6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Comparison of Object-Oriented and Paradigm Independent Software Complexity Metrics

Zoltán Porkoláb[a], Ádám Sillye[b]

Department of Programming Languages and Compilers,
Eötvös Loránd University, Faculty of Informatics
e-mail: gsd@elte.hu[a], madic@elte.hu[b]

Abstract

Structural complexity metrics play important role in modern software engineering. Testing, bug-fixing and maintenance covers more and more percentage of the software lifecycle. The cost of software maintenance is mostly depends on the structural complexity of the code. A good complexity measurement tool can trigger critical parts of the software even in development phase, measure the quality of the code, predict the cost of testing efforts and later modifications.

With the raise of object-oriented paradigm, research efforts at both the academic world and the IT industry has focused metrics based on special object-oriented features, like number of classes, depth of inheritance or number of children. Several implementations of such metrics are available for the most popular languages (like Java, C++, ...) and platforms (like Eclipse).

However object-orientation is not the only programming style used in software construction. We still have large number of legacy code written in procedural or even unstructured - way. For these codes, object-oriented metrics are not suitable. Also in modern programming languages (most importantly in C++) multiparadigm design is frequently used. An adequate measure therefore should not be based on special features of one paradigm, but on basic language elements and construction rules applied to different paradigms.

In this article, we make both theoretical and empirical comparison between such multiparadigm metrics and well-known object-oriented ones to decide their scope, identify strong and week points, and make suggestions on their practical usage.

Categories and Subject Descriptors: D.1.5 [Programming Techniques]: Object-oriented Programming; D.2.8 [Metrics]: Complexity Measures

Key Words and Phrases: Software complexity measures, Object-oriented complexity measures, Multiparadigm programming

1. Introduction

Structural complexity metrics play an important role in modern software engineering. Testing, bug fixing cover more and more percentage of the software lifecycle. The most significant part of the cost we spent on software connected to the maintenance of the software. The cost of software maintenance is mostly depends on the structural complexity of the code. A good complexity measurement tool can trigger critical parts of the software even in development phase. It can help to write good quality code, and can make assumptions on the predicted costs.

With the raise of the object oriented paradigm research efforts at both the academic world and the IT industry has have focused metrics based on special object oriented features, like number of classes, depth of inheritance tree or number of children classes. Several implementations of such metrics are available for the most popular languages (like Java, C#, C++) and platforms (like Eclipse [Eclip]).

However, object orientation is not the only programming style used in software construction. We still have a large amount of legacy code written in procedural or even in unstructured way. For that code, object oriented metrics are not suitable. Also in modern programming languages (most importantly in C++) multiparadigm design is frequently used [Cop98] [CE00]. An adequate measure therefore should not be based on special features of one paradigm, but on basic language elements and construction rules applied to different paradigms.

2. Currently popular metrics

2.1. Size metrics

Most primitive metrics are based on the physical attributes of the program code. The LOC (lines of code) measures the actual size of code, while eLOC (effective lines of code) differs only in ignoring the possibly ineffective lines like comments and block commands and so on. The number of commands (NOC) works similarly it counts the statements. While size metrics are cheap to compute and has correlation with many other more complex metrics, they have a critical weakness, namely they completely ignore the program semantics (thus they can produce misleading results).

2.2. Structural metrics

The first well-known structural measure was developed by McCabe. The cyclomatic complexity is based on the number of predicates (branches) in a program: The aim of this metrics was to approximate the testing effort of FORTRAN programs; therefore the cyclomatic number reflects the independent testing passes of the program. The inadequacy of the measure becomes clear, if we realize that the complexity also depends on the nesting level of the predicate nodes. According to

the McCabe cyclomatic complexity, a sequence of ten loops is identically complex to ten loops nested into each other.

Harrison and Magel [HM81] proposed a metric, which also takes into account the nesting depth of the predicate statements. The more nested statements weighted higher in the complexity. This concept was improved by Howatt and Baker [HB89] whose definition for nesting level was applicable for non-structured programs too.

