Bob's Concise Coding Conventions (C^3)*

Robert S Laramee†
Visual and Interactive Computing Group
Department of Computer Science
Swansea University
Swansea, Wales, UK
http://cs.swan.ac.uk/~csbob

July 19, 2013

Abstract

We introduce a set of concise coding conventions for general software development. The conventions are meant to be simple and concise and fit on one side of paper for ease of use. They represent the most essential rules to follow for implementing a large project. They're written with the C++ programming language in mind, but they are general enough to be applied to any imperative, object-oriented programming language. We also provide the background behind each rule including a description of where each comes from and why it was selected with pointers to further reading. This is followed by a description providing the main motivation behind introducing the conventions, namely, *Bob's Theory of Software Redevelopment*. This theory outlines a typical software development process that repeats itself in an essentially never ending cycle. The presented coding conventions are meant to serve as a tool to combat this unfortunate cycle and contribute to the *success* of a project.

keywords: software development good practice, software implementation, programming

*Started on 23 Oct 2007. Last updated on July 19, 2013.

†r.s.laramee@swansea.ac.uk

1 Bob's Concise Coding Conventions

The coding conventions are as follows:

1. All methods are 75 lines or less. All methods should be visible on a single screen/page. It should be possible to see the whole method from start to finish without scrolling.
   Exception(s): Methods with case tables (switch statements) and perhaps the main method.
2. No methods shall use more than *five* levels of indentation.
   Exception(s): none
3. No line of code shall exceed 80 characters. It should *not* be necessary to expand the code editor to the entire screen width in order to read a single line of code.
   Exception(s): none
4. All class variables start with the two character sequence "m_" (as in "member" variable) e.g., m_ClassVariable.
   Exception(s): symbolic constants. Symbolic constants should be written in `ALL_CAPITALS`.
5. All class variables are accessed through accessor methods, i.e. Get() and Set() methods, e.g.,
   `GetClassVariable(),SetClassVariable(int newValue)`.
   Exceptions: none
6. Accessor methods come at the top of both header files and implementation files.
   Exception(s): none
7. All member class variables are private.
   Exception(s): symbolic constants
8. Private methods begin with a lower-case letter whereas public methods begin with an upper-case letter.
   Exception(s): The Java Programming Language
9. In general, methods should not take more than five parameters.
   Exception(s): very rare
10. Do not use numbers in your code, but rather symbolic constants.
    Exception(s): 0 and 1.

2 Comments on the Conventions

1. The longer a method is, the less re-usable and the more difficult it is to modify. Also, the longer a procedure is, the more likely it is to contain bugs and the more difficult it is to debug. By confining the method to one screen, it gives the programmer (at least) a chance to keep track of the variables, i.e., the possible values they may contain, from the beginning to the end of the method. Many engineers resist this rule claiming it causes a performance slow down. However, software that follows this rule is easier to optimize with the help of a profiler [12]. Also, shorter methods are better candidates for inlining. It's poor algorithm or software design that leads to bad performance in general. See Chapter 6 on *Performance* by Dickheiser [1] for a more complete description of why this is such a good (and important) rule.

2. Too many levels of indentation quickly renders code illegible.

3. This is an interesting rule. Lines that are too long are less legible and more difficult to debug. This is because, the longer a line is, the more difficult it is for the eyes to move from the end of one line to the next line. Good publishers use a guideline of approximately 66 characters per line of text (so 80 is generally too much) [15]. Reading becomes more difficult as soon as there are more characters on a line. This is one reason why most newspapers and magazines are multi-column. Furthermore, object-oriented programming requires multiple windows to be open simultaneously. Thus having one window open occupying the entire screen makes the mechanics of the programmer's job much more difficult [14].

4. Class variables should be easily distinguishable from local variables or other types of variables.

5. The use of accessor methods enforces *encapsulation*, an extremely important concept in object-oriented methodology. (See Wirfs-Brock et al. for more on this topic [20].) Accessing member variables with methods makes the implementation easy to change, e.g., a `float` to an `int`. This methodology also prevents unwieldy (or even impossible) search-and-replace operations [4, 14].

