Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study

Uirá Kulesza[1] Cláudio Sant'Anna[1,2] Alessandro Garcia[2]
Roberta Coelho[1] Arndt von Staa[1] Carlos Lucena[1]

[1]Software Engineering Laboratory – Computer Science Department – PUC-Rio – Brazil
{uira, claudios, roberta, arndt, lucena}@inf.puc-rio.br

[2]Computing Department – InfoLab 21 – Lancaster University – UK
{garciaa}@comp.lancs.ac.uk

Abstract

One of the main promises of aspect-oriented programming (AOP) is to promote improved modularization of crosscutting concerns, thereby enhancing the software stability in the presence of changes. This paper presents a quantitative study that assesses the positive and negative effects of AOP on typical maintenance activities of a Web information system. The study consists of a systematic comparison between the object-oriented and the aspect-oriented versions of the same application in order to assess to what extent each solution provides maintainable software decompositions. Our analysis was driven by fundamental modularity attributes, such as coupling, cohesion, conciseness, and separation of concerns. We have found that the aspect-oriented design has exhibited superior stability and reusability through the changes, as it has resulted in fewer lines of code, improved separation of concerns, weaker coupling, and lower intra-component complexity.

1. Introduction

Aspect-oriented programming (AOP) [11] is a post-OO paradigm with the goal of enhancing software maintainability through new modularization mechanisms for encapsulating crosscutting concerns. However, it is not clear up to now how the aspectual decompositions scale in realistic maintenance scenarios involving more than one widely-scoped aspect, such as distribution and persistence. There is also no empirical evidence whether AOP improves the ability to preserve the architecture stability as the system evolves, thereby hindering the adoption of AOP in the development mainstream. The challenge is that system changes commonly manifest themselves in heterogeneous forms; they involve not only simple modifications in a single implementation module, but also encompass introductions of new use cases that naturally impact several classes and aspects implementing the coarse-grained architectural decisions.

In fact, while AOP [11] is fast gaining wide attention in both research and industry environments, the understanding of its impact on key maintainability-related software attributes are still deep challenges to software engineers. There is now some systematic case studies in the literature that analyses how AOP promotes superior separation of concerns in the implementation of crosscutting features, such as distribution [16, 17, 18], persistence [13, 14, 18], exception handling [6], and design patterns [3, 9]. However, they have not analyzed the scalability of AOP in the presence of widely-scoped design changes.

This paper presents a systematic case study in which we have compared the maintainability of aspect-oriented (AO) and object-oriented (OO) architectures of a typical web-based information system. Our investigation complements the existing empirical body of knowledge on the use of AOP, since our two-phase evaluation has respectively quantified: (i) the effects of AOP on the achievement of separation of concerns and other equally important maintainability attributes, such as low coupling, high cohesion, design simplicity, and conciseness; and (ii) the scalability of both AO and OO solutions with respect to the same attributes used in (i) while implementing a set of pervasive, broadly-scoped design modifications in the target system. The design of the case study is mainly structured according to a layered software architecture. It involves three classical crosscutting concerns – distribution, persistence, and concurrency – and other elementary non-crosscutting concerns, such as business and GUI elements. In order to better understand the positive and negative effects of AOP in the selected maintenance scenarios, our analysis was performed according two different architecture levels: at the system-level and at the "layer"-level.

The paper is organized as follows. Section 2 shows the design of the target web-based system. Section 3 presents the study setting. Section 4 presents the general system-level results. Section 5 describes a detailed analysis of the layer-level data. Section 6 provides more general discussion about the impact of AOP on maintainability. Section 7 discusses related work. Section 8 includes some concluding remarks.

2. Overview of the System Design

In our study, we have compared the AO and OO implementations of a same web-based information system, called HealthWatcher (HW). The main purpose of the HW system is to allow citizens to register complaints to the health public system. This system was the ideal for our case study due to several reasons. First, it has been developed out of our research environment. Both original AO and OO versions of the HealthWatcher system were developed by the Software Productivity research group from the Federal University of Pernambuco. Second, a preliminary qualitative assessment has been recently conducted and reported [16]. It has allowed us to supplement the qualitative focus on separation of concerns of the first study with both a broader quantitative analysis and a systematic investigation about the scalability of AOP in software maintenance scenarios.

