Compealing Code Chemistry

Paul Lynham FIAP

Introduction

After examining the concept of **Compealing Code** over previous articles, it is good to look at an extremely simple example, which will demonstrate and assess each attribute. While examining some code snippets to determine what makes code comprehensible, appealing and compelling, the following table of 'symbols' will be used. This will help highlight which attributes of *Compealing* have been achieved, are yet to be achieved or maybe in an intermediate state (undecided).

Compealing			
Attribute	**Comprehensible**	**Appealing**	**Compelling**
Symbol	Ce	A	Cg
Compliant	✓	✓	✓
Non-compliant	X	X	X
Undecided	?	?	?

Compealing Code Example

The following code snippet can be read, but our understanding is limited as it is hard to contextualise how this method is used.

```
double a(double x, double y)
{
    return x * y;
}
```

Ce A Cg
? X X

What do the identifiers *a*, *x* and *y* represent? Such questions limit our comprehension of this method, as if we have to ask such questions, it is not obvious what is happening. This method could also easily be manually inlined without losing the partial understanding we already have, so the justification for a method is limited:

```
r = x * y;
```

As can be seen, the inline version is more succinct than the method call below – it uses one less identifier and it is obvious what is being assigned to *r* so is simpler, whereas without looking at the method's implementation for *a,* the result is not obvious:

```
r = a(x, y);
```

Of course, using one character identifiers do not help as both the method name (*a*) and the parameters (*x* and *y*) do not give us much of a clue.

Inlining a one statement method is not usually justified when they are well named and called more than once, as such methods can both improve code comprehension by imparting further meaning,

as well as isolating the algorithm being used to a single location. This technique can be useful when, for example, a well-named method replaces a one-line complex calculation.

The next example extends the understanding a little more from the first example:

```
double area(double l, double w)
{
    return l * w;
}
```

It suggests that the method returns an area. We could guess that *l* and *w* parameters stand for *length* and *width* and we could further guess that this would return the area for a rectangle?

Using a method name that is a verb-noun combination and parameter or variable names that make their purpose plain, would be a big improvement:

```
double calculateRectangleArea(double length, double width)
{
    return length * width;
}
```

The above example produces a more appealing rendition because the intent is explicit. Yes, the method calculates the area of a rectangle, given its length and width as parameters. This example has therefore removed any doubt we may have from the previous example and we now know for sure what the intention is. We still have to make some assumptions, such as the units of `length` and `width` are the same, but even so, this code is becoming more compealing. However, would this work under all circumstances?

The subsequent example demonstrates this method will only accept arguments with positive non-zero values i.e. you can't have a negative area or a rectangle with a length or a width which are zero.

```
double calculateRectangleArea(double length, double width)
{
    if ((length > 0) && (width > 0))
    {
        return length * width;
    }
    else
    {
        string arguments = String.Format("length={0}, width={1}", length, width);
        throw new System.ArgumentException("Dimensions must be greater than zero: ", arguments);

    }
}
```

This is now compelling as it demonstrates the author has thought about defending against invalid inputs. Because it notifies us when there is a problem, you can't justify manually inlining this code. It has now become trustworthy and is compealing as it can be read, fully understood while being appealing and compelling.

Discussion of Examples

The method above could be part of a geometry utility class, which could be static and contain similar methods such as *calculateCircleArea()* etc. However, if the method is part of the implementation of a polymorphic class, say for example *Shape*, then the name could be changed to *area* so that it could be called on a *Rectangle* or a *Circle* derived from *Shape*. In such circumstances, then the combination of the object name and the method/property makes the meaning obvious, assuming `rectangle` is an instance of *Rectangle*:

```
tileCover = rectangle.area();
```


This last example could be further improved by renaming `rectangle` to the object it represents, such as a tile or more explicitly, the type of tile e.g. wallTile.

It may be more appropriate if the dimensions of the shape are part of the class and so does not need to be passed as parameters, as demonstrated above. The method signature can be processed quickly if well named, but parameters are a little more difficult. Zero parameters help to make the method easier to read – the brain doesn't have to process and comprehend each parameter, what it is supposed to represent, its type and any constraints it has. By minimising the amount of information about a method (most humans can keep 7 ± 2 items in short term memory[1]), the reader can keep the whole concept of the method in memory, without having to repeatedly re-read it.

In this case, the defensive programming could be moved to the dimension's property setters like so:

```
private double length;

public double Length
{
    get { return length; }

    set
    {
        if (value <= 0)
        {
            throw new System.ArgumentException("Length must be greater than zero.");
        }

        length = value;
     }
}
```

Only the *Length* dimension is shown above, but similar code would be used for *Width*. This would prevent invalid dimensions being assigned in the first place and so make the checks in any methods using those dimensions redundant. The implementation of the area method would then become cleaner while remaining compelling:

```
public double area()
{
    return length * width;
}
```

Refactoring

It is good to write compealing code from the outset, keeping the codebase both easy to understand and a pleasure to read while knowing it is also compelling. This latter attribute can be tested by utilising unit tests that exercises, for example, all outward-facing methods to ensure that the desired behaviour can be confirmed.

However, when reading code that has been written previously or by different authors, which may not conform to a compealing standard, there may be a need to refactor the code to raise the standard and bring it in line with the other compealing code. This may not be just a matter of using better-named identifiers or improving the persuasiveness or the code. The design may need to be improved, such as changing classes so they conform to a single responsibility and in a similar manner changing their methods so they do one thing only and have no side effects. Cohesiveness may need to be improved while ensuring classes are not tightly bound to each other.

In Martin Fowler's *Refactoring*[2] book, a catalogue of refactoring patterns are detailed, together with a walkthrough of an example application, using techniques to recognise poorer code and transforming it, which results in an improved structure and design. Some examples are 'Extracting a method', 'Encapsulate field' and 'Introduce explaining variable'. Utilising such techniques will lead to better design and help achieve more compealing code.

Parting Thoughts

The example covered is a simple one and often the *Appealing* attribute will be assessed against code structure.

Most programming languages offer the programmer much scope to make their programs compealing. As an example, choosing the name of each class and the name of each method is the programmer's opportunity to explain what they intend; the language doesn't enforce this, but the responsibility is with the author. If it is not exploited to get the best payback, then it is a missed opportunity.

The same can be said about duplicate code - the ideal is to keep the code DRY (**D**on't **R**epeat **Y**ourself) otherwise the code becomes WET (**W**rite **E**verything **T**wice).

Even simple measures such as not having nested levels of branching within a method, can keep methods short and sweet and simpler to comprehend. However, as with many things, this is a balancing act. Creating many small methods may decrease complexity within a method, but overall, it may increase the amount of interface that needs to be dealt with (even if they are not public methods), as there will be many more method signatures to understand.

In the same Martin Fowler book, Kent Beck discusses things that make programs hard to work with, which he lists as follows:

- *Programs that are hard to read are hard to modify.*
- *Programs that have duplicated logic are hard to modify.*
- *Programs that require additional behaviour that requires you to change running code are hard to modify.*
- *Programs with complex conditional logic are hard to modify.*

Avoiding complexity, making code both attractive and easy to understand so that readers of your code find it transparent (it does what it says on the tin), should be kept in your sights at all times.

---

[1] https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two
[2] Fowler, Martin (1999), Refactoring: Improving the Design of Existing Code, Addison Wesley.