This document is part of a series of papers that offer a practical approach to software development.

# Coding Standards

Paul Smietan

## Table of Contents

## Table of Figures

## An Evolution

This paper is all about constructing software, and more specifically, the coding standards subscribed to.  While our industry is constantly evolving with new technologies and languages engineering teams have to stay productive and meet deadlines.  The goal of this paper is to give you a clearer sense of what goes into developing software at the physical level with practical advice.

Our best practices have been gathered over decades of crafting software and surviving reviews by the most critical and discerning engineers.  Valuable lessons learned from failures as well as successes are shared throughout this series of papers.

So why should anyone be concerned about coding standards?

- ✓ Makes code easier to read and understand.
- ✓ Reduces the potential of introducing bugs or defects.
- ✓ Improves stability of the code.
- ✓ Simplifies the debugging process.
- ✓ Improves overall quality.
- ✓ Timeliness to delivering product.

The most *fundamental* best practice is to <u>keep the approach as simple and consistent</u> as possible.  Introducing layers of complexity due to ego or "pushing the envelope" ends up in cost overruns, dissatisfied customers, and loss of cycles.  That said, developers like to experiment and gather metrics as to code performance – prove out your ideas in your own developer's sandbox or virtual environments.  Once your code is stabilized you can introduce it at the code review process (discussed in this paper) where your approach can be defended and its benefits explained to the team; with a little hard work you can affect the standards and add to this paper.

Portions of this document are derived from coding standards and best practices from other publications, blogs, white papers, 2AM production outages, coding death marches, and collaborations with many talented people.  To them all, and even the tough times, we are grateful.

## *Philosophy*

In the spirit of the Open Source Initiative[1] and giving back to our community you are permitted to use and distribute this document as long as credit is given to the author, but most importantly, to those who paved the way:

Dr. Donald Knuth
Dr. Alan Turing
Dr. Ray Polivka
Dr. John R. Clark
Dr. Brian Kernighan
Dr. Dennis Ritchie
Dr. Ken Iverson
Dr. Grace Hopper
Grady Booch
Bill Joy
David Cutler

While this list is incomplete these brilliant people are foundational to the world of computer science and we are the better for their life's work.

## *Humor*

This is a tough business we've chosen. Things don't always go as expected and development sprints are sometimes an uphill battle. Take your dog to the park, listen to your favorite music, and by all means try to keep things light.

*"Each language has its purpose, however humble. Each language expresses the yin and yang of software. Each language has its place within the Tao. But do not program in COBOL if you can avoid it."*

The Tao of Programming
Geoffrey James

[1] https://opensource.org/

## Introduction

The overarching goal of this paper is to provide direction on what is required to write quality code that will not break in a production environment[2]. Achieving this level is obtainable; however, it only comes with devotion to practicing the art of computer science.  If you want to play Carnegie Hall better roll up the sleeves and get to work!

In its simplicity quality production code means that our code is:

- Reliable
- Maintainable
- Efficient

A seasoned developer will ask:

- Does my code meet the specification agreed upon?
- Is my code elegant and scalable?
- Can my code withstand a stringent review?

The consummate professional will then ask:

- Does the craftsmanship of the software speak for itself?
- Have I delivered a product that gives my client a competitive edge?
- Is my customer delighted?

Many of these questions are out of scope for this document and will be addressed in a forthcoming series of papers; however, if you apply these points to your workflow you will reach a level of software craftsmanship many aspire to, but only few achieve.

[2] A production application must run 24 hours per day 7 days per week implementing error and crash recovery, run with minimal downtime, and scale to the demands of the user base.

## *A World of Many Standards*

To develop reliable and maintainable applications you must follow a set of coding standards and best practices. It takes discipline and a spirit of cooperation.

The naming conventions, coding standards and best practices described in this document are drawn from our own experience interacting and learning from other developers. Don’t work in a vacuum! Reach out to other developers and share, learn, and grow.

Several standards exist in the programming industry. None of them are wrong and you may follow whichever makes the most sense for your given situation. What is most important is selecting a standard approach and ensuring that everyone is following it.

## *Teaming Model*

Every team has developers of different skill levels and philosophies. That said, we must all set aside our ego and follow the same standards. The best approach is to have a team meeting and develop your own standards document. You may use this document as a template: add, delete, edit, but do share your version with the community.

Distribute a copy of this document (or your own coding standard document) well ahead of the meeting. All members should come to the meeting prepared to discuss pros and cons of the various points in the document. Make sure someone is designated to make final rulings and 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 the team members. Store the document on your favorite repository and distribute printed copies of it to keep it visible.

During the review simply reference the source code from the projects in your repository. As an example, our team uses BitBucket to manage our source code tree and can easily access their project(s) to perform check-ins and branch management.

Set ground rules: keep it friendly, don’t take criticism personally, keep an open mind, and play nicely. We’re all on the same team.

