A Brief Look At The Attractiveness Of Source Code

Paul Lynham MSc PGCE FIAP

Introduction

In using FURST[1,2], the product must be shown to be **F**it **F**or **P**urpose (FFP). One of the attributes of FFP is **Attractive**. This states that for developers the design and code should be appealing. To go just a little deeper, it is appealing because it is simple, it is easy to understand, there is no doubt in what it is supposed to do and it abides by conventions, making it uniform with its surroundings, as opposed to being a 'blot on the landscape'!

The attractiveness of source code is a non-functional requirement and because of this fact, it has often been given cursory attention, even though failure to produce quality usually has some cost[3]. The testing is mainly carried out against the functional requirements, so testers are not going to complain about the look of the code! Often managers are only concerned with getting a working product out of the door. This usually means it is down to the programmers themselves to ensure that the structural quality of the software is good and this includes how easy the code is to read, understand and maintain.

If you are familiar with FURST you will also know that concepts often overlap and affect each over. This is true for **Attractive** and this concept can overlap with **Labelling**, **Intention** and **Transparent**, as in making the code attractive (such as using good naming), its intention becomes clear (transparent).

Legibility, Readability and Understandability

Readability is the ability to understand written text. The Readability of source code is often taken to be the ability to read and recognise code at a statement level. Steve McConnell defines Understandability as[4]:

> "*The ease with which you can comprehend a system at both the system organisational and detailed statement levels. Understandability has to do with coherence of the system at a more general level than readability does*."

There are many factors that affect the readability of text and researchers have examined factors such as speed of perception, perceptibility at a distance, perceptibility in peripheral vision, visibility, reflex blink technique, the rate of work (reading speed), eye movements and fatigue in reading[5].

Legibility is the ease with which a reader can recognise individual characters in the text. Factors that affect this include the font, size, colour, capitalisation, serifs and weight etc. It differs from readability in that readability involves recognising words, lines and statements.

It has been suggested[6] that the three concepts of legibility, readability and understandability can be layered, with legibility at the bottom, readability above and understandability at the top. However, just because code is legible, readable and understandable, it does not follow that it is believable! Magicians make all sorts of things disappear in front of your eyes, but does this mean you sincerely believe the person or object has become invisible?

It can be argued that there is at least another layer on top which measures the ability of the code to persuade the reader that it actually does what you think it does. The following diagram sets outs these layers.

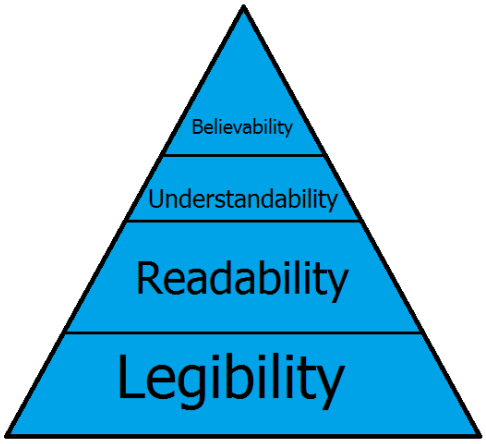

Figure 1. The abilities of source code.

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 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 is harder still.

Should the code be well written so that it meets all these criteria, then we could say it is attractive. Of course, we can use alternative words for some of these abilities, such as comprehensible instead of understandable and compelling instead of believable.

Compelling code

The following snippet checks if the weight of a parcel is neither 5Kg nor 11Kg and then block A will be executed, otherwise block B will be executed. How compelling is this code?

```
if (ParcelWeightKg <> 5) or (ParcelWeightKg <> 11) then
begin
  //Execute block A
end
else
begin
  //Execute block B
end;
```

What range of values must `ParcelWeightKg` be to execute block B?

The answer is that it will never be executed! It can't be both 5 and 11 at the same time, so because of the 'or', either the first or the second test will be true, making the whole expression true. This demonstrates that code can be readable and seemingly understandable, yet not necessarily compelling.

The intent could have been coded by reversing the logic and looking for a positive match with 5 or 11 first:

```
if ParcelWeightKg in [5,11] then
begin
  //Execute block B
end
else
begin
  //Execute block A
end;
```

Compealing

You may be wondering what **Compealing** is? It is a portmanteau like avionics, infotainment and webcast[7]. **Compealing** has been formed from *comprehensible*, *compelling* and *appealing*, as it may be easier to treat these three concepts together.

