Impact of Aspect Oriented Programming on Software Development Design Quality Metrics-A Comparative Study

Kotrappa Sirbi, Prakash Jayanth Kulkarni

Abstract—The aspect-oriented programming approach is supposed to enhance a system’s features, such as its modularity, readability and simplicity. Due to a better modularization of crosscutting concerns, the developed system implementation would be less complex and more readable. Thus software development efficiency would increase, so the system would be created faster than its object-oriented equivalent. The study reveals that the aspect-oriented programming approach appears to be a full-fledged alternative to the pure object-oriented approach. In this comparative study, we present existing and new Software metrics useful in the design phase of the Object Oriented Software Development (OOSD) and also which have extension in SDLC of Aspect Oriented Software Development (AOSD). In this paper, we will compare definitions for proposed metrics for AOSD design quality metrics. In this we will show all those metrics which are useful for improving the software design quality metrics using Aspect Oriented Software Development (AOSD).

Index Terms—Aspect Oriented Software Development (AOSD), Object Oriented Software Development (OOSD), OO metrics, AO metrics.

1 INTRODUCTION

In order to study the impact of aspect-oriented software development (AOSD) on evolution one has to study its impact on software characteristics such as evolvability, maintainability, understandability, and quality. To evaluate these characteristics not only qualitative but also quantitative techniques are of interest.

The key question is how we can quantify when applying AOSD is beneficial. Metrics are an important technique in quantifying software and software development characteristics. However, metrics have to be used knowledgeably and carefully. Theoretical and empirical validation of metrics and of their relation to software attributes is a cumbersome and long process. It is of paramount importance that we validate the utility of metrics we use in order to enable others to use them, too. Until now, the metrics used and proposed for AOSD are rarely validated. It is not sufficient to prove their definitions correct but also their usefulness to describe software characteristics has to be validated. In most cases, this can only be achieved through controlled experiments or through analysing large amounts of data, e.g., from case studies. In both cases, statistical evaluations are a key technique to examine hypotheses.

Aspect-orientation is one of the emerging paradigms that promise to enhance software design and promote reuse. It builds on Object-Orientation, and addresses some of the points that were not addressed by OO. Software metrics are the means by which software designs can be evaluated. In [1], Brito argues that “keeping on the evolution track means we must be able to quantify our improvements”. Some researchers argue that a quantitative analysis of software is indispensable for evaluating systems; others believe that software designs are not all quantitative. For example, Lord Kelvin states “when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, your knowledge is of a meager and unsatisfactory kind” [2]. On the other hand, Meyer argues that this saying is exaggerated [3]. Numbers in themselves do not have meanings unless they have an underlying rationale. The set of things that can be measured is infinite, and most of them are not interesting; for that reason, Meyer argues that any metrics applied to a system should be justified of what property the metric is intended to help estimate [3].

In section 2, we give a brief overview of aspect-orientation and AspectJ, explaining what aspects are and the objectives of aspect-oriented design and also metrics background theory pertaining to design phase. Section 3 discusses the need for metrics for object oriented and section 4 gave about definitions of aspect-oriented metrics. Section 5 lists the AOSD Design quality metrics and Chidamber and Kemerer (C&K) metrics suite. Section 6 explains AJHotDraw Graphical framework 2D drawing case.
study. We conclude by calling for more empirical research in relation with metrics in AOSD.

2 MOTIVATION & BACKGROUND

2.1 Aspect-Orientation & Software Metrics
Software metrics for aspect-orientation is needed to ensure that AO really accomplishes its objective of enhancing software reuse and providing better software designs. One other reason why we need AO metrics is to determine the correct design practices when designing AO systems. Just as one can write poor object-oriented designs, aspect-oriented systems can also be poorly designed. Even worse, a well designed object-oriented system might be made worse with the introduction of aspects, as aspects might increase the system complexity and reduce its understandability. Inaccurate designs that do not show the correct aspect-precedence, for example, might cause designs that do not even meet the system’s requirements. For these reasons, design styles for aspects are required, as well as metrics for measuring how efficient, understandable, and reusable an aspect-oriented design is.

