The Code Diet

Benjamin Kenwright[1]*

Abstract
Writing beautifully clear and efficient code is an art. Learning and developing skills and tricks to handle unforseen situations to get a 'feel' for the code and be able to identify and fix problems in a moments notice does not happen overnight. With software development experience really does count. This article introduces the reader to numerous engineering insights into writing better code. Better in the context of cleaner, more readable, robust, and computationally efficient. Analogous to the 20:80 principle. In practice, you can spend 20% of your time writing code, while the other 80% is editing and refining your code to be better. You have to work hard to get coding muscles. Lazy coding ultimately leads to unhealthy, inflexible, overweight code.

Keywords
Readable — Efficient — Robust

[1] *Edinburgh Napier University, School of Computer Science, United Kingdom*: b.kenwright@napier.ac.uk

Contents

Introduction 1
1 The Path To Working Smarter Not Harder 1
2 Reliable Optimised Code 1
3 Experience 1
3.1 Comments 2
4 Logic 2
4.1 Common Sense 2
4.2 Unpredictable 2
4.3 Impossible 2
4.4 Recovering 2
4.5 Double Checking 2
Two Different Methods
4.6 Security 2
4.7 Different Set of Eyes 3
5 Slow Same As The Fast 3
5.1 How Much Faster Before We Rewright The Code? 3
5.2 Lean and Mean 3
5.3 Loops (For, While) 3
6 High-Level vs Low-Level 3
7 Fast Isn't Always Faster 3
7.1 Tortoise and the Hare 3
7.2 C/C++/C#/Java 4
Pre and Post Incremental
7.3 Platform Specific 4
7.4 Don't Assume - Prove It! 4
7.5 Practical Test Cases 4
8 Reliable or Fast (Compromises) 4
9 Compiler 4
10 Sequential Thinking 4
10.1 Bigger Picture (Small Pieces) 4
10.2 Does Size Matter? 4
10.3 Naming Convention 4
Hungarian Notation
10.4 Colour Coding and Development Environments 5
10.5 External Data 5
10.6 Cache Hit 5
10.7 Asserts 5
11 Source Control 5
Acknowledgments 5

Introduction

There Is No Such Thing As Perfect Code We have all written what we think is beautiful code. Some might even say sexy, elegant and the undoubtedly the best. What made this piece of code so great? Was it the complexity? Robustness? Readability? Size? Have you ever had to debug someone elses code? When was the last time you saw a piece of code you thought was amazing? Have you ever not been able to fix or track down a bug in a large piece of code? We address and discuss these questions in this article. We try and open the readers eyes to more novel and flexible techniques for writing code. To think of sustainability and flexibility. To encourage the reader to ask 'what if' during the development of their code.

1. The Path To Working Smarter Not Harder

Be Open Minded The first step towards growing and improving is to be open minded. To accept new ideas and think outside the box. Tunnel vision (i.e., thinking in one direction without looking around or considering anything else) can keep you stuck in bad habbits. Has anyone ever given you some advice while you shunt it away since you think your method is better without reason or cause? Justification of a doing something one way and not another with 'just because' is not a reason enough.

Classification How do we classificy which piece of code is the best? For example, if two different people write two different pieces of code that perform the same operation which is the best?

2. Reliable Optimised Code

What do we mean by robust, optimized, and reliable? There is not such thing as perfect code. Human errors are inevitable. Furthermore, software evolves. What was considered as adaquatly code today might not be considered sufficient for tomorrow. New algorithms and hardware are always pushing the limits of what is possible. Nevertheless, when we say robust it means that it can handle a wide variety of unforseen circumstances. For example, when your algorithm uses external data, if that data is not as expected (e.g., corrupted), then your algorithm can identify this and cope without crashing to the ground. While optimized code, is code that runs sufficiently fast enought to meet the the systems desires.

3. Experience

A thousand mistakes is a thousand lessons A person who has never made a mistake has never learned anything new. Never be afraid to try new approaches. To go against tradition. You learn and grow as you experiment and through your failures. Each time you fix a bug or optimize an algorithm you expand your knowledge. For example, when you debug a huge piece of software for an error, it can take enourmous inginuity to track the problem down. You can be limited by what debugging facilities are available, such as prints or graphical output. How can you get more experience? Learning from those around you. Reading online articles and forums for advice and tips. Stretching your knowledge by taking on projects outside of your interest and scope to widen your eyes to new ideas. For example, you do not necessarily know if you like or dislike something until you have tried it. Analagous to if something can help and make your life easier better until you give it a try.

3.1 Comments

No One Else Will See My Code Typically, young developers get into their heads that as long as they understand their code at that moment they do not need to comment it. That the algorithm to them is easy to understand. That they do not have time to comment it. That they will come back and comment it later.

