Hindawi Publishing Corporation
ISRN Software Engineering
Volume 2013, Article ID 121692, 11 pages
http://dx.doi.org/10.1155/2013/121692

Research Article

An Assessment of Maintainability of an Aspect-Oriented System

Kagiso Mguni and Yirsaw Ayalew

Department of Computer Science, University of Botswana, Private Bag 0704, Gaborone, Botswana

Correspondence should be addressed to Yirsaw Ayalew; ayalew@mopipi.ub.bw

Received 30 November 2012; Accepted 23 December 2012

Academic Editors: C. Calero, R. J. Walker, and B. Yang

Software maintenance is an important activity in software development. Some development methodologies such as the object-oriented have contributed in improving maintainability of software. However, crosscutting concerns are still challenges that affect the maintainability of OO software. In this paper, we discuss our case study to assess the extent of maintainability improvement that can be achieved by employing aspect-oriented programming. Aspect-oriented programming (AOP) is a relatively new approach that emphasizes dealing with crosscutting concerns. To demonstrate the maintainability improvement, we refactored a COTS-based system known as OpenBravoPOS using AspectJ and compared its maintainability with the original OO version. We used both structural complexity and concern level metrics. Our results show an improvement of maintainability in the AOP version of OpenBravoPOS.

1. Introduction

Software maintenance is one of the most expensive activities that consume about 50–70 percent of the development cost [1]. Therefore, it is a very important activity that requires much attention. For this reason, people have attempted to find ways to minimize maintenance costs by introducing better development methodologies that can minimize the effects of change, simplify the understanding of programs, facilitate the early detection of faults, and so forth. In this regard, the object-oriented approach has played an important role by improving the maintainability of software using the concepts of object and encapsulation. Specifically, the OO approach has been hailed for providing constructs that minimize (i.e., localize) the impact of change. However, there are still some concerns that crosscut among a number of objects whose modification may be difficult as they are scattered across many objects. To address this issue, aspect-oriented programming (AOP) has been introduced as a way of improving the modularity of code there by facilitating maintenance. Concerns that crosscut across different components are represented by the construct aspect. A concern is a feature that a system should implement which includes all the functional, nonfunctional requirements and the design constraints in the system [2].

Aspect-oriented programming was developed to overcome the limitations of programming approaches such as OOP in handling crosscutting concerns. Therefore, it introduces several new constructs related to handling crosscutting concerns. The different constructs that are introduced by AOP are as follows.

(i) *Aspects*. Aspects are similar in nature to classes in OO. Aspects are also considered to be equivalent to the other OO constructs such as interface. Aspects are used to wrap up advices, pointcuts, and intertype declarations in a modular unit the same way classes and interfaces wrap up declarations and methods [3]. Aspects together with classes can be referred to as modules and they both represent the basic unit of modularity in AOP.

(ii) *Advices*. Advices are similar in nature to methods in OO. Aspects contain the body of the code that is executed at a given point that is captured by a join point. The advices together with their OO counterparts, methods, are referred to as operations.

(iii) *Intertype Declarations*. These are used to add new fields or methods to a class. In contrast, they are similar to static attributes in OO.

The main focus of AOP is to improve the implementation of crosscutting concerns (i.e., scattering and tangling). The question we would like to address in this paper is "*To what extent maintainability improves when an OO code is refactored using AOP?*" In other words, we want to assess the maintainability of AOP software. To answer this question, we need to understand how we measure maintainability. According to IEEE [4], maintainability can be defined as "*The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.*" In order to measure maintainability, we need to employ appropriate metrics. There are various maintainability metrics that are geared towards the different maintainability characteristics.

Quality models such as the ISO/IEC 9126 describe maintainability using the following characteristics: analyzability, changeability, stability, testability, and maintainability compliance [5]. The new quality model, ISO25010, provides additional maintainability characteristics such as reusability and modifiability. Measures such as the maintainability index [6] have been proposed to objectively determine the maintainability of software systems based on the status of the corresponding source code. To simplify the measurement of maintainability, Heitlager et al. [7] introduced a mapping of system characteristics onto source code properties. The source code properties that were introduced include volume, complexity per unit, duplication, unit size, and unit testing. For example, it was indicated that the degree of source code duplication (also called code cloning) influences analyzability and changeability. Some metrics are used to measure the internal quality (i.e., based on source code) and others are based on the external quality (i.e., based on sources other than source code). A discussion of metrics relevant to AOP software is provided in Sections 2 and 3. The focus of this paper is on maintainability that is grounded in source code analysis. To assess the maintainability of the original code and the refactored AOP code, we have chosen open-source COTS (commercial off the shelf) components. The open source COTS components are OpenBravoPOS and Jasperreports.

The rest of the paper is organized as follows. Section 2 presents related work on maintainability of AOP software. A discussion of how we conducted the study is provided in Section 3. The results of the study and issues emanating from the study are discussed in Section 4. Section 5 provides the main points of the study and future work.

2. Related Works

In this section, we discuss the relevant metrics used to assess maintainability and empirical studies that were conducted to determine the effectiveness of various maintainability metrics.

2.1. Maintainability Metrics. Most AOP maintainability metrics were derived from the maintainability metrics of OO systems [8]. However, in some cases, some new metrics which take care of the special characteristics of AOP need to be introduced. For example, the authors in [8] have introduced specialized maintainability metrics specific to AOP.

Table 1 summarizes the different studies and the corresponding metrics used to assess maintainability of AOP software.

2.2. Empirical Studies. Kvale et al. [15] conducted a study to show how AOP can be used to improve the maintainability of COTS-based systems. In their study, they argue that if the code for calling the COTS libraries is scattered all over the glue code, the maintenance of the system will be difficult. They showed that if the glue code is built using AOP, then the COTS-based system can be easily maintained. They used the Java Email Server in their experiments. They showed that AOP improves the changeability of a COTS-based system as the code that has to be modified is minimal in cases where AOP is used in the glue code. They used size-based metrics in their study.