2.3. Object oriented metrics

Chidamber and Kemerer [Chi94] and Henderson-Sellers [Hen96] gave one of the most complete discussions about object-oriented metrics. They encounter the following important object oriented complexity properties:

WMC (Weighted Methods per Class): The summarization of some structured complexity metrics (most frequently: McCabe) for the methods of a class.

DIT (Depth of Inheritance Tree): Shows the position of the class in the inheritance tree.

NOC (Number of Child Classes): This measures the importance of the class rather than the complexity.

CBO (Coupling Between Object Classes): The number of references to the class from others (fan-in) and those that are pointing out of the class itself (fan-out). This measures the vulnerability of the class.

RFC (Response for Class): Calling a method on the interface of the class generates other method calls. The more traffic is generated by an incoming call the testing efforts will be harder.

LCOM (Lack of Cohesion in Methods): This metrics based on an empirical result that the cohesion within a class reflects the better abstraction.

3. A paradigm independent metric

A paradigm-independent software metrics are applicable for programs written in different paradigms or in mixed-paradigm environment. Such metrics should be based on general programming language features, which are paradigm- and language independent. The paradigm-dependent attributes are derived from these features.

These features are the following:

- Control Structure of Program: most of the programs have the same control statements.
- Complexity of data types: reflects the complexity of data used (like classes)
- Complexity of data access: connection between control structure and data gives the direction of the data flow and nesting depth of the data handing.

Now we give a brief definition for the AV graphs: The precise introduction of the AV-graph metric can be found in [PZ1] and [PZ2]

Consider a data and flow graph (AV graph) $G' := (N \cup D, E \cup R, s, f)$ where N is the set of the control nodes, D the set of data nodes (or attributes), E the set of flow edges, R the set of data access edges. Let p denote a predicate node in the execution flow, so that p has at least two outgoing flow edges. (An AV graph for a class is a set of execution graph components and data nodes.)

The nesting depth of node p is $nd(p) := |pred(p)|$. This means that in the execution flow between start and the p node we can fount exactly $nd(p)$ predicate nodes. The total nesting depth of a graph is simply $ND'(G') := \sum_{p \in N'} nd(p)$, while the complexity of a class is: $SN'(G') := |N \cup D| + ND'(G')$.

The complexity of the AV graph depends on the control structure and the data handling. The control structure - with the help of predicate nodes - defines the nesting depth of control nodes and the depth of data handling. The total complexity is expressed by the nesting level of both data and control.

A *class-graph* $O = \{G|G\text{is an AV graph}\}$ is a finite set of AV graphs (the *member graphs*). The set of edges $\varepsilon = (E \cup R)$ represents the E edges belonging the control structure of one of the member graphs and R as the data reference edges of the attributes. As the control nodes (nodes belonging to the control structure of one of the member graphs) were unique, there is no path from one member graph to another one. However, there could be attributes (data nodes) which are used by more than one member graph. These attributes have data reference edges to different member graphs.

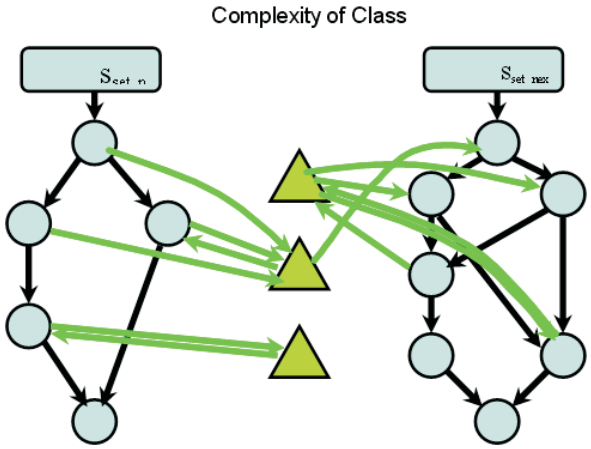
This is a natural model of the class. It reflects the fact that a class is a coherent set of attributes (data) and the methods working on the attributes. Here the *methods* (member functions) are procedures represented by individual data flow graphs (the member graphs). Every member graph has its own start node and terminal node, as they are individually callable functions. What makes this set of procedures more than an ordinary library is the *common set of attributes* used by the member procedures. Here the attributes are not local to one procedure but local to the object, and can be accessed by several procedures.

