General Style and Coding Standards for Software Projects

1. INTRODUCTION

1.1 PURPOSE

The goal of these guidelines is to create uniform coding habits among software personnel in the group project so that reading, checking, and maintaining code written by different persons becomes easier. The intent of these standards is to define a natural style and consistency, yet leave to the project team, the freedom to practice their craft without unnecessary burden.

When a project adheres to common standards many good things happen:

- Programmers can go into any code and figure out what's going on, so maintainability, readability, and reusability are increased. Code walk throughs become less painful.
- New people can get up to speed quickly.
- People new to a language are spared the need to develop a personal style and defend it to death.
- People new to a language are spared making the same mistakes over and over again, so reliability is increased.
- People make fewer mistakes in consistent environments.
- Idiosyncratic styles and college-learned behaviors are replaced with an emphasis on business concerns - high productivity, maintainability, shared authorship, etc.

Experience over many projects points to the conclusion that coding standards help the project to run smoothly. They aren't necessary for success, but they help. Most arguments against a particular standard come from the ego. Few decisions in a reasonable standard really can be said to be technically deficient, just matters of taste. So, in the interests of establishing the engineering department as a showcase software development environment, be flexible, control the ego a bit, and remember any project is a team effort.

A mixed coding style is harder to maintain than a bad coding style. So it's important to apply a consistent coding style across a project. When maintaining code, it's better to conform to the style of the existing code rather than blindly follow this document or your own coding style.

Since a very large portion of project scope is after-delivery maintenance or enhancement, coding standards reduce the cost of a project by easing the learning or re-learning task when code needs to be addressed by people other than the author, or by the author after a long absence. Coding standards help ensure that the author need not be present for the maintenance and enhancement phase.

1.2 Scope

This document describes general software coding standards for code written in any text based programming language (including high-level languages like C, C++, Basic, Visual Basic, and assembler languages). Many of the guidelines described in this document can be directly applied to programming practices in graphical based languages (such as PLC, graphical, and visual languages). This will be used as the base document for several language specific coding standard documents. Each language specific coding standard will be written to expand on these concepts with specific examples, and define additional guidelines unique to that language.

1.3 Terms Used In This Document

- The term "**program unit**" (or sometimes simply "unit") means a single function, procedure, subroutine or, in the case of various languages, an include file, a package, a task, a Pascal unit, etc.
- A "**function**" is a program unit whose primary purpose is to return a value.
- An "**identifier**" is the generic term referring to a name for any constant, variable, or program unit.
- A "**module**" is a collection of "units" that work on a common domain.

1.4 References

a. General Coding Standards, Author (s) unknown
b. Steve McConnell, Code Complete, Microsoft Press 1996 (ISBN 1-55615-484-4)
c. Writing Solid Code, Steve McConnell, Microsoft Press 1996 (ISBN 1-55615-551-4)
d. A Modest Software Standard, Jack G. Ganssle, March 1996 Embedded Systems Programming
e. Recommended C Style and Coding Standards, Indian Hill C Style and Coding Standards Updated by authors from Bell Labs, (http://www.cs.huji.ac.il/papers/cstyle/cstyle.html)
f. C coding Guidelines, Real Time Enterprises, Inc. Rev 7, 6/24/96
g. C++ Guidelines (companion document to C coding guidelines), Real Time Enterprises, Inc., Rev 0 11/8/1995
h. C style and Coding Standards for the SDM Project, ROUGH DRAFT, July 3, 1995 (http://www-c8.lanl.gov/sdm/DevelopmentEnv/SDM_C_Style_Guide.hyml
i. Software Design and Coding Standards for C++, Authors Unknown., 7/7/1994
j. C Programming Standards and Guidelines, Internal Document 8/10/93, Software Warehouse index # 100
k. C++ Style Guide, Internal Document, 6/22/90, Software Warehouse index # 201
l. Using Visual Basic for Applications (Appendix D- Style Guide for Professional Quality Code), QUE, (ISBN 1-56529-725-3)
m. Speaking the Language of the PM API, Part 4 (Overview of Hungarian Notation), PC Magazine March 14, 1989)
n. Programming Integrated Solutions with Microsoft Office, Appendix B, Visual Basic Variable Naming and Coding Standards.

1.5 The Emotional Topic of Coding Standards

Please be patient with these coding standards until they become natural... it is only then that an honest opinion as to correctness or utility can be formed. They need not impede the feeling of craftsmanship that comes with writing software. Consider the common good. Embrace the decisions of the group.

2 Constants

2.1 Use constants instead of hard coded literal values

- Literal values shall be avoided in code statements; rather, a symbolic constant for each shall be defined reflecting its intent. 0 should not be assumed to mean “OFF”, a symbolic constant OFF should be defined to be equal to 0.
- The numeric literals 0 and 1 shall not be used as Boolean constants. Booleans are not to be treated as integers.

The exceptions to this rule include

- The numeric literals 0 and 1 (where their use is to initialize, increment or to test).
- Certain fixed-purpose character literals (e.g., ‘ ‘ will always be a blank), and strings giving messages or labels.