Tizzei et al. [16] undertook a study to assess the design stability of an application built using software components and aspects in the presence of changes. The components in their case study refer to components built using the Component Service Model with Semantics (COSMOS*). In their study, they compared eight releases of four different versions of a MobileMedia application: an OO version, AOP version, component-based version, and a hybrid version where both components and aspects were employed. The first two versions of the MobileMedia application already existed and the last two were refactored from the first two versions. In their study, they measured the impact of change by looking at the number of components/operations changed, added, and removed. Their study showed that the hybrid version required fewer changes as compared to the other versions.

Kumar et al. [11] undertook a study on the change impact in AOP systems. In their study, they used AOP systems that have been refactored from their OO versions. The systems that they used were refactored from 149 OO modules into 129 modules. For the OO versions, the module refers to the classes, and for the AOP version, module refers to classes and aspects. Their study used the metrics and tool for collecting metrics data as defined in [8]. In their study, they found out that the change impact is lower in AOP systems as compared to the OO systems. Also, they found out that if the concerns which are not crosscutting are moved to aspects, then the impact of change for these modules will be higher. They assessed the maintainability of the AOP based on the changeability of the system, hence their assessment was done at the module level.

Przybyek [14] compared the modularity of OO- and AOP-based systems using the GQM approach. She used several systems that have an OO and AOP versions. These systems had also been used in several other studies for studying the effect of AOP. The modularity was compared using CBO and LCOM metrics. The systems that they used in their study are Telestrada, Pet Store, CVS Core Eclipse plug-in, EImp, Health Watcher, JHotDraw, HyperCast, Prevayler, Berkely DB, and HyperSQL Database. In their study, they observed that AOP does not offer any benefits in terms of modularity.

Table 1: Summary of maintainability metrics.

Study	Dependent variable	MetricstTested	Summary of result
Burrows et al. [9]	Fault-proneness	All Ceccato and Tonella metrics	In addition, they introduced a new metric base aspect coupling (BAC), which measures the coupling between base class and aspect. The study showed that the two metrics that displayed the strongest correlation to faults were CDA and BAC
Eaddy et al. [10]	Fault-proneness	DOSC, DOSM, CDC, CDO	Assessed the correlation between faults and crosscutting concerns. They found out that the more scattered a concern is the more faults in its implementation are. Concern metrics used to predict the scattering of a concern. These metrics are independent of the program size
Kumar et al. [11]	Changeability	WOM	Assessed the correlation between changeability and WOM metric. They found that the WOM can be used as an indicator of maintainability but it is a weak indicator. Change impact is less in AOP systems as compared to OO systems. Maintenance effort was measured in terms of the number of modules changed
Kulesza et al. [12]	Coupling, cohesion, separation of concerns	Sant'Anna metrics which includes LCOO, WOC, VS	VS and WOC cannot be used as predictors of maintainability as the increase in such metrics was always accompanied by less development effort. LCOO metric inconclusive for measuring maintainability
Shen et al. [13]	Changeability (coupling and maintenance tasks)	Ceccato and Tonella metrics for coupling (CFA, CMC, RFM, CAE, and CDA)	Coupling metrics correlated with maintainability.
Przybyek [14]	Modularity	CBO and LCOM	CBO and LCOM used to measure the modularity of a system. Aggregate coupling and cohesion should not be considered as coupling should be measured independent of the number of modules in the system.

Lippert and Lopes [17] refactored exceptional handling using AspectJ in the JWAN framework and found out that the code was reduced by a factor of 4. The JWAN framework has been built using the design by contract; hence the refactoring was targeted towards these contracts. For example, in the JWAN framework, all methods that return an object have to ensure that the object returned is not null. This is a clear example of a feature that could be better implemented with aspects. Also, exception handling was a key design feature of the JWAN framework as about 11% of the code was targeted towards exception handling.

Our study is similar to the studies described above. However, we want our study to be wider in scope by assessing the different maintainability characteristics. For example, the authors in [11, 15] focused only on changeability. The study by Tizzei et al. [16] measured the impact of change by looking at the number of components/operations changed, added, and removed. The study by Przybyek [14] focused on assessing modularity among OO and AOP systems. Our study can complement these studies by including some additional metrics for evaluating the maintainability of the COTS-based system. These metrics include those used to measure coupling, cohesion and complexity. Most of these metrics address the different characteristics of maintainability. In addition, we want to assess maintainability at two levels: concern level and structural level. By doing so, we want to assess their effectiveness in measuring maintainability.

3. Study Setup

The main aim of this research is to assess the maintainability of COTS-based systems developed using AOP. Therefore, we have formulated the following two research questions.

Question 1: *Is a COTS-Based System Built Using AOP More Maintainable Than One Built Using OO?* To answer this question, we compare the maintainability of the OO and AOP versions of OpenBravoPOS and Jasperreports. The assessment of maintainability has been carried out using structural complexity metrics and concern level metrics.

Question 2: *Which Maintainability Metrics (Structural Complexity or Concern Level) Have a Better Potential for Predicting AOP Maintainability?* To answer this question, we compare the results of the two maintenance tasks for the OpenBravoPOS and Jasperreports systems.

