Impact of Aspect Oriented Programming on Software Development Quality Metrics

Kotrappa Sirbi[1] Prakash Jayanth Kulkarni[2]

GJCST Computing Classification
D 2.3,D 2.m,K 6.4, K 6.3

***Abstract*-The aspect-oriented programming (AOP) is a new paradigm for improving the system's features such as modularity, readability and maintainability. Owing to a better modularisation of cross-cutting 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 programming (OOP) equivalent. In this paper, we provide some insight into the OO software development quality metrics were significantly associated with using AOP.The method that we are currently studying is based on a popular C & K metrics suite that extends the metrics traditionally used with the OO paradigm and also extend to AO paradigm. 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.**

Keywords- Aspect Oriented Programming (AOP), Aspect Oriented (AO) system, AO metrics, AspectJ.

I. INTRODUCTION

THE past decade has seen the increased use of Aspect Oriented Programming (AOP) based software development techniques as a means to modularize crosscutting concerns in software systems, thereby improving a development organization's working practices and return on investment (ROI).Numerous industrial-strength aspect-oriented (AO) programming frameworks exist, including AspectJ, JBoss, and Spring, as do various aspect-oriented analysis and design techniques. The *"Major Industrial Projects Using AOP*" are many notable applications, of which the most prominent is the IBM WebSphere Application Server. Developers considering AOP techniques must ask three fundamental questions:

• How is AOP being used in industrial projects today?

Developers must determine whether AOP techniques are suited to the problem at hand and the particular project context.

Does the improved modularity yield real benefits when engineering and evolving software?

Developers must understand whether the potential benefits outweigh the costs of introducing a new technology and, if so, be able to convince management of its long-term profitability.

•*What do developers need to be aware of when using AOP techniques?*

Developers must avoid known pitfalls and deploy design strategies and tools to help counter their potential threat to product quality.

Answers to these questions are not readily available, and narrowing knowledge from existing literature on the topic is difficult, but there is some insight by working with many several medium and large-scale open source projects employing AOP techniques. Much AO adoption shows that software development projects mainly rely on basic features of AO languages to modularize well-known crosscutting problems; developers introduce AOP concepts incrementally, initially addressing evelopmental concerns and not core product features. In addition, AOP techniques improve design stability over a system's evolution and can substantially reduce design model size [1].

The aspect oriented programming (AOP) is a relatively recent approach that has been argued to better enable modularization of crosscutting concerns [2] and consequently accelerate the development process. The hypotheses are that well separated concerns are more easily maintained, changed and developed, so the total programmer's working time should be shorter than the development time of analogous system, realized without mechanisms offered by AOP.The validation of these hypotheses requires empirical studies. Many researchers in literature present results of preliminary empirical evaluation of the impact of AOP on software development efficiency and design quality. This paper includes a comparison of developed AOP and OOP systems, based on software metrics proposed by Chidamber and Kemerer (hereafter CK) [3], Distance from the Main Sequence metric proposed by Martin [4], external code quality metric (defined as a number of acceptance tests passed) [5, 6, 7], and programmers' productivity metric.CK software metrics [3] were adapted to new properties of aspect-oriented software [8, 9].

Subramanyam and Krishnan state that research on metrics for object oriented software development is limited, and empirical evidence, linking the object-oriented methodology and project outcomes, is scarce [10]. Even more scarce is empirical evidence of the effect of aspect-oriented programming on software design quality, or development efficiency metrics. Therefore, the aim of this paper is to fill this gap and provide empirical evidence of the impact of aspect-oriented programming on software development efficiency and design quality metrics, as design aspects are extremely important to produce high quality software [10]. The hypothesis that design quality metrics are good predictors of the fault proneness is supported in [11] and [12].

About-[1]Kotrappa Sirbi, Department of Computer Science & Engineering, K L E's Dr.M.S.Sheshagiri College of Engineering & Technology, Belgaum, India.

About-[2] Prakash Jayanth Kulkarni, Department of Computer Science & Engineering ,Walchand College of Engineering,Sangli, India.

