A Further Look At The Attractiveness Of Source Code

Paul Lynham MSc PGCE FIAP

Introduction

As discussed in the previous article *'A Brief Look At The Attractiveness Of Source Code'*[1], in using FURST[2,3] the product must be shown to be **F**it **F**or **P**urpose (FFP) and one of the attributes of FFP is **Attractive**. This states that for developers the design and code should be appealing.

Previously attractiveness was examined at a higher level. Now the reasons why attractiveness is important will be considered and then some attributes of attractiveness will be surveyed, a few at a high level and some at a more detailed level.

The Costs of Unattractive Code

Attractive code is easy to read and easy to understand. It can also make it easier for the reader to be confident that the code will do what it is supposed to do i.e. is compelling. It allows code to be changed with less effort and so enhances maintainability.

For both enhancing software features and fixing software bugs, code must be read and understood and the issues put in context. Therefore maintenance involves a lot of code reading. Robert Glass[4] found that software maintenance consumed from 40 – 80% of the total software cost, with a mean of 60%, while Boehm and Basili[5] reported a mean of 70%.

Other quality attributes that can benefit from attractiveness include reusability, reliability, complexity and portability[6, 7].

The converse is also true. Unattractive code can have a deleterious effect on readability, comprehension and maintainability and so reduces quality and increases costs.

The compiler, interpreter or runtime environment can transform source code to the target binary or bytecode even if the source code is unfathomable to all but the writer. However, code is read many more times than it is written and is usually read by human programmers.

Some developers have little patience in trying to understand unattractive code and may rewrite the code in their own way. However, unless this rewrite is carried out in a **compealing**[8] manner, the same thing may happen with the next programmer that comes across this same code. This can lead to continuous rewrites and a continuing accumulation of financial costs and time. Surely it makes sense to write this code in a compealing manner in the first place? It seems obvious, but code should be written so that everyone can understand the code written by everyone else.

Philip Crosby in his book *Quality is Free*, wrote:

> *Quality organisation is not very complicated, but establishing a good quality operation can be. If something is easy to understand and makes sense, and yet isn't always done, there has to be a reason for not doing it.*[9]

Crosby thinks that either management doesn’t trust anyone else to make the decisions about quality or that management doesn’t understand the value of a good quality operation.

The author’s own perspective is that even if management doesn't instil quality in the operation, the programmer must take responsibility and ensure that quality in producing and maintaining source code (and any other software artefacts) is taken seriously. This is one of the traits of a professional.

Why Do Programmers Write Unattractive Code?

Often software developers are under pressure to meet deadlines and the internal structure of the product may suffer by producing quick and dirty code that works. Many may not follow the ideas that have been covered in this article or the previous one on this topic. They write for the machine and not for humans and produce little documentation, even though this is even more essential with unattractive code. They may be tempted to just copy and paste code, rather than spend time making their code reusable. Taken together, such practices may lead to technical debt[10].

The assessment of the quality of the product may focus on the external attributes, such as how nice the product looks and how well it behaves. Testers and the QA department are the ones who decide if the product is good enough based upon this external perspective. So, who polices the internal quality?

Some organisations have procedures to maintain the internal quality, such as production of unit tests or other types of developer testing, pair programming, developer documentation and code reviews. They may also evaluate the source using software that analyses the code and produces reports that may contain metrics, warnings and recommendations. Such organisations are likely to have internal criteria such as UI and coding standards, which may be checked in product and code reviews.

What Documentation?

Although the focus of these articles has been on source code, it is important that all software artefacts are available and easy to comprehend. Most developers don’t like producing documentation. This can be crucial for recording, for example, the configuration that is required to set up the development environment or to configure the application under development, so it runs correctly. Without such documentation, it can be very difficult to reverse engineer what databases, tables, constraints, configuration files and application settings etc. are required. This can become very apparent when either a new developer starts with the organisation, or when the ‘font of all knowledge’ at the organisation leaves!

However, a fairly new idea is for the developer to video record what he or she is doing. Livecoding.tv[11] is a platform that focuses on providing a new way of handling the documentation issue. It can be used as a social coding platform for broadcasting live product development, as well as being utilised by the development team to record their development process and make it private for internal usage only.