3.1. Selection of COTS Components. The COTS-based system OpenBravoPOS was selected for our study. The main reason for the selection of this system is based on our experience

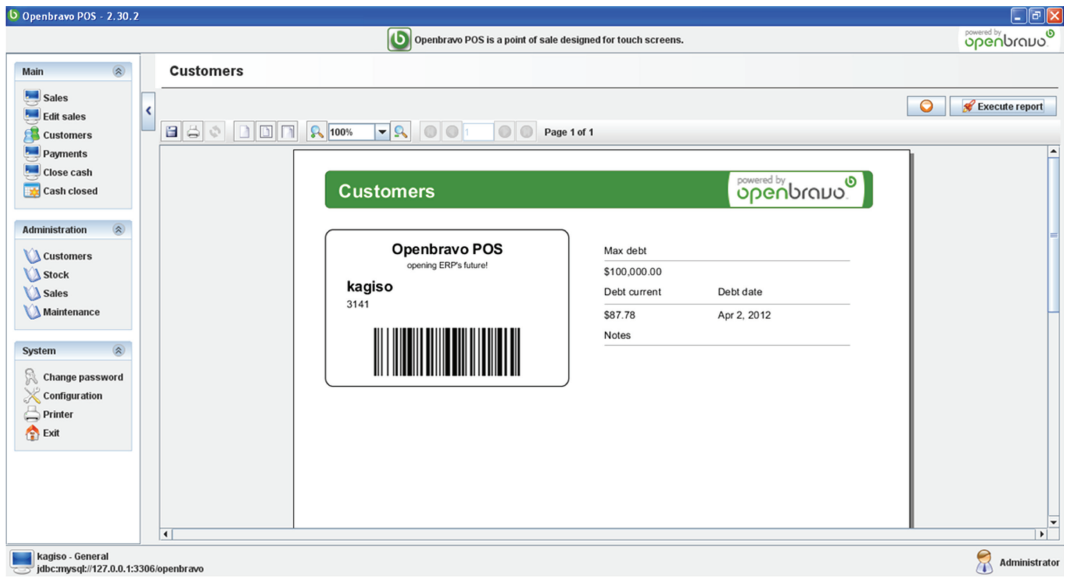

Figure 1: Screenshot showing OpenBravoPOS and Jasperreports in action.

on development on this system. OpenBravoPOS is a Java-based application that implements functionalities for the retail business [18]. It is an open-source system built with several components which include *Apache poi* which is used for reading and writing Microsoft Office files [19], *iText* which is used for reading and writing PDF files [20], *Commons-Logging* which is a component that provides logging features [21], and *jasperreports* which is used for the generation of reports [22]. Our study makes use of OpenBravoPOS and Jasperreports.

OpenBravoPOS is a typical example of a COTS-based system that is representative of a real world retail application. It implements common crosscutting concerns such as security, logging, persistence, and session management. The screenshot below (Figure 1) shows OpenBravoPOS and Jasperreports in action. Figure 1 shows a report running within the OpenBravoPOS. This report is generated by calling the Jasperreports component from the OpenBravoPOS.

Jasperreports is also built using several components which include *Commons-Logging*, *iText*, and *Apache Poi*. Jasperreports is a reporting tool and it accesses the database for retrieving data that is posted to the reports. Jasperreports implements crosscutting concerns such as persistence and transaction management in addition to the common crosscutting concerns such as logging and exceptional handling.

3.2. Refactoring and Maintenance of OpenBravoPOS and JasperReports. Refactoring is the process of changing the software component while leaving its functional state unchanged [10]. The COTS components OpenBravoPOS and Jasperreports were implemented using the OO programming language—Java. For the purpose of our experiment, we refactored OpenBravoPOS and Jasperreports to create equivalent implementations in AOP using the programming language AspectJ. The choice of AspectJ as the implementation language is mainly due to its popularity. In order to assess maintainability of the original OO versions of OpenBravoPOS and Jasperreports and the refactored AOP versions, we also carried out maintenance tasks on both versions. Table 2 shows the OO versions and their corresponding AOP versions of OpenBravoPOS and Jasperreports.

Changeability in a COTS-based system refers to either replacing/removing/adding a component or replacing/removing/adding a feature of code in a component. In this study, we assess the impact of AOP in a COTS-based system where changeability is a major issue.

The maintenance tasks M1 and MJ1 refer to the addition of a feature in the OpenBravoPOS and Jasperreports components, respectively. A feature for measuring how much time is taken during the execution of SQL statements was added to both components. A top-down approach using FEAT was used for identifying the different places where the concern is implemented. It identified the different places where the calls to the database for executing select statements were implemented. A feature was then added to measure the time taken for the execution of such statements.

The maintenance task M2 refers to the replacement of the logging component within OpenBravoPOS. The original OpenBravoPOS component uses the java.util.logging library for logging. This component was replaced with the Apache commons logging component. The logging concern was refactored earlier in the experiment; hence we already had information about the different places where the logging component is called. The calls to the java.util.logging component were replaced with the calls to the Apache commons logging concerns. The information about the different places where the logging concern was implemented was already collected using FEAT.

3.3. Aspect Mining and Concern Identification. The components OpenBravoPOS and Jasperreports were used in our previous study [23] where the refactoring was done by starting with a list of known concerns and then looking

Table 2: Refactoring of the original OO version to AOP.

OO version	AOP version	Description
OpenBravoPOS	OpenBravoPOS-AOP	Refactored from the original
OpenBravoPOS-M1	OpenBravoPOS-AOP-M1	After applying maintenance task 1 (same maintenance to both versions)
OpenBravoPOS-M2	OpenBravoPOS-AOP-M2	After applying maintenance task 2 (same maintenance to both versions)
Jasperreports	Jasperreports-AOP	Re-factored from the original
Jasperreports-MJ1	Jasperreports-AOP-MJ1	After applying maintenance task 1 (same maintenance to both versions)

for these concerns in the code. We followed the concern identification procedure used by Storzer et al. [24]. The Feature exploration and analysis (FEAT) [25] tool was also used for concern identification.

In this study, we also used an Eclipse plug-in called ConcernTagger [26] for collecting the concern level metrics which have been used in other related studies such as the one by Eaddy et al. [10]. The metrics that were implemented by ConcernTagger are CDC, CDO, DOSM, and DOSC [26]. The structural complexity metrics which are based on the Ceccato and Tonella metrics suite were collected using AOP metrics [27]. AOP metrics has been extended as an Eclipse plug-in as the developmental environment for our study has been Eclipse.

