Spatial measures of software complexity

C.R.Douce
ITRI, University of Brighton, Brighton, Sussex, UK.

P.J.Layzell
Software Management Group, UMIST, Manchester, UK.

J.Buckley
University of Limerick, Limerick, Ireland

January 1999

Abstract

This paper introduces a set of simple software complexity metrics that has been inspired by developments within cognitive psychology. Complexity measures are constructed by analysing the distance between components of a program. The greater the distance between program fragments, the greater the resulting spatial complexity of a program. Suggestions are made as to how spatial complexity measures can be tailored to individual programmer teams. Using these metrics, the complexity of a software system can be adjusted using subjective measures of programmer experience and knowledge. A related set of simple object-oriented metrics based around the same principles of are also suggested. Finally, a number of further research possibilities are suggested.

Index Terms : Software metrics, software complexity, psychological complexity, spatial reasoning, object-oriented programming, human-factors in software engineering, programmer experience, software maintenance.

1 Introduction

There exists the belief within engineering that if something can be measured, it can be controlled. This belief is no more evident than in the field of software engineering, where a large number of different software metrics have proliferated. One of the most important metric to receive attention has been that of the complexity metric. The motivation is simple : the more complex a software system is, the more difficult the software is to comprehend and maintain. If 'complexity' can be measured in some way, then we step towards managing and understanding software production and correction. Software complexity has been measured in a number of different ways. The simplest

of all software complexity measurements is the number of lines of code; the greater the number of lines, the more sophisticated a software system will be. Finer measurement of complexity includes simple counts of program statements and analysis of a programs control structures [1, 2]. Studying the source listing of software has caused two forms of measures to be defined, control flow complexity and data flow complexity.

Recently, object-oriented metrics has been an area of increasing interest, not only from the understanding that data and procedure are brought together and so necessitate the formation of new metrics, but also from a practical perspective. Object-oriented languages are becoming increasingly popular as a vehicle for the construction of significant software systems. A number of object-oriented metrics have been proposed by Chiadamber and Kemerer that attempt to describe the design and complexity of object-oriented software [3]. There are three types of metric, those that relate to object definitions, those that relate to object attributes or object data items, and those that relate to object communication or relations. An object definition metric is a measure of the depth of inheritance, along with a measure of how many methods are used. Data metrics are metrics that count the relationships between classes and their member functions.

It can be argued that contemporary software metrics, in part describe the software but cannot not describe how difficult parts of the software would be able to be comprehend, modify and change. Empirical software engineering practitioners have called for empirical assessment of software engineering practices and approaches; software metrics is one of the approaches that a software engineer can use [4]. Psychological complexity and software complexity are different but similar conceptions. A program that is 'psychologically complex' is a program that is difficult to understand. A program may be difficult to understand and yet still have a small number of lines, a small number of statements and low levels for certain types of complexity measure[1]. The spatial metrics that follow has been primarily inspired by theories of working memory [5]. Their intention is to measure psychological complexity simply, and in a way that can be directly related to the processes that occur during the comprehension of program code.

2 Spatial complexity metrics

Intelligence tests examine a number of cognitive abilities. Verbal ability is tested. Graphical and textual based tests are used to test induction, and spatial abilities are tested using mental rotation tasks. Spatial ability is a term that is used to refer to an individuals cognitive abilities relating to orientation, the location of objects in space, and the processing of location

[1] Within this paper, a program is considered to be a set of executable instructions that are written in a textual format.

related visual information. Spatial ability has been correlated with the selection of problem solving strategy, and has played an important role in the formulation of an influential model of working memory.

To successfully solve debugging, maintenance and comprehension tasks, programmers must posses knowledge of the programming language, have an understanding of the application domain and develop an appreciation of the relationships that can exist between the two [6]. Program comprehension and software maintenance are considered to substantially use programmers spatial abilities. To develop an understanding of non-trivial software systems, a programmer must begin to know where significant parts of the program lie and have an appreciation of their relevance to other parts of a program. Important parts of the program lie in the program 'space', which is the source file. Program space is not only one dimensional, but multidimensional. Software is not simply encoded within a single source file but can be distributed amongst any number of other files.

