Software Coding Guidelines

Last update 30 Aug 2016 [MLS]

General

Coding is the last step in translating a design into an executable program. As such, coding is different from design or documentation. There is an unavoidable amount of overlap among these activities, but good development process helps to minimize this.

Note that the code examples in this document refer to C/C++ language semantics, but the principals apply to any language implementation. Technical specifics may vary, but not the approach.

Code Follows Design

The design document will contain enough information to identify pseudo-classes that are needed to model hardware and software components, the hardware and software interfaces, the data being exchanged, and most of the format of this data.

All software components will offer public access to services and provide a well-defined response to service requests. Parameters and other interchange data for service requests must be identified.

The code will flesh out the design without deviating from the integrity of the design structure. Should you find yourself creating too much code which is at variance from the design then there is a problem with either the code or the design, perhaps both. Well-maintained documents will help sort that out.

Low Level Design in File and Function Document Blocks

The design calls out all of the publically-available interface elements of a component and specifies the data that needs to change hands for the component to do its work. The internal functions of the component, the algorithms involved, and the internal data being exchanged are described in the document blocks for the files and functions that make up the component.

If there is insufficient detail in the design to allow such documentation to take place then the design needs to be refined to address additional detail at the lower level. This process is iterative and not necessarily entirely predictive. This is acceptable; it is rare that a planned implementation will follow its original path without any mid-course corrections during discovery, testing, feature tuning, etc.

The use of document blocks to provide the lowest level of design documentation is the best compromise solution between having insufficient design documentation and having too much inertia and rigidity in the design. It is up to the skill, experience, and good sense of the design/coding team to determine the proper cutoff point between design and coding, just as it was between the architecture and design. As with any layered plan, the design should identify more of the what (higher level) and code more of the how (lower level).

The actual code must closely follow the algorithm or pseudo-code logic expressed in the file or function document block. If the code needs to change then the document block must be updated to reflect the changes. It is not at all hard to keep these miniature "documents" up to date, since they are very terse and are co-located with the code in question.

Code Composition Needs Standards

By applying certain standards to coding the results will conform to a predefined pattern, common to all products. This commonality will assist in the composition, enhancement, and maintenance of the code. Having a common template and format for the code alleviates the typical problems of useless revision

diffs, "format wars," and errors introduced during maintenance.

The composition standard also affects the integration of code composed by multiple developers on the same team or from teams that are geographically isolated. It is not always possible to answer questions about code face to face. Developers may also move on to other projects or leave the firm.

Sometimes requirements will shift during development in such a way that previous code no longer applies to the problem at hand. As long as the file and function document blocks are properly maintained these questions can be answered even without the aid of the original developer.

Having code that all "feels" the same makes for a commonality that will, itself, resolve certain problems that arise during development and maintenance. Therefore, these standards must be universally applied and religiously followed, as appropriate.

File Organization

Source code files are organized around the components that they support. Software components that are based on hardware can model that hardware through objects. The design will have these pseudo-classes spelled out.

All external interfaces (function calls) and data types (constant references, macros, symbols, structures, enums, etc.) are contained in a header file with the same base name as the C/C++ source file containing the component. The header file contains only those elements needed by external modules to use the component. All of the interface elements needed to use a component will be identified in the design. If the coding process reveals the need for a change in the interface then the design is at fault and needs to be reevaluated.

No elements (variables, pseudo-classes, defines, etc.) are shared between C source files without being an explicit part of the design. It is not permissible to make a local variable into a global just to eliminate a problem in sharing information. Internal data types, symbols, and other elements are defined in the C source where they are used. These elements are created as necessary and are typically not defined in the design.

Each file begins with a documentation block that contains the file name, summarizes the purpose of the module in a very brief description, and shows a VCS-generated maintenance log. A typical documentation block for a source code file looks like this:

```
/* FILENAME.EXT - very terse description


Summary:
This file contains some really useful stuff, briefly described in this and perhaps one
or two more lines. [This should come right out of the design.]


$VCS_SYMBOL_START$  [This is determined by the VCS in use.]

Rev      date      dev    notes
--------------------------------
1.00    01Jan94    MLS    original code
1.01    11Feb94    MLS    changed FunctionTwo() to fix something
                          added FunctionThree() to fix something else
1.10    22Mar94    MLS    changed FunctionTwo() back to previous condition

$VCS_SYMBOL_END$  [This is determined by the VCS in use.]
*/
```

Code files should be ordered as follows:

- System includes
- Project includes
- Manifest constants and other symbol definitions
- Type definitions
- Type constants (Note that constants actually assign storage space, so must be placed into C source files. Headers having shared constants can only contain references.)
- Function prototypes
- Code (for C source)