3.4. Maintainability Metrics and Tools. Software metrics can be classified as either product, resource, or process metrics [28]. Different researchers have proposed different metrics [8–10, 13, 29] for assessing the impact of AOP on software quality. The proposed metrics can be classified as either concern level metrics or structural complexity metrics.

3.4.1. Concern Level Metrics. Concern level metrics consider the lowest level of granularity of a software system to be a concern. Concerns are identified from project-related documentations and they should account for most of the source code [10]. The modules and/or operations responsible for the implementation of a given concern are grouped together and the concern level metrics collected and evaluated. Eaddy et al. [10] introduced the term *concern implementation plan*, to show the realization of concerns in the source code. Developers create concern implementation plans by creating or modifying the source code constructs in order to realize the concerns. The concerns can be mapped to different program elements in the source code such as modules, fields, operations, and statements [10]. In an ideal scenario where there are no crosscutting concerns, there would be a many-to-one relationship between the program elements and the concerns that is, several program elements mapping to one concern [30]. The concern level metrics as defined by Eaddy et al. [10] and Sant'Anna et al. [31] are as follows.

(i) *Program Element Contribution (CONT)*. This is the number of lines of code associated with the implementation of a given concern.

(ii) *Concern Diffusion over Components (CDC)*. This metric counts the number of classes and aspects responsible for the implementation of a concern.

(iii) *Concern Diffusion over Operations (CDO)*. This metric counts the number of advices and methods responsible for the implementation of a concern.

(iv) *Degree of Scattering across Classes (DOSC)*. It is the degree to which the concern code is distributed across classes. When DOSC is 0, all the code for the implementation of a concern is in one class, but when DOSC is 1, the code for the implementation is equally distributed across all classes implementing a given concern.

(v) *Degree of Scattering across Methods (DOSM)*. It is the degree to which the concern code is distributed across operations.

(vi) *Lines of Concern Code (LOCC)*. There are the lines of code in the implementation of a concern.

The collection of these metrics requires the assessor to construct a concern implementation plan that covers the concerns of interest in the source code of a given AOP software. There are different techniques and tools that have been proposed for identifying concerns in the source code. However, there are no widely accepted techniques and tools for locating concerns. The existing tools for measuring concern level metrics require the researcher to manually select the code associated with a given concern [32]. Such tools include ConcernTagger and AspectJ Assessment Tool (AJATO).

3.4.2. Structural Complexity Metrics. The AOP structural complexity metrics used in this study were defined by Ceccato and Tonella [8] which were adapted from the popular CK metrics as proposed by Chidamber and Kemerer [33]. The metrics as defined by Ceccato and Tonella and Burrows et al. [9] are as follows.

(i) *Weighted Operations per Module (WOM)*. This is number of operations (methods or advices) in a module. This is equivalent to the weighted operations per class (WMC) of the CK metrics [8]. A class with a higher number of operations is considered to be more complex, and hence it is fault prone [8, 34]. The complexity of the operations is considered to be equal. Also, more effort is needed to test a class with a higher WOM value [34]. A lower value of WOM is desired per module.

(ii) *Depth of Inheritance Tree (DIT)*. This is the length of the class from a given module to the class/aspect hierarchy root. This is equivalent to the DIT of the

CK metrics. The deeper a module is in the inheritance hierarchy, the more operations it will inherit and therefore the more complex the module will be thus making it fault prone [34]. Studies have also shown that a system with less inheritance can be easier to modify and understand [34]. Hence a lower value of DIT is desired as it means that the system will be easier to maintain.

(iii) *Number of Children (NOC)*. This is the number of immediate subaspects or subclasses of a given module [8]. In contrast with the DIT metric which measures the depth in the system, the NOC metric measures the breadth of the system. A lower value of NOC is desired.

(iv) *Coupling on Field Access (CFA)*. This is the number of modules that have fields that are called by a given module. This metric measures the coupling between modules based on field access [8]. A higher value of CFA implies tight coupling between the modules which indicates complexity, increase in the module being fault prone, and also decrease in the testability [34]. A lower value of CFA is desired.

(v) *Coupling on Method Calls (CMC)*. This the number of modules declaring operations that are called by a given module. Similar to the CFA metric, this measures the coupling between modules based on operation access [8]. A lower value of CMC is desired.

(vi) *Coupling between Modules (CBM)*. This is the number of operations and fields that are accessed by a given class [34–36]. This can be represented by the number of outward arrows from a given module [36]. This metric is a combination of the CMC and CFA metrics as it measures the coupling based on both the field access (CFA) and operations (CMC). HA's lower value of CBM is desired.

(vii) *Crosscutting Degree of an Aspect (CDA)*. This is the number of modules affected by *pointcuts* and *introductions* in a given aspect. This measures the number of modules that are affected by an aspect [8].

(viii) *Coupling on Advice Execution (CAE)*. This is the number of aspects containing advices triggered by the execution of operations in a given module. This is the number of inward arrows from aspects to a particular module [36]. This metric is used to measure the dependence of the operation on the advices; hence a change in the advice might impact the operation [8]. A higher value of this metric for a given module means that the module is coupled with more aspects [36].

(ix) *Response for Module (RFM)*. This is the number of operations that are executed by a given class in response to the messages received by a given class [34–36]. This includes operations that are called both implicitly and explicitly [34]. This metric can be showed pictorially in a sequence diagram [36]. Modules with a higher RFM value are more complex and also their testing is more complicated [34, 36]. A lower value of RFM is more desired.

(x) *Lack of Cohesion of Operations (LCO)*. This metric measures the relationship between the methods in a given module [34]. High cohesion is desired in a system as it shows that the system is modularized. If a module implements several concerns, then the operations in that module will be accessing different fields and the LCO will be higher [8]. A lower value of LCO is desired for a software system.

