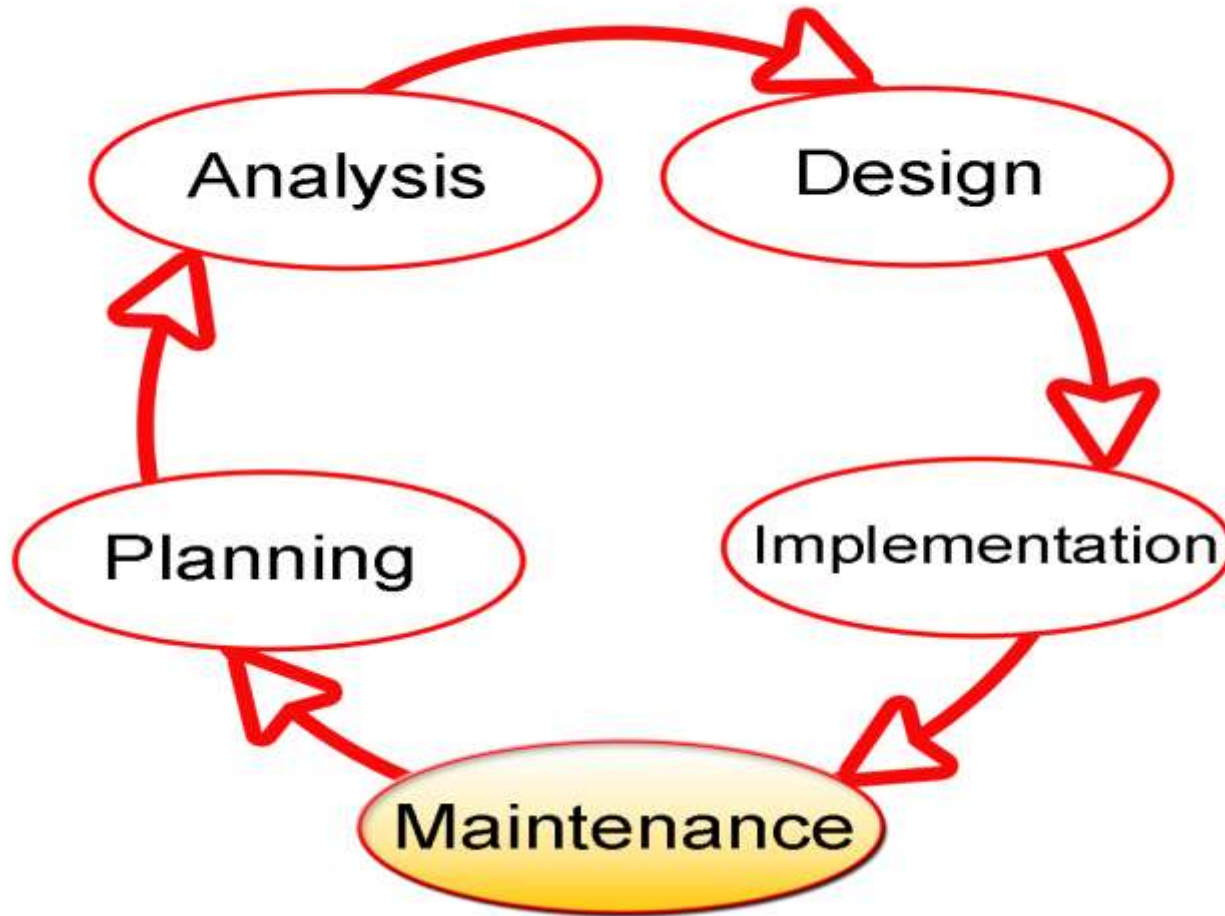
Software Maintenance

- Reducing the cost of **software maintenance** is the most often cited reason for following coding conventions. Code conventions are important to programmers for a number of reasons:

- 40%-80% of the lifetime cost of a piece of software goes to maintenance.

- Hardly any software is maintained for its whole life by the original author.

- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

Definitions

- A coding standard defines the style of your source code including (but not limited to):

    - Indentation, Bracket Placement
    - Variable, Function and Class Names
    - Comments
    - Declaring Pointers
    - And more

- A standard is defined for a language or project
- If everyone follows the standard the code is easy to read
- Readability is enhanced if some coding conventions are followed by all
- Coding standards provide these guidelines for programmers

Coding Conventions

- **Coding conventions** are a set of guidelines for a specific programming language that recommend **programming style**.

- These conventions usually cover **file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, etc**.

- Software programmers are highly recommended to follow these guidelines to help improve the **readability** of their source code and make software maintenance easier.

- Coding conventions are only applicable to the human maintainers and peer reviewers of a software project.

- Coding conventions are not enforced by compilers. As a result, not following some or all of the rules has no impact on the executable programs created from the source code.

Coding Standards

- Where coding conventions have been specifically designed to produce high quality code, and have then been formally adopted, they then become coding standards.

Coding

- Goal is to implement the design in best possible manner
- Coding affects testing and maintenance
- As testing and maintenance costs are high, aim of coding activity should be to write code that reduces them
- Hence, goal should not be to reduce coding cost, but testing and maintenance cost, i.e. make the job of tester and maintainer easier

Purpose of coding standards and best practices

To develop reliable and maintainable applications, you must follow coding standards and best practices.