The AV graph metric is based only on the paradigm independent properties, so it a multi paradigm metric.

- Control structures: the complexity of method control structures are expressed directly in the AV graph.

- Complexity of data types: local variables in methods and attributes are expressed as data nodes.

- Complexity of data handling: the interconnection between control structure and data is also expressed directly in the AV graph.




```
class date {
    int year, month, day;
public:
    void set_next_month() {
        if ( month == 12 ) {    month = 1; year = year + 1; }
        else               {    month = month + 1; }     }
    void set_next_day() {
        if ( month == 1 || month == 3 || ... || month == 12 )
            if ( day == 31 )  set_next_month();
            else              day = day + 1;
        else
            if ( day == 30 )  set_next_month();
            else              day = day + 1;    }
};
```

The figure shows the AV graph of an example class. The round nodes are the instructions in the execution flow which is represented by black arrows, the triangles are the data nodes with green arrows representing the data handling.

4. About the comparison process

4.1. The measuring tool

We implemented measuring tool that computes several software metrics. It supports both Java 1.3 and 1.4 standards. (We are planning to extend the support

for C# and C++.)

The calculated metrics are the following:

Size metrics	eLOC, Number of Statements
Structural complexity	McCabe, Howatt-Baker
Object oriented metrics	Inner Class Depth, Inheritance level, Number of Children, Number of Methods, Number of Fields, Lack of Cohesion, normalized Lack of Cohesion (Henderson-Sellers), Fan-out
Multi paradigm metrics	AV graph

4.2. The test data

Our test data for the comparison process was a selection of large Java modules and libraries. The overall size of the test input was more than 1.5 million effective lines of code that contain more than 17.000 classes.

The following table shows the test modules and their physical sizes:

Module	**eLOC**
Java Standard Library 1.4.2	367.000
jBOSS 3.2.3	300.000
Omg.org.CORBA	5.000
The measure tool	7.000
Eclipse 3.0M6	900.000
Overall	>1.500.000

4.3. Calibration of the tool

During the comparison process, our goal was to tune our program to achieve statistically similar results that other measurer tools provide.

Difficulties

Comparing several the outputs of several tools and several metrics raises many important questions that make the evaluation complicated:

- Metrics are not defined precisely enough from the view of implementation. Almost every tool uses slightly different interpretation of software complexity metrics.
- It is quite common from the implementers that they are simplifying the definitions of the metrics; therefore, each tool has somewhat different implementation.

- Tools have several major or minor bugs, so only partial measurements are available.

- Available tools working on different abstract program trees. This means that they are parsing the input sources with different parser engines that generate different syntax trees. Theoretically, this would not be a problem if we suppose that these abstract trees contain all necessary information for the metrics computation. However, the practice shows that the different input data structure leads different implementation compromises, consequently the results could not be accurate enough to compare.

- The different output formats of the tools need conversion to make them uniform.

Example

One of the typical problems that complicates the comparison is the computation of the LCOM (Lack of COhesion in Methods) metric. Remember that this number is illustrating the quality of the abstraction for a class. The LCOM number according to the definition is LCOM=|P|if |P|-|Q|$\geq$ 0, otherwise 0. The P and Q are standing for:

- The number of method pairs within the class that access different fields (P).

- The number of method pairs that access at least one common field (Q).

The problem rise when the source uses the this and super keywords to refer the appropriate class. These references must not be taken in account in P and Q because these do not affect the cohesion within a method. Even though these references are not explicitly part of the abstraction, some tools count them as a class field.

