A Software Complexity Metric Based On Cognitive Principles

Thomas Mullen
tom@tom-mullen.com

Abstract

The metric is a measure of the cost of understanding software and can be evaluated at all levels of the code (from expression/statement through to library/application). A working prototype has been validated against manufactured examples for refactoring, design patterns and cohesion levels. The suggestion is that the metric could be used as part of an approach to automate a large part of refactoring and software design.

Categories and Subject Descriptors D.2.8 [**Metrics**]: Complexity measures..

General Terms Design, Human Factors, Theory.

1. Introduction

This paper introduces a metric of software design complexity that represents the cost of understanding unfamiliar code. The constituent parts of the metric are as follows (a greater value corresponds to a greater cost):

1) *Element cost*: Each code element will become an element in the memory network of the code reader and will incur the time costs of committing to long term memory.

2) *Cognitive overload penalty*: A highly non-linear function that represents the burden of presenting too many items at once to the reader.

3) *Coupling and cohesion costs*: To promote structures similar to the chunking exhibited in long term memory.

4) *Semantic penalty*: Not covered directly in the metric this represents the transmitting of semantic knowledge about the purpose of the code (primarily by accurate names on the code elements). Section 9 discusses this in more detail.

The metric is validated against manufactured examples for refactoring [6], design patterns [7] and cohesion levels [14]. I do not attempt any formal, mathematical validation as the complexity perceived by the mind may not necessarily prescribe to mathematical theory.

The metric is applied to each element (expression, statements, method, class etc.) and aggregated up to get a total measure for each method, class, package, application etc.

A comparison is made with the force directed graph algorithm (used to display graphs clearly). The next section describes the Crofton School problem, a common analogy amongst the structure of memory, software complexity and force directed graphs.

2. Crofton School Problem

Crofton primary school (taking children from aged 4 to 11) is one of the largest in the UK with six classes in each year (each class has approx. 25-30 children). In the first 4 years the children remain together in the same class. For the fifth year the classes are mixed up. Children are allowed to state who their closest friends are and the school will attempt to put them together in the same class. Although there is no limit to the number of friends they can specify, this is typically just one or two. In Figure 1, Michael has chosen Avinash as his friend and Charles is the only one who has stated two children as his friends (Omene and Veronica)

This is the background to the problem that is stated more formally in the next subsection.

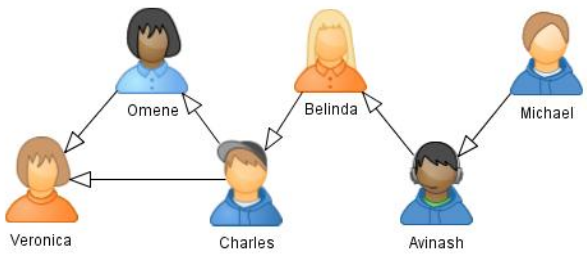


Figure 1. Crofton School Stated Friends Example.

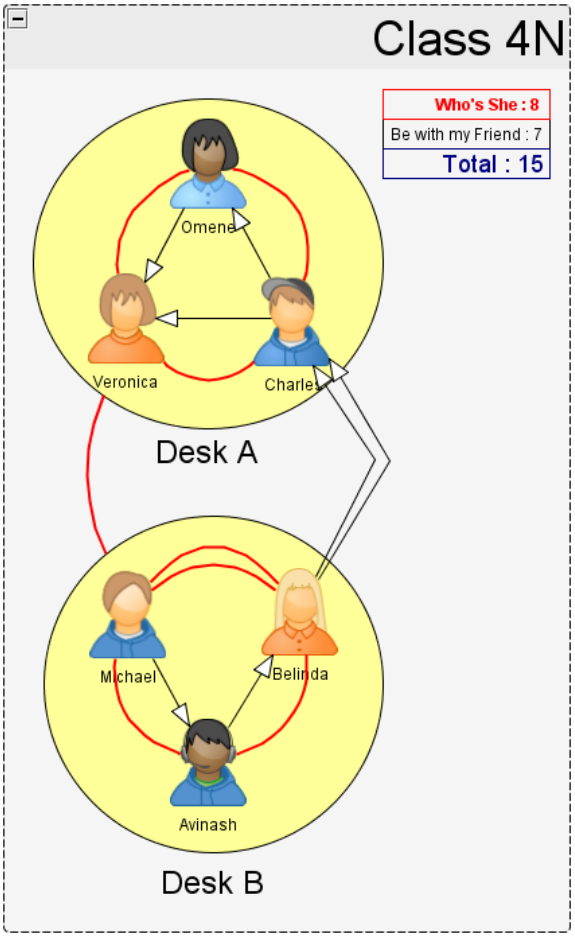


Figure 2. Optimum Seating Plan.

2.1 Statement of Problem

Given a set of children and their stated friendships what is the optimum seating plan (at desks within classes) that minimises unhappiness. Unhappiness occurs from two sources:

1) "I want to sit with my friend". If a child has stated a friendship but is not seated close to that friend then they will be unhappy. The further away they are from their friend the greater the unhappiness. The value used for unhappiness is 3 if the child and their friend are in different classes, 2 if they are in the same class but different desks and 1 if they are seated at the same desk.

2) "Who's she?" Children are unhappy if they're seated at the same desk as a child they have no common friendship with. The value will be the "degree of separation" between the two children. In the above example if Omene and Veronica are sat at the same desk then this will be 1, but if Belinda and Veronica are sat at the same desk this will be 2 (the shortest path length on the graph of friendships). This is also extended to the association between desks in the same class (again, the unhappiness value will be the shortest path length on the graph of friendships).

Figure 2 shows the optimal seating plan for the example scenario from Figure 1 (there are only 203 unique solutions if we restrict the children to one class). The red lines (without arrows) show where the "Who's She?" unhappiness values exist and the black arrowed lines are the "Be with my friend" unhappiness values. In each case the number of lines is indicative of the value.