Finally, it is a realistic system that involves a number of common concerns, such as GUI, persistence, concurrency, and distribution; it also encompasses the application of mainstream technologies commonly used in industrial contexts, such as Java Remote Method Invocation (RMI), Servlets, and Java Database Connectivity (JDBC). Fourth, both OO design (Section 2.1) and AO design (Section 2.2) of the HW system were developed with modularity and changeability principles as main driving design criteria. Each design choice for both OO and AO solutions have been deeply discussed and documented elsewhere [16, 17, 18].

2.1. Object-Oriented Design

The OO version of the HW system is implemented using the Java programming language. The Layer [2] architectural pattern is used to structure the system classes in four main layers: GUI (Graphical User Interface), Distribution, Business and Data. Figure 1a presents a partial class diagram of the OO implementation, illustrating the main architectural elements. The GUI layer implements a web user interface for the system. The Java Servlet API is used to codify the classes of this layer. The Distribution layer is responsible for making distributed the system services provided by the Business layer. It is implemented using the RMI technology. The Business layer aggregates the classes that define the system business rules. Finally, the Data layer defines the functionality of database persistence using the JDBC API. Also, several design patterns [1, 11, 16, 17] are used in the design of the HW layers to achieve a reusable and maintainable implementation.

The aforementioned design decisions have shown to be effective to modularize most of the driving system concerns: the graphical user interface, distribution, business, and data access concerns. However, code relative to some distribution, persistence, concurrency issues still remain spread and tangled in the system modules. Figure 1a illustrates how some of these concerns crosscut the coarse-grained structures in the existing OO implementation of the system, such as [16, 17, 18]:

- the need to make serializable the entity system classes (the Complaint and Employee classes, for example) in order to allow them to be transmitted over the network;
- the transparent configuration of GUI layer servlet classes to make it possible the remote access of the business services using the Distribution layer;
- existing transaction demarcation code in methods of the HealthWatcherFacade business class;
- initialization of a persistence mechanism that manages database initialization and connections;
- transparent configuration of the business classes to use persistent or nonpersistent data access classes;
- implementation of concurrency control mechanisms (such as, timestamp or code synchronization) in business and data classes.

2.2. Aspect-Oriented Design

The AO version of the HW system was implemented using AspectJ [12]. The design followed the same principles of reusability and maintainability of the OO version, modularizing the same main concerns of interest. The only difference was that the AO design was conceived to also isolate the crosscutting issues relative to distribution, persistence, and concurrency (Section 2.1), which naturally could not be separated in the OO system version. The AO implementation is still structured following the Layer architectural pattern. However, only the Distribution concern is no longer implemented as a "layer" (Fig 1b). Aspects are the abstractions used to implement this concern.

Figure 1b shows the design of the AO system version. An UML stereotype <<aspect>> is used to represent the aspects of the system. Moreover, UML dependency relations with the <<crosscuts>> stereotype indicate that an aspect introduces or modifies the structure and/or behavior of system classes. As we can see in the Figure 1b, different