High Level Concepts

Some of the following concepts were mentioned in passing in the previous article and will now be explained briefly.

Command-Query Separation Principle

A **query** is a routine that returns information only and doesn't alter the state of any data. In OO terms this means a method that returns data but doesn't change the object it is called from (or any other data). It is a well-known notion that a function should do one thing and one thing only and a function that returns a result should never change data (often referred to as a side-effect).

A routine that changes data should not return a result – again obeying the rule that it should do one thing only. This routine is called a **command**, although Martin Fowler prefers the term **modifier**, but also notes that they are also termed **mutators**[12]. A command may call a query but a query can never call a command.

Examples of queries that could be called on a collection of customers would be:

```
Customer GetCustomerByID(CustomerID)  // Returns a Customer given the customer's ID

CustomerCount GetCount()  // Returns the number of customers in the collection
```

Examples of commands that could be called on a collection of customers would be:

```
AddCustomer(NewCustomer)  // Adds a new customer to the collection

Clear()  // Removes all customers from the collection
```

The original idea about separating routines cleanly into queries and commands came from Bertrand Meyer. He stated:

From the Command-Query Separation principle follows a style of design that yields simple and readable software, and tremendously helps reliability, reusability and extendibility.[13]

There are some classic concepts in software development that break this principle, including the `Pop()` method of a stack that both returns the item at the top of the stack (query) and removes it from the stack at the same time (command). However, such concepts are well understood and developers know that `Pop()` has a side effect.

Keeping the Command-Query Separation principle in mind when writing or refactoring code, allows subsequent readers of the code to know which methods just return information and which ones alter the object, allowing them to be certain what the consequences are going to be, without worrying about side effects.

Single Level of Abstraction (SLA)

This principle relies on all statements of a method being at the same level of abstraction. If there are statements which belong to a lower level of abstraction, they should go to one or more private methods which comprise statements on this level. The result of this will be smaller methods[14].

A simple example is shown in listing 1 which mixes both low-level statements and higher level statements within the same method.

```
function GetGrossTotalPay(HoursWorked : double) : double;
var
  BasicHours, OvertimeHours : double;
begin
  //The following are decisions and mathematical operations
  // You can't get much lower than this (except perhaps bitwise operations?)
  if HoursWorked > STANDARD_HOURS then
  begin
    BasicHours := STANDARD_HOURS;
    OvertimeHours := HoursWorked - STANDARD_HOURS;
  end
  else
  begin
    BasicHours := HoursWorked;
    OvertimeHours := 0;
  end;

  // These call functions which hide the implementation
  Result := GetBasicPay(BasicHours) + GetOvertimePay(OvertimeHours);
end;
```

Listing 1. Mixed level statements

The statements that calculate the basic hours and overtime hours worked from the total hours worked could be extracted out into separate methods and the original method refactored as shown in listing 2.

```
function GetGrossTotalPay(HoursWorked : double) : double;
var
  BasicHours, OvertimeHours : double;
begin
  // All these statements are at the same (higher) level of abstraction
  BasicHours := GetBasicHoursWorked(HoursWorked);
  OvertimeHours := GetOvertimeHoursWorked(HoursWorked);
  Result := GetBasicPay(BasicHours) + GetOvertimePay(OvertimeHours);
end;
```

Listing 2. All high-level statements

Besides encouraging smaller, more focused methods, statements at the same level of abstraction make the code more compealing. Mixed levels of abstraction within a method can increase the viscosity of the code, making harder to read, understand and move on.

Parameter Lists

It is generally accepted that global data should be avoided where possible. In procedural programming, this often resulted in routines having longer parameter lists. In the first edition of *Code Complete* (aimed at procedural programming), it was advised to keep the number of parameters passed to a routine to be 7 or less[15]. Since then, OO has become the main paradigm in software development. Data can be stored as fields within the object so as there is much scope to reduce the number of parameters passed to methods to a minimum. In Clean Code it states:

> *The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification—and then shouldn't be used anyway.*[16]

Parameters are at a different level of abstraction to the method name and takes effort to understand and to put into context. The more parameters, the more effort is required to conceptualise what is going to happen when this method is called with actual arguments (values given to the formal parameters).