In the optimum solution. all children are sat at the same desk as their friends ("Be with my friend" value is 1) with the exception of Belinda and Charles ("Be with my friend" value is 2 as they are on different desks in the same class). Also, the children within each desk are direct friends, with the exception of Belinda and Michael who's friendship is via Avinash (which is why the "Who's She?" value between Michael and Belinda is 2).

Moving Belinda to the same table as Charles (Figure 3) would reduce the "Be with my friend" value between them from 2 to 1 but this would be offset by an increase of the "Be with My Friend" value between Belinda and Avinash. The additional "Who's She?" unhappiness between Belinda and the rest of the Desk A is more than the removal of the "Who's She?" unhappiness between Belinda and the rest of

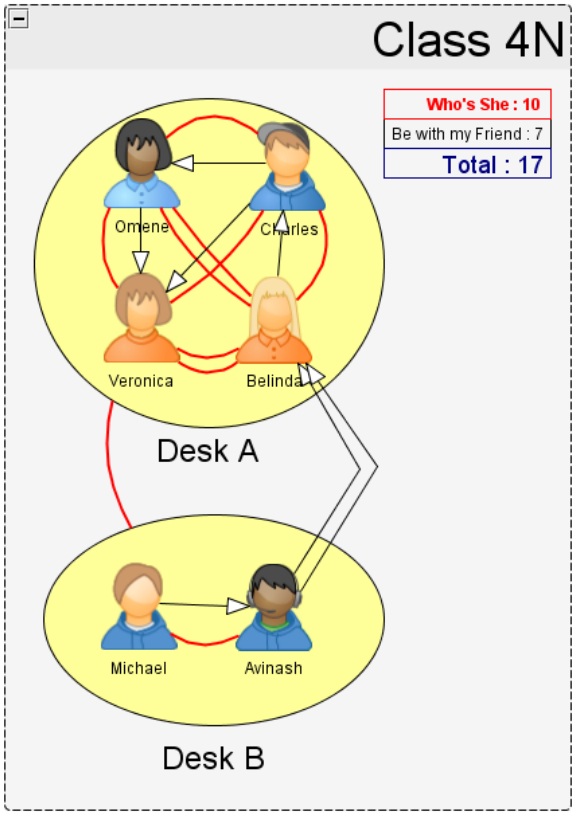


Figure 3. Non-Optimal Seating Plan.

Desk B.

This puzzle was developed as an analogy to illustrate the similarity between coupling /cohesion, the structure of long term memory and the force directed graph algorithm (Figure 5) and the mapping is discussed more in the sections below. Software metrics are used to produce code that is easy to understand and the force directed graph algorithm is used to aid clarity in visualising graphs. It is not mere coincidence that a common analogy exists between the cognitive model and these two.

2.2 Coupling and Cohesion

In software, the children of the Crofton school problem are code statements and the stated friendships are the dependencies between statements. The "Who's She?" unhappiness penalty relates [reciprocally] to cohesion (a large value of "Who's She?" corresponds to a low Cohesion and a small value to high cohesion). The "Be with My Friend" unhappiness penalty relates to coupling, the greater the value the more the coupling. The desks are the code structures we use to group statements (e.g. methods, classes, package etc.)

By minimising the combined value of "Who's She?" and "Be with My Friend" we are finding code structures that reduce coupling and improve cohesion.

2.3 Long term Memory structures

Memory Elements are stored as chunks in the long term memory network (a chunk is "a collection of memory elements having strong associations with one another but weak associations with elements within other chunks" [10]). The children in the Crofton School puzzle are representative of memory elements, and the friendships are the association amongst memory elements. The desks/classes are the chunks of memory elements. Minimising the "Who's She?" penalty will improve the associations amongst elements within a chunk and minimising "Be with my friend" will promote only weak associations of elements in other chunks.

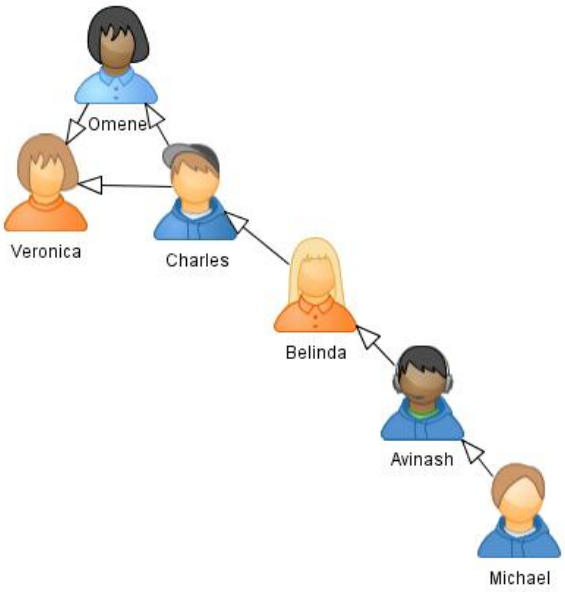


Figure 4. Crofton School mapping displayed using Force Directed Graph algorithm.

2.4 Force Directed Graphs

Many drawing applications use a force directed algorithm to display connected items. The algorithm works by modelling a spring between connected elements and a weaker repelling force between all items. Both forces vary according to the distance between the elements but in different ways, the spring force gets larger for larger distances but the repelling force gets smaller for larger distances. The resultant force upon each element is calculated, which implies an acceleration. The movement of the elements are allowed to follow the normal Newtonian model to find a structure that will result in a resting state (a damping on the velocities is used to prevent oscillations). The solution for the example scenario is shown in Figure 4.

The children of the Crofton School puzzle are the vertices of the graph and the friendships are the edges. The

Crofton School Problem	**Software Metric**	**Cognition**	**Force Directed Graph**
Friends	Dependencies between statements	Associations between memory elements	Edges on Graph
Desk/Class Structure	Code structure (statement groups, methods, classes, packages etc.)	Chunks in long term memory structure	(x,y) position of graph vertices
"Who's she?"	Cohesion (penalising code elements that have no association with other statements in a group produces cohesive statement groups)	"Memory elements within a chunk should be strongly associated with each other….	Spring between connected vertices
"Be with my friend"	Coupling (penalising the separation of associated statements will reduce coupling)	…and weakly associated with memory elements in other chunks"	Repelling force between all vertices

Figure 5. Mapping between Crofton school problem and Software Metric, Cognition & Force Directed Graph.

