General Coding Guidelines

Version No.: 1.1

Date: 21-May-2013

Revision History

Version No.	Date	Prepared by / Modified by	Significant Changes
1.1	21-May-2013	QMG	Formatting done as per Org. standards
1.0	23-Jan-2013	QMG	QMS-IT Phase 4 release

Glossary

Abbreviation	Description

Table of Contents

1 Introduction

1.1 Purpose

This document is intended to help developers in creating higher-quality software and quicken the development process with fewer problems. In order to develop high-quality software, attention to quality must be part of the software development process from the beginning to the end. This section discusses quality aspects related to coding.

1.2 Scope

This document provides a larger perspective on the common practice in the process of software construction. The Guidelines described in this document are applicable to any procedural language in any computing environment.

1.3 Intended Audience

Developers

2 Prerequisites to Software Construction

2.1 Problem Definition

A problem definition defines what the problem is without any reference to possible solutions. This is the first prerequisite for the software construction and there should be a clear statement of the problem that the system is supposed to solve.

2.2 Requirements Analysis

Requirements describe in detail what a software system is supposed to do and they are the first step to solution. Paying attention to requirements helps minimize changes to a system after development begins.

2.3 System Architecture

Software Architecture is the high-level part of the software design. And it is the frame that holds the more detailed parts of the design. The quality of the Architecture determines the conceptual integrity of the system and also makes the construction easy.

2.4 Choice of the Programming Language

The choice of programming language in which the system will be implemented affects productivity and code quality in several ways. Careful evaluation of a specific language for a particular project should be done based on the kind of the project with considerations such as dynamic memory use, real-time requirements etc.

2.5 Programming Conventions

The implementation must be consistent with the architecture that guides it and consistent internally. Any large program requires a controlling structure that unifies its programming-language details. Hence, establish programming conventions before programming begins. It is nearly impossible to change code to match them later.

3 Routines

A routine is an individual function or procedure invocable for a single purpose.

The following steps are involved in building a routine

- Decide that a routine is necessary
- Design the routine. Check the design. Repeat the step as many times as necessary.
- Code the routine. Check the code. Repeat the step as many times as necessary. If the design is of poor quality, go back to step 2.

3.1 When to Create a Routine?

Given below is a set of valid conditions to create a routine. The reasons are not mutually exclusive.

3.1.1 Reducing complexity

The single most important reason to create a routine is to reduce a program's complexity. Create a routine to hide information so that you do not have to think about it except when writing that routine.

3.1.2 Avoiding duplicate code

The most popular reason to create a routine is to avoid duplicate code. With code in one place, you save the space that would have been used by the duplicated code. Modifications are easier and more reliable because you need to make the modifications in only one place. The code is likely to be more reliable because you have to check only one place to ensure that it is correct.

3.1.3 Limiting effects of changes

Isolate areas that are likely to change so that the effects of changes are limited to the scope of a single routine or, at most, a few routines. Design so that the areas that are the most likely to change are the easiest to change. Areas likely to change include hardware dependencies, input/output, and business function rules.

3.1.4 Improving performance

You can optimize the code in one place instead of several places. Having code in one place means that a single optimization benefits all the routines that use that code, whether they use it directly or indirectly. Having the code in one place makes it more practical to recode the routine with a more efficient algorithm or a faster, more difficult language such as an assembler.

3.1.5 Making central points of control

It is a good idea to keep control for each task in one place. Control assumes many forms. Knowledge of the number of entries in a table is one form. Control of hardware devices is another. Using one routine to read from a file and another to write to it is a form of centralized control. This is especially useful because if the file needs to be converted to an in-memory data structure the changes only affect the access routines.

3.1.6 Hiding information

It is a good idea to hide information like order of processing of events, and implementation details of data structures. Hiding information provides a valuable level of abstraction that reduces a program's complexity. Routines that hide implementation details of data structures centralize operations on the data structures, reduce the chances of errors in working with the data structures and make it easy to change the data structures.

3.1.7 Promoting code reuse

Code put into modular routines can be reused in other programs more easily than the same code embedded in one larger routine. Even if a section of code is called from only one place in the program and is understandable as part of a larger routine, it makes sense to put it into its own routine if that piece of code might be reused in another program.

3.1.8 Planning for a family of programs

If you expect a program to be modified, it is a good idea to isolate the parts that you expect to change by putting them into their own routines. You can then modify the routines without affecting the rest of the program, or you can put in completely new routines instead.

3.1.9 Making a section of code readable

Putting a section of code into a well-named routine is one of the best ways to document its function.

3.1.10 Improving portability

Use of routines isolates non-portable capabilities, explicitly identifying and isolating future portability work. Non-portable capabilities include nonstandard language features, hardware dependencies, operating-system dependencies, and so on.