Comprehensible describes code that is simple and understandable and doesn't hide anything in complexity. **Compelling** paints a picture of code demonstrating what it is supposed to do. **Appealing** is code that is easy on the eye and thus enjoyable to read. Because these three concepts overlap frequently, **Compealing** can be used instead of using three separate words.

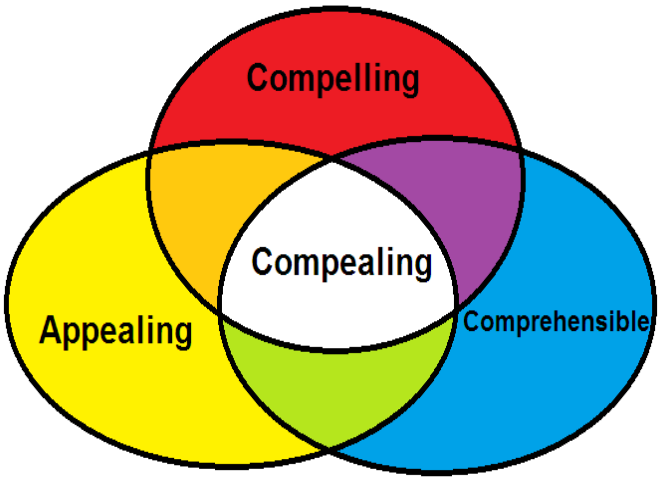

Figure 2. The overlapping model

The author feels that the layered idea shown in figure 1 may be too simplistic. Let us assume that code is presented so that it is legible e.g. the font and background colours are not the same! Then we can read the code. However it may not make any sense or it may need a long time before we realise what it means – it is not appealing. At this point it may not be compelling either e.g. it can be so complicated that although we now understand what it does, we can't be sure that this will work under all circumstances.

It may also be possible to read code that is not appealing, yet it is comprehensible and compelling. Thus although it is unlikely that appealing code is compelling without being comprehensible (the orange area in figure 2), it is possible to be in the green and purple areas without stretching our imagination. Of course it is the sweet spot in the middle where we would like our code to be.

How do we make code compealing?

Much has been written about *Clean Code*[8] and the idea of making code **self-documenting**[9]. It says that code should be continually assessed for clarity and refactored (Boy Scout Rule). One way of making code self-documenting is to make it look like English. Some people may be happy for their code to look like mathematics while others are attracted to cookery (especially spaghetti). However, whatever your favourite subject is, if it is written in plain English, it is easy to understand.

What does the following snippet of code do and what is the value of *c* after execution?

```
int a=5,b=5,c;
c=a+++++b;
```

We only have 3 variables and 2 lines of code, so this should not be too difficult to understand?

There are a few issues here. First is a question of context. If this is part of a larger body of code, you may ask "what do *a*, *b* and *c* represent"? From these names, we don't know! One character variables don't seem to give us any clue. Secondly, this is a straightforward calculation, but it doesn't look like one. The reason is the formatting. The compiler doesn't mind about spaces around operators, however, if we insert a few spaces the intention becomes a little clearer:

```
c = a++ + ++b;
```

The second version[10] is far easier to read and be understood by a human being. Code is read many more times than it is written, so it is better to write it correctly the first time, hence other developers (or even yourself looking at this again 3 weeks later) will understand. A little whitespace can go a long way in making code compealing. By the way, *c* should hold the value of 11[11].

Whitespace consists of any character that can't be seen such as the space, the tab, a carriage return or form feed. All these can be put to good use in helping code become more compealing. Inserting blank lines between sections of code such as initialisations, branching and iteration etc., helps in allowing the mind to match patterns that we are instinctively wired to recognise.

Naming

Years ago when computers had limited memory, one character variable names may have been defendable. There may be a case for using *i*, *j* and *k* for looping variables, where these variables don't hold any intrinsic meaning. In any case, non-mathematicians or developers who have not used FORTRAN may not understand the reason why these particular letters are used.