The rest of the paper is organized as follows, Section II provides a related work in the field of OO and AO metrics, Section III give a brief overview of Aspect-Orientated Programming(AOP), Section IV explaining importance of OO software metrics in OOD, Section V explaining requirements of AO software metrics, Section VI explains the potential effect of AO on the C&K metrics and Section VII implementation of case study AJHotDraw and Section VIII shows the impact of AO metrics on AJHotDraw and includes the results of the experiment. The conclusion of the paper is presented in Section IX.

II. Related Work

The literature available on the quantitative assessment of aspect-oriented solutions [14]. Kersten and Murphy [15] described the effect of aspects on object-oriented development practices, as well as some rules and policies that were employed to achieve maintainability and modifiability. Walker et al. [16] provided initial insights into the usefulness and usability of aspect-oriented programming.Soares et al. [17] reported that the AspectJ implementation of the Web-based information system has significant advantages over the corresponding pure Java implementation. Garcia et al. [18] presented a quantitative study, designed to compare the maintenance and reuse support of a pattern-oriented approach, and an aspect-oriented approach for a multi-agent system. It turned out that the aspect-oriented approach allowed the construction of the investigated system with improved modularization of the crosscutting agent-specific concerns. The use of aspects resulted in superior separation of the agent-related concerns, lower coupling (although less cohesive) and fewer lines of code. Tsang et al. [19] evaluated the effectiveness of AOP for separation of concerns. They applied the CK metrics suite to assess and compare an aspect-oriented and object-oriented real-time system in terms of system properties. They found improved modularity of aspect- oriented system over object-oriented system, indicated by the reduction in coupling and lack of cohesion values of the CK metrics. Hannemann and Kiczales [20], as well as Garcia et al. [21], have developed systematic studies that investigated the use of aspect-oriented programming to implement classical design patterns. It is worth mentioning that
Tonella and Ceccato [22] performed an empirical assessment of refactoring the aspectizable interfaces. This study indicates that migration of the aspectizable interfaces has a limited impact on the principal decomposition size, but, at the same time, it produces an improvement of the code modularity. From the point of view of the external quality attributes, modularization of the implementation of the crosscutting interfaces clearly simplifies the comprehension of the source code. Unfortunately, most empirical studies involving aspects have been based on subjective criteria and qualitative investigation [14].

III. Aspect Orientation Programming(AOp)

Aspect Oriented Programming (AOP) is a novel software development paradigm that aims at modularizing *aspects*, which are defined as well-modularized crosscutting concerns [23][24].This type of concerns cuts across traditional module boundaries such as classes and interfaces, and their implementation is scattered and tangled with the implementation of other concerns. *AspectJ* is the popular Java extension language of AOP [23].This basic constructs of the language are

Join point: A join point is a well-defined point in the execution of a component. It can be a method call or execution, an access to an attribute, or the execution of a constructor.

Pointcut: A pointcut is the mechanism that encapsulates join points. It can be compose of one or more join point.

Advice: An advice specifies the action (i.e., code) that must take place at a certain pointcut (i.e., a group of join points). With both abstractions mentioned above, advice gives developer the ability to implement crosscutting concerns.
There are three types of advice:
–before: The code declared is executed before the join point.
–after: The code declared is executed after the Join point.
–around: The code declared is executed instead of the one in the join point.

Inter-type declaration: This mechanism allows the developer to crosscut concerns in a static way. It permits alterations to classes and inheritance hierarchies from outside the original class definition. We enumerate below the types of possible changes through Inter-type declaration:
–Add members (methods, constructors, fields) to types (including other aspects).
–Add concrete implementation to interfaces.
–Declare that types extend new types or implement new interfaces.
–Declare aspect precedence.
–Declare custom compilation errors or warnings.
–Convert checked exceptions to unchecked.

Aspect*:* An aspect is the container for the encapsulation of pointcuts, advice code, and inter-type declaration. Acting like a Java classes, it can contain its own attributes and methods.

In AspectJ, an application consists of two parts: *base code* which corresponds to standard Java classes and interfaces, and *aspect code* which contains the crosscutting code. Next we describe the two types of crosscuts that AspectJ provides.