3.1.11 Isolating complex operations

Complex operations - complicated algorithms, communications protocols, tricky Boolean tests, operations on complex data, etc. - are prone to errors. If an error does occur, it will be easier to find it if it isn't spread through the code but is contained in a routine. The error will not affect other code because only one routine will have to be fixed - other code will not be touched. If you find a better, simpler, or more reliable algorithm, it will be easier to replace the old algorithm if it has been isolated into a routine. During development, it will be easier to try several designs and use the one that works best.

3.2 Designing the Routine

Represent the routine's design using pseudo-code. Some of the guidelines for using pseudo-code effectively are

- Use English-like statements that precisely describe specific operations.
- Avoid syntactic elements from the target programming language.

Follow the steps given below to arrive at the design for the routine. If the detailed design phase has been done to sufficient detail, many of these steps will already have been done.

3.2.1 Check the prerequisites

Before doing any work on the routine itself, check to see that the job of the routine is well defined and fits neatly into the overall design. Check to be very sure that the routine is really called for.

3.2.2 Define the problem the routine will solve

State the problem the routine will solve. This statement should indicate the following:

- The information the routine will hide
- Inputs to the routine
- Outputs from the routine
- How the routine will handle errors

3.2.3 Name the routine

A routine should have a clear, unambiguous name. If you have trouble creating a good name that usually indicates that the purpose of the routine is not clear. Here are a few guidelines for naming routines.

- For a procedure name, use a strong verb followed by an object, for example, PrintReport(), CalcMonthlyRevenues(), etc.
- In object-oriented languages, you need not include the name of the object in the procedure name because the object itself is included in the call. You invoke routines with statements like Report.Print(), MonthlyRevenues.Calc(), etc.

- For a function name, use a description of the return value, for example, cos(), NextCustomerId(), PrinterReady(), etc.
- Avoid meaningless and wishy-washy verbs. Routine names like HandleCalculation(), ProcessInput(), HandleOutput() do not tell you what the routines do. Sometimes the only problem with a routine is that its name is wishy-washy; the routine itself might be well designed. If HandleOutput() is replaced with FormatAndReportOutput(), you have a pretty good idea of what the routine does.

In other cases, the verb is vague because the operations performed by the routine are vague. The routine suffers from a weakness of purpose, and the weak name is a symptom. If that is the case, the best solution is to restructure the routine and any related routines so that they all have stronger purposes and stronger names that accurately describe them.

3.2.4 Decide how to test the routine

As you are writing the routine, think about how you can test it. This is useful for you when you are planning for unit testing and integration testing.

3.2.5 Think about efficiency

Depending on your situation, you can address efficiency in one of two ways. In the first situation, in the vast majority of systems, performance isn't critical. In such a case, see that the routine is well modularized and readable so that you can improve it later if you need to. If you have good modularization, you can replace a slow high-level language routine with a better algorithm or a fast assembly language routine and you will not affect any other routines.

In the second situation, in the minority of systems, performance is critical and the design should indicate how fast a routine should be, and, maybe, how much memory it can use. Design your routine so that it will meet the space and speed goals. If either space or speed seems more critical, design so that you trade space for speed or vice versa.

3.2.6 Research the algorithms and data structures

The single biggest way to improve both the quality of your code and your productivity is to reuse good code. Many algorithms have already been invented, tested, discussed in literature, reviewed and improved. Rather than spending your time reinventing something, spend a few minutes to look through an algorithms book or your local reusable library to see what is already available.

3.2.7 Write the pseudo-code

You might not have much to write after you finish the preceding steps. The main purpose of the steps is to establish a mental orientation that is useful when you actually write the routine.

Once you have written the pseudo-code, take a minute to review it. Back away from it and think about how you would explain it to someone else. Try as many ideas as you can in the pseudo-code, before you start coding.

3.3 Coding the Routine

The following steps are involved in coding a routine.

3.3.1 Write the routine declaration

Write the routine interface statement. This is referred to by many terms in many languages. For example, function declaration Pascal and C, subroutine declaration in FORTRAN etc.

3.3.2 Turn the pseudo-code into high-level comments

Write the first and last statements of the routine and turn all the pseudo code into comments. At this point, the character of the routine is evident.

3.3.3 Fill the code below each comment

Fill the code below each line of the comment. At the end of this, each comment has given rise to one or more lines of code. Each block of code forms a complete thought based on the comment.

3.3.4 Check the code informally

Mentally test each block of the code as you fill it in below the comment. Try to think of what it would take to break that block and then prove yourself that it would not happen.

3.3.5 Clean up the leftovers

When you have finished checking the code for problems, check it for some general characteristics. You can take a number of cleanup steps to make sure that the routine's quality is up to your standards.

