Coding Techniques and Programming Practices

3rd October, 2016,

Modified 2nd November, 2017

Table of Contents

Introduction

Superior coding techniques and programming practices are hallmarks of a professional programmer. The coding techniques are primarily those that improve the readability and maintainability of code, whereas the programming practices are mostly performance enhancements.

Coding Techniques

Coding techniques incorporate many facets of software development and, although they usually have no impact on the functionality of the application, they contribute to an improved comprehension of source code. For the purpose of this document, all forms of source code are considered, including programming, scripting, markup, and query languages.

The coding techniques defined here are not proposed to form an inflexible set of coding standards. Rather, they are meant to serve as a guide for developing a coding standard for a specific software project.

The coding techniques are divided into three sections:

1. Names
2. Comments
3. Format
4. ASPX Structure
5. Controls
6. CS File Structure
7. Files
8. Web Config

1. Names

A name should tell "what" rather than "how."

A principle of naming is that difficulty in selecting a proper name may indicate that you need to further analyze or define the purpose of an item. Make names long enough to be meaningful but short enough to avoid being wordy. Programmatically, a unique name serves only to differentiate one item from another. Expressive names function as an aid to the human reader; therefore, it makes sense to provide a name that the human reader can comprehend. However, be certain that the names chosen are in compliance with the applicable language's rules and standards.

Routines

1. Avoid elusive names that are open to subjective interpretation, such as Analyze() for a routine, or xxK8 for a variable. Such names contribute to ambiguity more than abstraction.
2. Use the verb-noun method for naming routines that perform some operation on a given object, such as CalculateInvoiceTotal().
3. When naming functions, include a description of the value being returned, such as GetCurrentWindowName().

Variables

1. Append computation qualifiers (Avg, Sum, Min, Max, Index) to the end of a variable name where appropriate.
2. Use customary opposite pairs in variable names, such as min/max, begin/end, and open/close.
3. Since most names are constructed by concatenating several words together, use mixed-case formatting to simplify reading them as explained in points 4 and 5.
4. camelCasing for parameters and local variables. (camelCasing is

where the first letter of the first word is in lowercase and the first letter of the second word is capitalized.)

5. PascalCasing for function, property, event, and class names. (PascalCasing is where the first letter of both the first and second words are capitalized.).
6. Boolean variable names should contain Is which implies Yes/No or True/False values, such as fileIsFound.
7. Avoid using terms such as Flag when naming status variables, which differ from Boolean variables in that they may have more than two possible values. Instead of documentFlag, use a more descriptive name such as documentFormatType.
8. Even for a short-lived variable that may appear in only a few lines of code, **still use a meaningful name**. **Use single-letter variable names, such as i, or j, for short-loop indexes only.**
9. For variable names, it is sometimes useful to include notation that indicates the scope of the variable, such as prefixing a g_ for global variables.
10. Constants should be all uppercase with underscores between words, such as NUM_DAYS_IN_WEEK. Also, begin groups of enumerated types with a common prefix, such as FONT_ARIAL and FONT_ROMAN.
11. **Variable names are expressed in singular form.**

Tables

1. When naming tables, express the name in the singular form. For example, use Employee instead of Employees.
2. When naming columns of tables, do not repeat the table name; for example, avoid having a field called EmployeeLastName in a table called Employee.

Miscellaneous

1. Minimize the use of abbreviations. If abbreviations are used, be consistent in their use. An abbreviation should have only one meaning and likewise, each abbreviated word should have only one abbreviation. For example, if using min to abbreviate minimum, do so everywhere and do not later use it to abbreviate minute.
2. Avoid reusing names for different elements, such as a routine called ProcessSales() and a variable called iProcessSales.
3. Avoid homonyms when naming elements to prevent confusion during code reviews, such as write and right.
4. When naming elements, avoid using commonly misspelled words. Also, be aware of differences that exist between American and British English, such as color/colour and check/cheque.
5. Avoid using typographical marks to identify data types, such as $ for strings or % for integers.

2. Comments

Software documentation exists in two forms, external and internal. External documentation is maintained outside of the source code, such as specifications, help files, and design documents. Internal documentation is composed of comments that developers write within the source code at development time.