4. Logic

4.1 Common Sense

Not magical Software algorithms should now work using magic or random luck. Always take a straightforward view to programming. Basically, as with mathematics, software follows a set of rules that you can break down into smaller easy to digest pieces. Each piece should be clear to understand. When you are looking at values ask yourself if the values seem reasonable (e.g., NAN - not a number, square root of a negative number) and what could have caused this. When developing your programs you need to take a critical thinking approach. Always look for ways that it could break and ways that it could be made better. Follow your implementation through sequentially and ask if the values or logic is following what you expected. Do random unforseens numbers pop up? Is there an automatic way of checking that your algorithm is working? Can you pass in a range of pre-set values to check that you get the correct output to show that your algorithm works? Similary, pass in a set of invalid values to see what happens and if your algorithm can recover gracefully.

4.2 Unpredictable

Sometimes? Have you ever run a piece of code that produces a different output each time? Software is not random. You should be able to predict what your algorithm will do. Even if you have random elements within your code, you should be able to determine the range of output based upon the range on inputs. For example, if you have not initialized all your variables in non-managed languages, such as C and C++, your program will possess random numerical errors. Imagine you did not initialize a loop counter. Each time you ran the code it would perform a different number of iterations. The random loop could cause random stalls or buffer overflows.

4.3 Impossible

Even the word 'imposssible' contains the word 'possible' Nothing is impossible. New inovative interesting and valuable algorithms are constantly being developed. What today you might thought impossible may be common and unnoteworthy tomorrow. For example, since computer speeds are growing exponentially faster, algorithms from decades ago are now becoming usable and viable options in real-time environments. Furthermore, due to the trend in highly parallel systems you need to think differently in some cases to take advantage of the full set of resources in your hand and accomplish the impossible.

4.4 Recovering

What to do when things go wrong? When your program does identify an error what should you make your code do? There are numerous chains of thought on this question. Developers usually like to be warned and hit with the line number and what caused the problem. While users do not want to be troubled and want to be able to continue working. Hence, it is usually necessary to put in a dual set of recovery and diagnostic code. One set warns and logs the problem, possibly causing an assert in the debugger, while the setting piece of code resets the problem to some stable value so the program can continue running. This means you need to be able to do 'release' and 'debug' builds. Also, don't fall into the trap of always working in debug mode and never testing or updating the seperate set of release code. Furthermore, always be aware that your debug code can cause issues. For example, cause a slow-down or modify the output.

4.5 Double Checking

Everybody checks everybody else This might seem a bit much. But having multiple 'redundant' versions of code checking one another means that your implementation will be less prone to errors. While it might run mega slow in diagnoistic mode it will however identify errors immediately instead of when the program goes live and out into the open world. For example, an algorithmically simplier and slower implementation testing a highly optimized assembly code version.

4.5.1 Two Different Methods

There are two ways to skin a cat Typically, there can be multiple ways to accomplish a task. Different algorithms can offer different features, for example, less bandwidth, less memory footprint, more computational speed, numerically less accurate. Understanding and knowing which tool to fit the job is important. As the old saying goes, *if all you have is a hammer, every problem looks like a nail*. Similary, if you only know one algorithm, each time you come across a problem, you try and adapt and fit that particular algorithm to the task. For example, if you are only familiar with the slower exponential bubble sort algorithm, each time you need to sort a set of numbers, it is your first choice. Nevertheless, if the numbers are all ready nearly sorted, it is miles more efficient to use a merge sort and offers all the same features but is vastly computationally quicker when the values are nearly sorted. This is useful where you can be sorting the data constantly and only minor changes happen between updates. Be aware of the task and the tools.

4.6 Security

Vulnurabilty Software flaws open the door to security issues. For example, assuming a string buffer will never go past a maximum length and not incorporating any checks. Then some scrupulous individual could identify this flaw and pass in data that would go outside the buffer range causing memory corruption. Overwritting code on the fly to perform unintented tasks.

4.7 Different Set of Eyes

Two Heads Are better than one Sometimes just explaining the problem to another person helps you clarify and solve the problem yourself. Alternatively, bouncing ideas of a person with a different background helps give you a view on the subject that you never even considered.

5. Slow Same As The Fast

Redundancy When you are developing a newer inovative and untested algorithm, you might want to run it in parallel with an older proven possibly slower and bulkier approach. The method allows you to guarantee that the algorithm is producing the same numerical results, furthermore it gives you a benchmarch on how much faster your new algorithm is and how less memory or bandwith it is using.

