Measuring the Complexity of Aspect-Oriented Programs with Multiparadigm Metric*

Norbert Pataki, Ádám Sipos, and Zoltán Porkoláb

Department of Programming Languages and Compilers
Eötvös Loránd University, Faculty of Informatics
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
`patakino@elte.hu, shp@elte.hu, gsd@elte.hu`

Abstract. Aspect-oriented programming (AOP) is a promising new software development technique claimed to improve code modularization and therefore reduce complexity of object-oriented programs. However, exact quantitative inspections on the problem details are still under way. In this paper we describe a multiparadigm software metric and its extension to AspectJ. We use the metric to compute structural complexity of all the object-oriented, aspect-related and procedural components of AOP code. We tested our metric on two functionally equal implementations of GoF design patterns made in aspect-oriented way and in pure object-oriented style and compared the results.

1 Introduction

Metrics play an important role in modern software engineering. Testing, bugfixing cover an increasing percentage of the software lifecycle. In software design the most significant part of the cost is spent on the maintenance of the product. The cost of software maintenance highly correlates with the structural complexity of the code. The critical parts of the software can be identified in the early stages of the developement process with the aid of a good complexity-measurement tool. Based on software metrics we can give recommendations and define coding conventions on the development of sound, manageable and hygienic code. Even though general recommendations for specifying sensible metrics exist [23], the concrete measurement tools are typically paradigm-, and language-dependent.

In the software development process *abstractions* play a central role. An abstraction focuses on the essence of a problem and excludes the special details [4]. Abstractions depend on many factors: user requirements, technical environment, and the key design decisions. In software technology a *paradigm* represents the directives in creating abstractions. The paradigm is the principle by which a problem can be comprehended and decomposed into manageable components [1]. In practice, a paradigm directs us in identifying the elements in which a problem will be decomposed and projected. The paradigm sets up the rules and

* Supported by GVOP-3.2.2.-2004-07-0005/3.0

properties, but also offers tools for developing applications. These methods and tools are not independent of their environment in which they occur.

The last 50 years of software design has seen several programming paradigms from *automated programming* and the FORTRAN language in the mid-fifties, to *procedural programming* with structured imperative languages (ALGOL, Pascal), to the *object-oriented* paradigm with languages like Smalltalk, C++ and Java. However, it is important to understand that new paradigms cannot entirely replace the previous ones, but rather form a new structural layer on the top of them. Object-orientation is a new form of expressing relations between data and functions, however, these relations implicitly existed in the procedural paradigm.

The need for new programming paradigms is a result of the ever-growing complexity of software. Object-oriented programming (OOP) is widely used in the software industry for managing large projects, but recently some of the weaknesses emerged. Problems like cross-cutting concerns, multi-dimensional separation of concerns, symmetric extension of a class hierarchy [22] are hard to handle. Modern programming languages have made possible the birth of new programming paradigms like *(C++) template metaprogramming* (TMP) [5], *generic programming* (GP) [24], and *aspect-oriented programming* (AOP) [15].

Software metrics have always been strongly related to the paradigm used in the respective period. The McCabe *Cyclomatic complexity* number [2] was designed for measuring the testing efforts of non-structural FORTRAN programs. Piwowarksi [17], Howatt and Baker [11] extended the cyclomatic complexity with the notion of *nesting level* in order to describe structured programs better. After the object-oriented paradigm became widely accepted and used, both the academic world, and the IT industry focused on metrics based on special object-oriented features, like *number of classes*, *depth of inheritance tree*, *number of children classes*, etc. [3]. Several implementations of such metrics are available for the most popular languages (like Java, C#, C++) and platforms (like Eclipse) [25].

Most programs are written by using more paradigms. Object-oriented programs have large procedural components in implementations of methods. AOP implementations (among which the most widely-used is AspectJ), highly rely on OOP principles. AspectJ essentially integrates tools for modularizing crosscutting concerns into object-oriented programs. Moreover *multiparadigm programs* [4] appear in C++, Java, on the .NET platform, and others.

Metrics applied to different paradigms than the one they were designed for, might report false results [21]. Therefore an adequate measure applied to multiparadigm programs should not be based on special features of only one programming paradigm. A multiparadigm metric has to be based on basic language elements and constuction rules applied to different paradigms. A paradigm-independent software metric is applicable to programs using different paradigms or in a multiparadigm environment. The paradigm-independent metric should be based on general programming language features which are paradigm- and language independent.