"Who's She?" penalty is represented by the repelling force between all elements and the "Be With My Friend" is the spring force between connected vertices. The crucial difference for the force directed graph algorithm is that rather than have fixed desks/methods/chunks the (x,y) position of the element is the variable we optimise upon. For the other three cases (Crofton school problem, coupling/cohesion and long term memory) there is no information as to how to improve any individual solution. For example when we moved Belinda to the left hand desk, the only way of discovering the variation was to calculate the unhappiness value before and after. However, to minimise the speed in the system, the force directed approach utilizes the first order derivatives (acceleration). This gives a direction to move in that will (at least most times) improve our solution (this is also identified by Feathers [4]).

3. Metric Derivation

The metric has been developed to illustrate the conjecture that "software code is a textual representation of the memory structure in the mind" [12]. The metric uses a mixture of the cognitive psychologist's models and software design principles to produce a measure of complexity and is a prototype implementation of the four minus analogies rule [12].

3.1 Code Elements

Code elements are expressions, statements, methods, classes, packages, libraries, modules, applications, suites of applications and so on. Code elements are presented to the reader as a structure graph, where statements are child elements of methods, methods are child elements of classes etc. The optimum design for the structure graph would be one capable of being used directly as the long term memory network within the reader's mind. Each code element will incur a time cost to persist to long term memory.

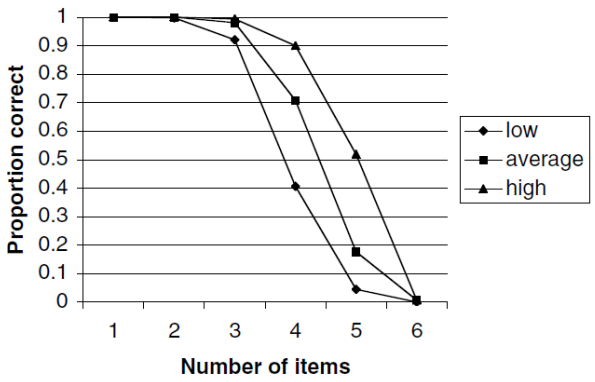


Figure 6: Illustration of the Short term memory capacity limit [2].

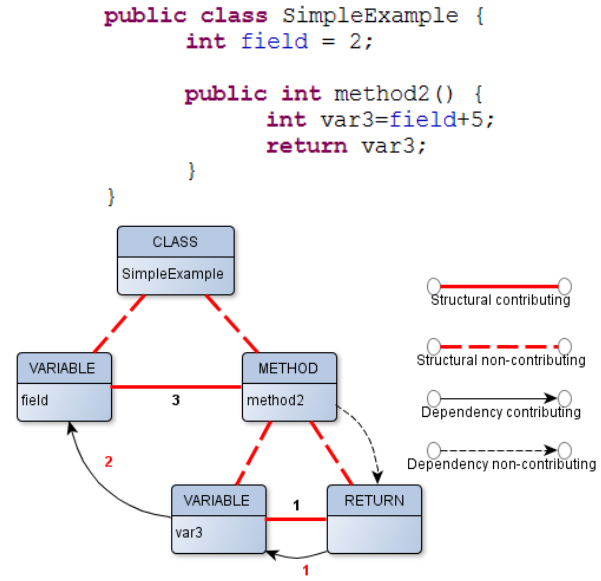


Figure 7. Illustrative Example of Structural and Dependency Graphs.

3.2 Cognitive Overload Penalty

The capacity limits of short term memory restrict the number of elements that can be presented at any time [2]. This has an effect on the number of child elements that each node has in long term memory [3,11]. To model this in the metric each code element will carry a penalty based on the number of children it has on the structure graph (as the structure graph is representative of long term memory). There are a number of strategies for the new reader to identify candidate associations. Looking purely at pairs of elements would perform like n^2, unordered combinations of all subsets (performing like 2^n) or ordered combinations of all subsets (performing like n!). I have used n! for the prototype as this is more of the near barrier function identified by Taatgen (Figure 6, the reciprocal of the proportion correct is a measure of the complexity)

The exception to this function is when varieties of analogies are presented to the reader [12].

3.3 Coupling And Cohesion

In addition to the structural graph there is a dependency graph. This is an extended call graph which contains dependencies between expressions and statements as well as those amongst methods and classes.

The child elements in the structure graph are representative of memory chunks. A chunk is defined [10] as a collection of memory elements that are closely related to each other and loosely related to elements in other chunks. Cohesion metrics measure how closely related code elements within a chunk are and coupling metrics measure how much code elements are dependent on elements in other chunks.

Programmers use spatial imagery for the abstract mental representation of code [5]. I propose to measure the cohesion amongst code elements within a chunk as a distance measure on the dependency graph. Additionally to measure coupling between dependent elements as a distance measure on the structure graph.

All connections (on both graphs) have a path length and can either be contributing or non-contributing. Coupling/cohesion costs are only calculated for contributing links. For the structural graph the parent/child links are non-contributing but contributing links exist amongst all siblings. The determination of the contributing links on the dependency graph is less precise and driven by whether coupling needs to be calculated. For example, a method must have a dependency upon the return statements within the method. The requirement that the code reader understand the software language means that this connection is already understood. The metric described in this paper attempts to measure the cost of understanding new code and so it is not necessary to measure the cost of a dependency that is already understood.

3.4 Illustration

An example of the structural and dependency graph is shown in Figure 7. The Structural graph is coloured bold red and the dependency graph in black (with arrows). Contributing links are solid lines and non-contributing links are dashed. For the purposes of simplicity the path lengths are 1 for all links on the structural and dependency graphs. The numbers on the structural contribution links are the minimum path lengths on the dependency graph (this is used to calculate coupling). The numbers on the dependency contribution links are the minimum path lengths on the structural graph (used to calculate cohesion).

