Towards a multiparadigm complexity measure

Zoltán Porkoláb and Ádám Sillye

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary,
phone: +36 1 381-2319, fax: +36 1 381-2185
{gsd, madic}@elte.hu

Abstract. Structural complexity metrics play important role in modern software engineering. 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. However object-orientation is not the only programming style used in software construction. In modern programming languages 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 propose such a multiparadigm metrics and evaluate it.

Area M (General Aspects of Measurement), Metrics for multiparadigm programs

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).

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. Also in modern programming languages

(most importantly in C++) *template metaprograms* are often used [Cza00]. In Java *generics* just have been introduced [Jav04], but *Aspect Oriented Programming* [Kic96] is already highly popular. For these code object oriented metrics are not suitable. More interestingly programs frequently constructed with the mixture of paradigms above. Such *multiparadigm programs* [Cop98] can appear in C++ and Java specially on the .NET platform.

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. 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.

Our proposal is, that when counting the complexity of a program, we should take the complexity of the data used and the complexity of data handling into consideration; we should see the decreasing of complexity through hiding techniques. Accordingly, the complexity of a program is a sum of three components

1. Control structure of program: most of the programs have the same control statements irrespectively of paradigm used.
2. Complexity of data types: reflects the complexity of data used (like classes)
3. Complexity of data access: connection between control structure and data gives the direction of the data flow and nesting depth of the data handing.

In the following sections we define our measure called AV-garph. Then we define the complexity of class. Class is defined as a set of data (attributes) and control structures (member functions, methods) working on the attributes. We define the complexity of class as a sum of the complexity of attributes and the complexity of member functions. The definition reflects the common experience that good object-oriented programs have very strong bindings between the attributes and the methods inside the class and have week connections between different classes. The measure is also examined on special cases such as member functions calling other member functions, and classes with no attributes (old-style libraries), etc.

Afterwards we examine the complexity issues of the connection of classes. Inheritance and aggregate relationships between classes can increase global complexity of the program. However, every time we use the same class we can see the benefit of these constructions. We show on some classical example, how complexity depends on binding between classes.

2 The proposed metrics

The well-known measure of McCabe (cyclomatic complexity) is based only on the number of predicates in a program: $V(G) = p + 1$. The inadequacy of the measure becomes clear, if we realize that the complexity depends basically on the nesting level of the predicate nodes. The measures proposed by Harrison and

Magel [Har81] and Piwowarski [Piw82] proven to be equivalent in principle by Howatt and Baker [How89] take this lack into account.

The complexity of the control structure of a program is defined with the help of the nesting depth. It is important to carefully define the complexity, not to exclude non-structural programs. Programming languages, like C#, C++, Java reintroduced non-structured control facilities with the exception handling.

Definition Given control garph $G = (N, E, s, t)$ and $p \in N$ predicate node. The scope of p is: $Scope(p)$.

Definition Given control graph $G = (N, E, s, t)$, and $x \in N$ node. For a node x the predicate set was defined as:

$$Pred(x) = \{p \mid x \in Scope(p)\}$$

.

Definition Given control graph $G = (N, E, s, t)$ and $x \in N$ the nesting depth of node x is:

$$nd(x) =\mid Pred(x) \mid$$

Definition Given control graph $G = (N, E, s, t)$ the total nesting depth of the graph is:

$$ND(G) = \sum_{n \in N'} nd(n)$$

Definition Given control graph $G = (N, E, s, t)$ the scope number of the graph is defined as:

$$SN(G) = \sum_{n \in N'} (\mid Scope(n) \mid +1) =\mid N' \mid +ND(G)$$

There is a clear analogy with the McCabe metrics. Here however we summarize the nesting depth of nodes rather than summarize the number of predicate nodes.

2.1 The role of data handling

An important feature of our software metrics is that it doesn't count the complexity of data handling based on the place of the declaration. The metrics encounter that value exactly at the point of data handling. This of course also measure the place of declaration in an implicite way: local variables are used only in the local code context (in the subprogram).