   Another advantage of using accessor methods concerns object state. If class variable assignment is performed exclusively through Set() methods, then you can ensure that your objects are always in a valid state. This is due to the fact that Set() methods perform error and bounds checking on the parameters passed to the procedure. Following this convention leads to very robust code.

6. Accessor methods are the most common to use, as such, it is most convenient when they are defined at the "top" of the file or class definition.

7. Keeping class variables private enforces encapsulation. Only the class itself should know about the specific implementation details of its own data [13].

8. It is *very* nice to be able to tell whether a method is private or public simply by looking at it (without having to look it up) [14].

9. The more parameters a method takes, the less re-usable it is. We prefer to have several different implementations of the same method taking different (but only a few) parameters. In general, too many method parameters, say six or more, is indicative of a problem(s) with the software *design*. A long list of parameters probably indicates that changes to the design are necessary, e.g., the introduction of a new class(es) or the re-arrangement of existing classes [14].

10. Using symbolic constants instead of typing numbers into your code makes it much more legible. Maybe the original author of the code knows what the number is, but others may not. Even the original author will eventually forget. Plus, the values of symbolic constants are easy to change. Trying to changing the values of numbers directly in the code causes bugs, especially when the number appears in multiple places [14]. Horstmann articulates this rule as "*Do Not Use Magic Numbers*" [3] and provides a nice explanation as to why in chapter 2 of his book.

3 Bob's Theory of Software Redevelopment

> *"There is never enough time to do it right the first time, but there is always enough time to do it over."*–Unknown

If you ever take on the job of software developer, in either industry or academia, you will find yourself in the following scenario on your first (or second) day of work:

At your first (or second) meeting, your manager provides a general description, with great enthusiasm, of an amazing software project you are to work on. He describes the application, in what seems like a lot of detail when hearing it for the first time. As the meeting evolves, he talks in more detail about the software that you are to work on and all the wonderful features you are to implement. As he talks, you nod your head in agreement–as a general sign of understanding and politeness. At the same time, it sounds complicated, and you wonder how it's all going to work out. At the end of the conversation, your manager says,

> *''Manager: And, what do you think?"*
> *"You: Sounds good!"*

is your reply. The project is big however, so instead of starting from scratch, you are to build upon an existing piece of software.

A little while later, the technician has (hopefully) already set up a computer for you to work on and a jolly colleague shows you how to access the existing software that will form the basis of your project. With some luck, a fellow engineer will give you a quick and flashy introduction to the development environment (IDE) you'll be working with (See the Appendix for an examples.). Then you catch your first glimpse of the existing source code. At first it looks fine and sophisticated, but the more you look at it, the more it strikes:

> *'Holy guacamole!"*

you think to yourself,

> *"How on earth am I supposed to work with this?"*

The source code before your eyes is the most unbelievable, *illegible*, sloppy, careless, and *endless* heap of spaghetti you have ever seen in your life. You are appalled at the lack of care that has been given to the body of existing software code that *you* are supposed to work with.

After catching your breath, the next question that comes to mind is,

> *"How did this happen?"*

Figure 1, left, shows a typical software development cycle presented in the average object-oriented methodology course at university (taken from Wirfs-Brock et al. [20]). The figure on the left depicts four stages of the software engineering cycle described in hundreds (if not thousands) of textbooks on the topic of software development [11, 17, 18, 19]. Every book

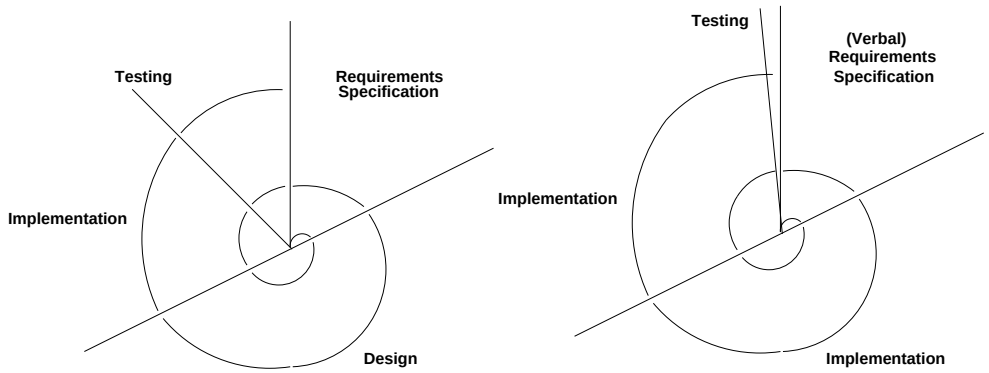

Figure 1: (left) The software development cycle presented in a typical object-oriented software engineering course at university [20], (right) an often-used software development cycle found in industry.