The code in the last example can be improved by meaningful variable names and some thoughtful formatting, perhaps a coding standard that states "operators should be separated from code elements by a single space". Standards for naming variable, constants, enumerations and classes (mainly nouns) and methods (mainly verb and object combinations) will help in improving understanding and readability.

> "*There are two ways of constructing a software design: one way is to make it so simple there are obviously no deficiencies, and the other is to make it so complicated that there are no obvious deficiencies.*" – Tony Hoare

There is a whole chapter in *Code Complete* dedicated to naming because it is such an important aspect of making code compealing. When you read code you need to quickly progress through it and not be stuck at a point where you can't understand what a particular symbol means. You should normally know if this is an object, a variable, constant or a method etc., (it is assumed you know the keywords of the language) and this will help in the next task, which is understanding. If the code is compelling, then eventually, you can put the whole thing together within the context and it has become compealing i.e. you have reached the zenith in figure 1 or the sweet spot in figure 2. Compealing code allows you to reach this point quickly.

Code for mathematicians

One character identifier names may be appealing to mathematicians as they are happy using algebra. In some circumstances, such identifiers may seem self-explanatory, as the following example may demonstrate:

```
function HasMassEnergyEquivalence(E, m : Double) : Boolean;
begin
  Result := E = m * c * c;
end;
```

Einstein's formula **E = mc^2** is one of the most recognisable equations, so it may appear the above function needs little explanation, especially to a mathematician. However there are a few problems here. The first is that direct comparisons between floating point types have issues. The second is perhaps the parameters may want to follow the same order as the function name (for positional parameters) i.e. *m* for Mass should be first and *E* for Energy second. Thirdly, what units are *E*, *m* and *c* measured in? It is assumed *c* is a constant defined elsewhere, but is this defined as miles per second or metres per second?

Taking into account some of the problems highlighted above, the following amendments could be made:

```
function HasMassEnergyEquivalence(
m,                                      // m = Mass measured in Kg
E : Double                              // E = Energy measured in Joules
) : Boolean;
begin
  Result := SameValues(E, m * c * c); // c = Speed of light in metres per second
end;
```

Is this an improvement or is the following better?

```
function HasMassEnergyEquivalence(MassKg, EnergyJoules : Double) : Boolean;
begin
  Result := SameValues(EnergyJoules, MassKg * LIGHT_SPEED_METRES_PER_SEC *
                                              LIGHT_SPEED_METRES_PER_SEC);
end;
```

This version leaves no doubt as to the units being used, resulting in the comments becoming redundant. It makes it plain that there are parameters and a constant being used. Of course this really comes down to the audience being targeted. If these are mathematicians who know the units and the context in which this is used, then the one character identifier version may be adequate (if the audience never changes). If this is not the case, the last version may be better. Mathematicians can also understand the latter, so it could be the lowest common factor!

Comments

Instead of the end of line comments above, *JavaDoc* type tags could have been used such as *@param* so that a documentation tool could extract the tagged information and use it to create context sensitive help in an IDE or help files.

The greatest problem with comments is that they are ignored by the compiler or interpreter and therefore do not need to be current nor correct and are often neither! *Clean Code* principles as well as current refactoring ideas prefer comments to be eliminated where possible. This school of thought states that comments can be used where a library is being developed for third party use or when an explanation is impossible to be expressed in the code, such as domain knowledge or a fudge which cannot be eliminated. However, comments should be the last resort after all other possibilities have been ruled out, such as making code self-documenting. Code has to be maintained whereas comments rarely are.

Get closer to English

How far should we go in getting the code to read like English? Here is an example snippet which deals with a bank account:

```
if Account.Balance < 0 then
begin
  //do something with an overdrawn account
end;
```

This code would appear to be straightforward in its intention. We have an *Account* object and are calling a member named *Balance* to test if the account is overdrawn (it is irrelevant at this point if *Balance* is an attribute, property or a method).

Account could possibly be renamed to reflect what type of account we are dealing with depending on the context. If we are dealing with a homogeneous collection of accounts and manipulating each account polymorphically, then this name may be appropriate. However, dilemmas may arise in becoming more specific. You may think if you are iterating a collection of accounts then the account you are currently manipulating could be called *CurrentAccount*? However, this may be misleading – a *current account* is an account that you can draw cheques on (sometimes called a checking account) as opposed to a *deposit account* or *savings account* used to save money. Thus, ambiguity over its type or temporal status could ensue.