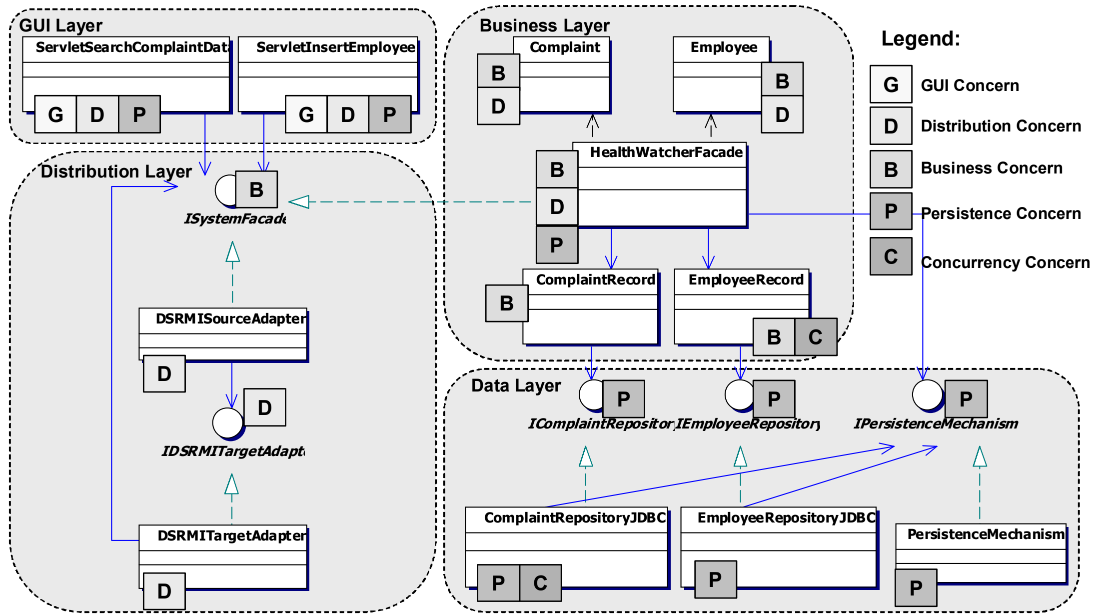


Figure 1: (a) HealthWatcher Object-Oriented Implementation

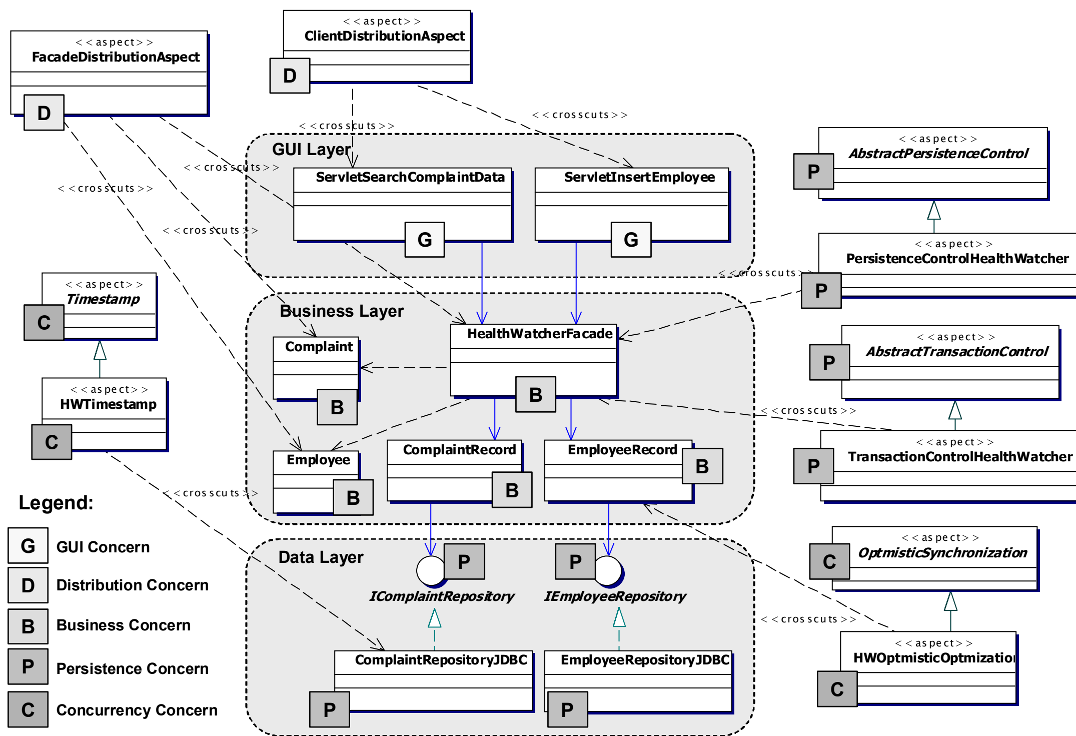


Figure 1: (b) HealthWatcher Aspect-Oriented Implementation

aspects modularize the crosscutting concerns existing in the OO implementation. Due to space limitation, for complete implementation descriptions refer to [16-18].

3. Study Setting

Following the selection of the system to be assessed, our quantitative study proceeded according to several steps: (i) definition of the assessment criteria; (ii) selection of the software metrics; (iii) execution of important assessment procedures; (iv) data collection; and (v) data analysis. In the following sections, we respectively describe the assessment criteria and metrics, and the most relevant assessment procedures.