Three types of code reviews are recommended:

- Peer review
    - Another team member reviews the code to ensure that the code follows the coding standards and meets requirements.
    - This 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 as time permits.
    - Distribute a printed copy of the files to all team members before the meeting.
    - Everyone is encouraged to come up with points for discussion.
    - Visually inspect every section of the code to elicit suggestions on how that piece of code can be written in a better way.

## Best Practices: Coding

### *Development Tools*

Tools like Microsoft Visual Studio™ provides an excellent IDE that can be configured to your standardized conventions and shared among the team. Other operating systems like Linux provide Eclipse and Google Android Developer Studio which offer similar functionality. Regardless of operating system platform the goal is to standardize the coding approach.

### *Naming Conventions*

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 FinancialTransaction
{
}
```

2. Use Pascal casing for Method names:

```
private void GrabDataPacket(string pktKey)
{
}
```

3. Many Java developers use Camel casing for both variables and method parameters. C# developers tend to use Pascal for variables and Camel for parameters.
4. Use the prefix “I” with Pascal Casing for interfaces (e.g. ITransaction).

5. Do not use Hungarian notation to name variables.

In earlier days most programmers liked it as it gave a sense of consistency and type-describing naming. Having the data type as a prefix for the variable name and using m_ as prefix for member variables:

```
string m_sName;
int nAge;
```

This is notation is no longer recommended nor should usage of data type and m_ to represent member variables be used. All variables should use Pascal 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 and descriptive words to name objects. Do not use abbreviations especially if they make the code cryptic!

7. Do not use single character variable names like `i, n, s` etc. Use names like `IndexToDelimeter, TotalCount`
   a. One exception in this case would be variables used for iterations in loops:

```
for(int i = 0; i < 20; i++)
{
}
```

   b. If you need to access a variable beyond the scope of the loop use a meaningful Pascal cased variable.

8. Do not use underscores (_) for local variable names.
   a. We've made an exception for constants since there are typically far less than working variables. We name constants as UPPER_CASENAME to keep them clearly separate and distinct.

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 (e.g. break, while).

11. Prefix Boolean variables, properties and methods with “Is” or similar prefixes.

```
private bool IsTransactionComplete;
```

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. Here are two different approaches:

    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 Visual Studio™ IntelliSense.

    2. Use appropriate prefix for each of the UI elements. Since .NET offers many 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.

    3. If your team decides this is too similar to Hungarian Notation and prefer to have flexibility in their naming than use descriptive control names in a Camel case format (e.g. totalAmountLabel, labelForHeader).

| **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 |

*Figure 1-UI Element Naming*

14. File name containing source should match with class name:

```
Class TransactionMaster() should be stored as TransactionMaster.cs
```

15. Use Pascal Case for data files and think of them in a case-sensitive way regardless of operating system:

```
FiscalYear2019.xml
```

16. Do not use default names assigned by the IDE.
    a. IDE's will assign default names to objects (e.g. variables, fields, class names) to name but a few. Please be sure to name each and every object with a meaningful Pascal cased name. This will make debugging your code way easier and readable.

## *Indentation and Spacing*

This section is typically the most controversial and causes all sorts of religious arguments. As in life everything is a matter of give and take – try to keep an open mind and allow the team to give input on these basic rules.

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).

3. Curly braces ({}) should be in the Allman format or the K&R style[3]. There are other formats but these are the two that my team subscribes to. A mixture is not acceptable – you only get to choose **one** style. Some languages like Java and JavaScript come with an IDE that enforces the K&R format but can usually be modified.

    a. Our team has broken out the bracing style based on language (e.g. C# and C++ is in Allman format, Java and JavaScript are in K&R format).
    b. Whichever route you take at least adhere to a consistent format.

```
while(x == y) {
    something();
    somethingelse();
}
```

*Figure 2-K&R*

```
while(x == y)
{
    something();
    somethingelse();
}
```

*Figure 3-Allman*

[3] Kernigham and Ritchie's seminal work: The C Programming Language (1978).

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

```
Private bool IsTransactionComplete(string TransactionID)
{
        char[] separator = new char[1] { '\\' };
        string result = "";
        string[] subResult = input.Split(separator);

        for(int i = 0; i <= subResult.Length - 1; i++)
        {
                result = i < subResult.Length - 1 ? result +
subResult[i] + "\\" : result + subResult[i];
        }