Such minor differences imprecise results that can make the evaluation and the comparison impossible.

5. Results and conclusion

Results

Our measurements produced several interesting results about the software metrics. We compared the several metric results using statistical correlation.

Comparing object-oriented metrics with each other resulted that there are no relationship between them in the sense of correlation. Most of the values were under 0.2 and all of them were under 0.5. (It is interesting that there are higher correlation values between LCOM and the Number of Fields, understanding this experience may need further research.) It is clear that the object-oriented metrics have very different meanings, so the low correlation between them is reasonable.

There is no statistical correlation between the object oriented and multi paradigm metrics. All correlation values were under this unexpected, but very important result points out that the importance of the structural complexity of the software is very high.

Tools are not reliable

There are several difficulties in the way of creating a reliable measuring tool, because every major step during the complexity computation has uncertain factors: there are no precise definitions for metrics, the different parsing engines produce different abstract syntax trees and the implementation compromises and simplifications are making the results almost incomparable.

Object oriented and multi paradigm metrics

Quantities from the Object-Oriented Metrics suite are showing some rough properties of the software. In other words, they are depicting only the big picture about source. A good example is the depth of inheritance tree: according to our measures Eclipse, Java and jBoss all have six levels deep inheritance trees, even though they have completely different properties.

By considering more (detailed) properties of the software, multi paradigm metrics have much higher density. Evaluation of the results shows that the structural complexity of methods is extremely increasing the overall complexity of the software product, so it is not satisfactory to consider only its object-oriented properties.

Appendix

The following table shows the statistical correlations between several metrics calculated on jBoss application server. The results for the other tested software were similar to this. The low correlation numbers (under 0.5) are highlighted by blue while the high correlations (above 0.9) are highlighted by red.

jboss	Inheritance level	LCOM	Henderson-Sellers LCOM*	Number of child classes	Fan out	Number of fields	Number of methods	Effective code lines	McCabe	Howatt Baker	AV graph
Inheritance level	1.00	0.01	0.12	0.02	0.18	0.04	0.04	0.06	0.03	0.08	0.03
LCOM	0.01	1.00	0.04	0.01	0.06	0.53	0.77	0.18	0.36	0.21	0.03
Henderson-Sellers LCOM*	0.12	0.04	1.00	0.06	0.30	0.20	0.33	0.23	0.22	0.11	0.10
Number of child classes	0.02	0.01	0.06	1.00	0.01	0.02	0.06	0.03	0.06	0.03	0.01
Fan out	0.18	0.06	0.30	0.01	1.00	0.08	0.34	0.52	0.43	0.18	0.32
Number of fields	0.04	0.53	0.20	0.02	0.08	1.00	0.51	0.40	0.23	0.25	0.05
Number of methods	0.04	0.77	0.33	0.06	0.34	0.51	1.00	0.50	0.70	0.38	0.23
Effective code lines	0.06	0.18	0.23	0.03	0.52	0.40	0.50	1.00	0.66	0.45	0.57
McCabe	0.03	0.36	0.22	0.06	0.43	0.23	0.70	0.66	1.00	0.74	0.67
Howatt Baker	0.08	0.21	0.11	0.03	0.18	0.25	0.38	0.45	0.74	1.00	0.78
AV graph	**0.03**	**0.03**	**0.10**	**0.01**	**0.32**	**0.05**	**0.23**	**0.57**	**0.67**	**0.78**	**1.00**

References

[Bal01] Balla, K. The Complex Quality World, Eindhoven University Press, Eindhoven (2001).

[CE00] Czarnecki K., Eisenecker U.W. Generative Programming Addison-Wesley, (2000).

[Chi94] Chidamber S.R., Kemerer, C.F. A metrics suit for object oriented design, IEEE Trans. Software Engeneering, vol.20, pp.476-498, (1994).

[Cop98] Coplien J.O. Multi-Paradigm Design for C++ Addison-Wesley, (1998).