2.2 AspectJ
AspectJ [13] is simple general-purpose extension to Java that provides, through the definition of new constructors, support for modular implementation of crosscutting concerns. It enables plug and play implementations of crosscutting concerns. AspectJ has been successfully used for modularizing the crosscutting concerns such as synchronization, consistency checking, protocol management and others. AspectJ supports the definition of aspects’ pointcuts, advice and introduction.

• **Join points**: Join points represent well-defined points in a program’s execution. Typical join points in AspectJ include method calls, access to class members, and the execution of exception handler blocks. Join points may contain other join points. For example, one method call may result in several other method calls before it returns.

• **Pointcuts**: Pointcut is a language construct that picks out a set of join points based on defined criteria. The criteria can be explicit function names, or function names specified by wildcards.

• **Advice**: Advice is code that executes before, after, or around a join point. You define advice relative to a pointcut, saying something like "run this code before every method call I want to log."

• **Introduction**: introduction allows aspects to modify the static structure of a program. Using introduction, aspects can add new methods and variables to a class, declare that a class implements an interface, or convert checked to unchecked exceptions.

2.3 Background
This section presents software metrics which can be used during the design phase: McCabe’s Cyclomatic number, Information Flow (fan in/out), Function Points and the Bang metric by Tom DeMarco. Design is in focus during the end of elaboration and the beginning of construction iterations. It contributes to a sound and stable architecture and creates a blueprint for the implementation model. During the construction phase, when the architecture is stable and requirements are well understood, the focus shifts to implementation.

2.3.1 Cyclomatic Complexity
The software metric known as Cyclomatic Complexity was proposed by McCabe [19] [20]. McCabe suggests seeing the program as a graph, and then finding out the number of different paths through it. One example of this graph is called a control graph. When programs become very large in length and complexity, the number of paths cannot be counted in a short period of time. Therefore, McCabe suggests counting the number of basic paths (all paths composed of basic paths). This is known as the “Cyclomatic Number”; which is the number of basic paths from one point on the graph to another, and we refer to the Cyclomatic Number by v(G). This is shown in equation 1,

\[ v(G) = E - N + 2P \]

Eq 1: Cyclomatic Complexity

Where

\[ V(G) = \text{Cyclomatic Complexity} \]
\[ E = \text{Number of edges} \]
\[ N = \text{Number of nodes} \]
\[ P = \text{Number of connected components or parts} \]

We do not have to construct a program control graph to compute \( v(G) \) for programs containing only binary decision nodes. We can just count the number of predicates (binary nodes), and add one to this, as shown in equation 2.

\[ v(G) = P + 1 \]

Eq 2 Cyclomatic Complexity for Binary Decision Nodes

Where:

\[ v(G) = \text{Cyclomatic Complexity} \]
\[ p = \text{number of binary nodes or predicates} \]

Cyclomatic complexity can help in the following situations:

• Identify software that may need inspection or redesign. We need to redesign all modules with \( v(G) > 10 \).

• Allocate resources for evaluation and test: Test all modules with \( v(G) > 5 \) first.

• Define test cases (basis path test case design): Define one test case for each independent path. Industry experience shows that Cyclomatic Complexity is more helpful for defining test cases (c), but doesn’t help that much on identifying software that may need inspection or redesign (a) and allocating resources for evaluation and test (b) as McCabe suggests. Studies have also found that the ideal \( v(G) = 7 \), which proves to be easier to understand for novices and experts.

2.3.2 Functions Points (FP)
This metric was developed by Albrecht [20] [21] [22]. It focuses on measuring the functionality of the software product according to the following parameters: user inputs, user outputs, user inquiries number of files and the number of external interfaces. Once the parameters are counted, a complexity (simple, average and complex) value is associated to each parameter. A complexity adjustment value is added to the previous count [23]. This value is obtained from the response to 14 questions related to reliability of the software product. Equation 3 presents the estimation of Function Points.

\[ FP = \text{count total} \times [0.65 + 0.01 \sum Fi] \]

**Eq 3: Function Points**

Where:
- \( FP \) = Total number of adjusted function points
- \( \sum Fi \) = The sum of all user inputs, Outputs, inquiries, files and external interfaces to which have been applied the weighting factor.
- \( Fi \) = A complexity adjustment value from the response to the 14 reliability questions.