Reducing and minimising the number of parameters passed to a routine helps in making the code easier to understand and less brittle.

SOLID Principles

Although SOLID principles[17] are related to writing higher quality code that is easier to maintain, they can have an effect on how compealing code is. Each principle will be briefly explained with any relation to its effect on compealing code.

S - Single responsibility principle

A class should have only a single responsibility i.e. only one potential change in the software's specification should be able to affect the specification of the class. This can be compared to the concept of a routine doing one thing and one thing only, but is extended to the class level. As with routines, if a class is small and specific, having one concern, then it is easier to understand (and so is easier to change).

O - Open/closed principle

This refers to the need for software entities such as classes to be open for extension but closed for modification. An example is a base class should easily allow one or more super classes to inherit from it (encapsulating common behaviour), but that common behaviour should be closed for change, as changes will have widespread implications (superclasses may be affected).

L - Liskov substitution principle

This states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

A simple example is a base class of Shape, with super classes being Ellipse, Triangle and Rectangle. A Circle is a special type of Ellipse (the 2 foci are in the same location called the centre) and a Square is a special type of Rectangle (all sides have equal length) as shown in the hierarchy tree in figure 1.

In practice, however, treating the special cases as their direct ancestor may cause issues (changing the height of a Square also changes its width, which doesn't happen with a Rectangle). This proves that Rectangles and Squares cannot be necessarily substituted.

Ensuring that objects can be substituted for their subtypes provides consistency, reliability and confidence. These can lead to the code being easier to understand.

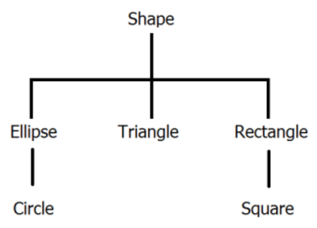

Figure 1. Shape objects hierarchy

I - Interface segregation principle

This is often defined as *"many client-specific interfaces are better than one general-purpose interface"*. It has parallels with both the *Single Level of Abstraction* and the focus of a routine – keeping it simple and specific.

An example may be instead of creating an interface for handling all types of user input (keyboard, mouse, gestures etc.) a specific interface should be made for each of these. In many languages that do not support multiple class inheritance, you can inherit from a single class but can also inherit or support multiple interfaces! This maintains flexibility, as for example if keyboard input is only required, you are not forced to implement gestures. If you want to cover all, then you can inherit from all 3 interfaces.

D - Dependency inversion principle

The idea is that you should "code against abstractions, not implementations". Abiding by this principle can lead to loosely coupled code and allows more flexibility in the choice of classes used to implement a specific interface.

Some Lower Level Ideas

These ideas look at concepts such as measuring aspects of methods and lines of code either using metrics that can easily be calculated by tools or by visually inspecting the code.

Method Length

The length of routines and methods were covered at a higher level in the previous article. Here the relationships between method length and other metrics will be briefly considered.

A relationship has been found between method size and cost[18]. It stands to reason that as a method increases in size it becomes more difficult to understand. Once it is larger than can be shown on a single screen, it is difficult to keep its full behaviour in your mind. Some of the properties that have been found to correlate with more expensive method code include:

- Size of the given method
- The number of dependencies leading from the method
- Number of conditionals in the method
- Quantity of Transitive dependencies involving the method
- Impact Set – the number of all other methods that the given method depends on, directly or transitively
- Absolute Potential coupling or the number of other methods visible to this method in the code base (not hidden by encapsulation)

Of these, the one which has found to have the greatest correlation to cost is the conditional count (cyclomatic complexity) so managing this will automatically manage the method size. Of the other properties, an acronym of SIPT[19] has been formed from **S**ize of method, **I**mpact set, **P**otential coupling and **T**ransitive dependencies. It has been proposed that **SIPT** is used as foundational properties by which programmers should evaluate method structure, with a view to minimising each of those properties.

Line Length

In general prose, one of the aspects that help in making it more readable is the sentence length[20]. In many programming languages, a statement can be spread over more than one line, but a line can be thought of as a basic aspect of source code. Like general prose, line length in programming has a correlation to readability and comprehension.