- Check the routine's interface. Make sure that all input and output data is accounted for and that all parameters are used.
- Check for general quality. Make sure that the routine does one thing and do it well.
- Check the routine's data. Check for inaccurate variable names, unused data, undeclared variables, and so on.
- Check the routine's control structures. Look for off-by-one errors, infinite loops and improper nesting.
- Check the routine's layout. Make sure that you have used white space to clarify the logical structure of the routine, expressions, and parameter lists.
- Check the routine's documentation. Make sure that the routine is adequately commented and that the comments are accurate. Check for algorithm descriptions, for documentation on interface assumptions and non-obvious dependencies, for justification of unclear or non-standard coding practices, and so on.

3.3.6 Repeat steps as needed

If the quality of the routine is poor, back up to the pseudo-code. High-quality programming is an iterative process.

3.3.7 Compile the routine

After mentally checking the routine, compile it. You will benefit in a number of ways by deferring compiling to as late a point as possible. The main reason is that when you compile new code, there is a strong internal pressure that just one more compile will get the routine right. This leads to hasty, error-prone changes that take more time in the long run. Avoid the rush to completion by not compiling until you have convinced yourself that the routine is right.

Here are some guidelines for getting the most out of compiling your routine:

- Set the compiler's warning level to the pickets level possible.
- Eliminate the causes of all compiler errors and warnings. Pay attention to what the compiler tells you about your code. A lot of warnings often indicate low-quality code.

3.3.8 Write the unit test programs

Develop the unit test programs as per the Unit Test Plan. Follow all the steps and guidelines given above for developing the routine while developing the test programs also.

3.3.9 Submit your code and test programs for a peer review

Reviews of the code can catch errors in a much shorter time than would be spent in discovering the errors through testing.

3.3.10 Test the routine

After the review process has been completed, run the routine step by step using a debugger. Make sure each line executes as you expect it to. You can find many errors by following this simple practice. After stepping through the code using the debugger, test it using the test cases planned earlier. You might have to develop test drivers or stubs to support your test cases.

3.3.11 Remove errors from the routine

Once an error has been detected, it has to be removed. If the routine you are developing is buggy at this point, chances are good that it will stay buggy. If you find that a routine is unusually buggy, start all over again and rewrite it. Do not patch it. Rewriting pays off in the long run.

3.4 Guidelines for creating better Routines

Given below are some general guidelines to write good quality code.

3.4.1 Design routines with high cohesion

Cohesion refers to how closely the operations in a routine are related. A function like sin() is perfectly cohesive because the whole routine is dedicated to performing one function. A function like SinAndTan() has lower cohesion because it tries to do more than one thing. Try to have each routine do one thing and not do anything else. This results in higher reliability.

3.4.2 Design the routines for loose coupling

The degree of coupling refers to the strength of a connection between two routines. Try to design routines with small, direct, visible, and flexible relations to other routines. Good coupling between routines is loose enough that one routine can easily be called by other routines. Try to create routines that depend little on other routines. A function like sin() is loosely coupled because everything it needs to know is passed to it with one value representing an angle in degrees. A function such as InitVars(var1, var2, var3,...,varN) is more tightly coupled because the calling routine virtually knows what is happening inside it. Two routines that depend on each other's use of the same global data are even more tightly coupled.

3.4.3 Keep routines short

A limit of 50 to 60 lines of code per routine makes it easier to understand the routine.

3.4.4 Defensive programming

Check for correctness of data at appropriate points. The best way, of course, is to prevent defects in the first place. Using iterative design, writing pseudo-code before writing the actual code, and having reviews are all activities that help prevent defects.

Assertions: Use assertions to check for correctness of state. An assertion is a function or macro that complains loudly if an assumption is not true. An assertion function usually takes two arguments: a Boolean expression that describes the assumption that is supposed to be true, and a message to print if it is not. The assertions can be left permanently in the code, or they can be removed before the product is released. However, remember not to put any executable code in the assertion, even though the programming language may allow you to do so. This will result in some code vanishing when you remove the assertions.

Garbage Management: A good program should follow "garbage in, nothing out", or "garbage in, error message out". Check input values for routines, and the values of data from external sources, for example, files. Check that the data falls in the allowable range. Make sure that numeric values are within tolerances and strings are short enough to be handled. Comment assumptions about acceptable input ranges in the code.

Exception Handling: Use exception handling to draw attention to unexpected cases. For example, if a case statement interprets 5 types of events, during development, the default case should generate a warning that the situation needs to be fixed. After release, however, it should do something more graceful, like, write a message to an error log file, and terminate cleanly.

Introducing Debugging Aids Early: Debugging aids are assertions, memory checksums, print statements and a variety of other coding practices that can help in debugging. The earlier you introduce debugging aids, the more they'll help.