The idea of the programming plan or program schema has been used as an explanatory tool to explain programmer expertise. A plan represents a conception of some predefined action. In computing terms this can be a sort or a searching algorithm, for example. Letovsky and Soloway believed that programming plans can be situated within different parts of a program, and this can make programs difficult to understand [7]. Wilde et.al. stated that programs written within an object-oriented language can be especially difficult to understand since a program plan can be distributed in different program parts, within classes, methods and object [8].

The more widely distributed the connections between program functions are, the more complex the relations between the program parts become. Complexity metrics have historically been of two main types; control flow oriented and data oriented. Spatial metrics, like the object-oriented metrics that were described represents a third category of metrics : code relation metrics.

The following sections present spatial complexity measures of increasing sophistication, beginning with measures of standard procedural code. This is followed with a discussion of related measures that can be applied to object-oriented code, derived primarily from examining the C++ language, where two main measures are presented; relations that may exist between classes and relations that may exist between objects.

3 A function complexity metric

Understanding the purpose of program of a significant size necessitates the understanding the functions or procedures that are contained within a program. The greater the distance in lines of code between related functions, the more cognitive effort is required to be expended to understand the connec-

tions between functions during the initial stages of program comprehension. If a function definition directly precedes a function call, no searching will have to be performed to locate portions of source code that are needed to facilitate understanding.

The function complexity value is derived in two parts; by determining how many functions are called within a program and calculating the distance in lines of code that lie between a function call and a functions declaration. A complexity measure for any particular function can be calculated by,

$$FC = \sum_{i=1}^{name} distance_i$$

where, name is the number of functions or procedures that are called, and distance is the number of lines of code from the functions declaration[2]. FC is an absolute value.

The entire spatial complexity for a program can be calculated by summing the complexity ratings for each function it contains,

$$PC = \sum_{i=1}^{n} FC_i$$

where, n is the total number of functions that exist within a program.

Since it is very unlikely that source code is contained within a single monolithic file, the function complexity value becomes more complex. It should be calculated by totalling the distance from the function call to the top of the current file with the line number of the file where the source code is contained. In the case where no source code for a function can be found, code is contained within a library which is only available within object form only, no measure can be produced.

Two levels of granularity can be used to derive a spatial complexity measure. Firstly there are those that can be measured in lines of code, and those that are related to the position of the function in relation to others. A complexity count for the distance in lines of code can be calculated using multiples. The lower the line of code multiple, the finer the level of complexity view.

4 Recursive function complexity metric

The simple function complexity metric does not consider that function calls are very often nested within one another. For example, a programmer may define multiple functions that are called from a larger 'higher level' function.

[2]The words *function* and *procedure* are used interchangeably. The C convention of calling everything a function is adopted

The recursive metric is a simple progression. As described, a function complexity value calculated using LOC measures is calculated by taking the sum of all the distances of the functions that it calls. The RFC for a function is also calculated by summing LOC distances from calling functions. The distances are the sum of the distances that its children call. Written more formally,

$$FC = \sum_{i=1}^{n} distance_i + FC_i$$

where$n$ is the number of functions that can be called, distance is the number of lines of code from the current function, and FC is the complexity of the function that is called. The greater the levels of nesting, the more navigation throughout the source text is required, the greater the spatial complexity.

5 Object-oriented spatial complexity metrics

The spatial complexity measures can be easily modified to assess the complexity of object-oriented code, just as it can be adopted to other textual programming languages without any great degree of difficulty. Three simple measures are proposed. The first of these is very closely related to the function complexity metrics previously described, while the other two metrics relate directly to inheritance. There are two main forms of inheritance relations that are used within object-oriented languages, inheritance through class reuse and inheritance through the construction of compound objects. A fourth measure, a composite measure, is also given.