Naming is extremely important and in considering the name of an entity it is vital that there is no confusion to what it is or what it does. A few minutes in thinking about what an entity should be called is a good investment as it could save many times this amount of effort in future when someone else reads the code and tries to understand what is happening. Consider if the name could be misconstrued. It is also important to be consistent in naming.

There may be some organisational coding standard for naming. For example, it could state that an entity currently being worked on should be prefixed with the word *The* such as *TheAccount*.

Going back to the snippet, we could refactor this code so that a new method is created that returns True if the *Balance* is below zero or False otherwise. We could call the method *IsOverdrawn()* - it is a common

convention to prefix functions which returns a Boolean with the *Is* prefix. In fact, the use of prefixes and suffixes can help the code pack much more compealing power. Prefixes such as *Is*, *Has*, *Min*, *Max* and suffixes such as units of measure – *chars*, *bytes*, *msecs* etc., make identifiers more explicit and understandable.

Thus, our example would look like this:

```
if TheAccount.IsOverdrawn then
begin
  //do something with an overdrawn account
end;
```

This would seem to take a statement that was originally more mathematically orientated to one that is more aligned to natural language. In fact, changing the dot notation for a space and inserting some spaces, we have a near English sentence – *if the account is overdrawn then…* – so that it becomes self-documenting.

A step too far?

In some OO languages, it is possible to have routines that do not belong to a class – these are standalone routines. It may also be possible to have local routines like local variables. In some languages, the implementation of the method is within the class definition while with others the class is defined in one section and implemented in another. In any case, depending on the language and the scope of the account entity, it may be possible to do this:

```
function TheAccountIsOverdrawn : Boolean;
begin
  Result := TheAccount.Balance < 0;
end;
```

Then our example would look like this:

```
if TheAccountIsOverdrawn then
begin
  //do something with an overdrawn account
end;
```

Or again, depending on style conventions, we could do this:

```
if The_Account_Is_Overdrawn then
begin
  //do something with an overdrawn account
end;
```

Thus, we have now eliminated the dot notation in the main body of the code!

How far should we go in trying to make our source code read like a book? Some would argue the last steps may have gone too far and in fact trying to subvert the dot notation is fighting the language. This strategy can be seen sometimes in developers that move from one language to another – they try to impose the style of their previous language on the new language. This is a mistake. It is akin to an English speaker learning French but sticking to putting the adjective before the noun – sacré bleu!

The opposite should be the strategy – make use of the strengths of the language! As an example, Pascal type languages are strictly typed (the compiler is your friend) and can make use of subranges and set

manipulation amongst others, while Python exploits dynamic typing and you can use heterogeneous collections and 'Monkey Patching' to good effect (the interpreter is your friend). Whatever language you use, it will have some features that make the language powerful in some way and you should aim to exploit these (where they are not to the detriment of maintainability).

Some of the core philosophies of Python[12] state:

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- Readability counts

If these and the other core Python philosophies are followed carefully, then the resulting code will be much more compealing.

> *"The most complicated skill is to be simple."* Dejan Stojanovic

Routines

Routines make up the vast majority of the total lines of code of a modern programme. Although some languages like COBOL had a lot of code in its setup divisions and sections, most modern languages don't have large initialisation sections and shouldn't have monolithic mains. Therefore, it is worth looking at improving routines to get the greatest return on our efforts.

How can we make routines compealing? This should not be difficult by using a few basic rules that are summarised as points here but more detail can be found in *Clean Code* and *Code Complete*[13] books:

- Get the prerequisites right – define the problem to be solved.
- What are the motivations for creating the routine?
- Consider how this is going to be tested.
- It is either a command or a query, but it can't be both[14].
- Good names that state what it does – usually a strong verb followed by an object.
- Short Parameter list.
- Consider design by contract - what are the pre and post conditions?
- Consider defensive programming.
- Does one thing and one thing only – SRP (Single Responsibility Principle).
- All statements at the same level of abstraction – SLAP (Single Layer of Abstraction Principle).
- Strong cohesion – highly focussed and related functionality.
- Loose coupling – minimise dependencies (code to an abstraction, not an implementation).
- Produces no side effects.
- Keep it simple stupid (KISS) first – only optimise later if necessary.
- Clean up any resources used.
- Consider how errors and exceptions are handled.
- The whole routine can be seen on the screen without scrolling.
- Ideally, they should be implemented directly after they have been called the first time.