3.1. The Metrics

In our study, a suite of metrics for separation of concerns (SoC), coupling, cohesion and size [15] was selected to evaluate the OO and AO implementations of the HW system. We have decided to focus on a restrict set of measures that are typically used to evaluate maintainability. The chosen metrics have already been successfully used in several case studies [3, 6, 8, 9, 10]. This metrics suite was defined based on the reuse and refinement of some classical and OO metrics [4]. Our assessment framework also encompasses new metrics for evaluating SoC dimensions. These metrics capture the degree to which a single system concern maps to the design components (classes and aspects), operations (methods and advice), and LOC. Table 1 briefly defines each metric, and associates it with the relevant software attribute.

3.2. Assessment Procedures

The study was organized in two phases: (i) assessment of the original implementations - the measurement and analysis of the original OO and AO versions for the HW system; and (ii) implementation and assessment of the evolved implementations. Both original versions implement the total of 13 use cases, presented in Table 2, related to the system domain. In the maintenance phase of our study, we changed both OO and AO architectures of the HW system to address a set of new 8 use cases, also showed in Table 2.

The functionalities introduced by these new use cases represent typical operations encountered in the maintenance of information systems. We have selected them because they naturally involve the modification of modules implementing several system concerns. It has allowed us to evaluate the degree to which both solutions scale in the presence of change scenarios not restricted to punctual modifications in the classes and/or aspects. All the new use cases required changes in the classes pertaining to the 4 layers of both system versions. With respect to the aspects, all the new use cases demand small changes in the distribution and persistence aspects, and the 4 new use cases related to insertion of system entities demand changes in the concurrency aspects..

In the measurement process of both original and maintenance HW versions, the data was partially gathered by the AJATO tool [5]. The data collection relative to the SoC metrics is preceded by the shadowing of every class, interface and aspect in both implementations of the system. Their code was shadowed according to the crosscutting concerns – distribution, persistence, and concurrency – that they implement. We treated these concerns as the issues driving the assessment because both designs and implementations of the HW system were motivated to separate them. We present the results of our evaluation process by describing the overall measures for the system viewpoint (Section 4), and the layer viewpoint of view (Section 5).

4. Results: The System Viewpoint

Tables 3 and 4 present the collected absolute values for all the metrics considering both AO and OO versions before and after their maintenance. Figures 2 and 3 compare the results obtained for the AO and OO implementations both before and after the introduced changes. The first column of the figures present the data gathered in the first phase – i.e. before the maintenance scenarios, and the second one describes the measures for the second phase. The measures shown in the graphics were gathered according to the system perspective; that is, they represent the tally of metric values associated with all the classes and aspects for the system implementation. The Y-axis presents the percentage relative to the absolute value of the system considering each metric. Each pair of bars, presented in the graphics, is attached to a percentage value, which represents the difference between the AO

Table 1. The Metrics Suite

Attributes	Metrics	Definitions
Separation of Concerns	Concern Diffusion over Components (CDC)	Counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them.
	Concern Diffusion over Operations (CDO)	Counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them.
	Concern Diffusion over LOC (CDLOC)	Counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch".
Coupling	Coupling Between Components (CBC)	Counts the number of other classes and aspects to which a class or an aspect is coupled.
	Depth Inheritance Tree (DIT)	Counts how far down in the inheritance hierarchy a class or aspect is declared.
Cohesion	Lack of Cohesion in Operations (LCOO)	Measures the lack of cohesion of a class or an aspect in terms of the amount of method and advice pairs that do not access the same instance variable.
Size	Lines of Code (LOC)	Counts the lines of code.
	Number of Attributes(NOA)	Counts the number of attributes of each class or aspect.
	Weighted Operations per Component (WOC)	Counts the number of methods and advice of each class or aspect and the number of its parameters.

and OO results. A positive percentage means that the AO implementation was superior, while a negative percentage implies that the AO implementation was inferior. As it can be observed in Figures 2 and 3, the results of the measurement process show favorable results for the AO implementation with respect to the majority of the metrics used.