The FP metric is difficult to use because one must identify all of the parameters of the software product. Somehow, this is subjective, and different organizations could interpret the definitions differently. Moreover, interpretations could be different from one project to another in the same organization, and this could be different from one software release to another.

### 2.3.3 Information Flow

Information Flow is a metric to measure the complexity of a software module. This metric was first proposed by Kafura and Henry [23]. The technique suggests identifying the number of calls to a module (i.e. the flows of local information entering: fan-in) and identifying the number of calls from a module (i.e. the flows of local information leaving: fan-out). The complexity in Information Flow metric is determined by equation 4.

\[ c = \text{[procedure length]}^2 \times \text{[fan- in \times fan- out]} \]

**Equation: 4 Information Flow**

Where:
- \( c \) = Complexity of the module
- \( \text{[procedure length]} \) = A length of the module, this value could be given in LOC or by using \( v(G) \) = Cyclomatic complexity
- \( \text{[fan-in]} \) = The number of calls to the module
- \( \text{[fan-out]} \) = The number of calls from the module

### 2.3.4 The Bang Metric

The Bang metric was first described by DeMarco [20] [25]. This metric can attempt to measure the size of the project based on the functionality of the system detected during the time of design. The diagrams generated during the design give the functional entities to count. The design diagrams to consider for the Bang metric are: data dictionary, entity relationship, data flow and state transition. Other design diagrams such as the Business Model, Use Cases, Sequence and Collaboration diagrams of the Unified Modeling Language (UML) can also be used to find the functional entities. The Bang metric can also be combined with other measures to estimate the cost of the project. To estimate the Bang metric, we classify the project in three major categories: function strong, data strong, or hybrid. We first divide the number of relationships between the numbers of functional primitives. If the ratio is less than 0.7 then the system is function strong. If the ratio is greater than 1.5 then the system is data strong. Otherwise, the system is hybrid. For Function strong systems, we first need to identify each primitive from the design diagrams. Then for each primitive, we identify how many tokens each primitive contains. We assign a corrected Functional Primitive Increment (CFPI) value according to the number of data tokens [20] [25]. At the same time, the primitive will be identified with one class, out of 16 classes. Each primitive will also be assigned a value for the increment. Afterward, the CFPI is multiplied by the increment and the resulting value is assigned to the Bang value. This process is repeated for each primitive and the values added all together to compute the final Bang for Function strong systems. For Hybrid and Data strong systems, we first identify the number of objects. Then, for each object, we count the number of relationships and then record the Corrected Object Increment (COBI) accordingly. The resulting value will be added to the values of each object and the total will be assigned to the final Bang computation for Hybrid and Data strong systems.

The benefit of AOSD is often studied in two ways. One is the comparison with an OO software, which already exists or which sometimes is developed in parallel. The other one is the independent evaluation of AOSD, often by assessing evolution scenarios. In both cases, the evaluation can employ design and program metrics to support and to quantify their observations. It has already noted by Chidamber and Kemerer [15]. We argue that a shift similar to the one leading to the Chidamber and Kemerer’s metrics is necessary when moving from OO to AOP software. Some notions used in the Chidamber and Kemerer’s suite
can be easily adapted to AOP software, by unifying classes and aspects, as well as methods and advices. Aspect introductions and static crosscutting require minor adaptations. However, novel kinds of coupling are introduced by AOP, demanding for specific measurements. For example, the possibility that a method execution is intercepted by an aspect pointcut, triggering the execution of an advice, makes the intercepted method coupled with the advice, in that its behavior is possibly altered by the advice. In the reverse direction, the aspect is affecting the module containing the intercepted operation, thus it depends on its internal properties (method names, control flow, etc.) in order to successfully redirect the operation’s execution and produce the desired effects. In the following section, the Chidamber and Kemerer’s metrics suite is revised. Some of the metrics are adapted or extended, in order to make them applicable to the AOP software.