Plan to Remove Debugging Aids: If You are writing code for commercial use, the performance penalty of debugging aids in size and speed is often prohibitive. So always plan to avoid debugging code in

and out of a program. Some of the techniques include using version control, using a built-in processor, writing your own processor and using debugging stubs

Checking Function return values: Check for error codes after each system call or other function call. If you detect an error, raise a warning or take appropriate action.

4 Data

Much of the power of good data structures manifests itself during requirements analysis and architecture.. For maximum advantage, consider defining major data structures then.

4.1 Guidelines for creating your own data types

- Create types with functionally oriented names. Avoid names that refer to the kind of computer data underlying the type. Use type names that refer to the parts of the real-world problem that the new type represents.
- Avoid predefined types. If there is a possibility that a type may change, avoid using predefined types anywhere but in typedef or equivalent statements.
- Do not redefine a predefined type. This can be confusing.
- Define substitute types for portability. In cases where portability is the main issue, one can define substitutes for the standard types so that on different hardware platforms You can make the variables represent exactly the same entities.

4.2 Variables

- Use each variable for exactly one purpose
- Avoid variables with hidden meanings. That is, avoid having different values of the variable mean different things, for example, PageCount might represent the number of pages printed, unless it equals -1, in which case it indicates that an error has occurred.
- Make sure that all declared variables are used

4.3 Guidelines for Variable Declarations

- Turn off implicit declarations
- Declare all variables
- Use naming conventions
- Check variable names. Use compiler-generated cross-references to spot similar sounding names, and variables declared but not used.

4.4 Guidelines for Initializing Data

Improper data initialization is one of the most fertile sources of error in computer programming. Developing effective techniques for avoiding initialization problems can save a lot of debugging time.

- Check input parameters for validity
- Initialize each variable close to where it is used
- Pay special attention to counters and accumulators. Always remember to reset counters and accumulators before they are used a second time.
- Check the need for re-initialization. A variable may be used many times in a loop, or a variable may retain its value across calls to the routine.
- Initialize named constants once; initialize variables with executable code
- Initialize each variable as it is declared. Although not a substitute for initializing variables close to where they are used, initializing variables as they are declared is an inexpensive form of defensive programming. If you make a habit of it, it will avoid many initialization problems.

- Listen to the compiler's warning messages. Many compilers give a warning when an un-initialized variable is used.
- Use a memory-access checker to check for bad pointers
- Initialize working memory at the beginning of Your program

4.5 Guidelines for Data Naming

- The name must fully and accurately describe the entity the variable represents
- Names should be neither too long, nor too short. 8 to 16 characters are appropriate name lengths.
- Short variable names like "i" may be used when the value will not be used outside a few lines of code, and the variable does not represent anything significant.
- If the variable contains computed values like totals, averages, maximum, minimum, etc., put the modifier like Ttl, Sum, Max, Min, Avg, etc. at the end of the variable name, for example, RevenueTtl, ExpenseAvg. An exception to this rule is the use of Num. Placed at the beginning of a variable name, Num refers to a total. NumSales is the total number of sales. Placed at the end of a variable name, Num refers to an index. SaleNum is the number of the current sale. It is better not to use Num, and instead use Count to refer to a total and Index to refer to a specific item.
- Simple loop counters may have simple names like i, j, k. If the variable is to be used outside the loop, it should be given a more meaningful name than j, k, or I.
- Think of better names than "flag" for status variables. For clarity, flags should be assigned values and their values should be tested with enumerated types, named constants or global variables that act as named constants. Thus,
- if (RecalcNeeded == TRUE)
- Is much clearer than
- if (ComputeFlag == 1)
- Name temporary variables more meaningfully than temp, tmp, etc., except when the temporary variable is used as temporary storage to swap two values, etc.
- Name Boolean variables names that imply TRUE or FALSE. Names like "done" and "success" are good Boolean names because the state is either true or false; something is done or it is not; it is a success or it is not. Names like Status and Source file are poor Boolean names because they are not obviously True or False.
- Use positive Boolean names. Negative names like NotFound, NotDone are difficult to read when they are negated.
- When you use an enumerated type, ensure that it is clear that all members of the type belong to the same group by using a prefix or suffix. For example, members of an enumerated type COLOR might be COLOR_RED, COLOR_GREEN, COLOR_BLUE.
- When naming constants, name the abstract entity that constant represents, rather than the number the constant refers to. Thus, CYCLES_NEEDED is better than FIVE.
- Identify variable that is used to return values and always initialize variables that return values especially for data types which have default values

4.6 Naming Conventions

Naming conventions are worthwhile

- When multiple people are working on a project

- When you plan to turn your work over to another person for modifications and maintenance
- When your programs are reviewed by another person
- When your program is very large

4.6.1 Guidelines for naming conventions

- Identify global variables.
- Identify module variables. Identify a variable that is used by several routines within a module. Make it clear that the variable is not a local variable, and it is not a global variable either.
- Identify type definitions.
- Identify named constants.
- Identify enumerated types.
- Identify input-only parameters in languages that do not enforce them.
- Format names to enhance readability.