5.1 Method location rating

The function location measure is a count of how close the definition of a member function (or method) is in lines of code to its class declaration. Within the language C++, the source code for member functions can be written next to the declarations. If this is the case, spatial complexity of the software is minimal and comprehension is eased since all the relevant information is contained within one place. The number of member functions used within a class affects the function location measure. It is a measure that is distinctly reminiscent of the weighted methods per class metric (WMC) as proposed by Chidamber and Kemerer.

Within C++ language, the method location metric is calculated by summing distances from a methods implementation and description. This is represented by,

$$MLR = \sum_{i=1}^{method} distance_i$$

method is the number of methods within a class and *distance* is a function that returns the number of lines of code. In the Java language, a slightly different approach can be considered. MLR can be approximated by taking the position of the current method, to the first line of its class.

5.2 Class relation measure

The class relation measure is a measure of how close an inherited class is situated to the class which it is inherited from. The greater the distance between the class declarations, the greater the role spatial memory will play during object-oriented code comprehension and maintenance. The CRM measure is considered to be important since the comprehension of inheritance structures requires an understanding of many different attributes, knowledge of methods and an appreciation of the differences between classes. Since a programmer is unlikely to hold all information within working memory at any one time, especially when performing 'cold comprehension', knowing where a class resides is considered to be of great importance.

The CRM is calculated by,

$$CRM = \sum_{i=1}^{class} distance_i + CRM_i$$

Where, *class* is the number of classes that a class inherits, *distance* is the number of lines of code from the top of the current class to the top of an inherited class, and CRM is the distance measure of this class. If classes are not defined within available code, once again the measure cannot be derived. If classes are located in more than one file, the number of lines from the definition of a class to top of the file is summed with the line position within the file where the definition can be found.

Take the following example: If a class 'a' multiply inherits classes 'b' and 'c', a CRM measure for 'b' and all its subclasses is taken. This is repeated for class 'b'. A CRM measure for class 'a', is then simply CRM(a) + CRM(b).

5.3 Object relation measure

This metric examines the usage of object types (or declarations) within classes. The object relation measure is calculated by summing the total distance in lines of code from each object declaration to their respective class declaration. Like with the other metrics that have been discussed, if declarations exist in other files (other than files that are purely intended to be header files) the rules that have been previously stated still apply. In the situations where the object definition is unavailable, code distances cannot calculated. A separate 'not available' or NA value should then be created.

This metric has some similarity with the Chidamber and Kemerer coupling metric, CBO, which stands for *coupling between objects*. Coupling can

be described to be the measure of interdependence between modules. If a module or object does not access others, then coupling will be low, creating low interdependence. The ORM further develops the conception of coupling. An object can be considered to have low spatial coupling, or low spatial interdependence if the used object or function is located near to where it is defined.

ORM is calculated simply by,

$$ORM = \sum_{i=1}^{object} distance_i$$

Where *object* is number of objects that are used within a class declaration, and *distance* is the distance in lines of code between its usage position and the class where it is defined.

5.4 Combining measures

These measures can be combined to produce a composite view of the spatial complexity of the most significant parts of an object-oriented program. No other methods of combining the methods have currently been devised apart from a simple summation operation. Obtaining a composite complexity measure is one that is considered to be important, but without understanding what the most cognitively demanding operations when manipulating and working with object-oriented source code are, it is difficult to see how such a value may relate to program comprehension and maintenance operations.

6 Complexity and Programmer Experience

Maintainers more often than not work on software systems for large amounts of time. The measures that have been described can be used to obtain an indication of how complex a software system is for programmers who have had no experience in using a particular software systems; programmers who undertake 'cold' comprehension. The complexity scores that can be derived from software may appear to be impressive but easily become meaningless to software development managers whose programmers have been working on a software development for a year or more, for example.

Over a period of months and years, it is safe to assume that programmers consign different types of information about a software system to memory. Such information can include data flow, control flow, knowledge of functional components and problem domain information. Spatial information about a programs terrain is also held; where information about a particular area of a large software system can be found as individual programmers become familiar with particular components of a system.