4 AO METRICS

Since the proposed metrics apply both to classes and aspects, in the following the term module will be used to indicate either of the two modularization units. Similarly, the term operation subsumes class methods and aspect advices/introductions.

- **WOM (Weighted Operations in Module): Number of operations in a given module.** Similarly to the related OO metric, WOM captures the internal complexity of a module in terms of the number of implemented functions. A more refined version of this metric can be obtained by giving different weights to operations with different internal complexity.

- **DIT (Depth of Inheritance Tree):** Length of the longest path from a given module to the class/aspect hierarchy root. Similarly to the related OO metric, DIT measures the scope of the properties. The deeper a class/aspect is in the hierarchy, the greater the number of operations it might inherit, thus making it more complex to understand and change. Since aspects can alter the inheritance relationship by means of static crosscutting, such effects of aspectization must be taken into account when computing this metric.

- **NOC (Number Of Children):** Number of immediate subclasses or sub-aspects of a given module. Similarly to DIT, NOC measures the scope of the properties, but in the reverse direction with respect to DIT. The number of children of a module indicates the proportion of modules potentially dependent on properties inherited from the given one.

- **CAE (Coupling on Advice Execution):** Number of aspects containing advices possibly triggered by the execution of operations in a given module. If the behavior of an operation can be altered by an aspect advice, due to a pointcut intercepting it, there is an (implicit) dependence of the operation from the advice. Thus, the given module is coupled with the aspect containing the advice and a change of the latter might impact the former. Such kind of coupling is absent in OO systems.

- **CIM (Coupling on Intercepted Modules):** Number of modules or interfaces explicitly named in the pointcuts belonging to a given aspect. This metric is the dual of CAE, being focused on the aspect that intercepts the operations of another module. However, CIM takes into account only those modules and interfaces an aspect is aware of – those that are explicitly mentioned in the pointcuts. Sub-modules, modules implementing named interfaces or modules referenced through wildcards are not counted in this metric, while they are in the metric CDA (see below), the rationale being that CIM (differently from CDA) captures the direct knowledge an aspect has of the rest of the system. High values of CIM indicate high coupling of the aspect with the given application and low generalizability/reusability.

- **CMC (Coupling on Method Call):** Number of modules or interfaces declaring methods that are possibly called by a given module. This metric descends from the OO metric CBO (Coupling Between Objects), which was split into two (CMC and CFA) to distinguish coupling on operations from coupling on attributes. Aspect introductions must be taken into account when the possibly invoked methods are determined. Usage of a high number of methods from many different modules indicates that the function of the given module cannot be easily isolated from the others. High coupling is associated with a high dependence from the functions in other modules.

- **CFA (Coupling on Field Access):** Number of modules or interfaces declaring fields that are accessed by a given module. Similarly to CMC, CFA measures the dependences of a given module on other modules, but in terms of accessed fields, instead of methods. In OO systems this metric is usually close to zero, but in AOP, aspects might access class fields to perform their function, so observing the new value in aspectized software may be important to assess the coupling of an aspect with other classes/aspects.

- **RFM (Response For a Module):** Methods and advices potentially executed in response to a message received by a given module. Similarly to the related OO metric, RFM measures the potential communication between the given module and the other ones. The main adaptation necessary to apply it to AOP software is associated with the implicit responses that are triggered whenever a pointcut intercepts an operation of the given module.

- **LCO (Lack of Cohesion in Operations):** Pairs of operations working on different class fields minus pairs of operations working on common fields (zero if negative). Similarly to the LCOM (Lack of Cohesion in Methods) OO metric, LCO is associated with the pairwise dissimilarity between different operations belonging to the same module. Operations working on separate subsets of the module fields are considered dissimilar and contribute to the increase of the metric’s value. LCO will be low if all operations in a class or an aspect share a common data structure being manipulated or accessed.