While the concern level metrics require the developer to decompose the system into concerns (construct the concern implementation plans), the structural complexity metrics are based on already decomposed modules.

4. Results and Discussion

The refactoring was limited to a few concerns; therefore not all the concerns that are implemented in the components under consideration have been refactored. For the OpenBravoPOS system, the modules that were refactored are those implementing the following crosscutting concerns:

(i) session management,

(ii) logging,

(iii) exceptional handling.

For the Jasperreports component, the modules that were refactored are those implementing the following crosscutting concerns:

(i) synchronization,

(ii) object retrieval,

(iii) exceptional handling.

Tables 3 and 4 show the amount of code in OpenBravoPOS and Jasperreports components that implements the concerns that were refactored. The OpenBravoPOS component was implemented in over 53,000 lines of code and the concerns under consideration in this study were implemented in 5,313 lines of code indicating a line of code coverage of 9.94%. The concerns under consideration were also implemented in 43 modules which have a total of 625 operations. The number of operations represents all the operations in the modules implementing the concerns being considered. Alternatively, we could have reported the operations implementing the concerns under consideration, but there were some operations that implemented more than one concern being considered which would have resulted in having overlaps as the concerns were tangled.

In the Jasperreports component, 6.78% of the total lines of code which corresponded to 9,329 lines of code were assessed. This code is responsible for the concerns under consideration.

4.1. AOP and OO Versions of OpenBravoPOS and Jasperreports. Tables 5 and 6 show the complexity metrics for the OO and AOP versions of OpenBravoPOS and Jasperreports,

Table 3: Code coverage for the concerns in OpenbravoPOS.

	Code coverage: OpenBravoPOS		
	Total	Concern coverage	Coverage (%)
Lines of code	53,433	5,313	9.94
Number of modules	978	43	4.40
Number of operations (all)	5,741	625	10.89

Table 4: Code coverage for the concerns in JasperReports.

	Code coverage: Jasperreports		
	Total	Concern coverage	Coverage (%)
Lines of code	137,495	9,329	6.78
Number of modules	13,606	43	0.32
Number of operations	5,741	625	10.89

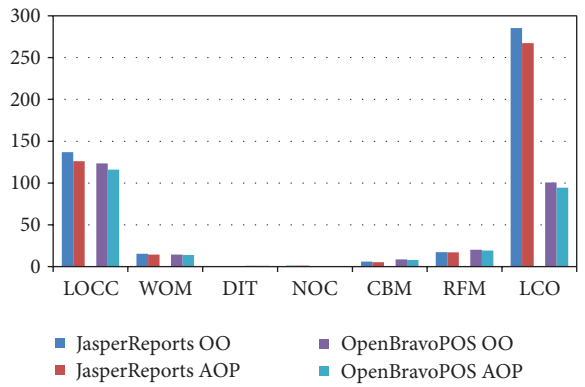

Figure 2: Comparison of the OO and AOP versions of OpenBravoPOS and Jasperreports.

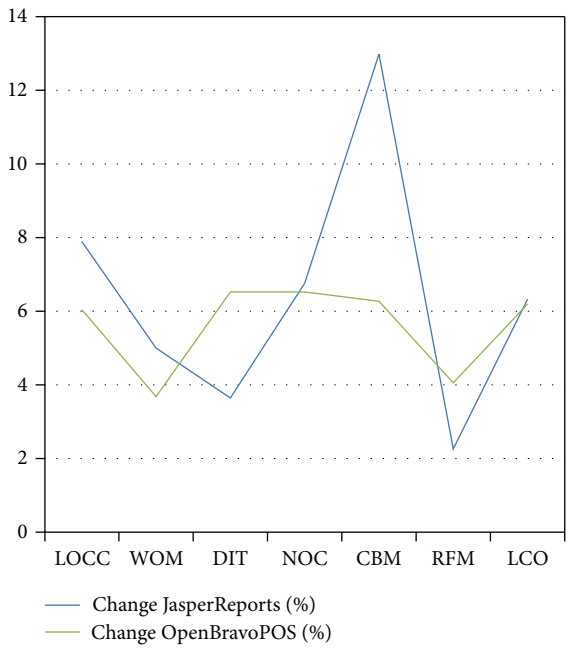

Figure 3: Comparison of the changes in OpenBravoPOS and Jasperreports following refactoring.

respectively. The values in Tables 5 and 6 are the average values of the metrics which are computed as the sum for the different modules divided by the number of modules under consideration.

Figures 2 and 3 indicate a comparison of structural metrics between the OO and AOP versions of OpenBravoPOS and Jasperreports.

The AOP versions of OpenBravoPOS and Jasperreports show an improvement in all the structural complexity metrics except for the CDA and CAE metrics which measure the coupling that is introduced by the aspects. CDA and CAE capture the AOP-specific coupling; hence they have a value of zero in the OO version while it shows an increase in the AOP version.

Figure 3 shows the comparison of the change in the metrics between the OpenBravoPOS and Jasperreports versions. The major improvement in the metrics is in the CBM metric for the Jasperreports component while for the OpenBravoPOS component it is for the NOC and DIT metrics. The CBM metric measures the OO coupling while the CDA metric measures the AOP-specific coupling. If the complexity of the AOP coupling is considered to be the same as the OO coupling, then the overall coupling which is obtained by adding the CBM and CDA values still shows an overall improvement in the coupling. For example, in the OpenBravoPOS component the improvement in CBM value is the .54 in the AOP version while the AOP coupling introduces a value of −0.15; hence the overall coupling will be 0.39. Burrows et al.'s [9] study showed that AOP coupling does not result in more fault-prone modules. Also, in a study by Przybyek [14], the CBM metric was redefined so as to capture both the traditional coupling and the AOP coupling and was able to measure the overall coupling in the AOP systems using a single metric. Based on the results shown above, the AOP implementations are less structurally complex as compared to their OO counterparts.