Definition Given an AV graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$, $\mathcal{N} = N \cup D$, where N is the set of the control nodes, D the set of data nodes, $p \in N$ (not neccessery) predicate node. The data-scope (D-scope) of node p is $DScope(p) = \{d \in D \mid$

$\exists n \in N \ \wedge \ n \in Scope(p) \cup \{p\} \ \wedge \ ((n, d) \in \mathcal{E} \vee (d, n) \in \mathcal{E})$

Definition Given an AV graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ $\mathcal{N} = N \cup D$, the set of control nodes is N, the set of data nodes is D, node $p \in N$ predicate. The data and control scope of node p

$$AVScope(p) = Scope(p) \cup DScope(p)$$

Given AV graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$, and node $x \in \mathcal{N}$. The set of nodes that predicate a node x is

$$AVPred(x) = \{p \mid x \in AVScope(p)\}$$

.

Definition. Given AV graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ the nesting depth of node $x \in \mathcal{N}$ is:

$$nd(x) = \mid AVPred(x) \mid$$

Definition Given Av graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ the total nesting depth is:

$$ND(\mathcal{G}) = \sum_{n \in \mathcal{N}'} nd(n)$$

Definition Given AV graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ the complexity of the graph is

$$\mathcal{C}(\mathcal{G}) = \mid \mathcal{N}' \mid + ND(\mathcal{G})$$

The complexity of th 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 exprssed by the nesting level of both data and control.

There is an another possible way to get these results. Let suppose we have no data nodes and data edges in our graph but we replace them whit special control nodes: „reader" and/or „writer" which do only receiving and sending information. These nodes will be inserted just before and after the real control nodes which read and/or write data. The nesting depth and complexity value we get with this model is the same we count based on AV graphs.

3 The complexity of class

We can naturally extend our model to object-oriented programs. In the centre of the object-oriented paradigm there is the class. Therefore we should first describe how we measure the complexity of a class. In the base of the previous sections we can see the class definition as a set of (local) data and a set of methods.

Definition A *class-graph* $\mathcal{O} = \{G \mid G\ AVgraph\}$ is a finite set of AV graphs (the *member graphs*). The set of nodes $\mathcal{N} = N \bigcup D$, where N represents

the nodes belonging the control structure of one of the member graphs and D represents the data nodes used by the member graphs. We can call D also as the **set of attributes** of the class. The set of edges $\mathcal{E} = E \bigcup R$ 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-flowgraphs (the member graphs). Every member graph has his 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.

Definition The *complexity of a class* can be computed in a very similar way to the complexity of the program:

$$\mathcal{C}(\mathcal{O}) = \mid N' \mid + \sum_{\mathcal{G} \in \mathcal{O}} ND(\mathcal{G})$$

It is important to stress that the complexity of the class is inherited from both the complexity of the control flow and the complexity of data structure.

Lemma The complexity of the class can be computed as the sum of the attributes and the sum of the complexity of disjunct memberfunctions.

$$\mathcal{C}(\mathcal{O}) \ = \mid \mathcal{A} \mid + \sum_{\mathcal{G} \in \mathcal{O}} (ND(\mathcal{G}) + \mid L_G \mid)$$

In the following example we represent a class with an AV graph. Let consider the *year* data node, as one of the nodes witch used by more than one method.