Here we give the structure of the paper. In section 2 we define our multiparadigm metric, the *AV-graph*. After that we define the complexity of the class, where class is defined as a set of data (attributes) and control structures (member functions, methods) carrying out operations on the attributes. Afterwards, in section 4 we explain how our metric applies to AOP notions and constructs. Section 5 describes our test results when applying the metric to the OOP and AOP versions of the GoF design patterns. We explain how AOP affects the complexity of these implementations, and in which cases AOP provided a simpler solution.

2 A multiparadigm metric

The well-known measure of McCabe, the cyclomatic complexity [2] 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 cyclomatic complexity ignores the nesting level of the predicate nodes. Improvements as weighting the control structure with the nesting level were proposed by Harrison and Magel [10], by Piwowarski [17] and by Howatt and Baker [11]. The *scope* of a predicate node is a set of statements, whose execution depends on the decision made in the predicate node. The nesting level of a statement is defined as the number of predicate nodes whose scope contains the statement.

The nesting level notation reflects procedural programs in an adequate way. At the same time, it does not take into account data handling, which has central role in modern programming languages. We have to take the complexity of the data defined and the complexity of data handling into consideration. Accordingly, the *AV complexity* of a program is a sum of three components:

1. *Control structure of program.* Most programs have the same control statements irrespectively of the paradigm used. The control structure is represented by a graph where nodes are statements and (directed) edges represent the possible flow of control. Nodes with more than one output edge are called predicate nodes. Nesting level is used weighting statement nodes.
2. *Complexity of data types.* It reflects the complexity of data used (like in the case of classes). Data nodes are represented as different type of nodes in the control graph.
3. *Complexity of data access.* Connection between control structure and data is represented by edges between the data nodes and statements that use them. The direction of these edges reflect the data flow (for example in case of reading data the edge is directed from the data node to the statement). Data edges are nested by the nesting level of their statement node.

An important feature of our metric is that it does not count the complexity of data handling based on the place of the declaration. The metric encounters that value exactly at the point of data handling. Of course, the metric also measures the declaration in an implicite way: local variables are used only in the local code context (in the subprogram).

There is an other possible way to get these results. Let us suppose that we have no data nodes and data edges in our graph, but we replace them whit special control nodes: „reader" and/or „writer". These control nodes only send and receive information. They 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 0same as the one calculated with AV graphs.

We can naturally extend our model to object-oriented programs. The central notion of the object-oriented paradigm is the class. Therefore we describe how we measure the complexity of a class first. On the base of the previous sections we can see the class definition as a set of (local) data and a set of methods accessing them.

The complexity of a class is the sum of the complexity of the methods and the data members (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 AV 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.

Let us consider that the definition of the AV graph permits the empty 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 – there is no common data shared between them –, we compute the complexity of the class as the sum of the complexities 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 to procedural, object-oriented, or even mixed-style programs. This was our goal.

3 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is one of the most promising new software development techniques. AOP aids a better handling of *crosscutting concerns* [12], as compared to object-orientation. Thus AOP is a generative programming paradigm that aims to help in writing more modularized, and more maintainable code. Today's AOP implementations (among which the most widely-used is AspectJ), mostly rely on OOP. AspectJ essentially integrates tools for modularizing crosscutting concerns into object-oriented programs. AOP defines the following important constructs, aside the OOP notions:

1. *Pointcut definitions* are made up of Pointcut type, and Pointcut signature. The pointcut type describes *what* happens, e.g. `call` stands for function call, `execution` stands for the execution of a function. The signature describes *which* kind of functions are monitored by the pointcut definition. `(void || int f(*))` means all the functions with the name `f`, that have either a `void` or an `int` return type, and receive one parameter of any type.
2. *Pointcuts* are sets of pointcut definitions bound by Pointcut operators (`||`, `&&`). A *named pointcut* is a pointcut that can be referred to by a name, therefore it is not necessary to be defined repeatedly.
   `pointcut p() : call(void || int f(*)) || execution(* g());`
3. *Advice* constructs specify the action to be taken at a certain pointcut (bound to the advice). The `before`, `after` and `around` keywords define *when* the body part of the advice is executed with respect to the pointcut. Otherwise the body of an advice is very similar to the body of a Java method.
4. *Inter-type declarations* allow among others declaring aspect precedences, custom compilation errors or warnings.
5. *Aspects* contain pointcuts, advices, and inter-type declarations. On the other hand, they also have a class-like behavior, as they can have their own attributes and methods.