Static Crosscuts

Static crosscuts affect the static structure of a program [25,33]. We consider *Inter-Type Declarations (ITDs)*, also known as *introductions*, that add fields, methods, and constructors to existing classes and interfaces [25, 33].

Dynamic Crosscuts

Dynamic crosscuts run additional code when certain events occur during program execution. The semantics of dynamic crosscuts are commonly described and defined in terms of an event-based model [26][27]. As a program executes, different events fire. These events are called *join points*. Examples of join points are: variable reference, variable assignment, execution of a method body, method call, etc. A

pointcut is a predicate that selects a set of join points. *Advice* is code executed before, after, or around each join point matched by a pointcut[26].

IV. Object Orientation (OO) Software Metrics

The inadequacy of the metrics in use with procedural code (size, complexity, etc.), when applied to OO systems, led to the investigation and definition of several metrics suites accounting for the specific features of OO software. However, among the available proposals, the one that is most commonly adopted and referenced is that by Chidamber and Kemerer [3].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.

V. Aspect Oriented(AO) Software Metrics

In this 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. 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 sub- classes 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. How- ever, CIM takes into account only those modules and inter- faces an aspect is aware of – those that are explicitly mentioned in the pointcuts. Submodules, modules implementing named interfaces or modules referenced through wild- cards 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 generality/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 (Cou- pling 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 ac- count 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 can- not 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 in- crease 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 introductions in a given aspect.*

  This is a brand new metric, specific to AOP software, that must be introduced as a completion of the CIM metric. While CIM considers only explicitly named modules, CDA measures all modules possibly affected by an aspect. This gives an idea of the overall impact an aspect has on the other modules. Moreover, the difference between CDA and CIM gives the number of modules that are affected by an aspect without being referenced explicitly by the aspect, which might indicate the degree of generality of an aspect, in terms of its independence from specific classes/aspects. High values of CDA and low values of CIM are usually desirable.
- **Weighted Methods per Class (WMC):** WMC is a measure of the number of methods implemented within a class. This metric measures understandability, maintainability, and reusability as follows:
  - The number of methods in a class reflects the time and effort required to develop and maintain the class.
  - The larger the number of methods, the greater the potential impact on children, since children inherit all of the methods defined in a class.

  A class with a large number of methods is more application-specific, and therefore is not likely to be reused.
- **Lack of Cohesion in Methods (LCOM):** LCOM is the degree to which methods within a class are related to one another and work together to provide well-bounded behavior. Well-designed systems should maximize cohesion, since it promotes encapsulation. LCOM measures the degree of similarity of methods by data input variables or class attributes. In [28], two ways of measuring LCOM are described:
  - Calculate for each data field in a class what percentage of the methods use that data field. Average the percentages then subtract from 100%. Lower percentages mean greater cohesion of data and methods in the class.
  - Methods are more similar if they operate on the same attributes. Count the number of disjoint sets produced from the intersection of the sets of attributes used by the methods.

  This metric evaluates efficiency and reusability. High cohesion indicates good class subdivision. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion.
- **Coupling Between Objects (CBO):** CBO is a count of the number of other classes to which a class is coupled. CBO is measured by counting the number of distinct non- inheritance related class hierarchies on which a class depends. Excessive coupling prevents reuse. The more independent a class is, the more likely it can be reused. The higher the coupling the more sensitive the system is to changes in other parts of the design, and therefore maintenance is more difficult. High coupling also reduces the system's understandability because it makes the module harder to understand, change, or correct by itself if it is interrelated with other modules.
- **Response For a Class (RFC):** RFC is the number of all methods that can be invoked in response to a message to an object of the class or by some method in the class. This measures the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated as it requires a greater level of understanding on the part of the developer. This metric evaluates understandability, maintainability, and testability.

VI. The Effect of AO on the C&K Suite

In this section, it provides an analysis of the effect of aspect-orientation on the C&K metrics suite. It is based on the case studies found in the literature about re-designing some existing software systems to incorporate the aspect-oriented paradigm.

- **Weighted Methods per Class:** aspects might help reduce the number of methods per class as follows:
  - Aspects combine crosscutting functionalities in modular, encapsulated units. Without aspect-oriented design, these crosscutting functionalities

would be tangled in the core class.