ND($set\_\ next\_\ month$) = 14
$\mid N' \mid = 4$
ND($set\_\ next\_\ year$) = 9
$\mid N' \mid = 3$
The number of the attributes in this graph:
$\mid A \mid = 3$
Total complexity:
$\mathcal{C}(date) = 18 + 12 + 3 = 33$

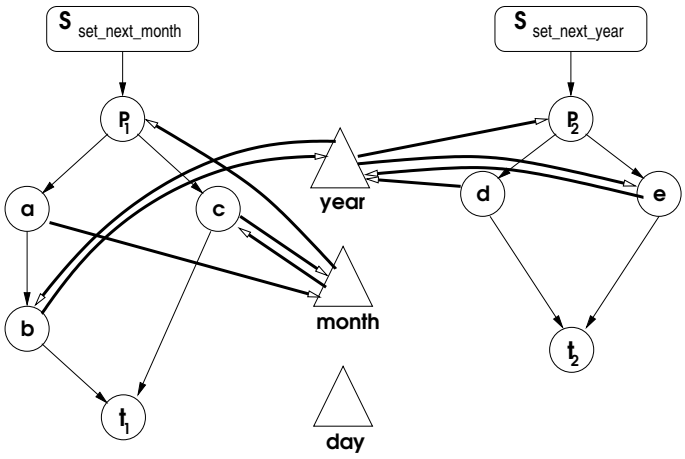

Fig. 1. example class

4 Evaluation of the metrics

Let consider that h definition of AV graph permits the emty set of control nodes. In that case we get a classical data structure. The complexity of a classical data structure is the sum of the data nodes.

The opposite situation is also possible. When a „class" contains disjunct methods, so there is no common data shared between them – there are no attributes – we compute the complexity of the class as the sum of the complexity of the disjunct functions. We can identify this construct as an ordinay function library.

These examples also point to the fact,that we use paradigm-independent notions, so we can apply our measure onto procedural, object-oriented or even mixed style programs. This was our goal.

Let suppose we have two identical classes. The only difference between the two code is that one of them using *private* declarations to hide those variables not to belong to interface. Can an ordinary C++ programmer see the differences *in complexity* between the two definitions? We can hardly say yes. However, there could be differences in the complexity of the *client code*, which uses the class. If the client accesses the attributes of the class via the *set_next_month* function, we can replace its subgraph in the client code in the known way. This decreases the complexity of the client code.

A data member of a class is marked with a single data node regardless of its internal complexity. If it represents a complex data type, its definition should be included in the program and its complexity is counted there. Up to the point, where we handle this data as an atomic entity, its effect to the complexity of the handler code doesn't differ from the effect of the most simple (built-in) types. From the viewpoint of the code using class *date*, the internal implementation of *date* makes no difference.

If T is a user-defined class, then the complexity of the whole system (containing the standard libraries, functions and the other user-defined classes earlier defined is increasing. The complexiy of T has been added to the system. This is however a good trade-of, because the constant effort we did here, will result a code decrease in linear, when we use that more intelligent class.

Here we use the member functions of *date*, the calls of which have constant complexity regardless of its implementation. Hovewer, if we break the encapsulation of class *date* (ie. we directly access its components), the data reference edges connect the handler code to the internal representation and increment its complexity. Once again, we stress this fact has to do with the private or public members only in an indirect way: as far as we use the methods to handle data, it doesn't matter whether the components are public or private. Of course, the compiler supports this strategy only when we made our components private.

Inheritance is handled in a similar way. Code of the derived class in most cases (but not neccessery) refers to the methods and/or data members of the base class(es). These references (method calls or data accesses) are described in the very same way as we did in the case of procedural programs. The motivation here is again to derive complexity from the basic, paradigm–independent program elements.

5 Weyuker axioms

Elaine J, Weyuker in 1988 proposed logical statements over complexity metrics based on syntactical features of program [Wey88]. The statements – often called Weyuker axioms, however they are not axioms in the mathematical sense – define the "expected behaviour" of software metrics. Here we show the results of the evaluation of some famous metrics and ours against the Weyuker axioms. (The + sign means that the appropriate metrics fulfills the certain "axiom".)