Smaller specific routines are easier to understand and offer less hiding places for errors and security loopholes[15], so it is better to have lots of smaller routines rather than a few larger general purpose routines.

> "*Controlling complexity is the essence of computer programming*" — Brian Kernighan

What else makes code attractive?

Wouldn't it be nice to have some constraints that when followed would make code attractive? The question about human beauty was given an insight by Langlois and Roggman[16] who found that the most attractive faces were composites i.e. a photograph averaged out from many faces. It seems that averaging out features seems to get rid of the unwanted features! Also, the morphing produced a better symmetry and this is also seen as attractive.

In terms of code this can be seen as producing code that blends in well with the rest of the code - it is consistent with no unwanted deviations. The unwanted features can often be mapped to special cases and it is common to find these in code. You may have heard the expression:

> "*the exception that makes the rule*"[17]

Even though this is often misused, in many fields you will hear a rule of thumb, with some sort of exception, such as:

> "*I before E except after C*" (but what about sc**ie**nce?)

> "*Only eat pork in months with an 'r' in them*" (i.e. all months except May to August)

Applications are bound to deal with such special cases and often the code that deals with these stick out like the notorious "sore thumb". So how can these special cases be dealt with in a compealing manner?

Martin Fowler[18] suggests using a subclass that provides special behaviour for particular cases:

> "*… return a Special Case that has the same interface as what the caller expects*"

This may work where there are a limited number of special cases. If there are standard cars, but also lots of different special cases such as producing a car that has different combinations of add-ons such as leather seats, metallic paint, fuel type, alloy wheels etc. the problem needs another approach.

One solution may be to use the **Decorator pattern**. In the classic Gang of Four Design Patterns book[19], this pattern is defined as:

> "*Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality*."

Yet another solution may be to use the **Visitor pattern**. From the same book, this pattern is defined as:

> "*Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates*."

There are many ways of solving this old problem, but by using an appropriate solution that makes the code look like a generic solution, rather than a special case, the code benefits from being reusable and extendable while making it look more symmetrical!

When a developer sees a pattern, his or her understanding switches to a different level and is immediately aware of the intimacies associated with a particular pattern.

Can we measure attractiveness?

Static analysis[20] tools can look at code and generate metrics that may be useful, but can they tell us if code is attractive? At this moment in time, it looks as though there are few simple direct metrics that can be used. Metrics which will give figures for min, max, median, mode, interquartiles etc., for lengths of identifiers and lines, number of parameters passed, executable lines in routines, lines of comments etc. These can be useful if you put them together.

As an example if you find the average number of lines of code in a routine and then look at the lines of comments in a routine, you can calculate the percentage of comments in a routine and this will give you an indication of possible code smells[21] (heavily commented areas of code could point to code that is difficult to understand). The tool needs to identify the worst cases to be useful, as averaging out might make the figures look reasonable and so bare statistics may be meaningless.

Let us look at the routine signature. If the average length of a routine name is 18 characters, then that may sound pleasing? However, what do you think of this example:

```
int aaaaaaBbbbbbCccccccc()
```

That doesn't mean a lot, although it is 18 characters long! Even if the name makes sense (to humans) we need to know the context. If it is a method, what is the name of the class and does this method fit expectations for such a class? This is the same for most of the metrics produced by such tools. They are good to get overall figures, but really don't tell us a great deal about attractiveness. However, the following metrics can be useful.

The number of parameters per routine may be an insight, especially if you can track ones down with more than 3 parameters so it is known where to refactor.

The number of local variables used for each routine is useful as this may point to how complex the routine is (if there are a high number, then the code is likely to be more complex). Another measure of complexity is the McCabe cyclomatic complexity measurement[22], which measures the number of linearly different paths through the code – thus the more branching (if then else, case/switch, ternary operator, goto, pattern matching) used, the more complicated the code.