4. Metric Formulae

The metric uses two graphs based on code elements. Firstly a structural graph **S** which is based on how the code is presented to the reader (e.g. a statement is a group of expressions, a method is a group of statements and fields etc.). Secondly, a dependency graph **D** which is an extended call graph that contains the dependencies between statements as well as methods.

For each code element i, the individual complexity metric value is defined in (1). Note: the full version would include semantic contributions.

$$CogCx_i = element_i + cognitive_i + cohesion_i + coupling_i \quad (1)$$

These values can be aggregated up through the structural graph (2) to get an overall measure for the class, package, application, etc.

$$x_i^{(agg)} = x_i + \sum_{j \in c_i} x_j^{(agg)} \quad (2)$$

Where x can be *CogCx*, *element*, *cognitive*, *cohesion* or *coupling* and c_i are the child elements of i on the structural graph **S**.

The $element_i$ cost is a simple static cost dependent upon the type of code element (e.g. 1 for a statement, 2 for a method and 5 for a class).

Defining a_i^{DEP} and a_i^{STRUCT} as the set of elements attached to element i by a contributing link on the dependency and structural graphs respectively. Also, p_{ij}^{DEP} and p_{ij}^{STRUCT} as the shortest path length between elements i and j on the dependency and structural graphs respectively. Then cohesion and coupling are specified as follows:

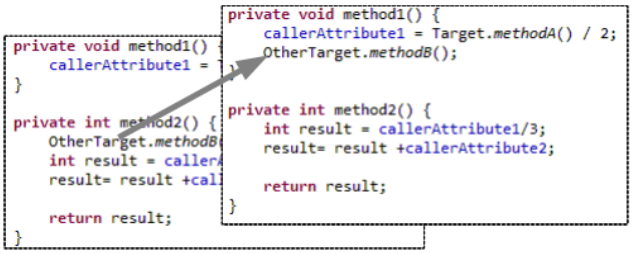
$$cohesion_i = \sum_{j \in a_i^{STRUCT}} \frac{1}{2} p_{ij}^{DEP} \log\left(p_{ij}^{DEP}\right) \quad (4)$$

$$coupling_i = \sum_{j \in a_i^{DEP}} \frac{1}{2} p_{ij}^{STRUCT} \log\left(p_{ij}^{STRUCT}\right) \quad (5)$$

The distance function applied here [$x\ log(x)$] was chosen to promote even spacing with the elements. For example, if two elements have an equal dependency on a third, then (ignoring other dependencies for the moment) the positioning of the third element on the structure graph should be equidistant from the both of its two dependencies. In some ways the choice of distance function here is similar to the choice of the functions used in the force-directed layout of graphs, where in some cases x and x^2 functions have been used. Empirical research would be necessary to establish which function produces the best metric value.

$$cognitive_i = n_i! \quad (6)$$

Where n_i is the number of children element i has on the structural graph allowing for any number of varieties of analogies. For example a package that contains classes that all extend the same abstract class (and are therefore varieties of that analogy) has n_i =1. Similarly the switch statement has n_i =1 as its children (the case statements) are varieties of an analogy [12].



	Element	Cognitive	Cohesion	Coupling	Total
Before	8	26	1381	174	1589
After	9	10	5	176	200

Figure 8 : Cohesion Level Example.

5. Results From Prototype

The following sections give the values of the metric for manufactured examples. A proof of concept prototype (written as an eclipse plugin) was used to calculate the values. All code is available from the author's website.

```
public int before() {
        int f =1;
        int g = f+4;
        int e=f*g;
        int t= 2;
        int u=t+5;
        int s=t*u;
        return e+s;
}
```

Is compared to :

```
public int after() {
        return afterPart1() + afterPart2();
}

public int afterPart1() {
        int g =1;
        int f = g+4;
        return g*f;
}

public int afterPart2() {
        int u= 2;
        int t=u+5;
        return u*t;
}
```

	Element	Cognitive	Cohesion	Coupling	Total
Before	9	5041	9.56	11.15	5070.71
After	12	15	2.94	33.45	63.39

Figure 9 :Refactoring Extract Method.

5.1 Cohesion Level

Sibling elements on the structural graph that have direct associations will have a low cohesion value within the metric. In this way sequential cohesion [14] is rewarded.

The example here (Figure 8) shows how the metric is consistent with the communicational level of cohesion. I am assuming that `method1` is always called before `method2`, so moving the statement that calls `methodB`, as shown, doesn't modify the behaviour of the application. In the initial position, the statement that calls `methodB` is a sibling of the other statements in `method2` and so the cohesion metric is calculated between them. This results in a high value as `methodB` and the variables in the other statements of `method2` are far apart on the dependency graph. When we move the `methodB` call, the cohesion metric is now calculated with the statement in `method1`. In our example, `Target.methodA()` and `OtherTarget.methodB()` are within the same package and are closer on the dependency graph due to the dependencies amongst that package. This results in an overall reduction in the cohesion metric.

5.1 Refactoring

5.1.1 Extract Method

The extract method refactoring [6] example is in Figure 9.

The biggest gain in splitting up the before method is with the cognitive penalty. Also of note is that cohesion is improved at the expense of worse coupling and element costs.

5.1.2 Inline Method

For the inline method example (Figure 10), moving all the

```
public int before() {
        int b =1;
        int a = b*beforePart1(b) ;
        return a ;
}

private int beforePart1(int b) {
        int c = b+4;
        return c;
}
```

Is Compared to

```
public int after() {
        int b =1;
        int c = b+4;
        int a=b*c;
        return a;
}
```

	Element	Cognitive	Cohesion	Coupling	Total
Before	10	11	1.92	37.60	60.52
After	6	25	2.22	8.74	41.96

Figure 10. Refactoring Inline Method.

behaviour to one method doesn't break the short term memory capacity limit (the cognitive penalty function is a near barrier function). Consequently the improvement in coupling is more than enough to offset degradation of the cohesion cost and cognitive penalty.