One of the challenges of software documentation is ensuring that the comments are maintained and updated in parallel with the source code. Although properly commenting source code serves no purpose at run time, it is invaluable to a developer who must maintain a particularly intricate or cumbersome piece of software.

Following are recommended commenting techniques:

1. Every file should have a header. The format to be followed is such

```
/*+======================================================
===============
File:      MYFILE.EXT
Summary:   Brief summary of the file contents and purpose.
Classes:   Classes declared or used (in source files).
Functions: Functions exported (in source files).
Origin:    Indications of where content may have come from. This is
not a change history but rather a reference to the editor-
inheritance behind the content or other indications about the origin
of the source.
========================================================
===========+*/
```

2. When modifying code, always keep the commenting around it up

to date.

3. At the beginning of every routine, it is helpful to provide standard, boilerplate comments, indicating the routine's purpose, assumptions, and limitations. A boilerplate comment should be a brief introduction to understand why the routine exists and what it can do.
4. Avoid adding comments at the end of a line of code; end-line comments make code more difficult to read. However, end-line comments are appropriate when annotating variable declarations. In this case, align all end-line comments at a common tab stop. There should be four spaces between the end of the code and the start of end-line comments.
5. Avoid using clutter comments, such as an entire line of asterisks. Instead, use white space to separate comments from code.
6. Avoid surrounding a block comment with a typographical frame. It may look attractive, but it is difficult to maintain.
7. Prior to deployment, remove all temporary or extraneous comments to avoid confusion during future maintenance work.
8. If you need comments to explain a complex section of code, examine the code to determine if you should rewrite it. If at all possible, do not document bad code—rewrite it. Although performance should not typically be sacrificed to make the code simpler for human consumption, a balance must be maintained between performance and maintainability.
9. Use complete sentences when writing comments. Comments should clarify the code, not add ambiguity.
10. Comment as you code, because most likely there won't be time to do it later. Also, should you get a chance to revisit code you've written, that which is obvious today probably won't be obvious six

weeks from now.

11. Avoid the use of superfluous or inappropriate comments, such as humorous sidebar remarks.
12. Use comments to explain the intent of the code. They should not serve as inline translations of the code.
13. Comment anything that is not readily obvious in the code.
14. To prevent recurring problems, always use comments on bug fixes and work-around code, especially in a team environment.
15. Use comments on code that consists of loops and logic branches. These are key areas that will assist the reader when reading source code.
16. Separate comments from comment delimiters with white space. Doing so will make comments stand out and easier to locate when viewed without color clues.
17. Throughout the application, construct comments using a uniform style, with consistent punctuation and structure.

Note - Despite the availability of external documentation, source code listings should be able to stand on their own because hard-copy documentation can be misplaced.

18. External documentation should consist of specifications, design documents, change requests, bug history, and the coding standard that was used, name of the programmer, date when created or modified.

3. Format

Formatting makes the logical organization of the code stand out. Taking the time to ensure that the source code is formatted in a consistent, logical manner is helpful to yourself and to other developers who must decipher the source code.

Following are recommended formatting techniques:

Tab characters (\0x09) should not be used in code. All indentation should be done with 4 space characters. Align sections of code using the prescribed indentation.

1. Use a monospace font when publishing hard-copy versions of the source code.
2. Except for constants, which are best expressed in all uppercase characters with underscores, use mixed case instead of underscores to make names easier to read.
3. Open braces should always be at the beginning of the line after the statement that begins the block. Contents of the brace should be indented by 4 spaces. For example:

```
for (i = 0; i < 100; i++)
{
…
}

if (someExpression)
{
    DoSomething();
}
else
{
    DoSomethingElse();
}
```

“case” statements should be indented from the switch statement like this:

```
switch (someExpression)
{
case 0:
        DoSomething();
        break;
case 1:
        DoSomethingElse();
        break;
case 2:
        {
        int n = 1;
        DoAnotherThing(n);
        }
        break;
}
```

Braces should never be considered optional. Even for single statement blocks, you should always use braces. This increases code readability and maintainability. For instance,

```
for (int i=0; i<100; i++) { DoSomething(i); }
```

Single line statements can have braces that begin and end on the same line.

```
public class Foo
{
```

```
int bar;
public int Bar
{
get { return bar; }
set { bar = value; }
}
}
```

It is suggested that all control structures (if, while, for, etc.) use braces, but it is not required.