5.1 How Much Faster Before We Rewright The Code?

Profiling If you are in doubt about where performance bottlnecks are, then run a profiler. You can pin-point specific areas of your code that are causing slow-downs. Furthermore, if you try out a new piece of code, be sure it is worth the extra effort. For example, it is not worth lots of extra time and effort rewriting a small section of code in assembly using some fancy algorithm if the problem is a high level one (i.e., badly designed system). You might need to take a different perspective and approach the problem from an alternative direction (i.e., smarter and less brute force).

5.2 Lean and Mean

Feature Creep Over time programs can be extended and engineered to solve multiple tasks. For example, flags and special cases can be incorporated into the algorithm. However, the time will come when you have to cut the fat. When you have to decide what is needed for what and split the overgrown program into smaller task specific modules rather than a single interweaved beast that is inflexible and bloated.

5.3 Loops (For, While)

Are Loops Really Bad? Loops allows us to develop small compact pieces of code. The small algorithm can typically fit in the cache and execute quickly. Of course, loops, especially nested loops can time consuming. If you can avoid loops then do.

Macros (Loop Unrolling) For programming languages, such as C and C++, that allow macros, you can have loops unrolled automatically at compile time. Of course, loop unrolling has problems. The compiled binary will be larger and can cause a cache hit if the algorithm cannot fit into memory. Nevertheless, for small pieces of code macros enable you to force inline optimizations.

Recursion Functions that call themselves are an alternative to loops. However, each time a function calls another function the current state registers and return address are pushed onto the stack. If you are not careful (e.g., infinitly call a function itself) you will hit a stack overflow assert. That is you will run out of stack memory allocated when the program starts. Recursive functions can be compact and beautiful. But the same can be said about loops. Recursion or loops depends upon the task at hand and in the majority of cases algorithms can be implemented both with loops and recursive methods.

The Big 'O' How can we measure the performance of our code? Classifly and compare two different algorithsm that perform the same operation? In computer science, the big 'O' notation describes the performance or complexity of an algorithm. Primarily, the big 'O' focuses on describing the worst-case scenario. For example, the execution time required or the space used (i.e., memory or disk space) by an algorithm in question.

Avoiding Conditional Statements An algorithm that is sequencial and does not require any conditional logic is easy to predict. The CPU predictive cache logic and pre-load and number crunch a small chunch of data. However, it can depend on memory accesses, floating point calculations (e.g., sine or squareroot operations), and bandwidth.

6. High-Level vs Low-Level

Do you really need to go down to assembly level optimization? Certain hard-core developers on occasion feel the need to sqeeze every single cycle out of a system. For example, those developing on older hardware, such as Sony's Playstation 2 or GPS hand held devices, where understanding processor instructions can dramatically effect the overall performance. However, in the majority of cases, you will find that a smarter algorithm does the job. As they say, there it is no use beating a dead horse. If you are working with an $O(n^2)$ algorithm (i.e., exponential growth efficiency algorithm) then it does not matter how fast the execution time through optimisation. While if you replace the exponentially slow algorithm with a log or linear approach will be dramatically faster without needed to jump into any assembly. Furthermore, the solution with be platform invariant and can be ported and modified without much pain.

7. Fast Isn't Always Faster

When you say fast what do you mean? When you say fast, do you mean computationally efficient or algorithmically uncomplicated? Do you make any assumptions, such as floating point precision? Is your code flexible and can it grow as the data grows?

7.1 Tortoise and the Hare

If Only The Hare Hadn't Stopped For A Nap Are you aiming for a long term solution or a temporary 'hack'? Will your program be use in navigating or controlling life support equipment on planes or in hospitals? Or will it be number crunching some approximate visual lighting values for a 3D video game?

7.2 C/C++/C#/Java

What Does Your Language Bring To The Table? Managed languages are popular and provide an assortment of tested libraries. They are platform independent and cut you free from worrying about memory and low-level issues. While hard-core programming languages, such as C and C++, allow you total control. You can mix in assembly language, manipulate the raw memory contents, or call specific hardware instructions. Again it depends upon the task at hand. For real-time environments, such as games and graphical solutions, C and C++ are the prefered solution, while for applications and interfaces managed languages are easier.

7.2.1 Pre and Post Incremental

++i or i++ Why does it matter if we use pre or post incremental operations? Why is post incremental the common and default method we are used to using? For languages that allow operator overloading (e.g., C++ and C#) pre and post incremental operations can produce code that is computationally faster and more robust. While for uncomplicated values such as integers and floats the pre and post incremental operations don't matter. However, for custom classes it can result in the creation, copying, and destruction of classes each time the post incremental operation is used.

