

# On Multiparadigm Software Complexity Metrics

Ádám Sipos, Norbert Pataki, Zoltán Porkoláb  
Dept. of Programming Languages and Compilers,  
Faculty of Informatics, Eötvös Loránd University  
{shp,patakino,gsd}@elte.hu

## Abstract

Structural complexity metrics play important role in modern software engineering. Based on software metrics we can identify critical parts of software, give recommendations, and define coding conventions for the development of sound and manageable code. With the emerging of the object-oriented paradigm, research efforts focused on metrics based on special object-oriented features. However, in modern software construction a multiparadigm design is frequently used. Since metrics might report false results when applied to different paradigms than the one they were designed for, there is an urgent need for paradigm-neutral metrics. In this article a rigorous definition is given for a multiparadigm metric. We discuss the metric on procedural and object-oriented paradigms, and evaluate it in more details applied on aspect-oriented programming.

## 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 development 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 [3]. 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 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) [4], *generic programming* (GP), and *aspect-oriented programming* (AOP) [13].

Software metrics have always been strongly related to the paradigm used in the respective period. The McCabe *Cyclomatic complexity* number [15] was designed for measuring the testing efforts of non-structural FORTRAN programs. Piwowarski [16], Howatt and Baker [10] 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. [2]. Several implementations of such metrics are available for the most popular languages (like Java, C#,

C++) and platforms (like Eclipse).

Most modern 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* [3] 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 construction 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.

The rest of this paper is organized as follows: In Section 2 we give a rigorous definition of our multiparadigm metric called *AV-graph*. A detailed analysis of the metric is given in Section 3. Aspect-oriented programming, a new emerging paradigm is discussed in Section 4. We show the application of our metric in details on aspect-oriented paradigm in Section 5. Empirical results presented in Section 6.

## 2 A multiparadigm metric

Since McCabe's Cyclomatic complexity number it is a common idea for software metrics to map the programs to directed graphs ( $\mathbb{G}$ ). Unfortunately, most metrics are paradigm-dependent (i.e. realize a  $\mathbb{S}_{P_i} \rightarrow \mathbb{G}$  mapping, where  $P_i$  is a currently used paradigm and  $\mathbb{S}$  is the space of programs written in the current paradigm( $P_i$ )). Contrarily, our metrics does not depend on the used paradigm (i.e. our metric realizes a  $\mathbb{S}_{P_1 \tilde{\cup} P_2 \tilde{\cup} \dots \tilde{\cup} P_n} \rightarrow \mathbb{G}$  mapping. Here the  $\tilde{\cup}$  means the operation of combinations of the paradigms.) At the case of these graph-based metrics the graphs must be evaluated to maps the graphs to numbers (these functions realize a  $\mathbb{G} \rightarrow \mathbb{N}$  mapping).

We define our metrics on the basis of the complexity of nested control structures. The definitions follow the „rigorous” description of J. Howatt and A. Baker [10], but extend their formalism to the data components of the program.

**Definition 2.1.** An *AV graph*  $\overline{G} = (\overline{N}, \overline{E})$  is an ordered pair of nodes  $\overline{N}$

and edges  $\overline{E}$ . Nodes  $\overline{N} = N \cup D$  consist of the union of *control nodes* ( $N$ ) and *data nodes* ( $D$ ). Edges  $\overline{E} = E \cup R$  consist of the union of *control edges* ( $E$ ) and *data reference edges* ( $R$ ), where  $E \subseteq (N \times N)$  and  $R \subseteq (N \times D) \cup (D \times N)$ .

An edge is an ordered pair of nodes  $(x, y)$ . If  $(x, y)$  is an edge then node  $x$  is an *immediate predecessor* of node  $y$  and  $y$  is an *immediate successor* of node  $x$ . The set of all immediate predecessors of a node  $y$  is denoted  $\mathbf{IP}(\mathbf{y})$  and the set of all immediate successors of a node  $x$  is denoted  $\mathbf{IS}(\mathbf{x})$ . A node has *indegree*  $n$  if  $\overline{E}$  contains exactly  $n$  edges of the form  $(w, z)$ , similarly a node has *outdegree*  $m$  if  $\overline{E}$  contains exactly  $m$  edges of the form  $(z, w)$ .

A *control path*  $P$  in an AV graph  $\overline{G} = (\overline{N}, \overline{E})$  is a sequence of control edges  $(x_1, x_2), (x_2, x_3), \dots, (x_{k-2}, x_{k-1}), (x_{k-1}, x_k)$ , where  $\forall i[1 \leq i < k] \Rightarrow (x_i, x_{i+1}) \in E \subseteq \overline{E}$ . In this case  $P$  is a *control path* from  $x_1$  to  $x_k$ .