Newspapers have traditionally used columns, to layout text, which limits line length. This is done on purpose so that the text is easier to read. Imagine if the text was spread across the entire width of the newspaper. It would be difficult tracking your eyes from left to right over a large distance and once reaching the end of the line, snapping your focus back to the left again and trying to locate which line you were on. It has been asserted[21] that humans can only read between 10 to 12 words in a line before they lose track of which line they are on.

Depending on the width of the edit window within an IDE or editor, the maximum number of characters in a line that can be displayed may exceed the traditional 80 character width many times. Often the line length hasn't a maximum setting, so scroll bars can be used to create line widths exceeding the width of the editing window. When a method size exceeds the number of lines that can be shown on a single screen (without scrolling), readability and comprehension suffer, so the same effect occurs when a whole line cannot be seen without scrolling.

Splitting long lines may not help – it may cause other problems such as with layout. Use of good names for identifiers is a well-known method of increasing comprehension, but overly long identifier names may cause an increase in line length. Therefore it is important to manage line length effectively.

Buse and Weimer[22] conducted experiments on the readability of source code snippets. They attempted to link simple code metrics to predict the readability of code. Figure 2 shows a graph of the correlation (0 is low and 1 is high) of simple code metrics. It can be seen that both maximum line length and average line length are in the top 4 highly correlated metrics to readability. These are negative aspects so the larger the average line length, the lower the readability. Therefore minimising both maximum and average line length will aid readability.

However, as with any statistic, it is important to put these findings into context. Line length is a simple metric – it just measures the number of characters in a line of source code. It does not measure how complicated the line is e.g. does it contain a large number of identifiers or does it call a large number of routines?

Keeping line length small, but using high complexity within a line, will not help readability. To determine the complexity of a line another aspect needs to be considered. This is line density.

Line Density

Line density is a measure of how much information is encoded within a line. This is not a simple metric but composed of other metrics. The density can be measured by the number of methods called, the number of variables used, or the number of parameters passed. It could even measure how many periods or full stops are used.

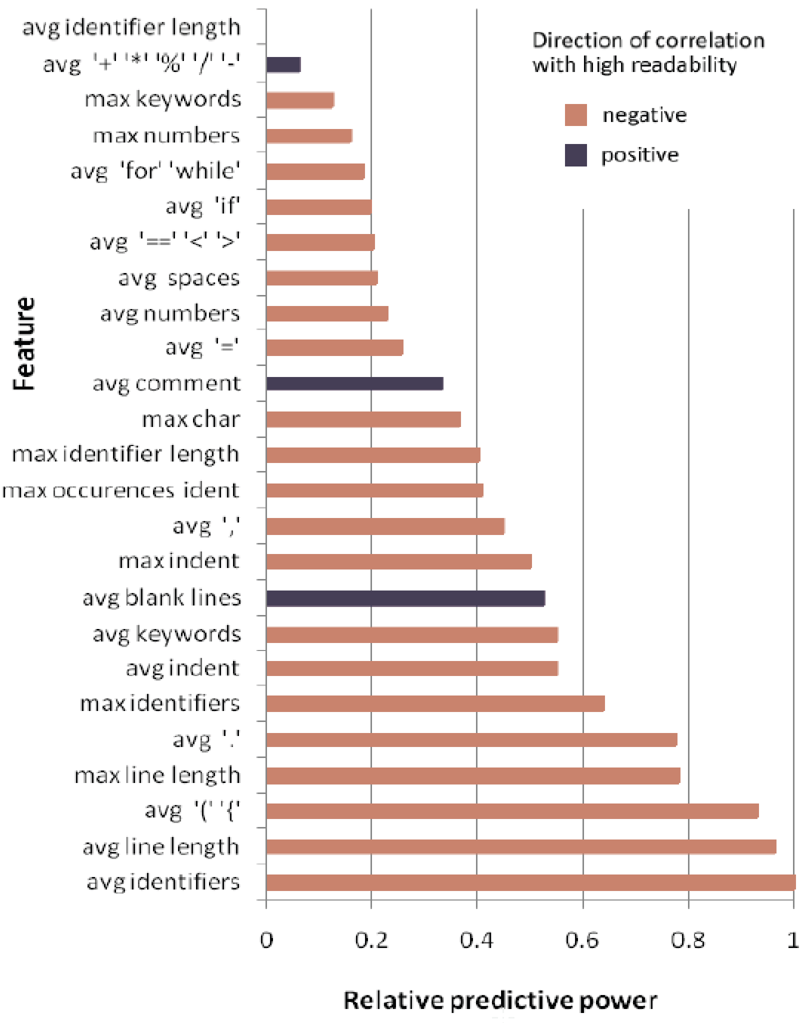

Figure 2. Simple code metrics used to predict readability found by Buse and Weimer.