- **CDA (Crosscutting Degree of an Aspect):** Number of modules affected by the pointcuts and by the introduc-
5 AOSD DESIGN QUALITY METRICS

Aspect-oriented software development introduces new abstractions to software engineering. For that reason, existing object metrics cannot be used directly. Presently, only few papers address aspect-oriented programs quality have been published in the literature. Zhao and Xu’s approach [6, 7] is the first proposal in the field of aspect cohesion and coupling measurement. It is based on a dependency model for aspect-oriented software that consists of a group of dependency graphs. Ceccato and Tonella [8] proposed an aspect revision of Chidamber and Kemerer’s metrics suite [15]. Sant’Anna et al. also proposed a metric suite for AOP in [9]. Zhang and Jacobsen [10] used the cyclomatic complexity number, size, weight of class, coupling between objects and response time for their evaluation. Clarke and Baniassad [11] applied the C&K metrics suite in their evaluation of aspect oriented techniques. They focused on the quality factors understandability, maintainability, reusability and testability, and the C&K definitions of metrics suited for measuring these factors. We have selected AJHotDraw case study to show that different ideas exist on how to define AOSD metrics. For defining AOSD metrics it is obvious to build on existing general software metrics and especially on existing OO metrics [12]. Thus, it is a key issue, how to extend metrics and how to define corresponding metrics besides defining completely new metrics. Most existing metrics cannot be applied straightforwardly to aspect-oriented software, since AOSD introduces new abstractions and new composition techniques. Consequently, there are new kinds of coupling and cohesion. Not only new metrics but also extended metrics to cover a new programming language have to be validated. Some characteristics and attributes have been widely acknowledged as playing a key role in the evolvability of software, both for OO and AO, such as size, coupling, cohesion and separation of concern, and because they are obviously related to aspect-oriented modularisation of crosscutting concerns. We can observe two tendencies with respect to defining metrics for these characteristics, extending metrics and defining new metrics.

5.1 Separation of concern metrics

Separation of concern metrics are new ones. In [14] the first separation of concern metrics were proposed. In [15] these metrics have been refined. Until now, these metrics requires manually identifying concerns.

5.2 Coupling metrics

Coupling is interesting because it is treated both ways. Initially, the coupling-between-objects metric [12] is defined as counting all classes (once) to which a class is coupled. This is in turn defined by counting the classes of the objects on which a given object/class “acts upon”. This refers to access or method calls on instance variables, local variables or formal parameters. Although the metric relies on different kinds of variables or parameter, essentially the same kind of access is counted, i.e., that an instance of another class is accessed via a reference and potentially via its attributes or methods. Also note that in the OO context, high numbers of coupling are considered as undesirable. In [15], this original metric is extended to cover also coupling between aspects. It includes new kinds of coupling as the existing definition cannot be straightforwardly applied to the coupling of an aspect to components. Thus it becomes a metric that counts different kinds of coupling whereas the original one relies on one kind. This allows for a uniform treatment of coupling. It implies high couplings are no longer a bad smell, e.g. between an aspect and an object. Other authors [16] have decided to provide separate metrics for the new kinds of couplings found in AO. Thereby it might be easier to use those metric as an indicator for different effects of the different kinds of coupling in a validation experiment. In [17] no new metrics are defined but the preservation of the existing definitions is assumed. They focus on the effects aspects typically have on these metrics, e.g., they discuss that OO coupling may be decreased. Thus, their work implies the necessity to be able to understand the OO part on its own, e.g. by preserving OO metrics.

5.3 Cohesion metrics

Cohesion has been addressed in both ways, too. [18] specifically looks at a new way for defining cohesion. Others extend existing metrics such as lack-of-cohesion in methods straightforwardly to advice [8].