**Definition 2.2.** A *flowgraph*  $G = (N, E, s, t)$  in  $\overline{G} = (\overline{N}, \overline{E})$  is a directed graph with a finite, nonempty set of nodes  $N \subseteq \overline{N}$ , a finite, nonempty set of edges  $E \subseteq \overline{E}$ ,  $s \in N$  is the start node,  $t \in N$  is the terminal node. For any flowgraph  $G$ , the  $s$  start node is the unique node with indegree zero; the  $t$  terminal node is the unique node with outdegree zero, and each node  $x \in N$  lies on some path in  $G$  from  $s$  to  $t$ . Let  $N'$  denote the set  $N \setminus \{s, t\}$ .

A flowgraph reflects the control structure of a program. A program may contain zero or more flowgraphs. The former happens in the case of pure data structures, like a *record* which is a set of data nodes without control structures. In object-oriented languages, a *class* denotes a set of data nodes (i.e. the *attributes*) and a number of flowgraphs (i.e. the *methods*).

A *basic block* is a sequential block of code with maximal length, where a sequential block of code in a  $G$  flowgraph is a sequence of tokens in  $G$  that is executed starting only with the first token in the sequence, all the tokens in the sequence are always executed sequentially, and the sequence is always exited at the end. Hence, basic blocks do not contain any loops or if statements.

In the following definitions we determine the nesting level of the control and data nodes. The rather complex indirect definitions are required because of non-structured programs.

**Definition 2.3.** Every node  $n \in N$  of a flowgraph  $G = (N, E, s, t)$  which has outdegree greater than one is a *predicate node* ( $|IS(n)| > 1$ ). Let  $Q$  denote the set of predicate nodes in  $G$  ( $Q = \{n \in N \mid |IS(n)| > 1\}$ ).

Given a flowgraph  $G = (N, E, s, t)$ , and  $p, q \in N$ , node  $p$  *dominates* node  $q \in G$  if  $p$  lies on every path from  $s$  to  $q$ . Node  $p$  *properly dominates* node  $q \in G$  if  $p$  dominates  $q$  and  $p \neq q$ . Let  $r \in N$ , node  $p$  is the *immediate dominator* of node  $q$  if (i)  $p$  properly dominates  $q$  and (ii) if  $r$  properly dominates  $q$  then  $r$  dominates  $p$ .

Given a flowgraph  $G = (N, E, s, t)$ , and  $p, q \in N$ , the set of *first occurrence*

paths from  $p$  to  $q$ ,  $FOP(p, q)$  is the set of all paths from  $p$  to  $q$  such that node  $q$  occurs exactly once on each path.

**Definition 2.4.** Given a flowgraph  $G = (N, E, s, t)$ , and nodes  $p, q \in N$ , the set of nodes that are on any path in  $FOP(p, q)$  is denoted by  $MP(p, q)$ :  $MP(p, q) = \{v \mid \exists P [ P \in FOP(p, q) \wedge v \in P ] \}$

The set of *lower bounds* of a predicate node  $p \in N$  is  $LB(p) = \{v \mid \forall r \forall P [ r \in IS(p) \wedge P \in FOP(r, t) \Rightarrow v \in P ]\}$ , and the *greatest lower bound* of  $p$  in  $G$  is  $GLB(p) = \{q \mid q \in LB(p) \wedge \forall r [ r \in (LB(p) \setminus \{q\}) \Rightarrow r \in LB(q) ]\}$

**Definition 2.5.** Given a flowgraph  $G = (N, E, s, t)$ , and a predicate node  $p \in N$ , the set of nodes predicated by node  $p$  is  $Scope(p) = \{n \mid \exists q [ q \in IS(p) \wedge n \in MP(q, GLB(p)) ] \} \setminus \{ GLB(p) \}$  and the set of nodes that *predicate* a node  $x \in N$ , is  $Pred(x) = \{p \mid x \in Scope(p)\}$ .

To extend the notion of *Scope* and *Pred* to the data nodes we extend *Scope* with the data nodes connected directly to the flowgraph via control nodes in  $Scope(p)$ :  $\overline{Scope(p)} = Scope(p) \cup \{d \mid \exists n \in Scope(p) (\exists(d, e) \in R \vee \exists(e, d) \in R)\}$  and  $\overline{Pred(x)} = \{p \mid x \in Scope(p)\}$ .