This ordering provides a buildup from the lowest level of definition to the highest. Example:

```
[File documentation block (see above)]

#include <SomeSystemFile.h>
#include "SomeLocalFile.h"

#define NameLength 4

typedef enum {
    1, 2, 3, 4;
}
CountEm;

typedef struct {
    int JustOne;
    char NameTwo[NameLength];
}
PlaceEm;

const int One = 1;
const char Two[] = "Two";

int Plebs(int);
void Nothing(void);


[Function documentation block (see below)]

int Plebs(int Ekmotz) {
    return (Ekmotz + One);
}
```

Note that this order produces some beneficial side effects for debugging. For instance, if a symbol's name is already in use in one of the system include files and one of the project includes tries to use that name then the preprocessor will error on the project include file. If the source file tries to reuse a symbol name the same error will arise.

If the local symbol is declared before the system include file is brought in then the preprocessor will complain about the symbol in the system include, which is obviously not the source of the problem. This solution applies to the types, constants, and function prototypes as well.

File Contents

C Source Files

C source files contain all executable code, module global variable declarations, and constant declarations. All C sources with a public interface have a corresponding header file that declares the public access to the functionality of that module. C sources that are used for internal module support need not have a header file. In that case, their function prototypes must still be declared in the source file where they are used.

Header Files

Header files must contain definitions for anything that can be referenced from outside the C source file: preprocessor macros, function prototypes, data structure definitions, type definitions, constant references (*not* declarations), and pseudo-class definitions. Any of these entities that are used within the module but not shared between C source files should not be put here.

Header files do NOT contain any code (other than macros or inline) or allocate any storage space for data. If the header file depends on information in some other header file then the header files involved must use some form of prior inclusion test, as shown below.

Header files should follow this model:

```
[documentation block (see above)]

// Prior inclusion test
#ifndef Headername__
#define Headername__

#define NameLength 4

typedef enum {
    1, 2, 3, 4;
}
CountEm;

typedef struct {
    int JustOne;
    char NameTwo[NameLength];
}
PlaceEm;

extern const int One;
extern const char Two[];

int Plebs(int);
void Nothing(void);

#endif  // close prior inclusion test
```

The prior inclusion test symbol ends with a double underscore. This convention makes finding inclusion test symbols easier for a regexp search. Note that leading underscores are not allowed, since this violates the ANSI namespace convention.

Functions

Precede each function with a documentation block. Complete the function documentation block as the final step of the detailed design of the module (i.e., before you write the code). Include a pseudo-code description of the algorithm. Include as much detail as is necessary to maintain the code, but no more.

A typical documentation block for a function looks like this:

```
/* FunctionName

A short description of the intended purpose of the function. Summarize why this
function exists. Use perhaps two lines of text to do this. [This should come right out
of the design.]

Input:
  <type> ArgumentOne - Description of first parameter and its range of values.
  <type> ArgumentTwo - Description of second parameter and its range of values.

Return:
  <type> - Possible return values, including error codes

Process:
  First, do this
  Then, do this
  After that, do this
  Last, do this

Magic:
Describe anything that is not intrinsically obvious to the most casual of observers.
Use as much verbiage as it takes, but don't use code to describe anything.
*/
```

Evaluation Layout

Any logic can test for early termination. In C the first failing evaluation (equals false) in a compound logic statement causes the entire statement to fail. Put the highest priority or most likely to fail evaluation first, then others in descending order of priority or likelihood of failure. Do the same thing in a function. The old construct of a single exit point is no longer useful. Exit the function as soon as it becomes clear that processing cannot continue.

Compare these two functions:

```
bool IsThisReady()
{
    bool Result;

    if (Goobers() == Yepper) {
        DoSomething();
        if (Plebs() == Yepper) {
```

```
            DoSomeMore();
            if (Oonyoffs() == Yepper) Result = true;
            else Result = false;
        }
        else Result = false;
    }
    else Result = false;
    Return (Result);
}


bool IsThatReady()
{
    if (Goobers() != Yepper) return (false);
    DoSomething();
    if (Plebs() != Yepper) return (false);
    DoSomeMore();
    if (Oonyoffs() != Yepper) return (false);
    return (true);
}
```

Ultimately, the logic is identical. Which would you rather read? Which would you rather maintain? Note that this has no impact at all on the design, since the same operations take place in the same order. Only the implementation in code changes from one function to the next.

Local Variable Declaration

Declare all local variables at the start of the function. Although most modern C compilers allow the C++ inline declaration syntax, use this sparingly (if at all) as it is non-portable and may cause compile and Lint errors. When the need for new local variables comes up during coding simply add the new variable at the top of the function with the other local variables. This is also a good indicator that a function may be outgrowing its context, and a re-evaluation of the code structure may be in order.