#	# statements	McCabe	Halstead	dataflow	AV graph
1	+	+	+	+	+
2	+	-	+	-	with mod.
3	+	+	+	+	+
4	+	+	+	+	+
5	+	+	-	-	+
6a	-	-	+	+	+
6b	-	-	+	+	-
7	-	-	-	+	+
8	+	+	+	+	+
9a	-	-	+	+	-
9b	-	-	+	+	+

6 Empirical results

There is a software implementation of our metrics working on Java sources, computing the AV-graph complexity. The implementation was written itself in Java, using an open source parser: CUP [Cup02]. The software is also able to measure some other complexity numbers, like

1. Size metrics: eLOC, Number of Statements.
2. Structured complexity: McCabe cyclomatic number, Howatt-Baker's nested complexity measure.
3. Object-oriented measures: 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

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:

1. Java Standard Library 1.4.2 367.000 lines
2. jBOSS 3.2.3 300.000 lines
3. Omg.org.CORBA 5.000 lines
4. The measure tool itself 7.000 lines
5. Eclipse 3.0M6 900.000 lines

Our measurements produced several interesting results about the software metrics. Comparing AV-graph metrics with the McCabe cyclomatic complexity number produced strong correlation for most of the test datas. Howatt-Baker's nested complexity measure produced similar values. However there were test samples were the correlation was under 0.8.

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 multiparadigm metrics. All correlation values were low.

References

[Che91] Cherniavsky J.C., Smith C.H. *On Weyuker's Axioms For Software Complexity Measures*, IEEE Trans. Software Engineering, vol.17, pp.1357-1365 (1991).

[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).

[Cur79] Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A., Love, T. *Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics*, IEEE Trans. Software Engineering, vol.5 no.2, pp.96-107, (1979).

[Cza00] Czarnecki K., Eisenecker U.W. *Generative Programming*, Addison-Wesley, (2000).

[Dun79] Dunsmore, H. E., Gannon, J. D. *An Empirical Investigation*, Computer, 12(12), pp.50-59 (1979).

[Dav88] Davis J.S., LeBlanc R.J. *A Study of the Applicability of Complexity Measures*, IEEE Trans. Software Engineering, vol.14, pp.1366-1372 (1988).

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

[Har81] Harrison,W.A. and Magel,K.I. *A Complexity Measure Based on Nesting Level*, ACM Sigplan Notices,16(3), pp.63-74 (1981).

[Mag81] Harrison,W.A. and Magel,K.I. *A Topological Analysis of the Complexity of Computer Programs with Less Than Three Binary Branches*, ACM Sigplan Notices,16(4), pp.51-63 (1981).

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

[Kaf81] Henry S., Kafura D. *Software Sructure Metrics Based of Information Flow*, IEEE Trans. Software Engineering, vol.7, pp.510-518 (1981).

[How89] 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).

[Kic96] Kiczales G. *Apect Oriented Programming*, AOP Computing surveys 28(es), 154-p (1996)

[Lak91] Lakshmanan K.B., Jayaprakash S., Sinha P.K. *Properties of Control-Flow Complexity Measures*, IEEE Trans. Software Engineering, vol.17, pp.1289-1295 (1991).

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

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

[Pra84] Prather, R. E. *An axiomatic theory of software complexity*, Comput. J., vol. 27. no. 4. pp.340-346, (1984).

[Str97] Stroustrup B. *The C++ Programming Language*, Addison-Wesley, (1997).

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

[Cup02] CUP Parser Generator for Java,
`http://www.cs.princeton.edu/~appel/modern/java/CUP`

[Ecl01] Eclipse.org formation,
`http://www.eclipse.org/org/index.html`

[Jav04] Java 1.5,
`http://java.sun.com/developer/technicalArticles/releases/j2se15`

[Fot02] Fóthi Á., Nyéky–Gaizler J., Porkoláb Z. *The Structured Complexity of Object-Oriented Programs*, Computers and Mathematics with Applications accepted for publication (2002).

[Fot99] Fóthi, Á., Nyéky-Gaizler, J., Porkoláb, Z. *On the Complexity of Class*, Proc. of the FUSST'99, Tallin, Estonia, pp.221-231 (1999).