**Definition 2.6.** The *nesting depth* of a node  $x \in \overline{N}$ , in a  $\overline{G} = (\overline{N}, \overline{E})$  is

$$nd(x) = | \overline{Pred(x)} |$$

Thus, the total nesting depth of an AV graph  $\overline{G}$  is counted as

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

The measure of program complexity given by Harrison and Magel is the sum of the adjusted complexity values of the control nodes. This value can be given (as proved by Howatt) as the scope number of a flowgraph. We extended this notion to the data structure of the program reflecting the behaviour of non-procedural paradigms too.

**Definition 2.7.** The *scope number*,  $SN(\overline{G})$  of an AV graph  $\overline{G} = (\overline{N}, \overline{E})$  is

$$SN(\overline{G}) = | \overline{N} | + ND(\overline{G})$$

### 3 Evaluation of the metric

The main concept behind the definition of AV graph is that the complexity of a certain code element – either data or control – is heavily depends it's environment. The execution of a control node and the possible value of a data node depends on the predicates dominating it. Thus understanding a node depends on its nesting depth.

There is an other possible way to get these results. We can map our AV graph model with control and data nodes to the Howatt's model without data nodes and data edges. Hence we replace data edges with 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 same as of the AV graph complexity.

This definition reflects our experience properly. For example, if we take a component out of the graph which does *not* contain a predicate node to form a procedure, (ie. a basic block, or a part of it – this means a single node), then we increase the complexity of the whole program according to our definition. This is a direct consequence of the fact that in our measures so far we contracted the statement-sequences that are reasonable according to this view of complexity. If we create procedures from sequences, the program becomes slightly difficult to follow. Since we can not read the program linearly, we have to ”jump” from the procedures back and forth. The reason for this is that a sequence of statements can always be viewed as a single transformation. This could of course be refined by counting the different transformations as being of different weight, but this approach would transgress the competence of the model used. The model mirrors these considerations since if we form a procedure from a subgraph containing no predicate nodes, then the complexity increases according to the complexity of the new procedure subgraph, (i.e. by 1).

On the other hand, if the procedure does contain predicate node(s), then by modularization we decrease the complexity of the whole program depending on the nesting level of the outlifted procedure. If we take a procedure out of the flowgraph, creating a new subgraph out of it, the measure of its complexity becomes independent of its nesting level. On the place of the call we may consider it as an elementary statement (as a basic block, or part of it).

As a matter of fact, we can decrease the complexity of a program in connection with data if we build abstract data types *hiding the representation*. In this case the references to data elements will be replaced by control nodes since data can only be handled through its operations. While computing the complexity of the whole program, we have to take into account not only the decreasing of the complexity, but also its increase by the added complexity determined by the implementation of the abstract data type. Nevertheless, this will only be an additive factor instead of the previous multiplicative factor.

That is the most important complexity-decreasing consequence of the object-oriented view of programming: *the class hides its representation* (both

data structure and algorithm) from the predicates (decisions) supervising the use of the object of class. We can naturally apply our model to object-oriented programs [17]. 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.

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 does not differ from the effect of the most simple (built-in) types. same as the one calculated with AV graphs.

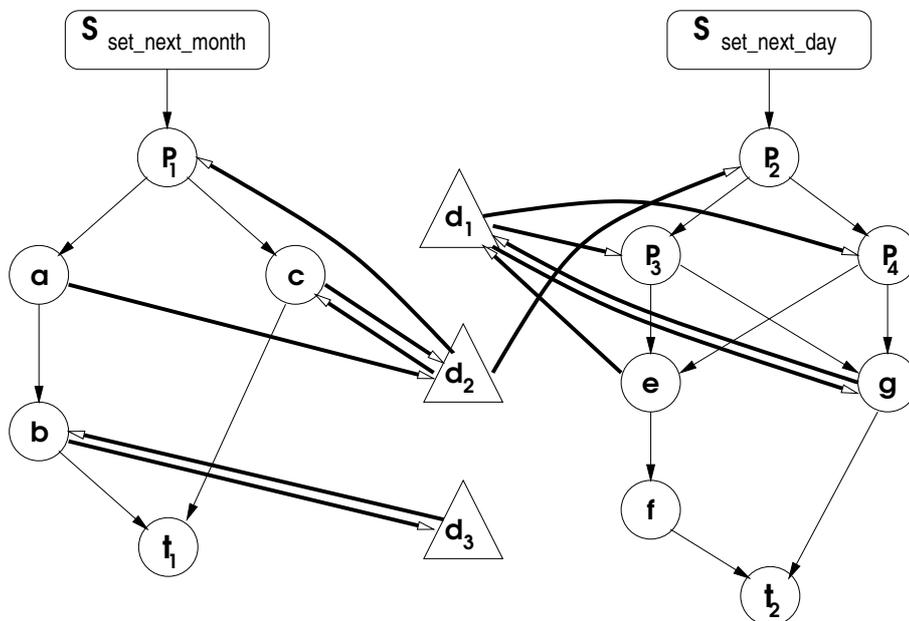


Figure 1: AV graph of a class with two methods and three attributes

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 ordinary 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, aspect-oriented or even mixed-style programs.