discusses how to write-up a software specification, a design process, how to implement a specification, and how to test the result. Figure 1 shows this as a cycle that iterates over time, visiting each stage repeatedly. There's an emphasis on the Design stage of the cycle, where the authors Wirfs-Brock et al. [20] claim that a lot of the product development time should take place.

Figure 1, right, depicts a software development life cycle that often resembles what happens in practice. The first thing to notice is that the requirements are often given verbally and may never be written down. The second thing to notice is that most of the time is spent on implementation. A third important point is that little-to-no time is spent on design. Many project managers want to see features, and they want to see them as fast as possible. Design is a very fuzzy concept, if it's a concept at all, in the mind of many project managers. It's strictly for academics. Testing is done by the users.

When your at your new software development job, take a good look around you. How many of your colleagues hold a degree in computer science? Do any of your managers hold a CS degree? Our experience in the software industry has lead to the development of *Bob's Theory of Software Redevelopment* which describes the average industry-based software development process as follows.

Stage 1–The Start: Someone, either a software engineer or manager comes up with the idea of a new software product–a new product that promises to be a big success and bring in lots of profits. The idea for the product is expressed *verbally* by someone with a convincing persona. Essentially, a typical industry-based software project starts out with an enthusiastic salesman who sells the idea to someone with the funding to make it a reality.

Stage 2–The Implementation: Amongst all of the excitement inspired by prospects of big success, the implementation starts *immediately*. And the implementation is usually lead by *one* or perhaps two software developers. The fact the implementation is done by only one or two people helps to create the illusion that this person (or team of two) is very important. Neither the lead software developer nor the manager of the project have a degree in computer science. The lead (or sole) developer has taken a few programming classes and the manager has a background in marketing, economics, or finance. The lead developer(s) works *hard* on the implementation and the release date for version 1.0 is already set to be one year after the start of the implementation.

The implementation starts off fine, in line with everything that the lead engineer learned in his programming course and consistent with his previous experience. He reports to the

management that everything is proceeding nicely.

Stage 3–One Year Later, Version 1.0: Version 1.0 is due. However, the project has gotten *big*. As a result, the implementation is becoming more difficult. The engineer(s) is starting to see that the size of the project is causing problems: bugs, cracks, and broken pieces. The code needs to be *organized*. Many things need fixing and not all of the promised features are quite there yet. Therefore the release date needs to be *delayed*.

Stage 4–Two Years Later, Version 1.0: Two years later is when version 1.0 is finally released. The "delay" is one year–much more common than one thinks. Although this "delay" is somewhat artificial. Since the product has not gone through a design process, no critical assessment or analytic thought has been given to constructing a feature list. The machinery under the hood has not been thought out. Nonetheless, after a year beyond the manager's original release date, the product now has to be released because the delay is perceived as being one year–a long time.

The application has quite a few bugs, more bugs than had been anticipated. It's not quite as stable as everyone had hoped it would be. But nonetheless, the product had to be rolled out do to the pressure of expectations. In the end, the delivery is not quite the success as had been imagined. No problem, the bugs will be fixed in time for the next version. There will be lots of great new features on top of that. And now that two years have been invested into the application and the product has been "successfully" released, additional engineers are assigned to participate its development, add to the feature list, and give the application the push it needs for big success.

Stage 5–Three Years Later, Version 2.0 and the Decline: Three years later, version 2.0 is due to be released. Many bugs should be fixed. The product should be stabilized. There should be some great new features. But the engineers are experiencing problems. The code base has *grown* rapidly to hundreds of thousands of lines of source code. There is no coordination amongst the engineers. Software design and coding convections are mere abstractions. As soon as one bug is fixed, another bug (or two) is introduced. The code is very difficult to manage. The whole project feels like a bulging barrel of water with holes and cracks. As soon as one hole is patched, another appears and water starts leaking everywhere uncontrollably.