Nowadays AOP is widely used in both academic, and industrial world. Practice shows that AOP programs are in many cases shorter, have more modular structure and are easier to understand. Numerous publications discuss the advantages of AOP design and implementation. However, we still have not found appropriate metric tools to present quantitative results on the structural complexity of AOP programs.

One possible reason might be the lack of multiparadigm metrics that are valid on both object-oriented and generative paradigm. There are proposals to measure specific features of AOP programs [6, 9], but our approach is that in practice a more suitable metric has to be able to measure soundly in more paradigms at once. The complexity of an AOP program depends on the OOP components and the AOP-specific constructs. Therefore the complexity could be scattered between the AOP-specific parts (in pointcut-definitions, advices, etc.), the object-oriented constructs (classes, inheritance, etc.), and even in the procedural-style implementation of the methods. Hence in our opinion we need to apply a metric that measures well more paradigms at the same time.

Experiences show that AOP provides a better solution for a certain set of problems (e.g. logging, debugging, etc.). In this paper we investigate what is common in these problem groups that renders the AOP solution intuitively easier. Can we find problem sets in which AOP provides a better solution? Why do we see one solution easier understandable than the other if they are implemented using different paradigms? How can we prove that for those aforementioned AOP problems the solutions are not only intuitively but also objectively better? In order to answer these questions, we aim to analyze the Gang-of-Four (GoF) Design patterns [7] and their implementations in pure Java and an AOP version in AspectJ [14].

4 Extending the metric

Extending the metric requires the identification of AOP-specific program elements, and their mapping to an AV-graph. In section 3 we have enumerated the most important AspectJ constructions, now we examine how our multiparadigm metric applies to them. In order to measure programs, we also needed to extend the measurement tool.

1. *Pointcut definitions, and pointcuts.* Aspect-oriented programming is a kind of metaprogramming. With the help of pointcut definitions we describe notions to control the compilation and weaving process. A pointcut defines a condition which triggers the possible execution of a code defined in the appropriate advice. In that sense a pointcut definition is a metaprogram conditional statement. Therefore we map pointcut definitions to the AV-graph as predicate nodes, and its constitutes (the pointcut type and the signature as input nodes). As in the case of run-time programs, where a predicate node might use complex expressions, a pointcut definition can use pointcut operators to express complex conditions.
   We measure pointcut definitions by summing up the value (1 by default) assigned to the definition's type (`call`, `execution`, etc) and the complexity of the signature. The complexity of pointcuts is the sum of the definitions' complexities. The signature can be expressed as a regular expression, for which metrics already exist [18]. We have decided to add a constant 1 complexity to each token occuring in the expression. A token is a string literal (like: `foo`), a keyword (like: `int`), or a regular expression metacharacter (like: `*`). The rationale behind the definition is the following. It takes the same effort to understand that a signature applies to all functions (in the form: `* *(*)`) or to exactly one (`void foo(int) throws IOException`). However, more complex patterns cause decisions harder to understand, like in `void f*oo(int,*) throws *`.
2. *Named pointcuts.* The complexity of named pointcuts is the sum of the complexity of their names (1 by default) and the pointcut itself. Thus if the programmer defines a certain pointcut, names it, and instead of repeatedly defining it again refers to the pointcut by its name, the complexity of the code can be reduced. In section 2 we have seen, that the usage of functions decreases the complexity, because by making a function call, the added complexity is only the function's name, and its parameters. The usage of named pointcuts is analogous to that procedure.
3. *Advices*, from our metric's point of view are built up from two parts: the function part, and the pointcut part.
   – The method for measuring the pointcut part has already been described in the item 1.