## 4 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* [13], 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 [11]. 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 [5, 8], 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 [6] and their implementations in pure Java and an AOP version in AspectJ [12].

## 5 Extending the metric

Extending the metric requires the identification of AOP-specific program elements, and their mapping to an AV-graph. In Section 4 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 [19]. We have decided to add a constant 1 complexity to each token occurring 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 preceded 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.

## 6 Results

To validate our metric we have chosen the GoF Design Patterns' ([6]) implementations ([12]) for measuring. One of the reasons was that in [12] 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 constructions.

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 test results encountering the AV-metric and the effective lines of code (ELOC) per patterns.

A comparison between the implementation of the design patterns in the OOP and AOP way can be found in [12, 14]. 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. 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%. As a result of these benefits allow the code-level reuse of some patterns. According to [12] some patterns' implementation may disappear into the code because of the AOP's constructions (e.g. the

decorator pattern). This can also lead to complexity decrease.

| 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            |

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. Here we can encounter 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` are

belong here. This shows that inadequate use of AOP can even be disadvantageous.

## 7 Conclusion and future work

In this paper a rigorous definition of a multiparadigm metric was given. The metric is equally able to measure the complexity of procedural, object-oriented, and aspect-related parts of multiparadigm programs. We implemented and tested our metric on two functionally equal implementations of GoF Design patterns: one of them is a pure Java, the other is based on AspectJ. The metric proved to be useful in such multiparadigm language environment. Empirical results also revealed that aspect orientation is not necessarily reduces the complexity of its own – the gain highly depends on the actual problem. Future investigations are necessary to clarify how complexity of aspect-oriented programs depends on the internal structure of the code.

## References

- [1] Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys* 17(4), pp. 471-522, 1985
- [2] Chidamber S.R., Kemerer, C.F., A metrics suit for object oriented design, *IEEE Trans. Software Engineering*, vol.20, pp.476-498, (1994).
- [3] Coplien, J.O.: *Multi-Paradigm Design for C++*, Addison-Wesley, 1998
- [4] Czarnecki K., Eisenecker, U.W.: *Generative Programming*, Addison-Wesley, 2000
- [5] Figueiredo, E., Garcia, A., Sant' Anna, C., Kulesza, U., Lucena, C.: Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method, *QAOOSE Workshop, ECOOP, Glasgow*, pp. 58-69, 2005
- [6] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns – Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995
- [7] Gradecki, J.D., Lesiecki, N.: *Mastering AspectJ*, Wiley, 2003
- [8] Guyomarc'h, J-Y., Guéhéneuc, Y-G.: On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics, *QAOOSE Workshop, ECOOP, Glasgow*, pp. 42-47, 2005

- [9] Harrison, W.A., Magel, K.I., A Complexity Measure Based on Nesting Level, ACM Sigplan Notices,16(3), pp.6
- [10] 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
- [11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ, LNCS vol. 2072, pp. 327-355, 2001
- [12] Kiczales G., Henneman, J.: Design Pattern Implementation in Java and AspectJ, OOPSLA, pp. 161-173, 2002
- [13] Kiczales, G.: Aspect-Oriented Programming, AOP Computing surveys 28(es), 154-p, 1996
- [14] Lesiecki, N.: Enhance design patterns with AspectJ, IBM <http://www.developers.net/external/730>
- [15] McCabe, T.J., A Complexity Measure, IEEE Trans. Software Engineering, SE-2(4), pp. 308-320, 1976
- [16] Piwowarski, R.E.: A Nesting Level Complexity Measure, ACM Sigplan Notices, 17(9), pp.44-50, 1982
- [17] Porkoláb, Z., Sillye, Á.: Comparison of Object-Oriented and Paradigm Independent Software Complexity Metrics, ICAI'04, Eger, 2004
- [18] Porkoláb, Z., Sillye, Á.: Towards a multiparadigm complexity measure, QAOOSE Workshop, ECOOP, Glasgow, pp.134-142, 2005
- [19] Power, J. F., Malloy, B. A.: A metrics suite for grammar-based software. Journal of Software Maintenance 16(6): 405-426 (2004)
- [20] Schmidmeier, A., Hanenberg S., Unland, R.: Implementing Known Concepts in AspectJ, 2003
- [21] Seront, G., Lopez, M., Paulus, V., Habra, N.: 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