And it does not seem to matter what the engineers do or how much effort is invested into the product. Thousands of lines of code and engineering manpower are invested into the application to get it into industry shape. But the product cannot seem to be brought under control no matter what is thrown at it and the engineers (and managers) are getting *frustrated*.

Stage 6–The Departure: After three years on the project the original engineer(s) have become very frustrated and can see that the product has gotten out of control. Their hard work and serious efforts have not been rewarded with the success as originally thought. The size of source code keeps growing and growing and the bugs and problems keep coming and coming with no sign of let up. Then, under their frustration, the unthinkable happens... the original lead software developer(s) on the project *quit* the job and move on.

There was a conflict between management and the lead engineer(s). The lead engineer(s) wanted a pay raise. Afterall, they were the mastermind behind the whole project, the one(s) who developed it from scratch. The manager(s) however, never saw a stable, profitable product. "Their" product is not the success story that they had thought it should be. And this was the engineer's fault.

Stage 7–The Rescue Attempt: After the lead engineer(s) quits the project, an undertaking for which there is very little documentation (Documentation is work-in-progress and is *much more difficult* to write after the lead engineer has left the company.) the project appears virtually dead to the remaining engineers. However, there is no way the responsible manager(s) is going to let the project die. More than three years have already been invested with multiple employees representing hundreds of thousands (if not millions) of dollars (or Euros etc). A product was promised and a product will be delivered. Plus, a failed application would be a major embarrassment for the management.

Thus a rescue attempt is undertaken. The plan is simply hire replacement engineers and pretend nothing has gone wrong. New engineers arrive and are handed the tasks of fixing the bugs left behind by developers that have quit and adding new features. For six months everything runs quietly. Six months is generally the grace period new engineers are given to understand the existing code base. After six months of trying to comprehend and work with the existing source code, the no-longer-so-new engineers start to fix bugs and add features. However, as the second generation of engineers modify the application, they encounter the same problems. Bugs are fixed and replaced by new bugs. In some ways they are even more frustrated than the first generation of engineers. The existing source code is cryptic and sloppy. There are a multitude of quick-and-dirty hacks that were, in theory, meant to be fixed at a later date. No rules were enforced on coding style and the engineers did not coordinate. Code legibility did not seem to be an issue when only one (or two) engineer was starting the project. Had the second generation engineers started the project from scratch, they would have done things much differently.

Stage 8–A Slow Death: After some time, the second generation of engineers will reach the same conclusion as the first generation, i.e., that the project they have been assigned to cannot be rescued. Thus they will either (1) quit after a few years or (2) start a new project. For the managers of the project, they may decide to hire a third generation of engineers who will repeat stages 6 and 7. Engineers who join another existing project will likely end up assigned to another rescue attempt as described in the introduction. And starting a new project doesn't fix the problem either, since it will generally evolve as described in stages 1-7.

4 Why Have Coding Conventions?

Coding conventions, such as those listed on page one, are major constituent of the solution to this problem. The idea of coding conventions and code comment conventions [8] is often met with strong resistance from industry, especially management. "That's not the real world," is a typical response. In "reality" the majority of industry software projects fail [2, 5]. What does it mean to fail? From a business perspective, failing means not generating a profit. That's right, the majority of software projects in industry never generate a profit. However, this information is never advertised. We only hear about the small minority of success stories.

There are other ways of failing as well. A subjective measure of failure is happiness or unhappiness. The majority of industry applications fail using this metric as well. The developers are unhappy because the product is unstable and they don't like repairing the bugs left behind by others. The managers are unhappy because the application is failing to turn a profit. The users are unhappy because the product they are using crashes too often.

We claim that following these coding conventions helps pave the way to a successful software application. Why? Because software that is very legible is better. It has fewer bugs, is more stable, and makes developers happier. The other two key ingredients are code com-

ment conventions [8] and design [20]. The commenting, design, and modification of design is facilitated by these coding conventions.

Big projects require multiple, coordinated developers over several years. And, applications should not generally be started from scratch [9]. But yet, we in the software industry start projects from scratch over and over again. We also re-invent the wheel over and over again. One of the major problems stems from source code that does not follow any conventions and is not very legible. As such it quickly turns into *legacy* code. Writing illegible code is easy and is generally the default. We have encountered numerous instances of programmers who cannot even read their own code.