  - The purely function part is as follows. An advice's header is like that of a special function's, with the keywords `before`, `after`, or `around` as the name, followed by the regular parameter list. The "name" might be preceeded by a return type. The body of an advice does not seem different for the programmer than the body of a function would. Even in an `around` advice, the keyword `proceed` does not seem different from an ordinary function call. Therefore the function part's complexity is measured the same way as Java methods.

   The pointcut decides when a certain advice's body part is executed. This is as if the body part of the advice would be in the *scope* of the predicates defined by the pointcut. Complex pointcut definitions behave like nested predicates. Thus the complexity of an advice is the complexity of advice's body multiplied by the complexity of its pointcut.
4. *Aspects* and classes have a lot in common from the complexity point of view. Both may include data, and member functions. Thus these members of aspects can be measured the same way as if they were in classes, for this the method is described in 2. Aspects may also have members of AOP-specific constructs. We have classified these constructs into two groups.
   - The complexity of *advices*, and *named pointcuts* is taken into consideration when measuring the aspect. These constructs directly affect the way the programmer sees the code. She needs to understand these members to be able to comprehend the complex construct described by the program.
   - As of now inter-type declarations like *declare parents*, *declare errors*, *declare warning*, and others are not taken into account when measuring complexity. We consider these auxiliary constructs in AspectJ which do not directly affect the complexity seen by the programmer, but rather as tools to easier express certain notions.

   These complexity values are summed up with the complexities of the data, and the member functions of the aspects.

We have seen in section 2 that the visibility of classes, and its members does not influence their complexity. For the same reasons we do not take this attribute into consideration in the case of aspects, and its members either.

5 Results

To validate our metric we have chosen the GoF Design Patterns' ([7]) implementations ([14]) for measuring. One of the reasons was that in [14] we find a functionally equivalent implementation of each pattern in AOP and OOP. At the same time, the renowned authors behind the implementations let us assume that the aspect-oriented techniques were handled correctly. We also supposed that DPs are neutral to crosscutting concerns. We did not choose examples that are well-known crosscutting problems (e.g. logging, tracing, etc), but more general ones, that *might* be in this problem set.

Many people think AOP reduces the complexity of the design patterns' implementations because of the patterns' crosscutting approaches. Obviously, the design patterns have been created as solutions for the non-trivial problems in OOP. The approach of AOP can describe these solutions easier by AOP's new language constructs.

The structure of these implementations is as follows. Each pattern has a Java and an AspectJ implementation. The AspectJ implementations also utilize a common library, otherwise independent of the patterns. As of now our measurement tool is able to parse and measure 14 out of 23 design pattern implementations. The table shows the AV-metric values, and the effective lines of code (ELOC) per patterns.

Design Pattern	Implementation	AV Complexity	Effective LOC
`adapter`	java	77	27
	AOP	51	22
`bridge`	java	235	75
	AOP	237	79
`builder`	java	219	55
	AOP	201	66
`decorator`	java	91	34
	AOP	93	25
`factoryMethod`	java	113	54
	AOP	129	67
`flyweight`	java	299	66
	AOP	286	71
`interpreter`	java	567	115
	AOP	567	113
`memento`	java	99	33
	AOP	186	47
`observer`	java	374	93
	AOP	305	87
`prototype`	java	187	53
	AOP	204	56
`state`	java	259	97
	AOP	179	103
`strategy`	java	265	56
	AOP	732	68
`templateMethod`	java	158	43
	AOP	158	45
`visitor`	java	300	83
	AOP	362	85

A comparison between the implementation of the design patterns in the OOP and AOP way can be found in [14, 16]. These papers explain that 17 out of 23 patterns had exhibited some degree of crosscutting. They also declare that implementing the patterns in AOP has many benefits, among them the most important being the ability to localize the code for a given pattern. Many patterns can be implemented as a single aspect, or as 2 closely related aspects. Our metric especially rewards code localization. The OOP versions can not be as well-structured as the AOP versions, where the code is more maintainable, and comprehensible. Another important benefit is the code's obliviousness. This benefit results directly from localization: as the pattern is localized in an aspect, it does not invade its participants. Henneman and Kiczales stated that the AOP versions are more modular by 74%, and more reuseable by 52%. According to [14] some patterns' implementation may disappear into the code because of AOP's constructs (e.g. the decorator pattern). This can also lead to complexity decrease.

