# Compealing Skills

Paul Lynham FIAP

## Introduction

Code rises above functionality. To some, this may seem a strange thing to say. The real meaning of code is determined not only by the programmer's intention, or by the operation it triggers, but also by how it is received and recirculated[1].

Code should communicate; indeed it should speak to you as you read it – although it is executed by the computer, developers should be the audience. The culture of the coding can be perceived, exposing aspects of the programmer, their way of thinking as well as the ethos of the coding language used. It is not a one-dimensional artefact but entangles much which cannot be glimpsed by the untrained eye.

Imagine a program's source code as a piece of sculpture. If it is taken off its plinth and examined from the back, the side, the top and the bottom, a better picture of its form can be perceived than just a casual view from the front.

In the introduction to Beautiful Code[2], Greg Wilson declares:

> "… for the first time, I saw that programs could be more than just instructions for computers. They could be as elegant as well-made kitchen cabinets, as graceful as a suspension bridge, or as eloquent as one of George Orwell's essays."

In the third part of this series on *Compealing Code*, some of the skills required to achieve this goal will be touched upon.

## Reading and Writing

Poetry, pottery, prose, painting or performance - all may communicate with the observer at several different levels. They can expend little effort by just letting the experience wash over them, or exert more effort by thinking deeper about what this means and its consequence. The experience may be delightful, nonchalant, boring or stressful. It could lead to confusion or may convey a message or a story, elicit changes in emotion, attitude or behaviour, providing inspiration, desperation, bewilderment or insight! Often it depends on the approach of the spectator. It would seem plain which set of responses is desired when dealing with source code.

Reading code and writing code requires different skills. Let us compare these skills by examining them in the context of reading and writing poetry. Fred Brooks said about creating software:

> "… there is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures." [3]

### Reading

If the poem is a simplistic piece, then it should only require basic reading skills. If it is a little more complex, then it may require more effort, including deciphering and then understanding the context, with perhaps empathy for the subject matter, possibly gained through experience. With a more esoteric poem then in addition to the skills noted, some further knowledge or experience in the subject matter is useful, a sprinkling of fancy, with an enquiring mind that seeks out hidden

  

meanings or relationships which others may miss. It requires the reader to pick up clues, make linkages, note weaknesses, flaws or omissions, fashion one or more interpretations, draw conclusions and finally form an opinion.

### Writing

When creating a poem, the writer should know the context, so that isn't an issue. Certainly domain knowledge is useful, so the subject matter can be portrayed realistically; otherwise, a good imagination is helpful. There needs to be a motivation and the ability to set out thoughts in a way that not only conveys the concepts to be imparted but also in a manner that makes the poem delightful to read. A good command of the language, with a creative mind, together with fertile constructional skills is needed to turn the work into something that conveys meaning, is pleasing and compelling, so has merit.

### Poetry and Programming

However, the world of art for art's sake and the world of producing software that produces hands-on results are very different. Yet there are some similarities as well as contrasts. Programmers usually don't appreciate the challenge of trying to understand esoteric code! To be effective, they need to quickly grasp the context, appreciate the intention and confirm that this is indeed what is implemented. The easier and quicker this can be done, the less stressed they are and the more productive they can become.

With programming, the developer learns by repeated cycles of writing and running the software. Initially, they only read their own code. Anyone can write code that a computer can understand - good programmers write code that humans can understand[4]. However, when working as part of a collaboration or team, the developer has to read and understand other people's code. This can be difficult compared to writing and is often the reason why some developers will routinely completely rewrite some functionality rather than try reading, understanding and changing the existing code. This is not because the code needs refactoring; rather these developers lack the skill or cannot be bothered to understand the current code. This not only creates churn[5] but leads to inefficiencies and bad practice, the potential introduction of new deficiencies, as well as a lost learning opportunity.

## Nothing is new

The idea of a programmer developing a craft not only to instruct the computer what to do but also describe the intention in an easily understood way is not a new idea. In 1984 Donald Knuth stated in his book *Literate Programming*[6] that the programmer needs to become an essayist, so the author's concern is the exposition of excellence of style. His idea was to produce programs that represent logic in an ordinary human language, with macros included to hide abstractions and traditional source code. Tools would then be used to provide two representations from the 'literate' source file. One would be suitable for compilation or execution by the computer and another for viewing the formatted documentation by humans.

However, *Compealing Code* aims to combine the idea of representing the logic so it can be executed by the computer and yet make these intentions explicitly simple for a human to comprehend using conventional programming language text. If this process was commonplace, then there would be no need for these words.

The inference must be developers need to learn to read code written by others but also to write code that speaks to their fellow developers, rather than just the computer. These skills are not easy to acquire. The former skill allows the reader to gain a wider perspective of programming and the language used, perhaps perceiving things they have not imagined or seen previously. It may also

present an opportunity to also pick up domain knowledge. Besides learning, they may also identify issues, which can lead to a higher quality product. The writing skill turns their code from something which may run at this point in time, to a lasting legacy and resource, which can be used, understood and extended for a long time to come. It serves functionality, specification and example, as well as a learning resource. The concept of *Compealing Code* applies directly to these skills.

## Software as an art

Is writing software an art? Leonardo Da Vinci said:

> "Art is never finished, only abandoned."

Good code can always be reworked and improved, so it can be reimagined and repurposed. It has been noted by others that producing software may be more an art than engineering. The goal isn't necessary to produce a piece of 'software art' of the highest order, but it should go further than just writing instructions. Well written code may be deemed art by those who can appreciate all its aspects.

Such skills needed to achieve this purpose require discipline as there is frequently a tendency to get something out of the door as quick as possible. Yet the saying, "more haste, less speed" is often true, as writing too quickly, without due diligence, focus, and attention to detail can result in avoidable mistakes. The result is the product will require even more time to complete the task satisfactorily. This is compounded when the developer or someone else comes back to this code at a later date to find the code is unappealing, difficult to understand and it is hard to assert that it does what it is supposed to do.

## Summary

There are many levels of appreciating code. *Critical Code Studies*[7] is a new way of examining and interpreting code beyond its functionality in a similar way that scholars read religious texts or literature, putting their own interpretations to what has been written. It applies methodologies from the humanities and may assess the text as a work of art. Although this is an interesting area of study, the author takes a more pragmatic view, as software developers need to create applications and systems that are fit for purpose, yet are flexible and easily maintained, to allow the software to live, rather than become brittle, stagnant and eventually die. Such programs serve a practical purpose.

Some may consider it as high art, but the true artisan will take the view that this is just normal quality, produced in a professional and efficient manner. Such craftsmen are courteous to the product, their organisation, as well as their fellow developers, who hopefully will appreciate their *Compealing Code* in the future.

---

[1] Marino, M.C. (2020) *Critical Code Studies*. Cambridge: MIT Press, p. 4
[2] Oram, A., Wilson, G. (2007) *Beautiful Code*. Sebastopol: O'Reilly Media, p. XV
[3] Brooks, F.P. (1995) *The Mythical Man-Month – anniversary edition*. Addison-Wesley Publishing Company, p. 7
[4] https://en.wikiquote.org/wiki/Martin_Fowler
[5] https://blog.drinkbird.com/code-churn
[6] Knuth, D. (1984) *Literate Programming*. Stanford: CSLI Publications
[7] Marino, M.C. (2020) *Critical Code Studies*. Cambridge: MIT Press