Table 2. HealthWatcher Use Cases

System Version	Use Cases
Original	- Insert, Update, Search Employee - Insert, Update, Search Complaint - Update and Search list of Health Unit - Search Specialties by Health Unit - Search Health Units by Specialty - Search List of Specialties - Search Disease Type - Search list of Disease Type
Maintenance	- Insert and Search HealthUnit - Insert and Search Symptoms - Insert and Search Medical Specialty - Insert Disease Type - Search list of Symptoms

4.1. Quantifying Separation of Concerns

The application of the SoC metrics was useful to quantify how effective was the separation of the distribution, persistence and concurrency concerns in the AO implementation of the system (Figure 2). Since the main objective of the AO solution [16, 17, 18] was to provide the isolation of these crosscutting concerns using AspectJ, we could expect superior SoC outcomes in favour of the AO implementation. In fact, all the measures in Figure 2 confirm our hypothesis: the AO implementation exhibits better results with respect to all the concerns investigated. The graphics show significant differences in favor of the AO implementation in terms of the concern diffusion over components (CDC), over operations (CDO) and over lines of code (CDLOC). There are cases where the superiority of the AspectJ solution is higher than 50%.

The distribution concern, for example, is spread over 35 components (classes and aspects) in the OO implementation, while in the AO solution it is only involves 6 components. The CDC metric in Figure 2a shows the percentage difference between both versions. Also, the distribution concern presents better results for the CDO metric. It is scattered over 98 operations (methods and advices) in the OO solution, while over only 41 operations contains code relative to distribution in the AO solution. Finally, the distribution concern is more tangled in the OO solution than in the AO implementation (CDLOC metric). The OO solution presents 77 “concern switches” over the system code, while the AO solution brings only 1, because the aspects fully modularize the distribution concerns. Thus, this difference also reflects the superiority of the AO solution in terms of the CDLOC metric.

The AO implementation after the system changes also exhibits better results for all the SoC metrics compared to the OO version. In some cases, the percentage difference between both versions is increased after the maintenance activities. For example, the CDO metric for both the persistence and concurrency concerns. It is also interesting to observe that the percentage differences between the AO and OO systems after the maintenance scenarios are relatively the same as before the changes for the distribution and persistence concerns. Figure 2 shows that the AspectJ solution has scaled well with respect to separation of concerns. The concurrency concern revealed a different situation: the superiority of AspectJ was even higher in the maintenance phase. The reason was that the introduced functionalities required the implementation of a number of extra synchronization behaviors. These behaviors were successfully captured by the concurrency aspects, but they were replicated over several methods in the OO version.

Table 3. Collected Values for the Separation of Concerns Metrics

Concern		Distribution			Persistence			Concurency		
Metric		CDC	CDO	CDLOC	CDC	CDO	CDLOC	CDC	CDO	CDLOC
Before Maintenance	OO	35	98	77	56	286	353	16	36	85
	AO	6	41	1	35	206	17	11	30	1
After Maintenance	OO	43	145	115	60	372	503	23	70	163
	AO	6	62	1	36	233	19	11	38	1

Table 4. Collected Values for the Coupling, Cohesion and Size Metrics

Metric		Coupling		Cohesion	Size			
		CBC	DIT	LCOO	VS	LOC	NOA	WOC
Before Maintenance	OO	517	145	767	89	6239	148	1003
	AO	495	142	838	96	5521	143	1015
After Maintenance	OO	728	158	814	102	7597	171	1214
	AO	687	155	960	109	6685	161	1227

4.2. Quantifying Coupling, Cohesion and Size

We have also analyzed how the AO implementation has impacted positively or negatively on the coupling, cohesion and size measures in comparison with its OO implementation. Figure 3 presents graphics with the results for these metrics for both original and evolved system versions. The graphic structures are similar to the ones in Figure 2, with the exception that Figure 3 also highlights the contribution of the aspects in the overall system measures. For example, aspectual modules consist of 20% of the total number of components (VS metric) in the original AspectJ implementation (Figure 3a) and 18% in the evolved version (Figure 3b). Both graphics show that the AO implementation exhibits better results for many of the metrics, such as: the lines of code (LOC), number of attributes (NOA), and both the coupling metrics (CBC and DIT). On the other hand, the OO implementation brings better results for the vocabulary size (VS) and cohesion (LCOO) metrics. Both AO and OO implementations present similar results for the WOC metric.