4.6.2 Kinds of Names to Avoid

- Avoid misleading names or abbreviations
- Avoid names with similar meanings
- Avoid variables with different meanings but similar names
- Avoid names that sound similar, for example, wrap and rap
- As far as possible, avoid numerals in names
- Avoid misspelled words in names
- Avoid words that are commonly misspelt in English
- Do not differentiate names solely by capitalization
- Avoid the names of standard library routines and predefined variables
- Do not use names that are totally unrelated to what the variable represents
- Avoid names containing hard-to-read characters

4.6.3 Global/Static Variables

Global variables are accessible everywhere in a program. Global variables have some advantages as well as disadvantages. Use global variables judiciously.

4.6.4 Common problems with Global data

- Inadvertent changes to global data. Some routine might change the value of a global variable as a side effect and that is easily forgotten.
- Re-entrant code cannot be written with global data
- Code-reuse is hindered by global data
- Global data weakens a program's modularity

4.6.5 Reasons to use Global Data

- Preservation of global values, for example data that every routine in the program uses, a variable that reflects the state of the program, such as interactive vs. batch mode.
- Substitution for named constants
- Streamlining use of extremely common data. For example, if a variable appears on the parameter list of many routines, it may be useful to make it a global variable.

- Eliminating tramp data. Sometimes you pass data to a routine merely so that it can pass the data to another routine. When the routine in the middle of the chain does not use the data, the data is called "tramp data". Global variables eliminate tramp data.

4.6.6 How to Reduce Risks of Using Global Data

- Begin by making each variable local and make variables global only as you need to
- Make all variables local to individual routines initially. If you find they are needed elsewhere, make them module variables first. If you finally find that you have to make them global, do it, but only when you are sure you have to.
- Distinguish between global and module variables
- It is okay to access a module variable any way you want to within the set of routines that use it heavily. If other routines need to use it, provide the variable's value by means of an access routine.
- Develop a naming convention that makes global variables obvious
- Create a well-annotated list of all your global variables

5 Code Organization

Code organization influence code quality, correctness, readability, and maintainability. When statements have dependencies that require you top put them in a certain order, take steps to make the dependencies clear. Some guidelines are given below:

- Organize code so that dependencies are obvious
- Name routines so that dependencies are obvious
- Use routine parameters to make dependencies obvious
- Document unclear dependencies

5.1 Control Structures

Control structures are a big contributor to overall program complexity. Poor use of control structures increases complexity: good use decreases it.

5.1.1 Guidelines for writing if statements

- Write the nominal path through the code first; then write the exceptions
- Make sure that You branch correctly on equality
- Put the normal case after the if rather than after the else
- Test the else clause for correctness

5.1.2 Guidelines for writing case Statements

- Keep the actions for each case simple
- Use the default clause only to detect legitimate defaults
- Use the default clause to detect errors

5.1.3 Guidelines for writing Loops

- Enter a loop from one location only
- Put initialization code directly before the loop
- Don’t use a for loop when a while loop is more appropriate
- Use begin and end or {and} to enclose the statements in a loop
- Avoid empty loops
- Make each loop perform only one function
- Make loop-termination conditions obvious
- Avoid code that depends on the loop index’s final value

5.1.4 Guidelines for writing Recursions

- Make sure the recursion stops
- Use safety counters to prevent infinite recursion
- Limit recursion to one routine

6 Layout and Style

Layout is a useful clue to the structure of a program. Layout affect how easy it is to understand the code, review it, and revise it months after it is written. Over the life of a project, attention to such details makes a difference in the initial quality and the ultimate maintainability of the code. Some of the Layout techniques include White Space, Grouping, Blank lines, Alignment, Indentation and parentheses.