5.2 Design Patterns

The design pattern tests centre around a fictitious requirement that a calling abstraction needs to call behaviour(s) on a target abstraction. The three versions differ based on the complexity of the caller and target behaviours and the detailed relationships between caller and target. In the diagrams The pink boxes in Figures 11-13 represent the abstractions of the problem (with <<analogy>> stereotypes to represent abstractions and <<variety>> to represent an implementation of an abstraction).

1) Simple (Figure 11): A simple, single target behaviour for a number of varieties and there is only one caller of this behaviour.

2) Complex 1 to 1 (Figure 12): Complex set of target behaviours for a number of varieties. Each of the varieties has a one-to-one dependency with the varieties of the calling abstraction.

3) Complex independent (Figure 13): Complex set of behaviours for a number of varieties. Behaviours are called by various unrelated callers.

These versions have been chosen so that the approaches below will be the preferred designs of `behaviourA` for each version in turn

a) Switch: Simple method that utilises switch

b) Hierarchy: target behaviours are incorporated into the varieties of the calling code, similar to the Template pattern.

c) Strategy: Strategy pattern, behaviours are grouped independent of calling code.

Table I shows that the metric values for the preferred designs are the smallest for the three versions of requirements.

Table I : Metric Values for Design Pattern Examples

	Proposed Design		
	Switch	Hierarchy	Strategy
Simple	107	171	421
Complex 1 to 1	3386	1374	1513
Complex Independent	10724	3640	1321

As the code author deliberates over candidate designs they will try and determine which choice will provide the minimum complexity (whilst still being consistent with requirements, both implicit and explicit). The essential complexity can be considered the choice with the global minimum complexity value. Note that this is not a measure of just algorithmic complexity; it rewards code structures that promote ease of understanding. Code simplicity is arguably as valid a requirement as the algorithmic behaviour needed, since the majority of software development costs are in support and maintenance. Further, complexity is as much a function of human cognition as it is any objective measure of the algorithm.

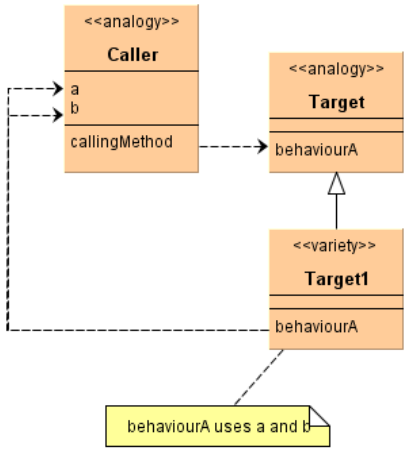


Figure 11. Simple Design Example.

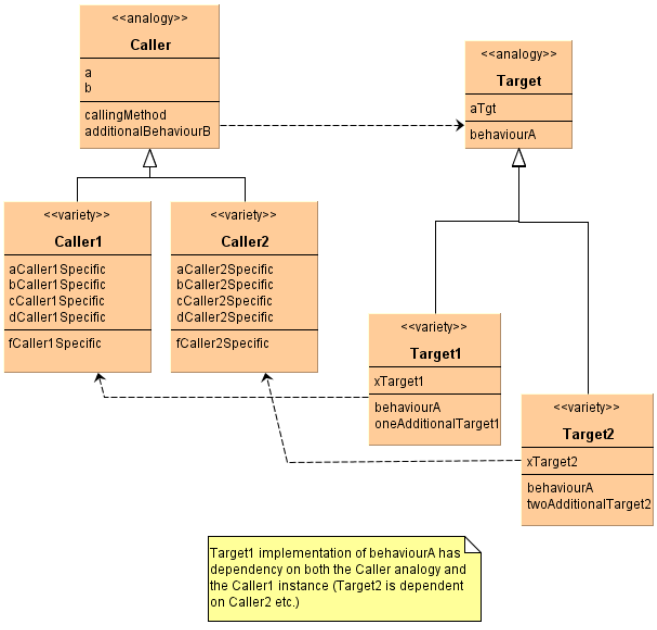


Figure 12. Complex 1to1 Design Example

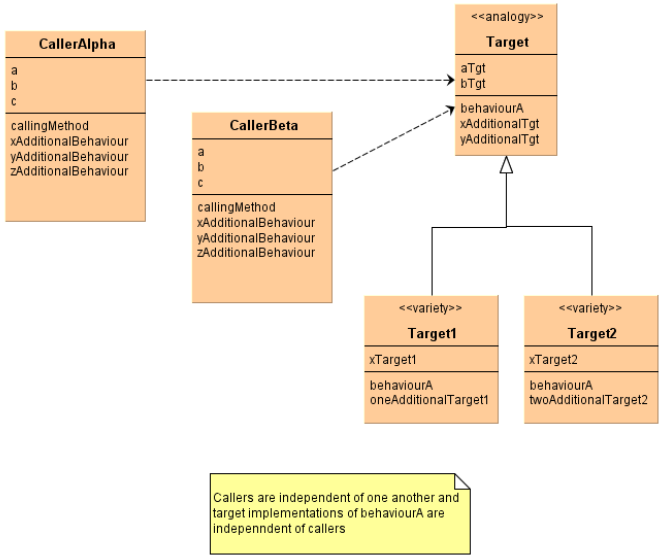


Figure 13. Complex Idependent Design Example.

6. Extensibility

To investigate extensibility we can measure each design choice for different numbers of varieties of the abstractions. In the example below I will compare the increase in complexity between implementing one and five varieties of the target and caller abstractions. The difference between the two measures must be normalised both by the number of additional varieties and the complexity of implementing just one variety:

$$Extensibility = \frac{CogCx^{(n)} - CogCx^{(1)}}{(n-1)CogCx_e^{(1)}} \quad (6)$$

Where $CogCx_e^{(1)}$ is the value for the essential complexity solution for 1 variety, and $CogCx^{(n)}$ is the value of the candidate solution with n varieties implemented.