With iteration (depending on the kind), common errors are that the loop is never entered, never leaves or is out by 1. It is also useful to know how many exits there are from the routine - ideally, there should only be one.

Metrics that measure such structures may be useful in deciding if a routine is too complex and perhaps is a candidate for further decomposition. However, they don't tell us how well the code is written and how good the identifiers are named and thus tell us little about attractiveness.

Let us examine a routine that may help in this regard.

```
int getAttractivenessScore(string identifierName)
{
    int attractiveScore = 0;
    CamelCaseTokeniser camelCaseTokeniser = new CamelCaseTokeniser;
    string [] identifiers = camelCaseTokeniser.tokenise(identifierName);
    string sentence = "";

    for (string identifier: identifiers)  //foreach (string identifier in identifiers)
    {
        if (englishDictionary.contains(identifier.toLower()))
            attractiveScore++;

        sentence += identifier + " ";
    }

    if (attractiveScore > 0)
        attractiveScore *= getSomeReadabilityIndex(sentence.trim());

    return attractiveScore;
}
```

This routine is passed the name of an identifier and uses a tokenizer to split the name up into words based on the assumption that camel casing is used. Each word is tested to see if it exists in an English dictionary and if it is, the total number of words found is updated. Also, a sentence is formed by inserting spaces between each word. If there are any recognised words, then a readability test is performed on the sentence. Lastly, the score is returned based upon the number of recognised words and their readability as a sentence.

Not so simple

Of course, this highlights some of the problems we are faced with. The identifier name could be 'students' or 'previousEmployees'. If we get a sentence of 3 words we are doing well! How can you check the readability of a 1 to 3-word sentence? To carry out a Gunning Fog Index[23] you need at least 100 words. Other readability tests also need a larger sample count than 3!

It may be possible to do what this routine does at a module level i.e. extract all the identifiers, split them into words, tagging them for their use and analyse these. A 'normal English' analysis is unlikely to work as sentence length is important (as is line length in code) and we would be just joining the words together arbitrarily. However some of the types of test that could be conducted are:

1. How well the words are related
2. How easy the words are to understand
3. How well the identifiers are named
4. How focussed the code is

How well the words are related

If the identifiers are from the same class or module, then it can be assumed (if the class or module is cohesive) they are related. Examples may be:

student, students, course, class, lesson, lecture, lecturer, tutor, tutorial, seminar, school, college, university, department, timetable, assignment, project, coursework, examination

To enable this analysis, perhaps some sort of relationship algorithm with a dictionary[24] of related words would be required.

How easy the words are to understand

The *Plain English Campaign*[25] uses a number of 'Marks' (e.g. Crystal Mark) to signify how clear the content of documents and websites are. They state:

> *"Since 1979, we have been campaigning against gobbledygook, jargon and misleading public information."*

It would be great if there was an 'Ada Lovelace Mark' for the clarity of source code. However until such a campaign is formed (or one taken up by the Plain English Campaign), it is up to the developers to ensure that code is plain to read. One way is to classify how simple or difficult these words are to understand. Why use *multifarious* instead of *complex* or use *pulchritude* instead of *beauty*?

There could be an issue in that the word or words used may be technical or from a domain that you are unfamiliar with or with terms that may not be found in an ordinary dictionary. You could possibly imagine a routine used in an application to analyse chemical compounds such as:

```
bool isDehydroepiandrosterone()
```

The solution would be to use a dictionary that has the words from the problem domain and understands some of the prefixes used in programming.

How well the identifiers are named

As the words have been tagged for their usage, they can be examined to see if the identifiers relate to their purpose e.g. nouns for variables – *totalPay* - and verb and noun combinations for methods – *calculateTotalPay*.

How focussed the code is

Counting the instant variables or fields of a class and the total times these are used in methods out of the possible maximum usage could be used as a measure for cohesiveness e.g. if a class has 4 fields and 2 methods, one method uses all 4, while the second method uses 2, then we have 6 usages out of a possible 8 (2 methods * 4 fields) or 75% cohesiveness.