6.1 Guidelines for Laying out Control structures

- Avoid unintended begin-end pairs
- Avoid double indentation with begin and end
- Use blank lines between paragraphs
- Format complicated expressions, put separate conditions on separate lines

6.2 Guidelines for Laying Out Individual Statements

- Use spaces to make logical expressions readable
- Use spaces to make array references readable
- Use spaces to make routine arguments readable

6.3 Guidelines for Laying Out Data Declarations

- Use alignment for data declarations
- Use only one data declaration per line

6.4 Guidelines for Laying Out Comments

- Indent a comment with its corresponding code
- Set off each comment with at least one blank line

6.5 Guidelines for Laying Out Routines

- Use blank lines to separate parts of a routine
- Use standard indentation for routine arguments

6.6 Guidelines for Laying Out Files, Modules and Programs

- Put one module in one file
- Separate routines within a file clearly
- If there are more than one logical module in a file, identify each module clearly
- Sequence routines alphabetically

7 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

Following are the recommended commenting techniques:

- When modifying code, always keep the commenting up to date
- 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.
- Avoid adding comments at the end of 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 the common tab stop.
- Avoid using clutter comments, such as an entire line of asterisks. Instead, use white space to separate comments from code.
- Avoid surrounding a block comment with a typographical frame. It may look attractive, but it is difficult to maintain.
- Prior to deployment, remove all temporary or extraneous comments to avoid confusion during future maintenance work.
- If You need comments to explain a complex section of code, examine the code to determine if you should rewrite it
- Use complete sentences when writing comments
- Comment as You code, because most likely there would not be time to do it later
- Avoid the use of superfluous or inappropriate comments, such as humorous sidebar remarks
- Use comments to explain the intent of the code.
- To prevent recurring problems, always use comments on bug fixes and work-around code, especially in a team environment.
- Use comments on code that consists of loops and logic branches
- Throughout the application, construct comments using a uniform style, with consistent punctuation and structure.

8 Code Review

Code should be submitted for peer reviews before unit testing is started. Both the program code and the test programs should be reviewed. The code reviews can detect a number of bugs quickly, thus saving testing time. It is important to review test programs also, because that reduces the amount of time that needs to be spent on debugging the test programs.

The main objective of code reviews is to ensure that the code correctly implements the design. In addition, the review should also ensure that all the coding practices recommended in the previous subsections have been followed.

8.1 Guidelines for Conducting Code Review

General	As you complete each review step, check that item in the box to the right. Complete the checklist for one program unit before you start to review the next.
Complete	Verify that the code covers all the design
What to check in a code?	
Coding Guidelines	Check that Coding Guidelines has been followed.
Includes	Verify that includes are complete.
Initialization	Check variable and parameter initialization: • at program initiation • at start of every loop • at function/procedure entry
Calls	Check function call formats: • pointers • parameters • use of '&'
Names	Check name spelling and use: • Is it consistent? • Is it within declared scope?
Strings	Check that all strings are • identified by pointers and • terminated in NULL
Pointers	Check that • pointers are initialized NULL • pointers are deleted only after being dynamically allocated, and • dynamically allocated pointers are always deleted after use

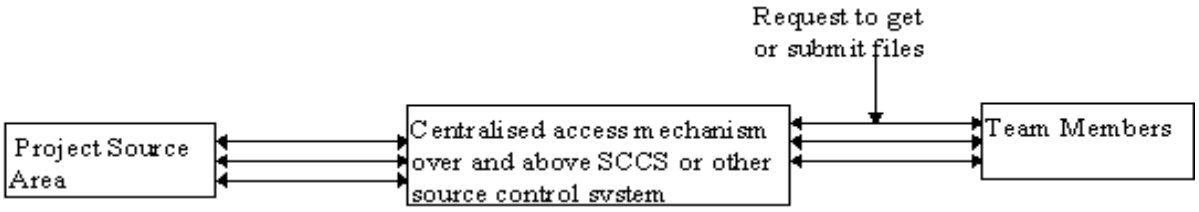
Output Formats	Check the output format: • line stepping is proper • spacing is proper
{} Pairs	Ensure that the {} are proper and matched Logic Operators Verify the proper use of ==, =, ||, and so on. Check every logic function for proper ().
Line by Line Check	Check every LOC for • instruction syntax and • proper punctuation
Standards	Ensure that the code conforms to the coding standards.
File Open and Close	Verify that all files are • properly declared, • opened, and • closed
Error Handling	Ensure that Error Handling has been done wherever necessary in the code.

9 Project Directory Organization Conventions

Before embarking on coding it is useful to spend some time and think about how to organize the source files and other files related to the project. One way is to keep the source file organizations very close to the way the software is structured. It is also safe and advisable to keep the project files separate from the work area of the team members.

Keeping these points in view the following guidelines are proposed:

- Identify the machine on which the files will be maintained. It is better to maintain all the files at a single place with the help of some source code control tool, even though work on the product may be undertaken on multiple platforms.
- Identify the HOME directory on the machine for the project source area and make sure adequate disk space is available. The project HOME directory should have the name /xxx/yyy/project where /xxx/yyy represents any path on the machine chosen.
- Create a login-id for each team member on the source machine.
- The work area or the HOME of individual team members should not be in the above source path.
- Create a group-id having the name project on the machine on which the files are to be maintained.
- Make every team member part of the project group.
- Provide a centralized access approach to the source area to all team members. A centralized access approach will help to maintain source integrity. It is useful to develop tools to achieve such a centralized access.