4.2. Concern Metrics. Tables 7 and 8 show the concern level metrics assessment of the OO versions of the OpenBravoPOS and Jasperreports components.

Table 5: Structural complexity metrics for the OpenBravoPOS original and refactored versions.

	OpenBravoPOS	OpenBravoPOS-AOP	Change	Change (%)
LOCC	123.56	116.09	7.47	6.05
WOM	14.53	14.00	0.53	3.68
DIT	1.16	1.09	0.08	6.52
NOC	0.49	0.46	0.03	6.52
CBM	8.65	8.11	0.54	6.27
CDA	0.00	0.15	−0.15	0.00
CAE	0.00	0.26	−0.26	0.00
RFM	20.26	19.43	0.82	4.05
LCO	100.79	94.54	6.25	6.20

Table 6: Structural complexity metrics for the Jasperreports original and refactored versions.

	Jasperreports	Jasperreports-AOP	Change	Change (%)
LOCC	136.87	126.07	10.80	7.89
WOM	15.38	14.61	0.77	5.00
DIT	0.43	0.42	0.02	3.65
NOC	1.32	1.23	0.09	6.76
CBM	6.09	5.30	0.79	12.97
CDA	0.00	0.64	−0.64	0.00
CAE	0.00	0.57	−0.57	0.00
RFM	17.45	17.05	0.40	2.27
LCO	285.36	267.31	18.05	6.33

For example, if we look at the concern for contract validation during the retrieval of the object in the concern "*RetrieveObject,*" it is implemented in 40 operations in 1 module. In contrast, the implementation of the concern in the AOP version shows that it has been implemented in 1 module and 1 operation over fewer lines of code; hence DOSC = 0, DOSM = 0, CDC = 1, and CDO = 1. The concern level metrics show that in the AOP implementation, the concerns are localized which is the main premise of AOP.

Table 9 shows the structural complexity metrics for the JRFillObjectFactory class in Jasperreports before and after the refactoring. After the refactoring, some parts of the code which deals with contract validation when retrieving objects was moved to the aspect and a new aspect called RetrieveObjectAspect was introduced in the AOP version. Table 9 shows that the refactoring resulted in reduction in the lines of code of this particular class which is 180 LOCC. This was in turn replaced by one module which is implemented in 12 LOCCs. In contrast, the concern level metrics shows that the concern is now implemented in 1 module and 1 operation.

4.3. Assessment of Changeability. To assess changeability of both the OO and AOP versions of the 2 components, we carried out 2 maintenance tasks. As mentioned earlier, maintenance task 1 refers to the introduction of a new feature to evaluate the execution time of SQL statements. Maintenance task 2 refers to the replacement of the logging component with another component of the same functionality.

4.3.1. Results of Maintenance Task 1. During this maintenance task, a feature was added so that the execution of the SQL statements against the database can be profiled; that is, see how long it takes for the execution of these statements. The impact of change or changeability was then assessed by counting the following code-level changes:

(i) number of modules added (MA),
(ii) number of modules removed (MR),
(iii) number of modules modified (MM),
(iv) number of operations added (OA),
(v) number of operations removed (OR),
(vi) number of operations modified (OM),
(vii) number of lines of code added (LA),
(viii) number of lines of code removed (LR),
(ix) number of lines of code modified (LM).

The results show that the change affected all the modules implementing a given concern (Table 10 and Figure 4). In the OpenBravoPOS implementation, the change affected 3 operations in 5 modules while the change affected 2 operations in 1 module in the AOP implementation. For the Jasperreports OO implementation, the change affected 10 operations in 2 modules while in the AOP implementation the change affected only 1 operation in 1 module.

The results indicate that the change impact is higher on the OO version as compared to the AOP version.

Table 7: Concern level metrics for the OpenbravoPOS component.

	OpenBravoPOS				
Concern name	DOSC	DOSM	CDC	CDO	SLOC
Session	0.903	0.96	34	62	518
SQLException	0.884	0.944	12	43	582
Logging	0.856	0.852	10	10	362

Table 8: Concern level metrics for the Jasperreports component.

	Jasperreports				
Concern name	DOSC	DOSM	CDC	CDO	SLOC
CloneNotSupportedException	0.965	0.964	37	36	779
RetrieveObject	0	0.974	1	40	494
Synchronization	0.977	0.977	43	43	473

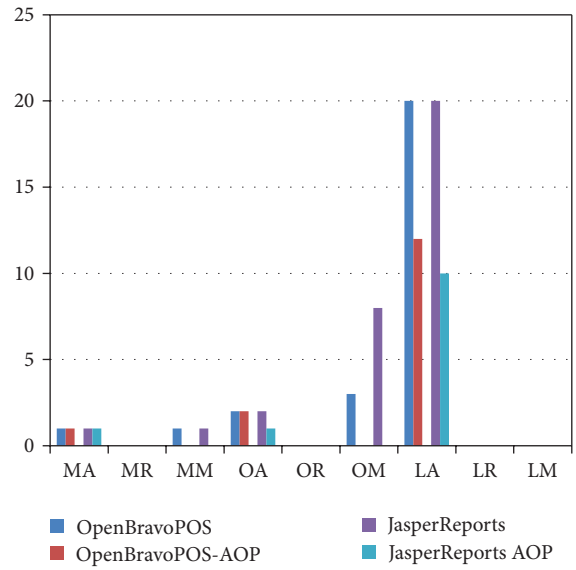

Figure 4: Comparison of the impact of change in OO and AOP versions of OpenBravoPOS and Jasperreports.