- In some cases, a sub- class might have to over- ride a function in its parent class in order to define its own aspectual behavior (not a core behavior). Let's take exception handling as an example. A function in the subclass might have to override a super- class function just to implement the sub-class's method of handling a certain exception. If exception handing was implemented as an aspect, the subclass will not have to add a function to implement its own exception handling technique. This reduces the WMC factor. Dealing with exception handling as aspects is discussed in more detail in [29].
- **Depth of Inheritance Tree:** subclasses that might be defined only for the purpose of applying their own implementation of aspectual behavior will not exist in systems designed using the AO Paradigm, because aspects will be responsible for that. This helps in reducing the depth of inheritance tree.
- **Number Of Children:** the same argument of "Depth of Inheritance Tree" is valid for this metric.
- **Lack of Cohesion in Methods:** aspects filter out crosscutting behavior, and therefore increases cohesion. Figure 1 is an example of this. The function Movable() is likely to contain synchronization checking that determine if the function Move can be invoked on an object of type Shape. This can be seen as a synchronization aspect, which uses its own flags to determine synchronization. Such a crosscutting function reduces the cohesion of the class Shape.
- **Coupling Between Objects:** the presence of aspects is likely to decrease the coupling between core classes, yet increase the coupling between core classes and aspect classes. This is because aspects are new entities on which core classes depend. It should be noted, however, that, unlike aspects, core classes are more likely to be reused. Decreasing the coupling between core -classes is a beneficial issue, and increasing coupling between aspects and core classes in return can be seen as a good trade- off. Given that a design might involve coupling between c lasses, it would be better to have this coupling occur between core and aspect classes, rather than having it happen between core classes.
- **Response For a Class:** RFC is likely to increase in the presence of aspects. This is because the number of entities that a class communicates with increases, and classes have to communicate with aspects. The positive point with using aspects is that they can be designed in a way that encapsulates the logic and the objects with which a class communicates in a modular way.