4. Indent code along the lines of logical construction. Without indenting, code becomes difficult to follow, such as:

```
If … Then
If … Then
…
Else
…
End If
Else
…
End If
```

5. Indenting the code yields easier-to-read code, such as:

```
If … Then
        If … Then
        …
        Else
        …
        End If
Else
…
End If
```

6. Use spaces before and after most operators when doing so does not alter the intent of the code. For example, an exception is the pointer notation used in C++.
7. Spaces improve readability by decreasing code density. Here are some guidelines for the use of space characters within code:
8. <u>Do</u> use a single space after a comma between function arguments.

   Right: Console.In.Read(myChar, 0, 1);

   Wrong: Console.In.Read(myChar,0,1);
9. <u>Do not</u> use a space after the parenthesis and function arguments

   Right: CreateFoo(myChar, 0, 1)

   Wrong: CreateFoo(myChar, 0, 1)
10. <u>Do not</u> use spaces between a function name and parenthesis.

    Right: CreateFoo()

    Wrong: CreateFoo ()
11. <u>Do not</u> use spaces inside brackets.

    Right: x = dataArray[index];

    Wrong: x = dataArray[index];
12. <u>Do</u> use a single space before flow control statements

    Right: while (x == y)

    Wrong: while(x==y)
13. <u>Do</u> use a single space before and after comparison operators

    Right: if (x == y)

    Wrong: if (x==y)

14. Put a space after each comma in comma-delimited lists, such as array
    values and arguments, when doing so does not alter the intent of the code.
15. Use **vertical** white space to provide organizational clues to source code. Doing so creates "paragraphs" of code, which aid the reader in comprehending the logical segmenting of the software.
16. When a line is broken across several lines, make it obvious that the line is incomplete without the following line. **This can be done by mentioning ellipsis (two full stops) in the comment. For instance "..".**
17. Where appropriate, avoid placing more than one statement per line. An exception is a loop in C, C++, Visual J++®, or JScript®, such as for (i = 0; i < 100; i++).
18. When writing SQL statements, use all uppercase for keywords and mixed case for database elements, such as tables, columns, and views.
19. Divide source code logically between physical files.
20. In ASP, use script delimiters around blocks of script rather than around each line of script or interspersing small HTML fragments with server-side scripting. Using script delimiters around each line or interspersing HTML fragments with server-side scripting increases the frequency of context switching on the server side, which hampers performance and degrades code readability.
21. Put each major SQL clause on a separate line so statements are easier to read and edit, for example:

```
SELECT FirstName, LastName
```

```
FROM Customers
WHERE State = 'WA'
```

22. Do not use literal numbers or literal strings, such as
    For i = 1 To 7. Instead, use named constants, such as For i = 1 To
    NUM_DAYS_IN_WEEK, for ease of maintenance and understanding.
23. Break large, complex sections of code into smaller, comprehensible modules.

4.ASPX structure

1. All CSS is to be maintained in one separate file. There **cannot** be Inline CSS.
2. Javascript is to be maintained in one separate file. If it has to be included in the aspx page, then it has to be included at the end of the aspx page after all the controls have been added. There should be just **one script tag for javascript**. Comments have to be included where necessary be it at the beginning of a function or an end of variable comment. Javascript functions should be formatted using the same guidelines.
3. Comments should be included before server controls where the purpose of a specific control could be difficult to understand. These comments would be before the control.
4. Naming conventions for variables and routines described above are to be followed for Javascript.
5. Head and Form tag should be named appropriately.
6. Every Page should have a title. This can be set using the title tag in HTML if it is a standalone page else it can be set through the properties window by selecting document.
7. The document type tag, DOCTYPE, should always refer to the latest HTML version.
8. All controls are to be names aptly including scriptmanager and contentplaceholders.
9. If there is a panel or a div that contains a number of elements. There is should a comment at before and after the panel/div indicating the start and end of that element. It should be commented as follows, <!--Panel_LeaveSummary starts--> and <!--Panel_LeaveSummary ends-->