There are several standards exists in the programming industry. None of them are wrong or bad and you may follow any of them. What is more important is, selecting one standard approach and ensuring that everyone is following it.

How to follow the standards across the team

If we have a team of different skills and tastes, you are going to have a tough time convincing everyone to follow the same standards. The best approach is to have a team meeting and developing your own standards document. You may use this document as a template to prepare your own document.

Distribute a copy of this document (or your own coding standard document) well ahead of the coding standards meeting. All members should come to the meeting prepared to discuss pros and cons of the various points in the document. Make sure you have a manager present in the meeting to resolve conflicts.

Discuss all points in the document. Everyone may have a different opinion about each point, but at the end of the discussion, all members must agree upon the standard you are going to follow. Prepare a new standards document with appropriate changes based on the suggestions from all of the team members. Print copies of it and post it in all workstations.

After you start the development, you must schedule code review meetings to ensure that everyone is following the rules. 3 types of code reviews are recommended:

Peer review – another team member review the code to ensure that the code follows the coding standards and meets 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 is 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 30 minutes before the meeting. Let them read and come up with points for discussion. In the group review meeting, use a projector to display the file content in the screen. 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.

Programming Principles

- The main goal of the programmer is write simple and easy to read programs with few bugs in it.
- There are various programming principles that can help write code that is easier to understand (and test)

- Structured programming
- Information hiding
- Programming practices
- **Coding standards**

File Organization

- *File organization* refers to the way records are physically arranged on a storage device.
- A file consists of various sections that should be separated by several blank lines.
  - Each file should contain one outer class and the name should be same as file
  - Line length should be less than 80
    - If longer continue on another line
  - Special characters should be avoided

File Naming Conventions :-

- File names are made up of a base name, and an optional period and suffix.
- The first character of the name should be a letter and all characters (except the period) should be lower-case letters and numbers.
- The base name should be eight or fewer characters and the suffix should be three or fewer characters
- These rules apply to both program files and default files used and produced by the program (e.g., ``mimoh.sav'').
- The following suffixes are required:

    C source file names must end in .c

    Assembler source file names must end in .s.

    Source files should have .java extension

Naming Conventions and Standards

C# :-

The terms <u>Pascal Casing</u> and <u>Camel Casing</u> are used throughout this document.

Pascal Casing - First character of all words are Upper Case and other characters are lower case.

Example: <u>B</u>ack<u>C</u>olor

Camel Casing **-** First character of all words, except <u>the first word</u> are Upper Case and other characters are lower case.

Example: <u>b</u>ack<u>C</u>olor

Use Pascal casing for Class names

```
public class HelloWorld
{
  ...
}
```

2. Use Pascal casing for Method names

```
void SayHello(string name)
{
    ...
}
```

3. Use Camel casing for variables and method parameters

```
int totalCount = 0;
void SayHello(string name)
{
  string fullMessage = "Hello " + name;
  ...
}
```

1.

Java camel casing

Identifier type	Rules for naming	Examples
Classes	Class names should be in UpperCamelCase, with the first letter of every word capitalised. Use whole words — avoid acronyms and abbreviations.	• class Raster; • class ImageSprite;
Methods	Methods should be verbs in lowerCamelCase	• run(); • runFast();
Variables	Local variables, instance variables, and class variables are also written in lowerCamelCase. Variable names should not start with underscore (_) or dollar sign ($) characters, even though both are allowed.	• int i; • char c; • float myWidth;
Constants	Constants should be written in uppercase characters separated by underscores. Constant names may also contain digits if appropriate, but not as the first character.	• final static int MAX_PARTICIPANTS = 10;

4. Use the prefix "I" with Camel Casing for interfaces (Example: **IEntity**)

5. 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;

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.

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

Good: **Not Good:**

string address   string nam
int salary   string addr
                                                                      int sal

7. 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.

8. Do not use underscores (_) for local variable names.

9. All member variables must be prefixed with underscore (_) so that they can be identified from other local variables.

10. Do not use variable names that resemble keywords.

11. Prefix boolean variables, properties and methods with "**is**" or similar prefixes.

Ex: **private bool _isFinished**

12. Namespace names should follow the standard pattern

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

13. 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.

   A. Use a common prefix (ui_) for all UI elements. This will help you group all of the UI elements together.
   B.   Use appropriate prefix for each of the ui element. A brief list is given below.