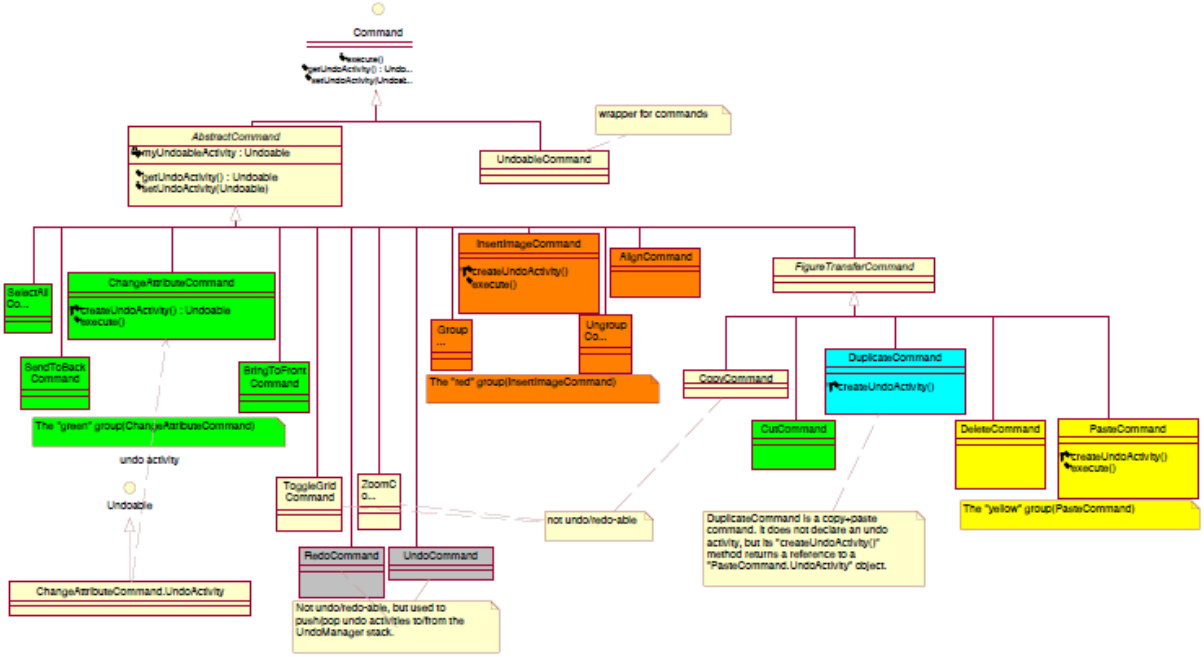

Figure 1:JHotDraw Command Hierarchy

VII. Case Study : AJHotDraw

The case study selected is AJHotDraw [30], an AspectJ implementation of JHotDraw [31]. The original JHotDraw project was developed by Erich Gamma and Thomas Eggenschwiler. 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 pat- terns. The AJHotDraw program contains more than 400 elements (classes, interfaces and aspects). To our best knowledge, there is no application of that size that has been carefully studied in the past regarding aspect- oriented quality. The *Command*

hierarchy in JHOTDRAW, shown in figure 1, implements the design pattern bearing the same name. The (12) undo-able commands store a reference to their associated undo activity. These command's execution through dedicated factory methods.AJHotDraw is an open source software project that provides numerous features for drawing and manipulating graphical and planar objects [1]. It consists of 13 features for a total of ~ 50KLOC. It is implemented with 279 classes and interfaces and only 31 aspects. Not surprisingly approximately 99% per-cent of the code is standard Java and only 1% of aspect code, of which almost all comes from ITDs.The modularized crosscutting concerns are persistence, design policies, contract enforcement, Undo command.

VIII. Impact of AO Metrics on AJHotDraw

The proposed metrics have been computed on an open source project AJHotDraw, taken from the implementation of some design patterns [32] provided by
Jan Hannemann both in Java and in AspectJ (appropriate AO Metrics are shown in Figure 2)
The practical implementation is based on *Observer* design pattern [32], in which there are two distinct roles, the *Subject* and the *Observer*. The Subject is an entity that can be in several different states. Some of the state changes are of interest to the Observer, which may take some actions in response to the change.

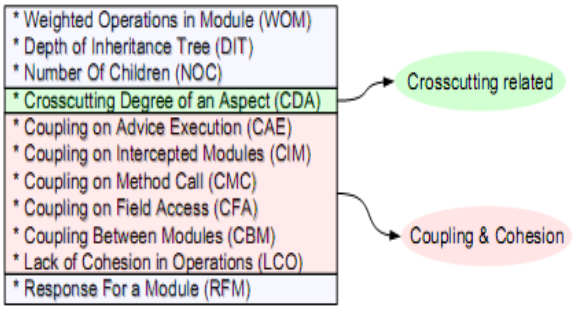

Figure 2: AOP C & K Metrics

The Observer pattern requires that the Observer registers itself on those Subjects it intends to observe. The Subject maintains a list of the Observers registered so far. When the Subject changes its state, it notifies the Observers of the change, so that the Observers can take the appropriate actions. In the OO implementation by Jan Hannemann, this design pattern consists of two interfaces, ChangeSubject and ChangeObserver, with the abstract definitions of the Subject and Observer roles. Moreover, the implementation contains the Point and the Screen classes, the first playing the role of Subject whereas the second plays both roles in two different instances of the pattern. The Main class contains the code to set up the two different pattern instances and run them. In the first pattern instance Point acts as the Subject and Screen as the Observer. In the second case, an instance of the class Screen is the Subject, while other instances of the same classes are its Observers. The AOP implementation contains a different version of the classes Point and Screen, with no code regarding the Subject/Observer roles. *ObserverProtocol* is an abstract aspect defining the general structure of the aspects that implement the Observer pattern. This abstract aspect is extended by *ScreenObserver*,*ColorObserver* and *CoordinateObserver*. These concrete aspects contain the actual implementation of the protocol. By means of inter-type declarations, they impose roles onto the involved classes and by means of appropriate pointcuts they specify the Subject actions to be observed. Moreover, these aspects contain the mapping that connects a Subject to its Observers. The class `Main` runs the code for the initialization of the patterns for their execution. The output of the metric suite to the two implementations of the Observer pattern and the median values produced by the tool are shown in Table 1. The value of LCO for the OO code is indicated as 1-12, since these two values are adjacent to the median point. We observer that the improvement in some metrics (WOM, LCO, CMC and RFM), no change in other metrics (NOC and CFA) and a worse value of DIT (due to the superaspect *ObserverProtocol*).But the general values change only a little bit, for RFM the change is relatively high, passing from 7 to 2. LCO is also affected positively, going from 1-12 to 0. The cost to be paid for such improvements is an increase of the CIM metric, due
to the aspects intercepting method executions (AOP coupling).