Table II shows the calculated extensibility factor for the design pattern examples (for each case I use an approximate to $CogCx_e^{(1)}$ by taking the minimum value of $CogCx^{(1)}$ amongst the three candidate solutions). In this example, the design choice with the lowest extensibility factor coincides with the lowest metric value. This may not always be true. A software designer may accept additional complexity when coding the first variety for the promise of improved overall complexity when all varieties are implemented at a later date. Accurate appreciation of the extensibility factor is a design skill.

Table II. : Extensibilty Measures For Design Pattern Examples

	Switch	Hierarchy	Strategy
Simple	0.23	0.39	0.77
Complex 1 to 1	4.95	0.78	0.85
Complex Independent	5.39	3.23	1.05

7. Comparison With Other Metrics

The simplest measure of complexity is counting the lines of code, and is directly related to the element cost. This can be a successful measure for applications that are already well designed (could be used as a crude comparative estimate of maintenance and support costs amongst applications).

The scope restrictions applied to method local variables means that coupling can't exist between statements in different methods. Most coupling and cohesion metrics operate solely at the level of classes and/or methods as this is where a good deal of the coupling/cohesion is present. But as we have seen in the method refactoring example, coupling and cohesion at the statement level can influence the design.

McCabe's cyclomatic complexity metric measures the number of possible paths in a method. This metric tries to warn against a large number of elements presented in a chunk (to keep within short term memory capacity limits). The cognitive penalty element of CogCx represents the complexity that McCabe measures.

The examples in Section 5 show that we rarely improve all the parts of the metric (element cost, coupling, cohesion, cognitive). Rather, some parts improve and some worsen but the overall measure of complexity can be reduced. Software design can be considered an optimisation problem where the objective is to minimise complexity (presented to the code reader) whilst satisfying all the requirements. Requirements can be explicit (e.g. VAT must be added to all invoices and it must be possible to change the rate) or implicit (e.g. unwritten performance requirements such as the application must respond to queries in a reasonable time and run on a pc with the standard cpu and memory limitations). The candidate solutions are like islands to one another and so our optimisation is most like a constrained Integer programming problem.

Perhaps the most useful values in optimisation problems are the first and second order derivatives as these suggest which direction to move for the greatest benefit. However, these are unavailable to us in our software design optimisation problem. This may explain why metrics aren't used as much as we would wish they were. We can identify individual values that seem to be outside of usual variance but we are not guaranteed that tactics to reduce those individual outliers will lead to a reduction in the overall complexity.

8. Metric Parameters

Although not explicitly stated in the metric formulae, there is the capability to apply constants to all or part of the equations. These parameters could be empirically researched to

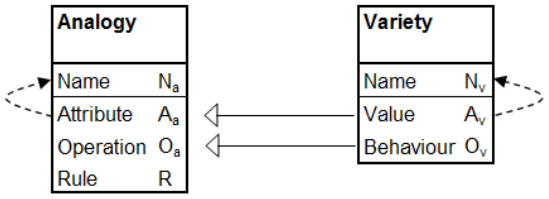


Figure 14.Analogy Structure.

allow the metric to match the minds perception of complexity. This would be influenced not only by the language but also whether tools are available to the developer. The ability to traverse straight to a method declaration via a single key press (e.g. F3 in eclipse) lessens the impact of our short term memory time limit when traversing links. Consequently we may allow the distance function on the structural graph to be reduced.

9. Semantic

Computer scientists refer to them as abstractions, cognitive psychologists as analogies and cognitive linguists as metaphors. They are the essence of intelligence, the building blocks of our cognitive abilities. In mapping the similarities between two or more objects and building up a structure, we lay down memories that allow us to use existing knowledge in new situations and can point to new understanding and breakthroughs. Mullen[12] showed the ubiquity of analogical code structures in code (not just as a class/interface construct). The description of an analogy is shown in Figure 14. The discussion in this section will also show that analogies are the common link between natural

Lexical Category		
Noun	Proper	Variety
	Common	Analogy
Verb		Behaviour
Adjective/ Adverb	Descriptive	Variety
	Modifier	Additional behaviours and/or attributes to existing analogy

Figure 15. Mapping of word types to analogical structure elements.

language and software (Figure 15 gives a précis of the mapping) .

As an example consider the following well known nursery rhyme as a statement of a fictional set of requirements functional requirements.

Mary had a little lamb.
Its fleece was white as snow
and everywhere that Mary went
the lamb was sure to go.

The complete analogical analysis of our requirements are discussed in the following sub-sections which will lead to the diagram in in Figure 16.

9.1 Nouns

Common nouns are analogies, proper nouns are varieties

As a proper noun “Mary” is a variety of some analogy as yet unstated. We may use our experience to establish that Mary is a variety of the “girl” analogy or a variety of the “shepherdess” analogy. Which analogy is appropriate will be dependent upon the purpose that we are modelling Mary (in