7.3 Platform Specific

Big Endian or Little Endian Which way do you eat your egg? If you are unfamililar with Gulliver's Travels story about Lilliput. Where two great nations are at constant war about which side of an egg you should eat from. Similarly, CPU architecture is primarily split into two main types, big-endian and little-endian, depending upon your byte ordering. The artictures can then be highly parallel (e.g., graphics processor unit), game orientated (e.g., xbox, ps4), higher level managed language such as C# and Java. Understanding where your code will be used can determine how you write it. How you structure array, how you initialized variables, and so on.

7.4 Don't Assume - Prove It!

That Won't Ever Happen The best rule of practice for assumption is to use 'asserts'. If you are so confident that it will never happen, then place an assert in the code to confirm that it. This has the added advantage of triggering and informing you if it does happen. Furthermore, if another person looks at your code they can see from the assert that you have specified that this should never occur and if it does then something has gone haywire.

7.5 Practical Test Cases

Making Your Program Jump Through Hoops Do you just write your algorithm, run it once, then assume it's working perfectly without flaws? If possible try break your algorithm. Run it with a wide set of randomly generated variables and see what happens. Possibly you didn't initialized all your variables or you made some assumptions while you were developing the code. Running your program through some test cases will ensure it doesn't come back an bit you at a later date. The test code will also help you identify bugs and problems if you modify the code at a later date to make it more faster more efficient and juicy.

8. Reliable or Fast (Compromises)

The Glass Shoe When the entire village of eligable women were trying on Cinderelas glass shoe, none of them fitted. Imagine it. No one has feet either as small or as big as Cinderela. With software you have to decide. For example, fast or accurate, big or small memory footprint, portable or platform-specific. Rarely, no solution fits every situation.

9. Compiler

The Compiler Is Trying To Help You Compile warnings are there for a reason. The compiler is informing you that the solution you have chosen can cause slowdowns or numerical problems. Try and avoid having hundreds or thousands of warnings fly through the output window each time you compile. Typically, if you truly understand, and I mean truly understand the warning, you can include a define in the program to disable it. But be warned. Don't go disabling a warning nilly-willy just because. Be sure you understand the consequences.

10. Sequential Thinking

Single Thought At A Single Moment Our mind cannot run in parallel. We systematically switch between tasks and problems. We cannot work on multiple problems at the same time. As the saying goes, if you try to chase two rabbits both rabbits will escape.

10.1 Bigger Picture (Small Pieces)

Building Blocks Methods and functions allow us to create reusable parts. Re-inventing the wheel all the time can be error prone and inefficient.

10.2 Does Size Matter?

Size Of What? Are long variable names clearer than shorter ones? Does a small algorithm run faster than a larger algorithm? What is your justification for your answer?

10.3 Naming Convention

Single Letter Variables Do you name your variables 'a', 'b', 'c', 'i', 'j', and so on. Why? When a variable name can help make your algorithm more readable.

10.3.1 Hungarian Notation

Overkill While the full hungarian notation might be overkill, a simple common sense naming convention, such as functions and classes starting with captial letters, and variables always with a starting lower case letter are common and form a set of consistent rules that enable people to easily see what an item is without jumping around in the development environment.

10.4 Colour Coding and Development Environments

Why Not? While you might not want to use a full development environments, such as Microsofts Visual Studio, you can still take advantage of open source programs, such as Notepad++, which provide you with colour coding. Make your life easier if you can.

10.5 External Data

Input From The Outside World A program needs input. Without some sort of input, either from data files or from user interaction, the program would be useless. Verifying that the external input data is valid and correct is important. Since bad data means unforseen results. Any time a function takes in values and returns values you should always check to ensure the paramater values are within specified tolerance ranges (e.g., using asserts).

10.6 Cache Hit

The cache is there to prevent performance hits from memory retrival. Small algorithms with a mimumum memory footprint that work with coherent sequential data from memory are ideal for the cache.

10.7 Asserts

A Must Novice and beginner programmers seems to avoid asserts. They cannot comprehend the true value of them. They allow you to define a set of rules and checks for your code both for yourself and other developers. A good place to always insert asserts is at the beginning and end of functions to ensure incoming and outgoing values are within predicted predifined limits.

11. Source Control

CVS, SVN, GIT If you have ever collaborated with multiple people on a project then you should have come across source control. Source control enables you to record and track changes. Some people refer to source control as a 'blame-tool' since it allows you to identify who changed what, when, and why.

Acknowledgments

I would like to thank all the reviewers for taking time out of their busy schedules to provide valuable and constructive feedback to make this article more concise, informative, and correct.

Recommended Reading

Code Complete: A Practical Handbook of Software Construction, Steve McConnell, ISBN: 978-0735619678
Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, ISBN: 978-0132350884