[Eclip] Eclipse.org formation `http://www.eclipse.org/org/index.html`

[FN93] Fóthi Á., Nyéky-Gaizler J. On the Complexity of Object-Oriented Programs Proc. of the 3rd Symp. on Programming Languages and Software Tools Kaariku, Estonia, 1993.

[FNP99] Fóthi, Á., Nyéky-Gaizler, J., Porkoláb, Z. On the Complexity of Class Proc. of the FUSST'99, Tallin, Estonia, pp.221-231 (1999).

[Fen00] Fenton, N.E., Neil, M. Software Metrics: Roadmap, The Future of Software Engineering. ACM Press, New York, (2000).

[Gra89] Grassberger, P. Estimating the information content of symbol sequences and efficient codes IEEE Transactions on Information Technology. Vol.35,669. (1989).

[HB89] Howatt,J.W. and Baker,A.L. Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting The Journal of Systems and Sofware 10, pp.139-150 (1989).

[HK81] Henry S., Kafura D. Software Sructure Metrics Based of Information Flow IEEE Trans. Software Engineering, vol.7, pp.510-518 (1981).

[HM81] Harrison,W.A. and Magel,K.I. A Complexity Measure Based on Nesting Level] ACM Sigplan Notices,16(3), pp.63-74 (1981).

[Hal72] Halstead, M. H. Natural laws controlling algorithm structure SIGPLAN Notices, vol.7. pp.19-26 (1972).

[Hen96] Henderson-Sellers, B., Object-oriented metrics: measures of complexity, Prentice-Hall, pp.142-147, (1996).

[Hum89] Humphrey, W.S. Managing the Software Process, Addison-Wesley, Reading, Massachusetts (1989).

[Koko99] Kokol, P., Podgorelec, V., Zorman, M. Universality - A Need For A New Software Metric International Workshop on Software Measurement (IWSM'99) Lac Supérieur, Canada, (1999).

[McC76] McCabe, T.J. A Complexity Measure, IEEE Trans. Software Engineering, SE-2(4), pp.308-320 (1976).

[Meth] Method Object `http://www.c2.com/cgi/wiki?MethodObject`

[Mor89] Morris, K.L. Metrics for Object-Oriented Software Development Environments, Master's Thesis, M. I. T. Sloan School of Management, (1989).

[PZ1] Fóthi Á., Nyéky-Gaizler J., Porkoláb Z The Structured Complexity of Object-Oriented Programs Computers and Mathematics with Applications accepted for publication (2002)

[PZ2] Porkoláb Zoltán: Programok strukturális bonyolultsági mérőszámai, Doktori Értekezés, ELTE 2002 (in hungarian)

[Piw82] Piwowarski,P. A Nesting Level Complexity Measure ACM Sigplan Notices, 17(9), pp.44-50 (1982).

[Sche93] Schenkel, A., Zhang, J., Zhang, Y. Long range correlations in human writings Fractals, Vol.1, no.1, pp.16-19. (1993).

[Schn91] Schneidewind, N., F. Validating Software Metrics: Producing Quality Discriminators Proceedings of the Conference on Software Maintenance (CSM91), Sorrento, Italy, (1991)

[Shan51] Shannon, C.E. Prediction and entropy of printed English Bell System Technical Journal. Vol.30,50. (1951).

[Vil] Vil - View Intermediate Language `http://www.1bot.com`

[Wey88] Weyuker, E.J. Evaluating software complexity measures] IEEE Trans. Software Engineering, vol.14, pp.1357-1365 (1988).

[Zus99] Zuse, H. Software Complexity: Measures and Methods De Gruyter, Berlin, (1991).

[Zus99] Zuse, H. Validation of Measures and Prediction Models International Workshop on Software Measurement (IWSM'99), Lac Supérieur, Canada, (1999).

Postal address

Zoltán Porkoláb, Ádám Sillye
Department of Programming Languages and Compilers
Eotvos Lorand University, Budapest,
H-1117 Pazmany Peter setany 1/c.
Hungary