In procedural programming, a lot of standalone functions may be used to manipulate a string. The following example formats a customer's name from a data set:

```
CustName := Trim(UpperCase(DataSet.FieldByName('CustomerName').AsString));
```

In modern OO languages, raw data types can be represented as classes, with their own methods. With the following example, the return of these methods is a string. By use of *Method Chaining*[23], the output string of a method calls its own method. Thus, the above functionality may look like this:

```
CustName := DataSet.FieldByName('CustomerName').AsString.ToUpper.Trim;
```

The latter is a little easier on the eye, as there are fewer identifiers within the parameters of methods i.e. only one method is explicitly passed a parameter, whereas, in the original code, every method is passed a parameter. This has reduced the density of the code.

Some programmers like to cram as many statements into a line as possible. If this is good practice, then why have lines at all (for languages that do not need whitespace to denote blocks or end of statement)? If so the code for listing 2 could look like listing 3.

```
function GetGrossTotalPay(HoursWorked:double):double;var BasicHours,OvertimeHours:double;begin
BasicHours:=GetBasicHoursWorked(HoursWorked);OvertimeHours:=GetOvertimeHoursWorked(HoursWorked);
Result:=GetBasicPay(BasicHours)+GetOvertimePay(OvertimeHours);end;
```

Listing 3. Statements compressed to a minimum number of lines

This would compile to exactly the same binary as listing 2 that was formatted and laid out better. Taking listing 2 again, it could be compressed to a lesser extent, as in listing 4:

```
function GetGrossTotalPay(HoursWorked : double) : double;
begin
  Result := GetBasicPay(GetBasicHoursWorked(HoursWorked)) + GetOvertimePay(GetOvertimeHoursWorked(
            HoursWorked));
end;
```

Listing 4. Statements compressed but with standard layout

Listing 2 is most probably the easiest to read and understand. Increasing line density clearly reduces readability. There is a trade-off between line volume and line density. However, line density should not be increased at the expense of readability. Reducing line density (without going to extremes) is likely to make the code more compealing.

Depth of Nesting

Nesting depth measures the maximum number of encapsulated scopes inside the body of the method. A testing condition with N conditions, such as

```
        if ( i > 3) and (i < 12) then
```

is considered as N scopes because it is possible to decompose such conditions into N atomic conditions. When a method has a nesting depth of 4 or more it can be difficult to understand and maintain[24].

The method in listing 5 has a nested depth of 3. The same is true for the method in listing 6, even though it has only 1 if statement (remember encapsulated scopes)?

```
procedure PrintStrings(AStringList : TList<String>);
var
  Index : integer;
  Str : String;
begin
  for Index := 1 to 10 do
    if Index > 4 then
      for Str in AStringList do
        Writeln(Str);
end;
```

Listing 5. Nested depth of 3

```
procedure PrintStrings(AName : String);
begin
  if (not AName.IsEmpty) and (AName.Length > 2) and (AName.IndexOf('Jon') >= 1) then
    Writeln(AName);
end;
```

Listing 6. Nested depth of 3

An example of a function that uses data to check if a rule is complied with is shown in listing 7. This function is passed a date which is used to access the Year and Month of birth and together with the person's married and employed status decides if the rule applies.

```
function IsRuleRelevant(DateOfBirth : Date; Married, Employed : boolean) : boolean;
begin
  Result := false;
  if (DateOfBirth.Year < 1954) or ((DateOfBirth.Month < 4) and (DateOfBirth.Year = 1954)) then
    Result := true
  else
  if Married and (DateOfBirth.Year = 1954) and Employed then
    Result := true;
end;
```