version	WOM	DIT	NOC	CAE	CIM
java	3	1	0	0	0
aspectj	1	2	0	0	2
version	**CMC**	**CFA**	**RFM**	**LCO**	**CDA**
java	2	0	7	1-12	0
aspectj	1	0	2	0	3

Table 1: AOP Metrics for AJHOTDRAW

IX. Conclusions and Future Work

Assessing the quality of software has been the preoccupation of software engineers for two decades. The problem of separation of concerns led to the apparition of the aspect-oriented paradigm. This new paradigm raises questions about quality, due to its close relations with object-oriented programming. In this paper, we argue that the impact of AOP on software development quality metrics is significant. The proposed work shall be validated through empirical studies. In fact, case study used here shall enable us to appraise the quality of an aspect-oriented system over object oriented system.
Some of the issues that require more research and metrics are:
Aspect Granularity: how many crosscutting functionalities should an aspect encapsulate. Dependency between aspect

and class: how aspects can be designed such that the dependency of core classes on them is minimal.

Understandability: how aspects affect the system's understandability.

Depth of aspect inheritance tree: are there limitations for aspect- inheritance? And how far does it affect the design understandability.

X. References

1) Awais Rashid et al "Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe", Lancaster University, UK.Published by the IEEE Computer Society, February, 2010.
2) Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J.: 'Aspect-Oriented Programming',Proc. European Conf. Object-Oriented Programming (ECOOP 1997), vol. 1241 of Lecture Notes in Computer Science, Jyv¨askyl¨a, Finland, June 1997, pp. 220–242
3) Chidamber, S. R., and Kemerer, C. F.: 'A Metrics Suite for Object Oriented Design', IEEE Trans. Softw. Eng., 1994, 20, (6), pp. 476– 493
4) Martin, R. C.: 'OO Design Quality Metrics: An Analysis of Dependencies', http://www.objectmentor.com/resources/articles/oodmetrc.pdf, accessed September 2006
5) George, B., and Williams, L. A.: 'An Initial Investigation of Test Driven Development in Industry', Proc. ACM Symposium on Applied Compupting (SAC 2003), Melbourne, USA, March 2003, pp. 1135–1139
6) George, B., and Williams, L. A.: 'A structured experiment of test driven development', Inf. Softw. Tech., 2004, 46, (5), pp. 337–342
7) Madeyski, L.: 'Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality', in Zieli´nski, K., and Szmuc, T. (Ed.): 'Software Engineering: Evolution and Emerging Technologies', vol. 130 of Frontiers in Artificial Intelligence and Applications, (IOS Press, 2005), pp. 113–123
8) Ceccato, M., and Tonella, P.: 'Measuring the Effects of Software Aspectization', Proc. Workshop on Aspect Reverse Engineering (WARE 2004), Delft, The Netherlands, November 2004 (Cd-rom)
9) Aopmetrics project, http://www.e-informatyka.pl/sens/Wiki.jsp?page=Projects.AOPMetrics, accessed September 2006
10) Subramanyam, R., and Krishnan, M. S.: 'Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects', IEEE Trans. Softw. Eng., 2003, 29, (4), pp. 297–310
11) Briand, L. C., W¨ust, J., Ikonomovski, S. V., and Lounis, H.: 'Investigating Quality Factors in Object-Oriented Designs: an Industrial Case Study', Proc. Int. Conf. on Software Engineering (ICSE 1999), Los Alamitos, USA, May 1999, pp. 345–354
12) Emam, K. E., Melo, W. L., and Machado, J. C.: 'The Prediction of Faulty Classes Using Object-Oriented Design Metrics', J. Syst. Softw., 2001, 56, (1), pp. 63–75
13) Basili, V. R., Caldiera, G., and Rombach, H. D.: 'The Goal Question Metric Approach', in Marciniak J.J. (Ed.): Encyclopedia of Software Engineering, (Wiley, 1994), pp. 528–532
14) Garcia, A. F., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C. J. P.de, and Staa, A.von: 'Modularizing Design Patterns with Aspects: A Quantitative Study', in Rashid, A., and Aksit, M. (Ed.): T. Aspect-Oriented Software Development I, 2006, vol. 3880 of Lecture Notes in Computer Science, pp. 36–74
15) Kersten, M., and Murphy, G. C.: 'Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-oriented Programming', Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1999), New York, USA, November 1999, pp. 340–352
16) Walker, R. J., Baniassad, E. L. A., and Murphy, G. C.: 'An Initial Assessment of Aspect-oriented Programming', Proc. Int. Conf. on Software Engineering (ICSE 1999), Los Alamitos, USA, May 1999, pp. 120–130
17) Soares, S., Laureano, E., and Borba, P.: 'Implementing Distribution and Persistence Aspects with AspectJ', Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002), New York, USA, November 2002, pp. 174–190
18) Garcia, A. F., Sant'Anna, C., Chavez, C., Silva, V. T.da, Lucena, C. J. P.de, and Staa, A.von: 'Separation of Concerns in Multi-agent Systems: An Empirical Study', Proc. Int. Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003), vol. 2940 of Lecture Notes in Computer Science, pp. 49–72
19) Tsang, S. L., Clarke, S., and Baniassad, E. L. A.: 'An Evaluation of Aspect-Oriented Programming for Java-Based Real-Time Systems Development', Proc. Int. Symposium on Object-Oriented Real-TimeDistributed Computing (ISORC 2004), Vienna, Austria, May 2004, pp. 291–300
20) Hannemann, J., and Kiczales, G.: 'Design Pattern Implementation in Java and AspectJ', Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002), New York, USA, November 2002, pp. 161–173
21) Garc´ıa, F., Bertoa, M. F., Calero, C., Vallecillo, A., Ru´ız-S´anchez, F., Piattini, M., and Genero, M.: 'Towards a consistent terminology for software