        return result;
}
```

5. There should be one and only one single blank line between each method inside the class followed by a meaningful comment block articulating the purpose of the method.

6. Add spaces between operators for readability:

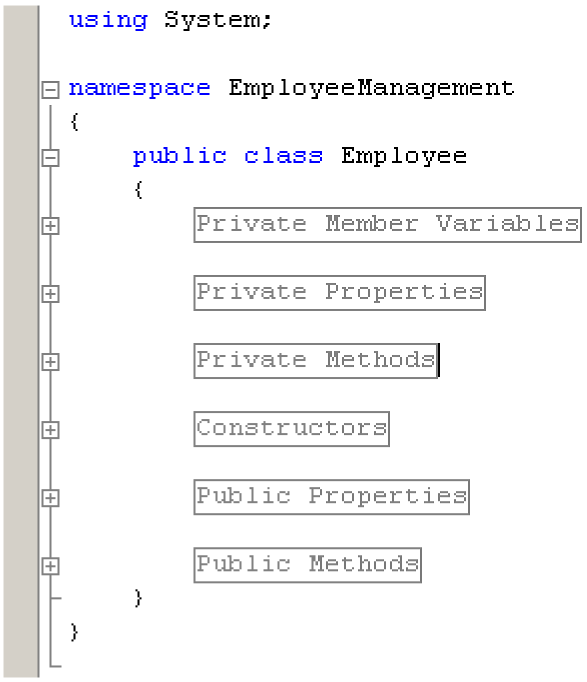
```
/// <summary>
/// CvtToOut: Converts an image as per its extension (e.g. foo.jpg will
create a
/// JPG file).  After the image file is converted a receipt file will
be generated
/// for server-side processing.
///
/// Note #1: Since TIFF is capable of multi-paged images we'll create
separate
/// images in sequence (e.g. foo_1, ... foo_n).  Typical usage of this
function would
/// be to extract images from a multi-page TIFF source into individual
destination
/// JPG/GIF files.
/// </summary>
/// <param name="src"></param>
/// <param name="dst"></param>
public void CvtToOut(string src, string dst)
{
        int iIdx = dst.LastIndexOf('.');
        string sBaseFile = dst.Substring(0, iIdx);
        string sExt = Path.GetExtension(dst);
        ushort nSEQ = 1;

        TiffFile outTIF = new TiffFile(src);

        for(int i = 0; i < outTIF.Pages.Count; i++, nSEQ++)
        {
            string sDstSeq = sBaseFile + "_" + nSEQ.ToString("D5") +
sExt;
            outTIF.Pages[i].Save(sDstSeq);
        }

        outTIF.Dispose();
}
```

7. Use `#region` to group related pieces of code together. This is a very good practice and really help code to be more readable (especially if you have particularly long blocks of code). If you use proper grouping using `#region`, the page should like this when all definitions are collapsed.



```
using System;

namespace EmployeeManagement
{
    public class Employee
    {
        Private Member Variables

        Private Properties

        Private Methods

        Constructors

        Public Properties

        Public Methods
    }
}
```

8. 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 should consider refactoring into separate methods[4].

2. Method name should tell what it does. Be explicit!

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.

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

```
int Age;                (not Int16)
string Name;            (not String)
object ContactInfo;     (not Object)
```

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

5. 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.

```
if(memberType == eMemberTypes.Registered)
{
        …
}
else if(memberType == eMemberTypes.Guest)
{
        …
}
```

[4] Sometimes you may choose to have a larger block of code to avoid duplication or want to implement a business requirement in a more consolidated way. The choice is yours – make it clean and readable.

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

   a. Constants are handy variables to use but increase maintenance efforts should their values or names need to change in the future.  Use them if they're fairly static in nature.

6. Do not hardcode strings. Use resource files or a static class.

7. 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(Path.GetExtension(txtSrcFile.Text).Substring(1).ToUpper().Equals(
CDefs.PDF))
{
        …
}
```

8. Use String.Empty instead of "".

9. 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.

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

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

12. The event handler should not contain the code to perform the required action. Rather call another method from the event handler.

13. 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.

14. Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.

15. 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:".

16. 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.

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

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

19. 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."

20. 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."

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

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

23. Have your own templates for each of the file types in Visual Studio. You can include your company name, copyright information etc. in the template.
    a. C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\IDE\ItemTemplates\CSharp\1033. (This folder has the templates for C#, but you can easily find the corresponding folders or any other language)

24. 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.

25. 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.

26. 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.

27. 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”.

28. Use the AssemblyInfo file to fill information like version number, description, company name, copyright notice etc.

29. 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.

30. 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.

31. 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.

32. Declare variables as close as possible to where it is first used. Use one variable declaration per line.

33. Use StringBuilder class instead of String when you have to manipulate string objects in a loop. String objects are immutable 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 operation.

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 procedure 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` namespace.

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. The "flowerbox" (/* … */) type of commenting is also acceptable as long as you take care of matching your syntax.
3. Write comments wherever required. But good readable code will require less comments and be self-documenting. If all variables and method names are meaningful, that would make the code highly readable and easier to diagnose problem blocks of code.
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. Some developers use 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.  It's advisable to always 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 if possible.

```
void ReadFromFile(string fileName)
{
        try
        {
                // read from file.
        }
        catch (FileIOException ex)
        {
                // log error.
                // re-throw exception depending on your case.
                throw;
        }
}
```

4. 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'.

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

6. 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 exist 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.

7. 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.

8. 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.