In some cases we can see that the AOP implementation was less complex by our metric even if the ELOC number was greater. These are the cases when using AOP was adequate, these are the `adapter`, `builder`, `observer`, and `state` patterns. However, we can see a number of patterns where the AOP implementations were at least as complex as their Java counterparts. Patterns like `memento`, `visitor` and most typically `strategy` belong here. This shows that inadequate use of AOP can even be disadvantageous.

6 Conclusion and future work

In this paper we described a multiparadigm metric which was extended for aspect-oriented programs. The metric can measure the complexity of procedural, object-oriented, and aspect-related parts of programs implemented in AspectJ. We tested our metric on two functionally equal implementations of GoF design patterns: one of them is written in pure Java, the other is based on AspectJ. The metric revealed that aspect-orientation does not necessarily reduce the complexity on its own – the gain highly depends on the actual problem. Future investigations are needed to clarify the details.

References

1. Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism, ACM Computing Surveys 17(4), pp. 471-522, 1985
2. McCabe, T.J., A Complexity Measure, IEEE Trans. Software Engineering, SE-2(4), pp. 308-320, 1976
3. Chidamber S.R., Kemerer, C.F., A metrics suit for object oriented design, IEEE Trans. Software Engeneering, vol.20, pp.476-498, (1994).
4. Coplien, J.O.: Multi-Paradigm Design for C++, Addison-Wesley, 1998
5. Czarnezki K., Eisenecker, U.W.: Generative Programming, Addison-Wesley, 2000

6. Figueiredo, E., Garcia, A., Sant' Anna, C., Kulesza, U., Lucena, C.: Assessing Aspect-Oriented Artifactsd: Towards a Tool-Supported Quantitative Method, QAOOSE Workshop, ECOOP, Glasgow, pp. 58-69, 2005
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object Oriented Software, Addison-Wesley, 1995
8. Gradecki, J.D., Lesiecki, N.: Mastering AspectJ, Wiley, 2003
9. Jean-Yves Guyomarc'h, Yann-Gael Guéhéneuc: On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics, QAOOSE Workshop, ECOOP, Glasgow, pp. 42-47, 2005
10. Harrison, W.A., Magel, K.I., A Complexity Measure Based on Nesting Level, ACM Sigplan Notices,16(3), pp.6
11. Howatt, J.W., Baker, A.L.: Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting, The Journal of Systems and Software 10, pp.139-150, 1989
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, Jean-Marc, Irwin, J.: Aspect-Oriented Programming, ECOOP, Finland, Springer-Verlang LNCS vol. 1241, pp. 220-242, 1997
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ, LNCS vol. 2072, pp. 327-355, 2001
14. Kiczales G., Henneman, J.: Design Pattern Implementation in Java and AspectJ, OOPSLA, pp. 161-173, 2002
15. Kiczales, G.: Aspect-Oriented Programming, AOP Computing surveys 28(es), 154-p, 1996
16. Lesiecki, N.: Enhance design patterns with AspectJ, IBM `http://www.developers.net/external/730`
17. Piwowarski, R.E.: A Nesting Level Complexity Measure, ACM Sigplan Notices, 17(9), pp.44-50, 1982
18. James F. Power, Brian A. Malloy: A metrics suite for grammar-based software. Journal of Software Maintenance 16(6): 405-426 (2004)
19. Porkoláb, Z., Sillye, Á.: Towards a multiparadigm complexity measure, QAOOSE Workshop, ECOOP, Glasgow, pp.134-142, 2005
20. Schmidmeier, A., Hanenberg S., Unland, R.: Implementing Known Concepts in AspectJ, 2003
21. Grégory Seront, Miguel Lopez, Valérie Paulus, Naji Habra: On the Relationship between Cyclomatic Complexity and the Degree of Object Orientation, QAOOSE Workshop, ECOOP, Glasgow, pp. 109-117
22. Wadler, P.: The expression problem, Posted on the Java Genericity mailing list, 1998
23. Weyuker, E.J.: Evaluating software complexity measures, IEEE Trans. Software Engineering, vol.14, pp.1357-1365, 1988
24. Java 1.5, `http://java.sun.com/developer/technicalArticles/releases/j2se15`
25. Eclipse.org formation, `http://www.eclipse.org/org/index.html`