measurement', Inf. Softw. Tech., 2006, 48, (8), pp. 631–644
22) Tonella, P., and Ceccato, M.: 'Refactoring the aspectizable interfaces: An empirical assessment.', IEEE Trans. Softw. Eng., 2005, 31, (10), pp. 819–832
23) AspectJ, http://eclipse.org/aspectj/
24) Kiczales, G., Hilsdale, E., Hugunin, J., Kirsten, M., Palm, J., Griswold, W.G.: 'An overview of AspectJ'. ECOOP (2001)
25) Laddad, R.:. 'AspectJ in Action. Practical Aspect-Oriented Programming'. Manning (2003)
26) Lämmel, R.: 'Declarative Aspect-Oriented Programming'. PEPM (1999),
27) Wand, M., Kiczales, G., Dutchyn, C.: 'A Semantics for Advice and Dynamic Join Points in Aspect Oriented Programming'. TOPLAS (2004)
28) Rosenberg, Linda H. and Lawrence E. Hyatt. "Software Quality Metrics for Object-Oriented Environments". NASA, SATC. http://www.satc.gsfc.nasa.gov/support/CROSS_APR97/oocross.html
29) Lippert, Martin, Cristina Videira Lopes. 'A Study on Exceptio n Detection and Handling Using Aspect- Oriented Programming'. Xerox Palo Alto Research Center. Technical Report, Dec. 99.
30) Ajhotdraw project. http://sourceforge.net/projects/ajhotdraw/.
31) Jhotdraw project. http://www.jhotdraw.org/.
32) E . Gamma, R. Helm, R.Johnson, and J. Vlissides. 'Design Patterns: Elements of Reusable Object Oriented Software'. Addison-Wesley Publishing Company, Reading, MA, 1995.
33) LADDAD, R. 'Enterprise AOP with Spring Applications: AspectJ in Action' 2nd Edition, Manning Publications, 2010.