9.1.1 Directory Naming Conventions

Directory names should be made up of alphanumeric characters and the underscore character. The name should begin with an alphabet.

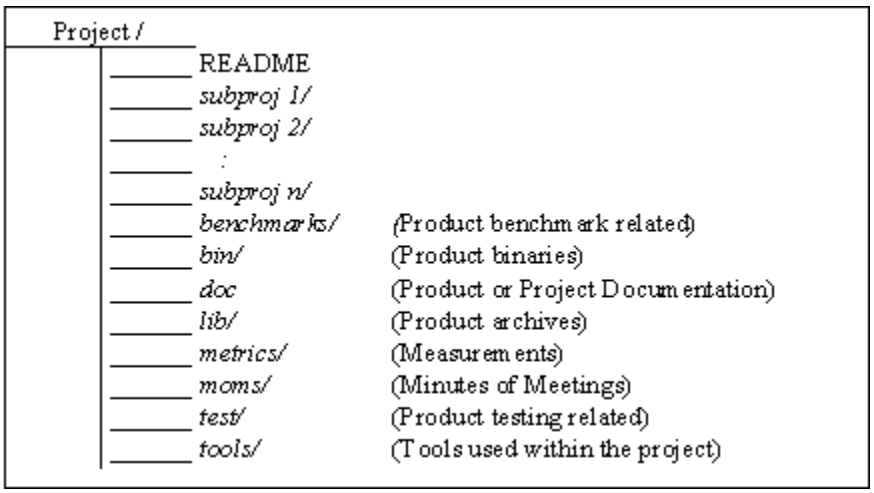
Directory names should be in lower case. However the first character of each word in a multi-word file or directory name can be in upper case (this convention is part of the notation known as "Hungarian Notation"). Directory names should be within 14 characters. Avoid use of special symbols in directory names. An exception is the underscore character, which may be used to separate words in a multi-word name. Always use a name that is unique in as large a context as possible.

9.1.2 Recommended Directory Structure

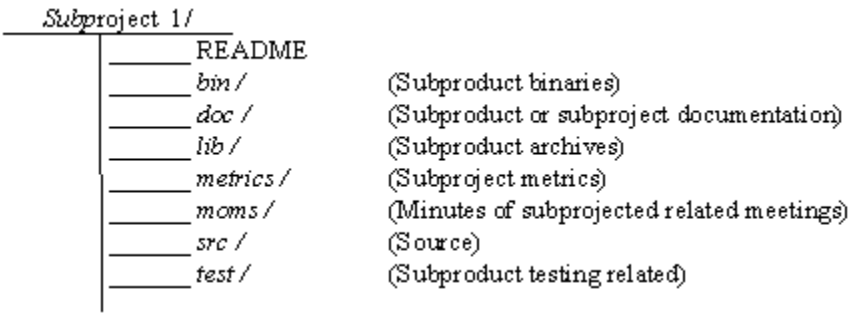
The project area should be organized such that

- Related files can be accessed easily
- Related files are grouped together
- Workspaces of different subprojects are separate

Usually, the directory structure is quite parallel to the product architecture. One possible way of organizing the directories is shown below.



subproj1 to *subprojn* represent subprojects of the main project. Each subproject's directory can further be organized, for example, as shown below:



Each src directory can be structured such that it reflects the design architecture of the product or subproduct. The directories and files in the src directory can be named after the entities identified in the design process.

Each project group should define the directory structure it follows and document it in the highest-level README file.

9.2 Source Code Organization Conventions

9.2.1 Source File and Directory Naming Conventions

UNIX file and directory names should be made up of alphanumeric characters and the underscore and dot character. The name should begin with an alphabet. File and directory names should be in lower case. However the first character of each word in a multi-word file or directory name can be in upper case (this convention is part of the notation known as "Hungarian Notation"). Directory names should be within 14 characters. Avoid use of special symbols in both file and directory names. An exception is the underscore character, which may be used to separate words in a multi-word name. Always use a file name that is unique in as large a context as possible.

DOS file and directory names should be made up of alphanumeric characters, underscore and dot. DOS is not case-sensitive. Directory and file names can be up to 12 characters long including a dot and a 3-character suffix.

File names typically contain a base name and a suffix. Some tools affix additional suffix/prefix characters, for example, SCCS. Typically such prefixes/suffixes take up two additional characters including the ".", for example, ".p", ".s". Taking these into consideration and keeping in view the limit in most UNIX systems (of 14 character names), it is advisable to limit file names to 12 characters including the basic suffices required for source files. Typically basic suffixes are up to 3 characters long on UNIX systems and up to 4 characters long on DOS systems, including the dot.