10. Unnecessary, unused or commented code and controls should be deleted. If they have to be kept for the future, a comment should be included mentioning the reason for its existence.
11. The aspx page should be formatted using cntrl+K,F.
12. Unnecessary vertical spaces should be deleted.
13. Always check for closing tags. All opening tags need to have a closing tag.
14. Avoid using text without a label or an appropriate html tag.
15. Tab Index should be included for every aspx. The initial value of Tabindex should be set to 1.
16. All for loops should be initialized with 1 and not 0.
17. By default, those controls which have to be hidden or displayed under certain circumstances, should be set to true/false in the aspx page and not in the page load event.
18. Unnecessary attributes have to be deleted as well.
19. Every attribute used should be understood and should exist for a reason.
20. In scenarios where the attributes list is long, it should end at the maximum line.
21. Certain controls should have an access key. There should be a list of access keys to avoid confusion.
22. There should be a vertical space between elements that are not related at all or comments. For instance, two divisions
23. There should be a rule to using html elements that need to be runat server or asp controls or just html controls. There might an advantage of a simple html element or a asp control.
24. Javascript code should not have the debugger.
25. Register assemblies should be one for a namespace. For instance Ajaxcontroltookit.

26. All javascript references are to be included at the end.
27. All unnecessary aspx pages should be deleted.
28. Should have a description of all javascript functions else we might have duplicate functions performing a similar function.
29. When writing HTML, establish a standard format for tags and attributes, such as using all uppercase for tags and all lowercase for attributes. As an alternative, adhere to the XHTML specification to ensure all HTML documents are valid. Although there are file size trade-offs to consider when creating Web pages, use quoted attribute values and closing tags to ease maintainability.
30. The maximum line length for comments and code to avoid having to scroll horizontally the source code window is 174. 174 is the column number. For instance, col 174 at the bottom right of the code editor window.

5.Controls

1. All controls should be named with the name of the control in **full** followed by an underscore and then the specific name. For instance, if it is a label to display an error message, it should be named as **Label_ErrorMessage**. Naming controls will use **PascalCasing** like in the previous example. Another example is as follows, for a dropdown of dates or textview of employee code, **DropDown_Date** and **TextView_EmployeeCode**.
2. All controls are expressed in singular form. For instance, it should be **DropDown_Date** and **NOT DropDown_Dates.**

6.CS File Structure

1. Controls and variables should be declared at the start separated by a blank line. Controls should be declared first and then variables.
2. All variable and control declarations should be alphabetically. For instance, hashtable, int, mysqlcommand, string.
3. The functions and events should be ordered alphabetically. The functions and page events have to be divided.
4. All connections are opened late and closed immediately.
5. All objects declared at the start and initialized when they have to be used.
6. The functions and events should be separated. There should be a start and end comment for functions and events. For instance, <!--events start--> and <!--events end--> & <!--functions start--> and <!—functions end-->.
7. Functions and events should be ordered alphabetically. For instance, Button_click followed by Dropdown_selectionchange for events and addsalary followed by calculateleavebalance for functions.
8. The maximum line length for comments and code to avoid having to scroll horizontally the source code window and to allow for clean hard-copy presentation is 174. 174 is the column number. For instance, col 174 at the bottom right of the code editor window.
9. One blank line should always be used in the following circumstances:
    1. Between methods.
    2. Between the local variables in a method and its first statement.
    3. Before a block or single-line comment.

iv. Between logical sections inside a method to improve readability. For instance, before and after a do-while, for, if and while loops and nested loops.

10. All nested loops should be numbered in the comments.
11. One declaration per line is recommended since it encourages commenting.
12. **Special Comments –** Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken.
13. When an expression will not fit on a single line, break it according to these general principles:

    i. Break after a comma.
    ii. Break before an operator.
    iii. Prefer higher-level breaks to lower-level breaks.
    iv. Align the new line with the beginning of the expression at the same level on the previous line.