This proposed measurement is based on how relevant the methods are to the class i.e. how focussed the code is. The assertion that cohesion can be measured by the use of instance variables in methods is based on *Clean Code*'s ideas of cohesiveness:

> *"Classes should have a small number of instance variables. Each of the methods of a class should manipulate one or more of those variables. In general, the more variables a method manipulates the more cohesive that method is to its class. A class in which each variable is used by each method is maximally cohesive."*[26]

It may be useful to disregard getters and setters as these are normally specific to a particular field.

Attractiveness Formula

The *Difficulty metric* of *Halstead's Complexity Measures*[27] can also be used as a general guide to how difficult code is to read and understand and no doubt, there other ways of measuring some aspect of attractiveness. By coming up with a formula that totals scores for the above metrics and perhaps using some of the other metrics such as cyclomatic complexity, a score can be achieved that can be compared to a scale that would indicate how attractive or ugly the code in the module or class is.

Summary

There are steps we can take to make our code understandable, attractive and compelling, although it may take a little more effort than writing quick code. These are some of the points that are pertinent to creating compealing code, but as with routines, more detail can be found in the classics books on this subject:

- Formatting
- Style
- Naming
- Self-documenting
- Principles for Routines listed earlier
- Use of language features
- Context
- Consistency
- Nothing unexpected
- Special cases handled carefully
- Use of design patterns
- No magic numbers
- No duplication (DRY)
- Loose coupling
- Strong cohesion
- SOLID principles

This is not an exhaustive list, but it can be used as a guide or checklist when refactoring and reviewing source code.

Current thinking and *Clean Code* principles suggest certain policies such as continual refactoring and making code self-documenting removing the need for commenting.

It is believed there is beauty in simplicity and another saying is that beauty is in the eye of the beholder; however with source code, it is easier to discern attractive code from ugly code. Attractive code is easy to recognise in hindsight but difficult to create initially.

David Gelernter said in "Machine Beauty - Elegance and the Heart of Technology":

> *"Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defence against complexity."*

[1] Professionalism FURST, IAPetus Newsletter, Paul Lynham, November 2015, http://www.iap.org.uk/main/wp-content/uploads/2015/12/Professionalism-FURST.pdf

[2] More FURST, IAPetus Newsletter, Paul Lynham, March 2016, http://www.iap.org.uk/main/wp-content/uploads/2016/03/More-FURST.pdf

[3] Quality Is Free, Philip B Crosby, 1979

[4] Code Complete: A Practical Handbook of Software Construction, Second Edition, Steve McConnell, 2004, p 465

[5] https://en.wikipedia.org/wiki/Readability

[6] http://wildlyinaccurate.com/what-makes-code-readable/

[7] https://en.wikipedia.org/wiki/List_of_portmanteaus

[8] Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, 2008

[9] https://en.wikipedia.org/wiki/Self-documenting_code

[10] Alan Mycroft, Professor of Computing at the Computer Laboratory, University of Cambridge says this actually lexes to c = a++ ++ +b, person communication

[11] http://www.c4learn.com/c-programming/c-increment-operator/

[12] https://www.python.org/dev/peps/pep-0020/

[13] Code Complete: A Practical Handbook of Software Construction, First Edition, Steve McConnell, 1993

[14] https://en.wikipedia.org/wiki/Command%E2%80%93query_separation

[15] http://www.mccabe.com/pdf/More%20Complex%20Equals%20Less%20Secure-McCabe.pdf

[16] Attractive Faces Are Only Average. *Psychological Science*, *1*(2), 115–121. Langlois, J. H., & Roggman, L. A., 1990 http://www.jstor.org/stable/40062595

[17] https://en.wikipedia.org/wiki/Exception_that_proves_the_rule

[18] Patterns of Enterprise Application Architecture, Martin Fowler, 2002

[19] Design Patterns: Elements of Reusable Object-Oriented Software, Gamma et al, 1994

[20] https://en.wikipedia.org/wiki/Static_program_analysis

[21] http://martinfowler.com/bliki/CodeSmell.html

[22] A Complexity Measure. *IEEE Transactions on Software Engineering,* SE-2(4), Thomas J McCabe, 1976

[23] https://en.wikipedia.org/wiki/Gunning_fog_index

[24] http://www.onelook.com/?w=*%3Astudent&ls=a

[25] http://www.plainenglish.co.uk/

[26] Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, 2008, p 140

[27] https://en.wikipedia.org/wiki/Halstead_complexity_measures