For a C++ source file which implements a class, the base name of the file can be the class name itself or an abbreviation of the class name if this name is longer than the limits discussed above.

The following are the suffixes commonly used for various kinds of files:

C source files	.c
UNIX C++ source files	.C
C++ inline function	I.h
'lex' input files	.l
'yacc' input files	.y
C and C++ header files	.h
Externs file	.e
Make files	.mk
UNIX object files	.o
UNIX object archive files	.a
UNIX assembly files	.s
DOS object files	.obj
DOS header files	.h or .inc
DOS C++ source files	
DOS object archive files	.lib
DOS assembly files	.asm
Shared objects in SVR4.2	.so
Shared objects in HP-UX	.sl

9.2.2 Recommended Organization of Source Files

Every individual involved in the coding step will be coding an "entity" at a time. Before starting to code, it will be useful if some time is spent in trying to organize or structure the source code into different files.

An entity may have different functional units, or it may perform different types of activities. It is quite useful to organize the source code into different files such that each file contains code that implements one function or activity. It is again quite useful to put together all "support code" for a particular function or activity in the same file.

Such a file organization leads to highly modular structure and considerably helps in cataloguing code for re-use. This also helps achieve the right type of cohesion and coupling of the modules.

Let us consider the example of a FORTRAN compiler. The language constructs are divided into broadly two types of statements viz., "EXECUTABLE" statements and "NON-EXECUTABLE" statements. "EXECUTABLE" statements can be further classified into "ASSIGNMENT" statements,

"IO" statements and so on. Among "EXECUTABLE" statements "EXPRESSIONS" may be a common feature occurring throughout. In such a situation it will be meaningful to divide the code into different files like "expressns.c", "assign.c", and "io.c", etc.

A little bit of time spent in the beginning of the coding step in organizing/planning will go a long way in clearly identifying common code that may be required, identifying dependencies, identifying the job steps needed to complete coding of an entity and to sequence and prioritize the coding activity.

Another point to consider in organizing the source file is the limit on the length of the source file. Although there is no maximum limit on the size of a file, files should generally not have more than 1000 lines. Files with more than 1000 lines are cumbersome and difficult to deal with. One may encounter practical problems with various utilities for example, editor not having sufficient temporary storage, compilations being slow etc. Also when coding, care should be taken to ensure that individual functions are not more than 4-60 lines of code and that no line goes beyond 72 characters. Lines longer than 72 characters may not be handled properly by all terminals and they may be difficult to understand.

Place all machine- or operating system-specific code in one file so that the code can be easily changed when porting the code to another machine or operating system.

When a single source file has to be split because the number of lines exceeds 1000, then the functions in the source file should be judiciously regrouped into a number of files so that closely related functions are in the same file. In this case the names of the files should be composed of an initial part (class name or entity name or an abbreviation of these) and possibly a trailing part reflecting the classification of the functions in the file.

10 Code File Header

10.1 .NET code file header

```
/// Method / Module Name            : <Name of the method / module>
/// Module Purpose(brief description)     : <Purpose of the module>
/// Created By                      : <Name of the Developer and ERP code >
/// Created On                      : <Creation Date in DD-Mon-YYYY format>
/// Reviewed By                     : <Name of the reviewer and ERP code >
/// Reviewed On                     : <Review Date in DD-Mon-YYYY format >
///
/// Modified By                     : <Name of the developer and ERP code >
/// Modified On                     : <Modified Date in DD-Mon-YYYY format>
/// Reviewed By                     : <Name of the reviewer and ERP code >
/// Reviewed On                     : <Review Date in DD-Mon-YYYY format>
/// Purpose for modification        : <Purpose of modification>
```

10.2 SQL code file header

```
/*
Project Name                        : <Project Name as in ERP>
Project Code                        : <Project Code as in ERP>
CR Number / Issue Number            : <SSD No>
File Name                           : < Name of the SQL object,
                                    databasename.schema.objectname>
Purpose                             : <Purpose of creating the object>
Input Parameters                    : <Input parameters, if any>
Output parameters                   : <Output parameters, if any>
Created By                          : <Name of the developer and ERP code>
Created On                          : <Created date in DD-Mon-YYYY format >
Reviewed By                         : <Name of the reviewer and ERP code>
Reviewed On                         : <Review Date in DD-Mon-YYYY format >
Modified By                         : <Name of the developer and ERP code >
Modified On                         : <Modified Date in DD-Mon-YYYY format>
Reviewed By                         : <Name of the reviewer and ERP code >
Reviewed On                         : <Review Date in DD-Mon-YYYY format >
Purpose for modification            : <Purpose of modification>

*/
```