Compealing Code for Improving Software Quality

Paul Lynham FIAP

Introduction

In a previous article looking at the attractiveness of source code[1], an overview of compealing software briefly summarised the concept and the three attributes that it is formed from. The rationale for the use the compealing concept will be the focus here, but future articles will give more details and examples of how compealing software can be achieved.

The case for compealing

Quality can be difficult to evaluate. Metrics can be used to some extent e.g. checks can be made to ensure all exceptions are caught. However, certain aspects of source code may be difficult to assess, such as how appealing the code appears to a human reader. Line length and density can be measured, but use of good names is more problematic. Einstein's once wrote on a blackboard:

> *Not everything that counts can be counted, and not everything that can be counted counts.*

So although some of the concepts that will now be examined may be difficult to measure, they certainly count and are worth assessing, even though specific metrics may not yet exist!

Compealing attributes

At the lowest level of reading any text, including source code, it must be legible. This in practical terms measure how easy each character can be deciphered, so as information can be gleaned from the text. Think of a prescription written by a doctor and how easy it is to decipher what has been written! At this point, you may not be able to form any sequence of characters to construct words and certainly no understanding.

The next level is being able to read groups of characters that are recognisable as words. Some of the words may be familiar, even if you don't know any Latin. If you look up the words, the sentences formed may now convey meaning. You could google the medicine in the prescription and together with the reason why you went to see the doctor, you can see why the prescription makes sense.

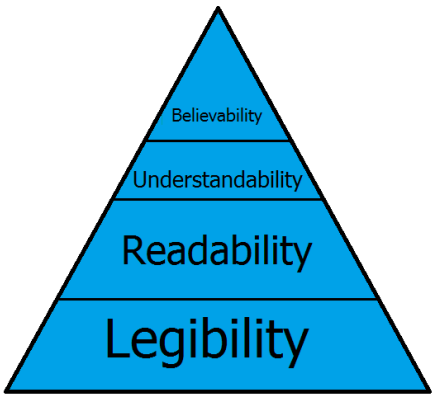

Figure 1. The abilities of source code

This analogy is summed up in Figure 1 which demonstrates the different 'abilities' of source code. We will take it for granted that source code is legible if the font and background colours have a contrast.

Readability is slightly more difficult as this has different connotations – it could mean the ability to recognise individual words or to recognise the concept conveyed by the statements formed from those words. In source code it would mean at least that the words used are understandable either as keywords in the language, language constructs or as identifiers.

Understandability means that after reading the source code you fully understand exactly what the intent of the code is.

Believability is achieved when the source code fully convinces you that this code will reliably achieve the intended behaviour when it is run and gracefully handle all circumstances.

As in a novel, you have to be able to read the prose, understand what is being read in context and you have to 'buy in' to the story, for it to be found appealing.

As the layers are ascended in Figure 1, it becomes harder to achieve the next ability e.g. making code readable is one thing, making it fully understandable is hard, while making it totally believable under all circumstances is harder still.

However, this model is rather too simplistic. We can read code, but it may not make any sense or it may need a long time to cogitate what we have read before we get any meaning out of it. If we get to the stage where we now understand what is trying to be achieved, the code may not convince us that this will always be achieved. However, this seems a rigid transition from one layer to another.

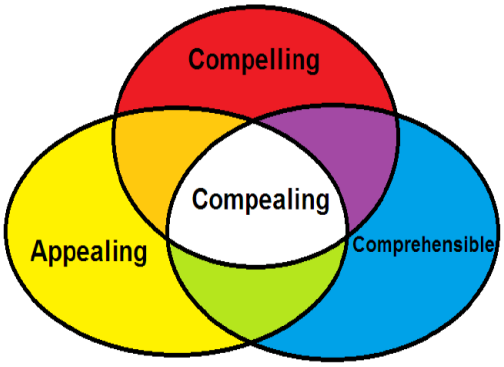

Figure 2. The overlapping model

Another model is demonstrated in Figure 2 which shows that the transition from one layer to another doesn't necessarily occur in an ordered manner. Rather, source code can show degrees of these abilities.

It also merges Readability with Understandability, as the code must be **comprehensible** as a bare minimum – if it doesn't reach this standard it is in severe trouble. Comprehensible describes code that is simple and understandable and doesn't hide anything in complexity.

Attractiveness may denote several things, such as the design of the code, how well identifiers are named and the absences of duplicate code. Such **appealing** properties adds not only beauty, so that it is easy on the eye and enjoyable to read, but allows the code to be comprehended very quickly.

For the software to be trustworthy, it must be **compelling** so we are sure that there are no hidden issues, it is robust, secure and the intended behaviour is reliably achieved when the code is run.

When the code hits the sweet spot in the middle of Figure 2, so that it is comprehensible, appealing and compelling, it can said to be ***compealing***.

Compealing is a portmanteau word like avionics, Brexit, brunch and webcast. Compealing has been formed from **comprehensible**, **compelling** and **appealing**, as sometimes it may be easier to treat these three concepts together.