  II. If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking method calls:

a. someMethod(longExpression1, longExpression2, longExpression3, longExpression4, longExpression5);

b. var = someMethod1(longExpression1,
    a. someMethod2(longExpression2,
        i. longExpression3));

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

a. longName1 = longName2 * (longName3 + longName4 -
   longName5)
   + 4 * longname6; // PREFER

b. longName1 = longName2 * (longName3 + longName4
   - longName5) + 4 * longname6; // AVOID

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//CONVENTIONAL INDENTATION (4 spaces)
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
    Object andStillAnother) {
    ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}
```

Line wrapping for if statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult.

For example:

```
//DON'T USE THIS INDENTATION(4 spaces)
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();           //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD(8 spaces)
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS(8 spaces)
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
```

: gamma;

7.Files

1. All asp.net applications should be stored in My Documents/Visual Studio 2012/Projects folder.
2. Files should follow PascalCasing. File names should not include numbers.
3. Images,which are files, should be named appropriately.
4. If there are any folders or files that have been created by the developer for a specific reason, then it should be mentioned in a text file with the same name as the folder/file that will be stored in a folder called documentation.
5. There should not be folders inside folders inside the solution folder.
6. There should be documentation for the version of visual studio, framework and database used.
7. All unnecessary files and folders should be deleted.
8. All jquery files should have documentation for future references or use by a new developer. This will also ensure that there aren't multiple files for the same purpose. This document can be maintained in the Jquery folder.
9. There cannot be folders called CSS and Styles that save the same file type i.e. .css
10. There should not be files with the name copy.
11. Files with similar file type should be saved in a folder named appropriately.
12. All Backup files or folders should be saved in backup files in visual studio. It should be renamed to backup. The projects folder should contain the live, current or working projects.
13. If there multiple copies of a folder or file, then there should be a documentation with the same name as the old folder or file. The

document should state the reason for the folder or file becoming obsolete.

14. All working code downloaded from the internet or code that is not currently being used but can be used in the future should be saved in code snippets folder under the visual studio. Also code that can be reused should be saved here.
15. All files or code needed to run the project should be saved in settings folder of visual studio. For instance, the code to allocate permissions to a local machine to connect with the database on the testing server. It could be a specific DLL.
16. If there are files, mostly xml or aspx files, that can be used as templates they should be saved in the templates folder of visual studio. For instance, if one specific login page is to be used across multiple projects then it is a template.
17. Obsolete files or folders should be renamed with the date appended when it last became obsolete.

8.Web config

1. The web config should have just one connection string named as connectionString.
2. In the connection string, the server should be 127.0.0.1 and not localhost. Using localhost makes the page load take time.

9.Build

1. Always build your applications before hitting F5.

Programming Practices

Experienced developers follow numerous programming practices or rules of thumb, which typically derived from hard-learned lessons. The practices listed below are not all-inclusive, and should not be used without due consideration. Veteran programmers deviate from these practices on occasion, but not without careful consideration of the potential repercussions. Using the best programming practice in the wrong context can cause more harm than good.

1. To conserve resources, be selective in the choice of data type to ensure the size of a variable is not excessively large.
2. Keep the lifetime of variables as short as possible when the variables represent a finite resource for which there may be contention, such as a database connection.
3. Keep the scope of variables as small as possible to avoid confusion and to ensure maintainability. Also, when maintaining legacy source code, the potential for inadvertently breaking other parts of the code can be minimized if variable scope is limited.
4. Use variables and routines for one and only one purpose. In addition, avoid creating multipurpose routines that perform a variety of unrelated functions.
5. When writing classes, avoid the use of public variables. Instead, use procedures to provide a layer of encapsulation and also to allow an opportunity to validate value changes.
6. When using objects pooled by MTS, acquire resources as late as possible and release them as soon as possible. As such, you should create objects as late as possible, and destroy them as early as possible to free resources.
7. When using objects that are not being pooled by MTS, it is necessary to examine the expense of the object creation and the