Bob's concise coding conventions are influenced by and drawn from other coding standards and guidelines including the VTK [4], Sun Microsystems [14], Meyers [12, 13] and Dickheiser [1]. They are meant to be concise so they can be printed out and hung up for ease of use. The basic philosophy behind the conventions is code legibility should be maximized. The hypothesis is that code with maximum legibility leads to a minimum number of bugs. Maximizing legibility also helps maximize code re-use, good design, and flexibility–all goals of these conventions.

5 The Author's Software Industry Experience

The author spent the summers of 1995 and 1996 working in the Information Technology (IT) Department of a company called Private Health Care Systems (PHCS) based in Waltham, MA. His job was to document source code. The undocumented, legacy applications were abandoned by the original lead developers. No one knew how the undocumented software worked. The IT department of PHCS experienced the highest employee turn-over rate the author has ever seen. PHCS was acquired by MultiPlan in October 2006.

Bob spent the summer of 1997 again as a documenter (documentator?) of source code for a small, internet start-up Company called Cambridge Interactive based in Cambridge, MA. The lead engineer of the primary product quit after building up the project implementation for about two years. Legend has it that the company CEO used to sleep in a server room to restart machines that crashed during the night. Cambridge Interactive is no longer in business.

The author spent five years (2001-2006) as a full-time software developer at a company called AVL (www.avl.com) in the department of Advanced Simulation Technologies (AST). (He was also an employee of VRVis, www.VRVis.at, during this time [7].) The product he worked on was a replacement for another legacy application. The release of version 1.0 was delayed by one year. Both of the original lead developers of the project quit. Although, they did last a little bit longer than the three years as described in Bob's Theory of Software Development. Bob was a second generation engineer and part of the team trying to save it. Legend has it, the application was described as, "The most unstable piece of software I have ever used," by one of the users. As of 2006 this software development department had never turned a profit since its founding in the mid 80's.

The author also has academic software development experience [6, 7, 10, 16] and is currently a lecturer at Swansea University in the Department of Computer Science.

6 Acknowledgements

Understanding sloppy code is difficult, time-consuming, and can even be impossible. Thanks to Tony McLoughlin of Swansea University for valuable discussions on this topic. Thanks to Edward Grundy of Swansea University for his feedback. Feedback on this document is

not only welcome but encouraged. Please send correspondence to the first author. Please see the related document on code comment conventions [8].