Who assess if code is compealing?

The code can be tested when it is run and its interfaces checked for suitability – this is where the quality assurance (QA) department determines if the software is a candidate for release. However, QA don't normally assess the quality of the source code by reviewing it! Someone else must review the code, assess it, noting any defects and omissions and hopefully point out the good parts as well as any areas that need to be improved. Often, colleagues are the ones who carry out the review.

Tools can be used to augment human reading and analysis of the code. Later it will be noted the role of compilers, static analysis tools, deobfuscators and other tools that can be utilised to visualise the source code in different ways. However, it must be stressed that *Compealing* is evaluated on the source code and not on the running software.

Balancing

The three *Compealing* constituents do not exist in isolation, but to varying extents, overlap with each other. In fact, we have achieved compealing code when the code exhibits all three where they overlap. However, they also must exist in balance. If one constituent is taken too far, it can adversely affect one or both of the others.

One example of this balance could be that making the code more *Compelling* in a thoughtless manner may detract from its attractiveness making it less *Appealing* such as when adding code to validate data in inappropriate places. Making code more concise, can help make the code more *Appealing*, but when it becomes too terse (the code is dense), it can slow down or hinder comprehension therefore making it less *Comprehensible*.

It can therefore be seen that when working on a particular aspect of *Compealing*, the correct balance must be borne in mind so that the work does not detract from the overall goal of increasing the *compealingness*.

Personal perception

Personal taste can have an effect on the perceived compealingness of code, but is likely to have a larger effect on *Appealing*, less on *Comprehensible* and little on *Compelling*, as demonstrated in Figure 3.

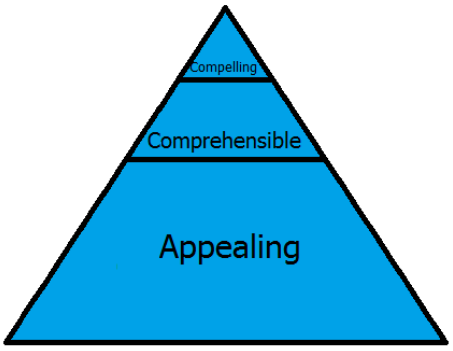

Figure 3. Varying effect of personal preferences on each concept

The reason is because beauty is held in the eye of the beholder. Often the code is either compelling or it is not e.g. it handles security and exceptions well or it does not. Personal preferences on layout

and style may have an effect on how comprehensible the code it, but not to a large extent, although it can affect the speed of this process. Experienced programmers usually acquire the skills to comprehend code presented in various different formats. However, the appealing quality of the code can be perceived differently by two people, according to their personal preferences.

Useful tools

With compiled languages, the compiler can help to some extent in analysing your code, generating different levels of warnings and tips etc. However, the compilers main job is to convert your high level language into binary or byte code that the OS or runtime environment can understand, so analysing your code for potential problems is not their highest priority.

Compilers are normally good at finding simple issues such as unused or unassigned variables or paths within a function that don't return a result. You can often configure the compiler to customise the warnings it either ignores or reports, so personal practices that are acceptable, don't continue to appear in the warnings.

However, to get a more detailed analysis of your code you need a different tool such as a static code analyser. You can consider such a tool as a debugger that doesn't work at runtime. They can be good at finding issues that the compiler and the programmer has missed, but could potentially cause issues. Examples of the issues that can be found include buffer overflows, SQL injection, cross site scripting and using components which have known vulnerabilities. However, such tools can also create false positives, so you need to be able to configure their output, so that only genuine concerns are raised. Getting the balance right can take time.

There are organisations that track and document the most common issues within code. There are tools which use these known problems and can analyse source code to find and highlight such problems. Examples of organisations that document vulnerabilities include OWASP and CISQ.

There are also tools that can take source code that has been obfuscated and then deobfuscate it, to make it more readable to human beings. There are several reasons to obfuscate code, some are to save space and download time, while it can also be used to protect intellectual property rights. However, hackers can also obfuscate code to prevent it being easily examined, so malevolent code in the software cannot easily be identified.

Code formatting tools, sometimes called beautifiers, can be used to change the code so it conforms to an organisation standard. This may enforce style and layout, such as ensuring a standard type and amount of indent used and where code blocks begin and end. Often programmers' use their own personal preferences when they write code, even though there may be organisational standards they should be following. However, passing committed code through a code formatter as part of the development or build process can enforce such standards and hence make the codebase consistent.

Summary

The rationale for the use of *Compealing* has been examined as well as getting the correct balance between the three aspects. Perspectives have been touched upon as well as tools that can be useful in keeping code compealing have also briefly been considered. Future articles will look at the concept in more depth with examples.

[1] A Brief Look at the Attractiveness of Source Code, available at http://www.iap.org.uk/main/wp-content/uploads/2016/07/OnThe-AttractivenessOfSourceCode.pdf