Source Code Format

General

The body of a function should not exceed one printed page (60 lines), including both vertical white space and inline comments. This does not necessarily include the function documentation block. This is a guideline. However, you must explain exceptions to your inspectors in such a way that they agree with the coherence of the code as a single large function. Under most circumstances long functions are composed of multiple coherent entities. These entities would be more clearly expressed as several individual functions.

Statements should not exceed 80 characters in total length. Break statements at the highest logical level that results in lines of less than 80 characters. When plausible, break statements outside of any grouped entities such as array elements, logic inside parens, sub-expressions, etc., and before any logical operators. Put the continuation of the statement on the proceeding line, indented one level. Further continuation stays at that same indentation level.

Code Indentation

Indentation is *[NOTE: Choose a value for this and stick with it. Industry standard is four.]* four spaces per level. Do not use tabs. Replace tabs with the appropriate number of spaces even in legacy code. All modern editors can replace tabs with spaces, so fixing this is trivial. Fixing tabs may generate one cycle of a large diff, but any further revisions will then be much smaller and easier to track.

Indent subordinate sections of code following block structure. Put the opening brace *[NOTE: Choose a standard for this and stick with it. There is no industry standard for this attribute of source code.]* on the end of the line containing the conditional statement. Indent subordinate code with respect to the enclosing braces.

Braces

Braces end a line except for the rare inline comment. Opening braces are at the end of the conditional. Put the closing brace on its own line, vertically aligned with the opening brace. For example:

```
SomeFunction()
{
    for (...) {
        if (...) {
            code();
        }
        else {
            othercode();
        }
    }
}
```

It is all right to use fully bracketed syntax to enclose logically conditional statements (if, do, while, switch, for) even if there is only one statement enclosed by the condition. *[NOTE: Yet another source attribute with no industry agreement.]* However, concise code that uses one-line conditionals is easier to read and maintain.

Switch Statements

Switch case statements indent the same level as the opening switch. *[Indent case statements if this is going to cause fights.]* Each value for a case should be on its own line, even if more than one case value is assigned to the same block of code. Each case block must end in a break unless it falls through to the next case, whereupon a comment annotates the place of the break. Every switch must have a default case value, even if that case only contains a break. Example:

```
switch (Plebs) {
case Dweeb:
    Bloofta();
    Oony = Offs;
    break;
case Dwab:
    Ek = Motz;

    // case falls through

case Dweezle:
case Wheezle:
case Sneezle:
```

```
        YoMomma();
        break;
    default:
        break;
    }
```

Returns

Use parens around return values, even if the value entity being returned is a single variable or constant. Separate the opening paren from the return keyword so that it can be distinguished from a function call by editor automation.

Use of Spaces for Entity Separation

Put blank lines

- before functions. Two blank lines ahead of the beginning of a function doc block makes for much easier reading and visual separation of functions.
- between the declaration of local functions and the beginning of code in a function.
- before block comments and inline comments on their own line. This visually separates important comments without the use of clutter such as long series of dashes or asterisks.

Put spaces

- around keywords, unless they are at the end of the line.
- on each side of operators, except for unary operators. For logical AND (&&) and OR (||) operators use two spaces on each side. This makes the distinction between a logical comparison and a bit melding very clear.
- after every comma. This makes separation of entities easier for both the developer and the IDE search to detect.
- after comment delimiters. Put two or more spaces in front of inline comment delimiters to visually distinguish them from operators.

Do not put spaces

- between function names and open parens.
- between variables and unary operators.
- before semicolons.
- after open and before close parens, brackets, and angle brackets.

Inline Comments

Comments describing variables can be added after the declaration, as follows:

```
bool IsDoorJammed;  // whether the door is jammed
```

However, good variable naming will preclude most usage of this type.

If a piece of code is particularly complex, then succinct inline comments may help clarify the purpose of the code. Describe only what you must, but leave no mysteries to be solved.

Inline comments in code appear before the code block being described, as follows:

```
/* Very tricky hardware usage because of stacked registers. The same I/O address must
be hit three times to get to the register containing the data direction bit. Read the
port twice, discarding the results, then write the value in question. */

Discard = ControlPort;
Discard = ControlPort;
ControlPort.Direction = OUT;
```

Conclusions

Source code is the fulfillment of a long process of refining product requirements into machine instructions. By the time the development team is composing code all of the significant questions about what the product is supposed to do and how it is to do it have already been answered. The code should simply and clearly state the means by which the machine is to implement the design of the product's components. Anything the development team can do to facilitate that is a step in the right direction.

Therefore, return to this and all the other guideline documents frequently to see if there needs to be a change in the way you do your work. When you find a practice that is no longer relevant or needs to be updated for whatever reason then change it here and make your work reflect that changed policy. Remember, having everyone play off the same sheet of music prevents the engineering symphony from becoming a cacophony.