level of contention for resources to determine when resources should be acquired and released.

8. Be wary of using ASP Session variables in a Web farm environment. At a minimum, do not place objects in ASP Session variables because session state is stored on a single machine. Consider storing session state in a database instead.
9. Do not open data connections using a specific user's credentials. Connections that have been opened using such credentials cannot be pooled and reused, thus losing the benefits of connection pooling.
10. Avoid the use of forced data conversion, sometimes referred to as variable coercion or casting, which may yield unanticipated results. This occurs when two or more variables of different data types are involved in the same expression. When it is necessary to perform a cast for other than a trivial reason, that reason should be provided in an accompanying comment.
11. Be specific when declaring objects, such as ADODB.Recordset instead of just Recordset, to avoid the risk of name collisions.
12. Require the use Option Explicit in Visual Basic and VBScript to encourage forethought in the use of variables and to minimize errors resulting from typographical errors.
13. Avoid the use of variables with application scope.
14. Use RETURN statements in stored procedures to help the calling program know whether the procedure worked properly.
15. Use early binding techniques whenever possible.
16. Use Select Case or Switch statements in lieu of repetitive checking of a common variable using If…Then statements.
17. Explicitly release object references.

Data-Specific

1. Never use SELECT *. Always be explicit in which columns to retrieve and retrieve only the columns that are required.
2. Refer to fields implicitly; do not reference fields by their ordinal placement in a Recordset.
3. Use stored procedures in lieu of SQL statements in source code to leverage the performance gains they provide.
4. Use a stored procedure with output parameters instead of single-record SELECT statements when retrieving one row of data.
5. Verify the row count when performing DELETE operations.
6. Perform data validation at the client during data entry. Doing so avoids unnecessary round trips to the database with invalid data.
7. Avoid using functions in WHERE clauses.
8. If possible, specify the primary key in the WHERE clause when updating a single row.
9. When using LIKE, do not begin the string with a wildcard character because SQL Server will not be able to use indexes to search for matching values.
10. Use WITH RECOMPILE in CREATE PROC when a wide variety of arguments are passed, because the plan stored for the procedure might not be optimal for a given set of parameters.
11. Stored procedure execution is faster when you pass parameters by position (the order in which the parameters are declared in the stored procedure) rather than by name.
12. Use triggers only for data integrity enforcement and business rule processing and not to return information.
13. After each data modification statement inside a transaction, check for an error by testing the global variable @@ERROR.
14. Use forward-only/read-only recordsets. To update data, use SQL

INSERT and UPDATE statements.

15. Never hold locks pending user input.
16. Use uncorrelated subqueries instead of correlated subqueries. Uncorrelated subqueries are those where the inner SELECT statement does not rely on the outer SELECT statement for information. In uncorrelated subqueries, the inner query is run once instead of being run for each row returned by the outer query.

ADO-Specific

1. Tune the RecordSet.CacheSize property to what is needed. Using too small or too large a setting will adversely impact the performance of an application.
2. Bind columns to field objects when looping through recordsets.
3. For Command objects, describe the parameters manually instead of using Parameters.Refresh to obtain parameter information.
4. Explicitly close ADO Recordset and Connection objects to insure that connections are promptly returned to the connection pool for use by other processes.
5. Use adExecuteNoRecords for non-row-returning commands.

Stubbing

Use stubbing to both speed development and avoid writing a ton of code just to get something useful to compile. Stubbing is a programming trick that is best illustrated by example.

Suppose your project requires you to display a text-based menu of program features on the screen. The user would then choose one of the menu items and press ENTER, thereby invoking that menu command. What you would really like to do first is write and test the menu's display and selection methods before worrying about having it actually perform

the indicated action. You can do exactly that with stubbing.
A stubbed method, in its simplest form, is a method with an empty body. It's also common to have a stubbed method display a simple message to the screen saying in effect, "*Yep, the program works great up to this point. If it were actually implemented, you'd be using this feature right now!*" Stubbing is a great way to incrementally develop your project. Stubbing will change your life!

Bibliography

Rob Carson(February, 2000). Article – Coding Techniques and Programming Practices.
Link - https://msdn.microsoft.com/en-us/library/aa260844(v=vs.60).aspx#cfr_standards

Brad A(January, 2005). Article – Internal Coding Guidelines.
Link - https://blogs.msdn.microsoft.com/brada/2005/01/26/internal-coding-guidelines/