4.3.2. Results of Maintenance Task 2. A second maintenance task was carried out in the OpenBravoPOS to supplement the results of the first maintenance task. During this maintenance task, the implementation of the logging concern in OpenBravoPOS which uses java.util.logging was replaced with the Apache commons logging. The Jasperreports component also uses the commons logging component for implementing its logging features. The implementations that were used in carrying out this task were the original unmodified OpenBravoPOS component and its AOP counterpart. The logging concern had been refactored in the AOP implementation. The changeability results are in Table 11.

Table 11 shows that the impact of the change is higher in the OO implementation as compared to the AOP counterpart.

4.4. Threats to Validity. The result of the case study is encouraging in that it indicates the viability of the aspect-oriented approach in improving maintainability. This is in line with other similar studies that have been conducted in the past. However, we need to be cautious before we try to generalize the results of our case study.

(i) The sample size is relatively small for our case study and would be difficult to predict if the same result holds for a very large system. In addition, in our case study, only selected crosscutting concerns were refactored from the OO versions of OpenBravo and Jasperreports using the AOP language—AspectJ. It would be good to have a complete AOP version of the OpenBravoPOS system to compare the maintainability of the two systems. Moreover, our maintainability assessment is based on a single system (OpenBravoPOS) which makes generalization of results difficult. One needs to carry out more experiments on various systems from various domains to have a more credible generalization.

(ii) Another issue related to our case study is that the people who did the refactoring of the aspects from the components were the same people who did the measurement of the maintainability of the system. This may create a potential bias. However, we tried by all means to restrict the refactoring such that the goal was to refactor the aspects and not to improve the maintainability of the system.

(iii) The assessment of changeability is not comprehensive enough. Even though we used similar procedures to other studies, the types of changes still need improvement. For example, Grover et al. [37] provided code-level changes that can be made at system level and component level. In addition, it provides criteria for minimum number of random changes. We will use this approach in our future work.

(iv) The structural complexity metrics selected considered each line of code to be uniform even though the effort required to make a call to a local module is less expensive as compared to a call to an external component [38]. This is an interesting phenomenon for COTS-based systems as the maintenance developer will have to learn how to call a COTS component. This

Table 9: Structural complexity metrics for the Jasperreports component.

Type name	Type kind	LOCC	WOM	DIT	NOC	CBM	CDA	CAE	RFM	LCO	
JRFillObjectFactory	class	1058	81	1	2	68	0	0	92	0	OO
JRFillObjectFactory	class	878	81	1	2	68	0	1	93	0	AOP
RetrieveObjectAspect	aspect	12	1	0	0	0	1	0	0	0	AOP

Table 10: Changeability results after maintenance task 1.

	OpenBravoPOS-SQLProfiling								
	MA	MR	MM	OA	OR	OM	LA	LR	LM
OpenBravoPOS	1	0	1	2	0	3	20	0	0
OpenBravoPOS-AOP	1	0	0	2	0	0	12	0	0
	Jasperreports-SQLProfiling								
	MA	MR	MM	OA	OR	OM	LA	LR	LM
Jasperreports	1	0	1	2	0	8	20	0	0
Jasperreports-AOP	1	0	0	1	0	0	10	0	0

Table 11: Changeability results after carrying out maintenance task 2.

	OpenbravoPOS-Logging								
	MA	MR	MM	OA	OR	OM	LA	LR	LM
OpenbravoPOS	0	0	7	0	0	7	7	0	0
OpenbravoPOS-AOP	0	0	1	0	0	0	0	0	1

effect of the calls to a COTS component will also be investigated in future studies.

5. Conclusion and Future Work

In this paper, we have presented a case study for the assessment of the maintainability of an AOP-based COTS system by comparing it with its OO version of implementation. The COTS-based system used in our case study is OpenBravoPOS which is an open-source software used in retail business. For the assessment of maintainability, we used concern level metrics and structural complexity metrics. For both metrics, the results show that AOP-based implementation is more maintainable. In other words, most metrics show better maintainability values for the AOP version of implementation as compared to the OO version of implementation. In our case study, we refactored only 9.94% of the code for OpenBravoPOS and 6.78% for the Jasperreports. This might have an impact on the degree of maintainability improvement observed. Moreover, there are no threshold values specified for the values of each of the metrics in the literature. Had there been such threshold values, it would have been easier to see the acceptability of the improvement. This is one area for further investigation so that a maintainability index similar to the maintainability index of the traditional systems can be established for the AOP software.

In our case study, the concern level metrics are found to be good indicators of changeability of the system. For example, if 40 operations implement a given concern, when the concern is subjected to a maintenance task, we found that almost all these operations are to be affected. On the other hand, the structural complexity metrics are found to be poor indicators of changeability as they do not capture some of the issues related to changeability such as changing a line of code. For example, if a line of code calling component A was replaced with a call to component B, then the structural complexity metrics would not change.

References

[1] P. Jalote, *An Integrated Approach to Software Engineering*, Springer, Berlin, Germany, 3rd edition, 2005.

[2] I. Jacobson and P. -W. Ng, *Aspect-Oriented Software Development with Use Cases*, Addison-Wesley Object Technology Series, Addison-Wesley Professional, Reading, Mass, USA, 2004.

[3] X. Corporation, "The AspectJ Programming Guide," 2002-2003, http://www.eclipse.org/aspectj/doc/released/progguide/index.html.

[4] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," IEEE Std 610.12-1990, 1990.

[5] H. Al-Kilidar, K. Cox, and B. Kitchenham, "The use and usefulness of the ISO/IEC 9126 quality standard," in *Proceedings of International Symposium on Empirical Software Engineering (ISESE '05)*, pp. 126–132, November 2005.

[6] P. Oman and J. Hagemeister, "Construction and testing of polynomials predicting software maintainability," *The Journal of Systems and Software*, vol. 24, no. 3, pp. 251–266, 1994.

[7] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability—a preliminary report," in *Proceedings of the 6th International Conference on the Quality of Information and Communications Technology (QUATIC '07)*, pp. 30–39, IEEE Computer Society, September 2007.