A complexity measure that differs between different groups of programmers can be an especially useful tool for cost and time estimation. Measurements about the complexity of particular sections can be *weighted* using a subjective knowledge measurement provided by a programmer. This can be obtained in the form of a percentage. Individual programmers can rate particular sections of a software system in terms of their familiarity. A rating of zero percent indicates that a programmer currently has no knowledge of a particular part of a system, while a maximum one hundred percent rating suggests that a programmer can recall the position and names of all of the program segments and reconstruct the key elements directly from memory. A complexity rating weighted by programmer knowledge can be simply calculated by subtracting the suggested percentage.

Group measurements for programming can be obtained by calculating simple averages of all collected data from all members of a programming team. Over time, subjective knowledge can change or even degrade through lack of use. To maintain a correct view of programmers experience and how they affect the complexity measures, subjective measures of knowledge should be taken at regular intervals to reassess the state of knowledge. These metrics, when combined with personal adjustments, haw the potential to provide the software developer with a view of how 'complicated' program comprehension can be, and indirectly, begin to gauge how costly it can be.

7 Discussion

The spatial metrics conform to many of Weyuker's desirable properties of complexity measures [9]. Metrics should neither be too coarse or too fine. In essence, a measure should not rank different programs as being equally complex. It will also follow that if two programs were joined together, in some cases, the resulting program will be more complex than the sum of its parts. It does not follow that in all cases, if statements were re-ordered, a different measurement will be obtained. It will follow if the function position is changed.

Further work is needed to understand the relationship between the spatial complexity measurements and cognitive effort needed to understand program code. Spatial memory is said to play an important role in cognition, specifically working memory. Baddeley proposes a theory of working memory that goes further than the simple distinction of short-term and long-term memory. Evidence from cognitive neuropsychological studies of the brain damaged gives weight to the conception that spatial processing involves a particular cognitive system.

The psychological complexity of software goes beyond simple considerations about the relative position of related fragments of software. The spatial location metrics one describes a very particular view of a software

system. Like all metrics, it should be used in combination with others to obtain a full picture of the sophistication of software.

Code position is a concept that can be expanded and used to create further metrics. The metrics that have been presented are by no means perfect. They are in need of refinement. The object-oriented spatial metrics do not attempt to address additional language features such as multi-tasking and exception handling, both of which are present within the Java language. Although no direct consideration has been given to these features, using the notion of distance functions, metrics can be constructed without great difficulty.

Further research is required to further assess the advantages and shortcomings of these metrics. These metrics are rich for empirical investigation. Correlations between other more established measures should be conducted and empirical evidence should be collected to begin validation and assessment. The spatial metric is a powerful conception. It presents a view of software complexity that is related to the cognitive demands of conducting programming tasks, rather than to simple counts of lines, operators and operands. It currently remains to be seen whether the artifacts that software engineers produce can be measured with accuracy, particularly in terms of their psychological complexity. If they can, then development and maintenance may indeed become an activity that can be controlled.

References

[1] Halstead, M.H., *Elements of software science. 1977*, New York: Elsevier.

[2] McCabe, T.J., *A complexity measure. IEEE Transactions on Software Engineering*, 1976. SE-4: p. 308-320.

[3] Chidamber, S.R. and C.F. Kemerer. *Towards a metrics suite for object-oriented design.* in OOPSLA '91. 1991: ACM SIGPLAN Notices.

[4] Fenton, N. and A. Melton, *Deriving structurally based software measures. Journal of Systems and Software*, 1990. 12: p. 177-178.

[5] Baddeley, A., *Human memory : theory and practice - Revised edition.* 1997, Hove: Psychology Press.

[6] Brooks, R., *Towards a theory of the comprehension of computer programs.* International Journal of Man-Machine Studies, 1983. 18: p. 543-554.

[7] Letovsky, S. and E. Soloway, *Delocalised plans and program comprehension.* IEEE Software, 1986. 3: p. 41-47.

[8] Wilde, N., P. Mathews, and R. Huitt, *Maintaining Object-Oriented software.* IEEE Software, 1993. 10: p. 75-80.

[9] Weyuker, R., *Evaluating software complexity measures.* IEEE Transactions on Software Engineering, 1988. SE-14