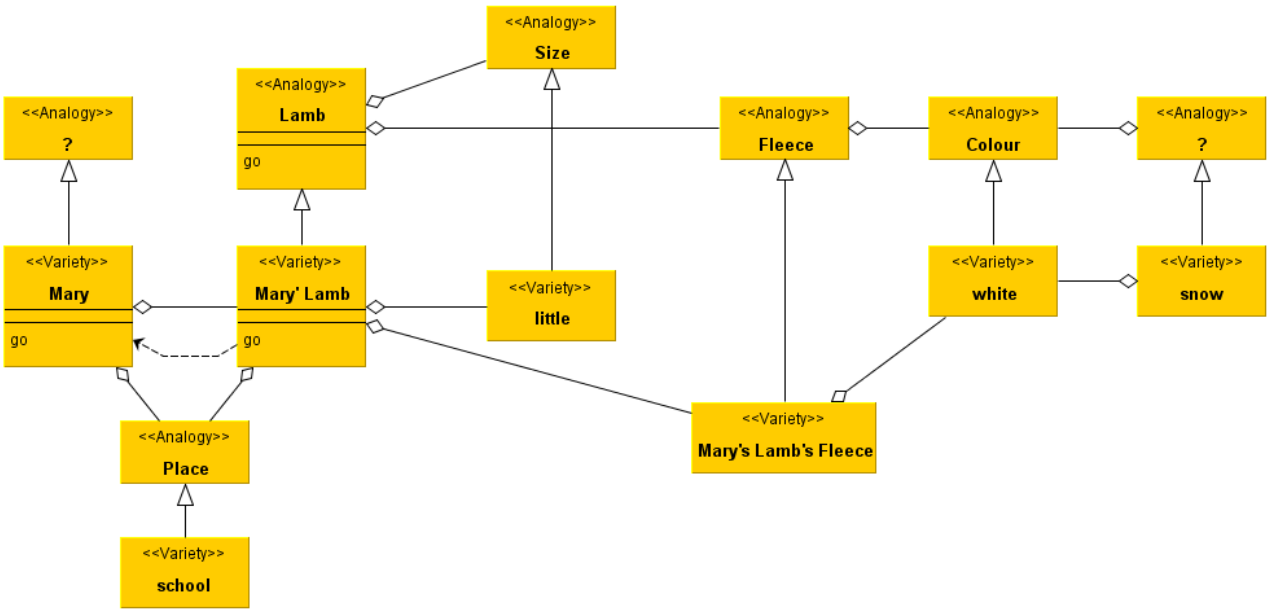


Figure 16. “Mary Had a Little Lamb” analogical diagram.

this example these are requirements that are yet to be stated or discovered). For the moment I will treat Mary as a singleton.

As a common noun, "lamb" is an analogy, the attributes and behaviours for which will become apparent..

At this point it is worth mentioning some of the exceptions and pitfalls. Nouns (as with all words) can be subject to ambiguity due to:

- homonyms, where the same sounding and/or written word can have more than one unrelated meaning (e.g. bank can be a financial institution or the bank of a river)

- polysemy, where the same sounding and/or written word has more than one related meaning. (e.g. something you can bank on is a certainty, and that has it's roots in what used to be perceived as the stability of banks)

- metonymy, where a simple attribute of a complex concept is used to identify the whole. For example "The White House" refers to the office of the President of the United States (i.e. we should not model a house that is white). The name comes from a relatively minor attribute (the colour of the external walls of a former home now used as the office building)

9.2 Adjectives

Adjectives can either be descriptive or noun modifiers.

Descriptive adjectives are varieties of analogies. When applied to a noun(analogy) this infers two things. Firstly that the lamb analogy has an attribute that the adjective is a variety of (the attribute is itself a noun/analogy). Secondly that the instance to which the adjective is applied to has the value of the adjective.

In our example "little" is a variety of the "size" analogy. So the lamb analogy has a size attribute and the particular instance (which I will call "Mary's Lamb") has "little" as it's variety/value of size. The choice of which attributes and/or behaviours we model for the lamb analogy is directly driven by the purpose that we need to view the lamb (for example our purpose does not require us to model the fact that the lamb has eyes or a mouth). We model attributes as specified by the requirements in the same way that our mind chooses analogies based on purpose. Similarly, size is an analogy and can have many attributed but as we don't need them to satisfy our requirements we model only the names of the analogy/variety.

Modifier adjectives are a way to append behaviours to an existing analogy

As will be discussed in the next section, behaviours are verbs. Verbs can sometimes be used with both an object noun (the thing doing the action) and a subject noun (the thing upon which the action is done). For example.

Dogs love bones

Pratibha compares the scores.

In these instances the nouns are already named, but if we needed to attach a name to them (to make the application of the behaviour/verb more abstract) what do we choose? To identify the object noun we nominise the verb usually by adding -or or -er to the verb root (e.g. the dog is the lover [of bones], Pratibha is the comparator). To identify the subject, we adjectivise the verb usually by adding -ible or -able to the verb root (scores are comparable, bones are loveable).

When the software programmer attempts to model such behaviour, it is the dependencies that will drive whether it should be placed with the subject or the object (more specifically the impact of the dependencies on the coupling and cohesion metric). For example, sometimes all the compare logic will exist in the comparator, in other cases our subject noun/analogy will need to have some of the compare logic, for which we will label it as a comparable element. Java interfaces are sometimes adjectives (Runnable, Comparable, etc.) and in these cases they almost certainly end in -able or -ible . These interfaces modify the existing analogy to implement some or all of a behaviour. This is why these interfaces typically have one significant method that carries the verb name (Runnable → run). Examples of these naming rules are evident across most good software.

Alternatively a number of attributes or behaviours can be wrapped up in one adjective. For example mercurial brings the attributes of eloquence, shrewdness and swiftness

Adverbs are varieties in the same way as adjectives (e.g. quickly is a type of speed so the attribute is speed and the variety is quickly). The attribute is local to the behaviour it modifies (e.g. "Mr Bolt runs quickly" and the knowledge that the adverb quickly is a type of speed [noun] shows that speed is an attribute used within the run behaviour).

9.3 Verbs

As discussed in many texts, verbs are behaviours.

In our example "everywhere that Mary went the Lamb was sure to go". There is a new noun "everywhere" which

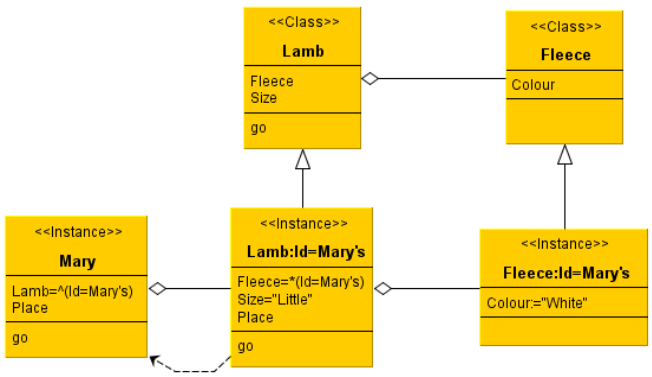


Figure 16. "Mary Had a Little Lamb" class diagram.

corresponds to “all places” or perhaps “all positions”. The verbs in the sentence are “went” (the past form of go) and go itself. There is also a dependency between where Mary goes and where the lamb goes.