5.4 Size metrics

Size metrics are often critized as being dependent on programming styles and being too simple. Size itself has an empirical counterpart in the physical length though there are numerous ways to define it, e.g., LOC. From this viewpoint it might be admissible to count a pointcut as a code line. In [15], experience with an extended LOC metric has been gathered. Size metrics can point to the fact that duplicated code is avoided. The question is, if LOC is a good indicator for maintainability and reusability when comparing AO with OO. Avoiding duplicate code might be achieved by writing difficult-to-maintain pointcuts. The C & K metrics for AOP are given in Figure 1.
6 A CASE STUDY-AJHOTDRAW

In order to present our approach, we selected the case study selected AJHotDraw [5], an AspectJ implementation of JHotDraw [4]. The original JHotDraw project was developed by Erich Gamma and Thomas Euggenschwiler. It is a Java GUI frame-work for technical and structured graphics. It has been developed as a design exercise but it is quite powerful. Its design relies heavily on some well-known design patterns. The AJHotDraw program contains more than 400 elements (classes, interfaces and aspects). As per the literature, there is no application of that size that has been carefully studied in the past regarding aspect-oriented quality. The first and most important observation we can talk about is that very few aspects were found in aspect-oriented programs. For example AJHotDraw had 407 classes and only 10 aspects. Secondly, aspects seem to crosscut classes that are complex and rarely crosscut small classes. This is even more easily observable in the bigger program. Thirdly; we observed that aspects were rarely built to crosscut many classes in the code. Usually one or two classes are attached to a certain aspect. This would mean that only a part of this aspect is dedicated to a large number of classes and that the main behaviour is done in the children aspects. And finally, we found that aspects are usually very cohesive. Once again the only exception to this rule was the large aspects with many children.

It is well known in the software community that low coupling and high cohesion are regarded as being quality characteristics of software systems. We can mention that the AJHotDraw, contained very few aspects (less than 10% of the total number of entities).

![Fig. 1: Selected AOP Design Metrics](image)

Table 1 gave corresponding metrics for both Java JHotDraw and AspectJ AJHotdraw implementation.

<table>
<thead>
<tr>
<th>Table 1: AOP METRICS FOR AJHOTDRAW</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>java</td>
</tr>
<tr>
<td>aspect</td>
</tr>
</tbody>
</table>

7 CONCLUSION

This paper gave an overview on the necessary steps for validating metrics that are to be used in an evaluation process. These steps are well-known in software engineering. The current state-of-the-art in AOSD is that one has started to work on the definition of apparently useful metrics. Now it is time to start with completing this research by providing empirical results. This will enable a larger to community to use AOSD metrics and more importantly, understand the benefits of AOSD. Thus, we have to strive for planning and carrying out the corresponding experiments. The results may also give hints as to for which purposes metrics extensions are useful and for which purposes separate metrics are useful.

REFERENCES


Kotrappa Sirbi having M Tech(CSE) from Visvesvaraya Technological University Belgaum,M S (Software System) from BITS,Pilani and B .E (EE) working with Department of Computer Science & Engineering, K L E’s College of Engineering & Technology,Belgaum since 1985.Presently working with K L E’s B C A, Belgaum(Deputation).He has 04 International Journal publications,02 International conference papers and 02 National Conference papers and 02 workshop papers for his credit.His areas of interest are Software Engineering, Object Technology and its evolution like., Design Patterns,AOP(Aspect Oriented Programming).He is member of ISTE,Members of CSTA,ACM,USA and Member of IAENG.

Prakash Jayanth Kulkarni having Ph.D (Electronics) and M E (Electronics) by Research from Shivaji University, Kolhapur and B.E (Electronics & Tele) from Poona University, Poona working with Walchand College of Engineering, Sangli since 1981 and 1980-81 worked with Trans Lines Division,M S E B haing 13 international conference papers and 10 National conference papers and 08 journal papers.His areas of interest are Digital Communication,Digital Image Processing and Computer vision ,Software Engineering, Artificial Neural Network and Genetic Algorithms.In 2001 he received a distinguish Samaj Shreee Award for rendering service to society.