2.2 Only Define constants once

Constants shall not be defined in more than one textual location in the program, even if the multiple definitions are the same.

3. Code Layout

3.1 One statement per line

- There shall normally be no more than one statement per line; this includes control statements such as “if” and “end” statements.
- When a single operation or expression is broken over several lines, break it between high-level components of the structure, not in the middle of a sub-component. Also, place operators at the

ends of lines, rather than at the beginning of the next line, to make it clear at a glance that more is coming.

3.2 Error Handling

- Functions that can fail (i.e. file I/O) should always return a success or error as a return code parameter.

4. Naming Convention for Identifiers (Variables, Constants)

4.1 Select Clear and Meaningful Names

The most important consideration in naming a variable, constant is that the name fully and accurately describes the entity or action that the structure represents. Clear, complete, and meaningful names make the code more readable and minimizes the need for comments.

4.2 Use of camel casing for naming function variables and functions

Camel case is the practice of writing phrases without spaces or punctuation. It indicates the separation of words with a single capitalized letter, and the first word starting with lower case letter.

4.3 Constants

Fixed identifiers such as Constants shall contain underscores between words and use all UPPER case letters.

4.4 Abbreviations

An abbreviation shall only be considered if it saves a considerable number of characters (e.g., “FFT” for “Fast Fourier Transform” is acceptable, but “Snd” for “Send” is not), as long as the language does to restrict identifier lengths severely. An abbreviation list shall be created during the design phase of each project. However, smaller groups of programmers may create their own abbreviations for terms which are used within the domain of the code to which they are assigned; again, the use of these abbreviations shall be consistent. Any abbreviations which are on the list must be used by all programmers for any identifiers which include the corresponding phrase. For example, if series of procedures sends various types of messages using the identifiers Send_Hello_Msg, Send_Connect_Msg, and Send_Data_Msg, then the name of a new procedure in the series should not be Send_Disconnect_Message. To do otherwise simply encourages coding errors and frustrates text searches.

5. Misc. Rules for Coding

5.1 Conditionals and comparisons

Always test floating-point numbers as <= or >= relational operator, never use exact comparisons (= = or !=).

No assumptions shall be made about the value of uninitialized variables, unless the language definition makes a clear statement about this.

5.2 Strive to develop clear code

Team should strive to develop code that is both clear, and efficient in its use of CPU time, memory, and other resources. However, when efficiency and clarity conflict, then clarity should take strong precedence over resource stinginess, unless it is proven that using the clear but less efficient method impairs the program critically. Micro-optimizations to small areas of code are especially to be avoided if they impair clarity in any way, since it is generally only the program's overall algorithms that affect resource utilization significantly.

5.3 Use libraries when available

Whenever library routines, graphics packages, compiler/assembler features, or other sorts of utilities are helpful to the program, they should be utilized. The danger of losing the access to the utility if the hardware, the compiler/assembler, or the operating system should change is generally overridden by the savings in software creation time.

6. Modularization and Information Hiding

6.1 Information Hiding , domain, and scope of variables

The use of information hiding is mandatory, to the extent allowed by the given language. The overall program shall be divided into various domains of interest, and different divisions of the code shall deal with different domains. The code that deals with a given domain shall protect, to the greatest extent possible in the given language, its data, its data structure design, and its internal operations on the data. It shall "export" to outside code modules only the operations required by the outside modules. The exported operations shall be presented to outside modules at a level of abstraction such that the internal implementation is not detectable.

6.2 Low Coupling , High Cohesion, and Clean interfaces

The quality of the modularization of a program depends on the linkage between modules, called coupling, and the binding within a module, called cohesion. Ideally, a module will have low coupling with other modules and a high level of cohesion with itself.[1]

It may not be easy to attain low coupling, high cohesion, and clean interfaces at the same time, but the final product will be cleaner and easier to maintain because of this extra effort.

6.3 Cohesion

Cohesion refers to the relation of the statements within a routine. There are different ways in which statements can be related. Functional cohesion, which means that the statements perform a single purpose or goal, is the strongest type of binding and the ideal to strive for.[1]

6.4 Coupling

Coupling refers to the degree to which two routines are dependent on each other, or the degree of difficulty one would have trying to change one routine without having to change the other. Data coupling would depend on the number and type of parameters passed into or out of a routine. Loose data coupling (low dependence between modules) is the ideal coupling method.[1]

6.5 Clean Interface

Clean interfaces refers to clear, standard and defined lines of communication between routines. In many languages, this is done by passing parameters. It is important to try to keep communication between procedures and functions confined to parameters, i.e. not referencing global data.

6.6 Minimize scope of variables

Whenever allowed by the language, all constants, types, and variables shall be declared only within the scope in which they need to be known.

Data items must not be accessed or altered by some obscure process. Data should be local if at all possible. "Pass through" parameters (or "Tramp Data"), whose only function is to pass data down to called routines creates readability and maintainability problems. Each data linkage makes integration problems more probable.

7. Merge Requests

Group member assigned as Repo Manager must complete all merge requests.

Adapted from https://cse.buffalo.edu/~rapaport/code.documentation.Excerpts.pdf