7 Appendix

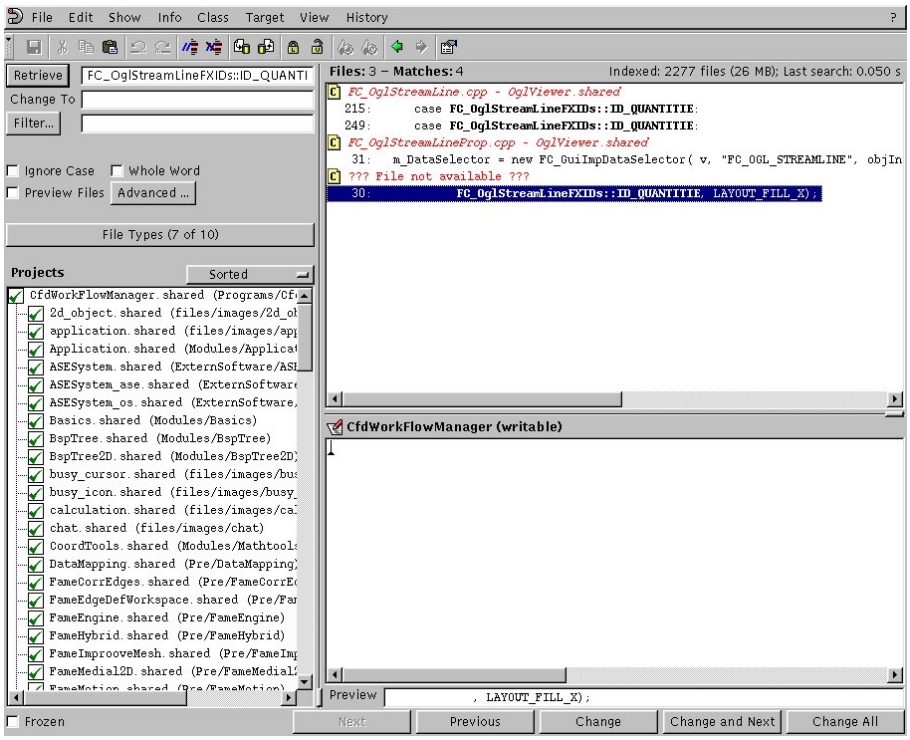

Figure 2: This figure shows a screen shot from the SNiFF+ IDE. The user has performed a source code search for the expression in the top, left next to "Retrieve". The results are shown in the larger top, right window. Line-by-line results are supposed to be shown in the bottom, right window. Notice the question marks. This project was so big and disorganized, that even the IDE could not always perform searches with correct results.

References

[1] M J. Dickheiser. *C++ for Game Programmers*. Charles River Media, Boston, MA, 2 edition, 2007.

[2] K. Ellis. Business Analysis Benchmark: The Impact of Business Requirements on the Success of Techonolgy Projects. Technical report, IAG Consulting, 2008. http://www.iag.biz/links/white-papers-and-articles/ [accessed: April 2009].

[3] C. Horstmann. *Computing Concepts with C++ Essentials*. Wiley, 3 edition, 2003.

[4] Kitware. *VTK Coding Standards*. http://www.vtk.org/Wiki/VTK_Coding_Standards [January 2009].

[5] M. Krigsman. Study: 68 Percent of IT Projects Fail, December 2008. http://blogs.zdnet.com/projectfailures/?p=1175 [accessed: April 2009].

[6] R. S. Laramee. Isosurface Rendering of Adaptive Resolution Data. Master's thesis, University of New Hampshire, Department of Computer Science, Durham, NH, December 2000.

[7] R. S. Laramee. *Interactive 3D Flow Visualization Using Textures and Geometric Primitives*. PhD thesis, Vienna University of Technology, Institute for Computer Graphics and Algorithms, Vienna, Austria, December 2004.

[8] R. S. Laramee. A Source Code Comment Standard. Technical report, The Visual and Interactive Computing Group, Computer Science Department, Swansea University, Wales, UK, 2007.

[9] R. S. Laramee. Comparing and Evaluating Computer Graphics and Visualization Software. *Software: Practice and Experience (SP&E)*, 38(7):735–760, June 2008.

[10] R. S. Laramee, G. Erlebacher, C. Garth, H. Theisel, X. Tricoche, T. Weinkauf, and D. Weiskopf. Applications of Texture-Based Flow Visualization. *Engineering Applications of Computational Fluid Mechanics (EACFM)*, 2(3):264–274, September 2008.

[11] J. A. Maciaszek and B. L. Liong. *Practical Software Engineering: A Case Study Approach*. Addison-Wesley, London, 2005.

[12] S. Meyers. *More Effective C++, 35 New Ways to Improve Your Programs and Design*. Addison-Wesley, 1996.

[13] S. Meyers. *Effective C++, 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 2005.

[14] Sun Microsystems. *Code Conventions for the Java Programming Language*, April 1999.

[15] T. Oetiker, H. Partl, I. Hyna, and E. Schlegl. *The Not So Short Introduction to LaTeX* 2_ϵ, 4.23 edition, January 2008. (published online, free).

[16] Z. Peng, R. S. Laramee, G. Chen, and E. Zhang. Glyph and Streamline Placement Algorithms for CFD Simulation Data. In *NAFEMS World Congress Conference Proceedings (NWC)*. NAFEMS–The International Association for the Engineering Analysis Community, June 2009. forthcoming.

[17] J. F. Peters and W. Pedrycz. *Software Engineering: An Engineering Approach*. John WIley & Sons, Inc, New York, 2000.

[18] I. Sommerville. *Software Engineering*. Addison-Wesley, London, 8 edition, 2007.

[19] H van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, Ltd, West Sussex, England, 3 edition, 2008.

[20] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

Figure 3: This figure shows an application attempting to output useful information in text, e.g., a stack trace, after a crash. If a product is big and sloppy enough, it might even have trouble outputting ASCII characters.