[8] M. Ceccato and P. Tonella, "Measuring the effects of software aspectization," in *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE '04)*, Delft, The Netherlands, 2004.

[9] R. Burrows, F. C. Ferrari, A. Garcia, and F. Taïani, "An empirical evaluation of coupling metrics on aspect-oriented programs," in *Proceedings of the ICSE Workshop on Emerging Trends in Software Metrics (ICSE '10)*, pp. 53–58, ACM, Cape Town, South Africa, May 2010.

[10] M. Eaddy, T. Zimmermann, K. D. Sherwood et al., "Do crosscutting concerns cause defects?" *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497–515, 2008.

[11] A. Kumar, R. Kumar, and P. S. Grover, "An evaluation of maintainability of aspect-oriented systems: a practical approach," *International Journal of Computer Science and Security*, vol. 1, no. 2, pp. 1–9, 2007.

[12] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. Von Staa, and C. Lucena, "Quantifying the effects of aspect-oriented programming: a maintenance study," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pp. 223–232, IEEE Computer Society, September 2006.

[13] H. Shen, S. Zhang, and J. Zhao, "An empirical study of maintainability in aspect-oriented system evolution using coupling metrics," in *Proceedings of the 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE '08)*, pp. 233–236, IEEE Computer Society, June 2008.

[14] A. Przybyek, "Where the truth lies: AOP and its impact on software modularity," in *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering*, Springer, Saarbrcken, Germany, 2011.

[15] A. A. Kvale, J. Li, and R. Conradi, "A case study on building cots-based system using aspect-oriented programming," in *Proceedings of the 20th Annual ACM Symposium on Applied Computing*, pp. 1491–1498, ACM, Santa Fe, NM, USA, March 2005.

[16] L. P. Tizzei, M. Dias, C. M. F. Rubira, A. Garcia, and J. Lee, "Components meet aspects: assessing design stability of a software product line," *Information and Software Technology*, vol. 53, no. 2, pp. 121–136, 2011.

[17] M. Lippert and C. V. Lopes, "Study on exception detection and handling using aspect-oriented programming," in *Proceedings of the 22nd International Conference on Software Engineering*, pp. 418–427, ACM, Limerick, Ireland, June 2000.

[18] OpenBravo, "Point of Sale Retail," 2010, http://www.openbravo.com/product/pos/.

[19] A. S. Foundation, "Apache POI—The Java API for Microsoft Documents," 2012, http://poi.apache.org/.

[20] BVBA, T.X., 2012, http://itextpdf.com/.

[21] A. S. Foundation, 2012, http://commons.apache.org/logging/.

[22] Jaspersoft, 2012, http://jasperforge.org/.

[23] K. Mguni and Y. Ayalew, "Improving maintainability in COTS based system using aspect oriented programming: an empirical evaluation," in *Proceedings of African Conference of Software Engineering and Applied Computing*, pp. 21–28, IEEE Computer Society, Gaborone, Botswana, 2012.

[24] M. Storzer, U. Eibauer, and S. Schoeffmann, "Aspect mining for aspect refactoring: an experience report," in *Proceedings of the 1st International Workshop Towards Evaluation of Aspect Mining (TEAM '06)*, Nantes, France, 2006.

[25] M. Robillard, "FEAT: An Eclipse Plug-in for Locating, Describing, and Analyzing Concerns in Source Code," 2005, http://www.cs.mcgill.ca/~swevo/feat/.

[26] M. Eaddy, "ConcernTagger," 2007, http://www1.cs.columbia.edu/~eaddy/concerntagger/.

[27] Metrics, 2012, http://aopmetrics.tigris.org/.

[28] N. Fenton and A. Melton, "Deriving structurally based software measures," *The Journal of Systems and Software*, vol. 12, no. 3, pp. 177–187, 1990.

[29] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa, "Modularizing design patterns with aspects: a quantitative study," in *Transactions on Aspect-Oriented Software Development I*, pp. 36–74, Springer, Berlin, Germany, 2006.

[30] M. Trifu, *Tool-Supported Identification of Functional Concerns in Object-Oriented Code*, KIT Scientific Publishing, Karlsruhe, Germany, 2010.

[31] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, and A. V. von Staa, "On the reuse and maintenance of aspect-oriented software: an assessment framework," in *Proceedings of the 17th Brazilian Symposium on Software Engineering*, pp. 19–34, 2003.

[32] J. C. Taveira, J. Saraiva, F. Castor, and S. Soares, "A concern-specific metrics collection tzool," in *Proceedings of the Assessment of Contemporary Modularization Techniques at Object-Oriented Programming Systems and Applications (OOPSLA '09)*, Orlando, Fla, USA, October 2009.

[33] S. R. Chidamber and C. F. Kemerer, "Metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[34] S. K. Dubey and A. Rana, "Assessment of maintainability metrics for object-oriented software system," *SIGSOFT Software Engineering Notes*, vol. 36, no. 5, pp. 1–7, 2011.

[35] U. L. Kulkarni, Y. R. Kalshetty, and V. G. Arde, "Validation of CK metrics for object oriented design measurement," in *Proceedings of the 3rd International Conference on Emerging Trends in Engineering and Technology (ICETET '10)*, pp. 646–651, Karjat, India, November 2010.

[36] C. Babu and R. Vijayalakshmi, "Metrics-based design selection tool for aspect oriented software development," *SIGSOFT Software Engineering Notes*, vol. 33, no. 5, pp. 1–10, 2008.

[37] P. S. Grover, R. Kumar, and A. Kumar, "Measuring changeability for generic aspect-oriented systems," *SIGSOFT Software Engineering Notes*, vol. 33, no. 6, pp. 1–5, 2008.

[38] A. Endres and H. Rombach, *A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories*, Fraunhofer IESE, Kaiserslautern, Germany; Pearson Addison-Wesley, Harlow, England, 2003.