Recent research [8] on verbs and nouns by psychologists show that nouns are easier to learn and understand than verbs.

- An analysis of the dictionary definitions of the 50 most used verbs and 50 most used nouns showed that the average number of different definitions for each verb was significantly more than the average number of definitions for nouns.

- Between most spoken languages there is a one-to-one correspondence between nouns, but the usage of verbs shows more divergence.

- Speech and language development starts with infants learning and repeating nouns before learning and using verbs.

The Google language “Go” is one of a number that identify analogies by pairing behaviours [verbs] rather than the name attached to an abstraction [noun] and so it could be argued that there is more likely to be confusion (due to the greater ambiguity of verbs). However, Gentner [9] argues that the mind chooses analogies using a highest rank of the behaviours, and so Go seems to be mimicking that process.

9.4 Converting Analogies To Classes

Where we do not need to model attributes or behaviours for analogies we can simply model their varieties as the strings of their names (or more formally as simple enums). For other analogies we can choose amongst the different analogical structures [12]. Collapsing our requirements and choosing to model all our non-empty analogies as classes gives the class diagram in in Figure 16.

9.5 The Semantic cost

The closer that our software is to natural language the easier it will be to understand. Even though language seems easy for us to understand, the rules that underpin it are by no means simple. We underestimate the complexities of cognitive processes because we are largely unaware of them in our conscious mind. IBM Watson's success in the “Jeopardy” challenge is a rare example of algorithmic translation of natural language. To deliver algorithmic understanding is a task that the AI researchers continue to wrestle with. It is likely that the correct identification of the analogies of the problem, and their dependencies, is a task that will continue to require human endeavor for some time yet.

9.6 Analogies In Humour

Growing up, my family and I were given a different version of the poem by our Dad:

Mary had a little lamb.
Her father shot it dead
and now it goes to school with her
between two chunks of bread

As with most humour the delight is an analogy (Mary owning a lamb that goes to school with her) between two scenarios that seem so ridiculously different (a strong bond of love between child & pet vs slaughtering animals to provide a lunchtime snack)

10. Conclusion

This paper presents a prototype of a complexity metric that is based on the principles identified from cognitive psychology and computer science. The metric measures the cost of understanding software code by employing the same rules as those applied to the structure of long term memory and the limitations of short term memory.

Most current software languages store code in flat text files (in OO languages the files typically have the structure to represent primary abstractions). The software developer employs design principles and refactoring to decide where to place the code within these files. This is effectively an optimisation problem to reduce the cognitive burden (for which the complexity metric here could serve as the optimisation function). Restricting to flat files means that, like the Crofton School problem, the derivatives of this function are unavailable and so it would be difficult to automate the recommendations of design improvements. However, if the code were placed more visually (as with the Self Language [13] and Code Bubbles [1]) then, like the force directed graph, the derivatives of the optimisation function could be used to automate refactoring.

In such a visually structured language the software developer is still needed to correctly identify the abstractions and dependencies defined by the requirements. However, it may be possible for manual refactoring to become unnecessary. Additionally, the same metric could help decide how to present abstractions to the code reader, automatically moving from a simple switch statement to a more formal class structure as the complexity and dependencies grow.

Acknowledgments

To Michael Feathers, Bernie Mullen and Vasanti Persad for their improvements to the paper.

References

[1] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr.. 2010.

Code bubbles: a working set-based interface for code understanding and maintenance. In Proceedings of the 28th international conference on Human factors in computing systems (CHI '10). ACM, New York, NY, USA, 2503-2512. DOI=10.1145/1753326.1753706

[2] N. Cowan. The Magical Number 4 in Short-term Memory: A Reconsideration of Mental Storage Capacity. In Behavioral and Brain Sciences, Vol. 24, No. 1. (February 2001), pp. 87--185. (2001)

[3] D. K. Dirlam. Most efficient chunk sizes. In Cognitive Psychology, 3:355--359, 1972.

[4] Michael Feathers. 2011. Discovering Hidden Design. http://drdobbs.com/architecture-and-design/231002664

[5] Maryanne Fisher and et al. Using Sex Differences to Link Spatial Cognition and Program Comprehension. In Proceedings of the 22nd IEEE International conference on software maintenance (2006) 289--298

[6] M. Fowler, K. Beck, J. Brant, and W. Opdyke. Refactoring: Improving the Design of Existing Code (1999). Addison-Wesley ISBN 0-201-48567-2.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[8] Gentner, D. (1982). Why nouns are learned before verbs: Linguistic relativity versus natural partitioning. In S. A. Kuczaj (Ed.), Language development: Vol. 2. Language, thought and culture (pp. 301-334). Hillsdale, NJ: Lawrence Erlbaum Associates.

[9] D. Gentner. Structure-mapping: A theoretical framework for analogy. In Cognitive Science, 7, pp 155-170 (1983).

[10] F. Gobet, P. C. R. Lane, S. Croker, P. C-H. Cheng, G. Jones, I. Oliver, and J. M. Pine. Chunking mechanisms in human learning. In TRENDS in Cognitive Sciences, 5, 236--243. (2001).

[11] J. N. MacGregor. Short-term memory capacity: Limitation or optimization? In Psychological Review, 94(1):107--108, 1987.

[12] Thomas Mullen. Writing code for other people: cognitive psychology and the fundamentals of good software design principles. In Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09). ACM, New York, NY, USA, 481-492. DOI=10.1145/1640089.1640126

[13] Randall B. Smith and David Ungar. 1994. Self: The Power of Simplicity. Technical Report. Sun Microsystems, Inc., Mountain View, CA, USA.

[14] E. Yourdon, L. Constantine. Structured Design (1979). Prentice-Hall.