Listing 7. Lines with high nesting depth

In listing 8, the logic has been extracted into well-named functions, making comprehension easier than lots of logical comparisons. This results in a lower nesting density for this routine (2 functions calls, rather than 6 evaluations), but instead 2 extra routines have been created resulting in an increase in line volume.

```
function IsRuleRelevant(DateOfBirth : Date; Married, Employed : boolean) : boolean;
begin
  Result := false;

  if BornBeforeApril1954(DateOfBirth) then
    Result := true
  else
  if IsEligibleFor1954Birth(DateOfBirth, Married, Employed) then
    Result := true;
end;
```

Listing 8. Refactored with lower nesting depth

In figure 2, it can be seen that *average identifies per line* is highly negatively correlated with code readability – the more identifiers per line, the less readable the code becomes. Listing 8 reduces the number of occurrences of identifiers compared to listing 7. Similarly, figure 2 shows that the third highest negative correlation to readability is the *number of parentheses used per line*. Again this has been reduced from 8 to 2 in the first 'if'.

Summary

Professional programmers realise that attractive code is not just a question of 'cosmetics', but are concerned with producing and refactoring code to make it compealing. This means writing code not just for themselves at that point in time, but for other programmers, or indeed for themselves in a few weeks' time[25].

Writing quick and dirty code might make sense to the writer at the time, as the context of the problem is 'current' and so badly named variables, poorly structured code and code shortcuts make sense. A few weeks later, the context is forgotten and then the code may not even make sense to the author, let alone anyone else.

Attractiveness can be subjective, with such factors as the experience of the programmer reading the code, the coding standards used in the source code and how familiar the reader is with the language. Keeping a common base of attractiveness is, therefore, important. Using concepts covered in these articles should transcend personal preferences so code can be made more compealing for everyone.

[1] A Brief Look At The Attractiveness Of Source Code, IAPetus Newsletter, July 2016, Paul Lynham http://www.iap.org.uk/main/wp-content/uploads/2016/07/OnThe-AttractivenessOfSourceCode.pdf

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

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

[4] Facts and Fallacies of Software Engineering, Addison-Wesley. 2002, ISBN: 0-321-11742-5, Glass, Robert L.

[5] Software Defect Reduction Top 10 List. IEEE: Computer Innovative Technology for Computer Professions. January 2001, Vol: 34, No:1, pp: 135-137, Boehm, Barry; Basili, Victor.

[6] Impact of Programming Features on Code Readability, International Journal of Software Engineering and Its Applications, Vol.7, No.6 (2013), pp.441-458, Yahya et al. http://www.sersc.org/journals/IJSEIA/vol7_no6_2013/38.pdf

[7] Impact and Comparison of Programming Constructs on JAVA and C# Source Code Readability, International Journal of Software Engineering and its Applications, November 2015, Batool et al. https://www.researchgate.net/publication/284219994

[8] Compealing is a portmanteau formed from comprehensible, compelling and appealing, as it is easier to treat the three concepts together. See 1.

[9] Quality Is Free, 1979, p 68, Philip B Crosby

[10] Laws, Debts and Smells, IAPetus Newsletter, May 2015, Paul Lynham. http://www.silverbirch.ltd.uk/share/Technical_Debt.pdf

[11] https://www.livecoding.tv/

[12] http://martinfowler.com/bliki/CommandQuerySeparation.html

[13] Object-Oriented Software Construction, 2nd Ed., 1997, p 752, Bertrand Meyer

[14] http://principles-wiki.net/principles:single_level_of_abstraction

[15] Code Complete: A Practical Handbook of Software Construction, First Edition, Steve McConnell, 1993, p 108

[16] Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, 2008, p 40

[17] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[18] http://edmundkirwan.com/general/pearson.html

[19] https://dzone.com/articles/if-you-like-good-java-sip-tea?edition=197642&utm_source=Daily

[20] https://strainindex.wordpress.com/2008/07/28/the-average-sentence-length/

[21] http://paul-m-jones.com/archives/276

[22] A Metric for Software Readability, ISSTA'08, 2008, Raymond P.L. Buse and Westley R. Weimer

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

[24] http://www.ndepend.com/docs/code-metrics

[25] https://xkcd.com/1421/