14. File name should match with class name.

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

15. Use Pascal Case for file names.

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

Indentation & Spacing

Language	Indent	
C#	4	
CSS	Varies	
HTML	Varies	
Java	4	
JavaScript	Varies	
Perl	4	
PHP	Varies	
Python	4	

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 );
```

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

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

        return false;
}
```

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

Good:

```csharp
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;
}
```

5. There should be one and only one single blank line between each method inside the class.

6. 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
}
```

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

Good:

```
if ( showResult == true )
{
        for ( int i = 0; i < 10; i++ )
        {
                //
        }
}
```
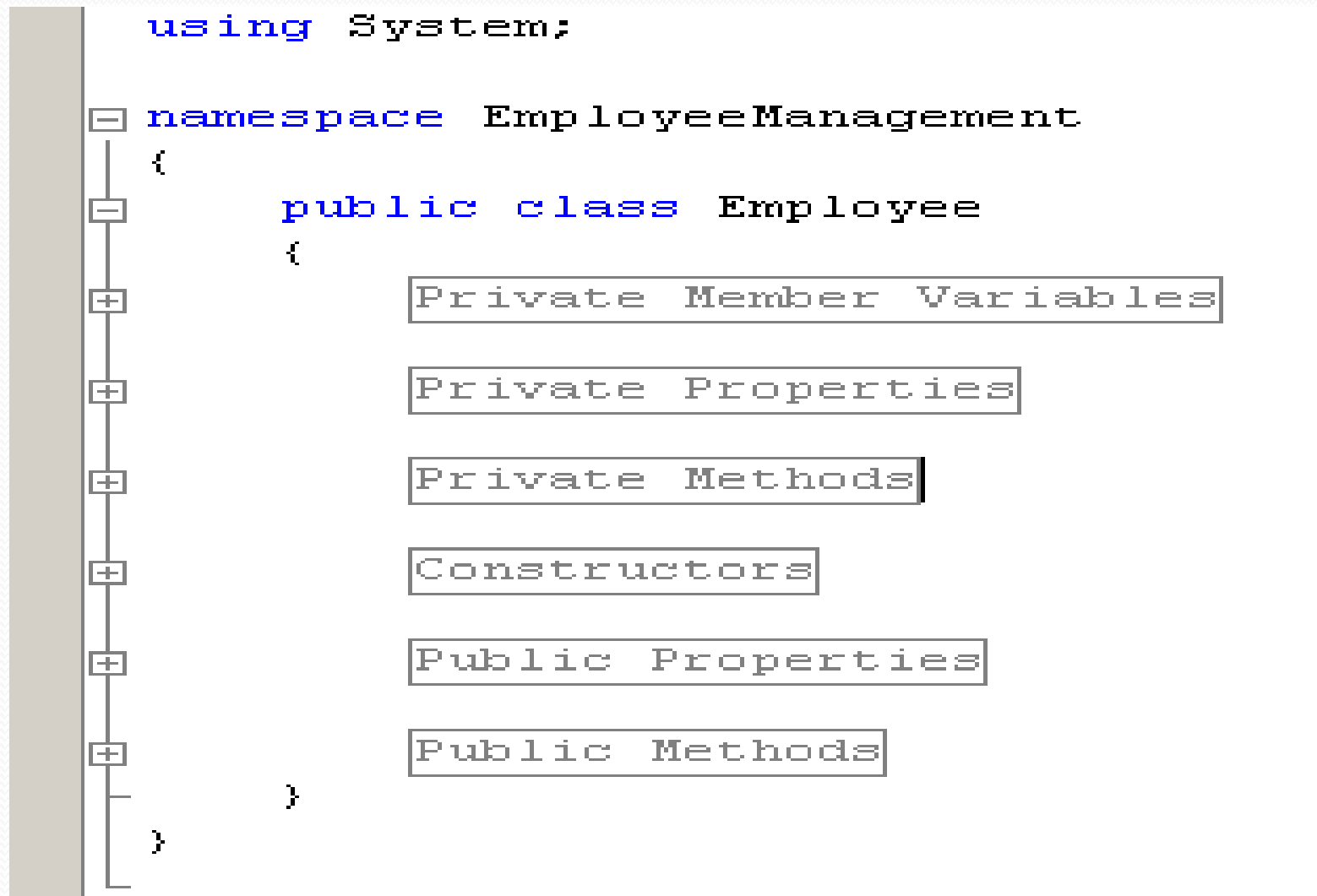
Not Good:

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

8. 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.

```csharp
using System;

namespace EmployeeManagement
{
    public class Employee
    {
        Private Member Variables

        Private Properties

        Private Methods

        Constructors

        Public Properties

        Public Methods
    }
}
```

9. 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 misleading 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.
```

3. 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.
                // ...
        }
```

4. 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.
}
```

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

```
class Calendar
{
    public const int months = 12;
}
```

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

6. 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" )
{
            //…
}
```

7. Use String.Empty instead of ""

Good:

```
If ( name == String.Empty )
{
        // do something
}
```

Not Good:

```
If ( name == "" )
{
        // do something
}
```

8. 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.

9. 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;
        }
}
```

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

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

12. 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."

13. 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.

14. Do not have more than one class in a single file.

15. 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.

16. 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.

17. 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.

18. 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".

19. Use the AssemblyInfofile to fill information like version number, description, company name, copyright notice it.

20. 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.

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.

Architecture

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

2. 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 process 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.

3. 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.

Comments

1. Good and meaningful comments make code more maintainable. However,

2. Do not write comments for every line of code and every variable declared.

3. Use **//** or /// for comments. Avoid using **/* … */**

4. 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.

5. 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.

6. 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.

7. If you have to use some complex or weird logic for any reason, document it very   well with sufficient comments.

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.

Thanks You