A Further Look at Compealing Software

Paul Lynham FIAP

Introduction

Software has become ubiquitous in modern society and is found in places that would have been unimaginable just a few years ago. However, although software can bring extended functionality and flexibility to devices and services we use, it can also cause problems. The trustworthiness of software is a major concern which covers security, reliability and its behaviour amongst other aspects. Cyber incidents see a 1087% increase year on year[1], yet the production of good quality software can go a long way to combat these issues. It has been calculated 90% of security incidents result from exploits against defects in software[2].

Trustworthiness is therefore directly related to how well the software is written and this depends on several factors including the specification, architecture, design, implementation and testing. However, for this consideration, the coding of the software will be the main focus of this series of articles and how the quality of the source code can be assessed using the concept of *Compealing*.

What is compealing again?

To recap, compealing is a portmanteau, in simple terms, a word made by joining two or more words together. Compealing is formed from 3 related concepts, used to describe computer program code, but can be applied to other concepts, for example, architecture and design:

Comprehensible – Understandable, Comprehendible, Clear, Intelligible, Lucid, Graspable
Compelling – Convincing, Persuasive, Undeniable, Captivating, Enthralling, Gripping
Appealing – Attractive, Interesting, Pleasing, Likeable, Engaging, Enticing, Grabbing

Thus compealing code can be clearly understood, convinces you that it works and is attractive and pleasing to read. The three ideas can be combined because they encapsulate the properties of well-written code – code that is written by programmers so it can be read and understood by other programmers. They are also linked by the idea of being able to grasp the intent of the code – notice this theme in the last meaning of each word shown above.

Why is compealing important?

Software compilers and interpreters are great at 'deciphering' poorly written code, producing machine instructions from the source code, with little effort. However, humans are not so adept at understanding such code. Poorly written code is not pleasing to read, it may not fully explain itself and the reader may be unconvinced that this code will work under different circumstances when it is run. As Martin Fowler has noted:

> *Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*

This is the essence of compealing. Code is read many more times than it is written. It is read by programmers long after the original programmer wrote the code. Thus code should not only be written so the author understands it, but it also needs to be written well enough so every programmer in the future that reads the code should find it compealing.

If code is not maintained, it dies. Business processes change. In modern society, change often occurs frequently and new ways of using the software are regularly found. For the software to remain

useful, it must also change. When code is easy to read and understand, it makes the task of changing it easier – you must understand what the code already does before you can change it. In medicine, the concept is called 'Do no harm' i.e. you must not do anything to the patient which would make their condition worse. With software, the same caution needs to be applied.

A professional programmer should read and fully understand the existing code and then consider the changes that are required to meet the new requirements and then assess how these changes could impact the system. Changing the software to enhance its capabilities in one area, while inadvertently causing errors or unwanted behaviour in other areas is an undesirable effect.

Where is compealing code written?

Software is malleable by nature, but this advantage can also be a disadvantage as the software ages. Code can become brittle i.e. become easy to break and hard to change. This is because the changes may make the software more difficult to understand e.g. the original design may have morphed so much, that its current intent is no longer clear. Such scenarios are considered to be 'Technical Debt' which is the accumulation of less than optimal code quality over time.

At some point, the code needs to be updated (refactored), to bring it back to an acceptable state. This may involve changing the design as well as the code and rewriting the code in several places. Besides ensuring the new code doesn't change the software's behaviour (the use of unit tests is a good solution), the new code needs to be written using a compealing approach, so its intent is clear and unambiguous.

Thus compealing software should not only be written with new functionality, but also when refactoring software. This latter concept has been labelled the 'Boy scout rule' – boy scouts always leave the environment they use in a better condition than when they arrived. This means every time you touch the code base and make any changes, it should leave the software in a more compealing state than it was before.

Some compealing attributes

For code to be compealing, it should be amongst other things, readable, clear, being as simple as possible, logical, transparent, with no 'code smells', while at the same time abiding with current best practices, such as those promoted with 'Clean Code'.

It is said beauty is within the eye of the beholder. However, there are many attributes that are commonly held to make code appealing. In other papers[3,4], I have looked at attributes which have an influence on making code attractive and a few are summarised here as an example.

Consistency is one which should be immediately apparent. If you read code and find there is more than one style, it can be disconcerting. Therefore the whole codebase should have a consistent format and style, so wherever you look, you will see nothing that gives you an initial uncomfortable feeling. This result can be achieved in several ways. The first is having style and a coding standard that mandates the expected look. Code can be checked as part of a code review. The software can be run through a 'beautifier' or a code formatting tool, which will ensure committed code abides by the set standards. This latter solution can be used as a backstop as part of the build process. There may be other standards used, such as those that lay out how user interfaces should look and behave.

Besides their style, the names of identifiers are important, so their purpose and meaning are immediately apparent. Thus constants, variables, types, fields, methods and classes etc. should use a

consistent rule for naming. One example is to use a noun for a variable, with a verb plus noun for a method. The use of one character names for identifiers should be avoided, with the possible exception of looping variables that have no intrinsic meaning. The aim is to write self-documenting code.

Understandable OO code will use classes that reflect the entities of the problem domain while keeping each class with a single purpose. Unit tests that exercise how the class should be used can also help understanding. Where doubt exists if a behaviour is not explicit to this class, then it should be moved elsewhere. As an example, a 'person' class may hold information on a person's name, address and contact details. It could be considered that code that validates the postcode is not really part of a 'person', so such validating methods should not reside within the person class. If a person has more than one address (home, work, holiday) then the address could be moved to its own class and these addresses linked to the person.

If the functionality demands, the person could be decomposed to encapsulate a name, date of birth, address and contacts (such as home phone, mobile phone, email etc.) which are all implemented as their own classes. Using the name class, the person's name could be returned as a full name, as family name/given name order or as initials e.g. GetFullName(), GetFamilyNameGivenName(), GetInitials(). The theme here is that smaller concepts are simpler to understand, especially when they are well delineated.

Code is convincing when it seriously takes account of security, exception handling, persistency, accuracy and transparency amongst other considerations. As a simple example, where a case or switch statement does not consider all possible alternatives, then a concern would be what happens when one of these situations arises? One solution is to use a 'default' or 'else' case, where unknown cases can be gracefully handled, such as raising an exception and logging the event. Defensive programming assumes things will go wrong and puts in place code that will handle or mitigate this.

Summary

It should be plain that each aspect of compealing is a subject within itself and these have only been briefly touched upon here. However, it is always worth reviewing the code you have written before committing it. Ask yourself some questions, such as:

- Do I really understand everything?
- Is this code self-documenting?
- Could a junior programmer understand the code I have written?
- Could I explain everything I have written in detail to someone else?
- Will I be able to understand this if I read it again in 6 months' time?
- Can it be made clearer?

There are numerous detailed checklists that can be used, such as in the book *Code Complete*, which contains comprehensive checklists e.g. ones for considering routines.

There is a lot to consider! It is often easier to recognise compealing code than to write it yourself!

[1] https://www.ftadviser.com/your-industry/2019/07/01/number-of-cyber-incidents-jumps-1087/
[2] https://www.csoonline.com/article/2978858/is-poor-software-development-the-biggest-cyber-threat.html
[3] http://www.iap.org.uk/main/wp-content/uploads/2016/07/OnThe-AttractivenessOfSourceCode.pdf
[4] http://www.iap.org.uk/main/wp-content/uploads/2017/01/A-Further-Look-at-The-Attractiveness-Of-Source-Code.pdf