The VS metric in Figure 3a shows that the AO implementation needed to define 7% more components (classes + aspects) than the OO version. In fact, the AO version involved 96 components while the OO implementation included only 89 components to implement the same functionalities. These differences are justified by the presence of several new aspects in the AO implementation of the system which are used to (Figure 1b): (i) modularize persistence and concurrency crosscutting concerns encountered in the implementation of the Business and Data system layers; or (ii) replace the original OO implementation of the Distribution layer.

Figure 3a also shows that there is a small difference in favor of the AO implementation with respect to the absolute value of the coupling metric (CBC). It happens mainly because, although many of the aspects reduce the coupling of system classes by modularizing their respective crosscutting concerns, they still need to hold references to the classes in which they introduce some state or behavior. But considering the AO implementation has more components (classes and aspects) demonstrated by the VS metric, we can observe that it has produced more decoupled classes and aspects. In addition, Figure 3b shows that the contribution of the aspectual modules in the overall system coupling has relatively decreased after the maintenance changes, showing a satisfactory stability of the AO design.

The DIT (depth inheritance tree) and NOA (number of attributes) metrics have presented similar results in the AO and OO implementations of the system, as shown in Figure 3a. For some system layers, such as the Distribution (see figures 4c and 4d), the DIT value has been reduced significantly, because the AO implementation does not explore a complex class hierarchy as in the OO solution. The DIT value of the AO solution is compensated by the creation of several aspect hierarchies that enable the reuse of crosscutting concerns implementations, thereby decreasing the number of "extension dependencies" and reducing code replication in the class hierarchies. In fact, the two coupling dimensions were lower in the AspectJ solution, which tended to present both weaker inter-component coupling (CBC) and weaker inheritance coupling (DIT).

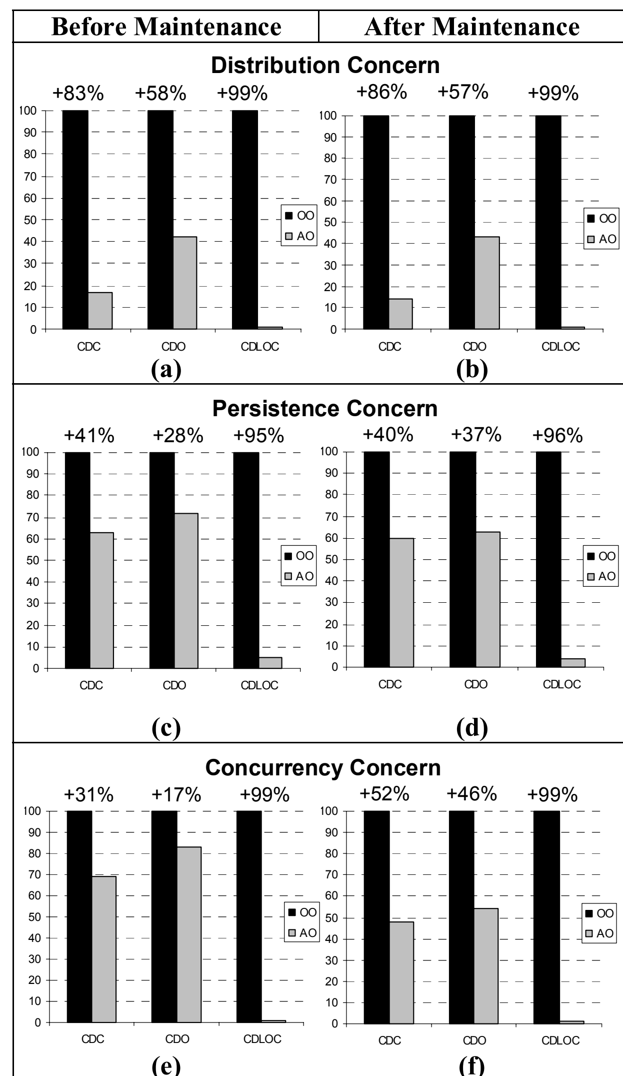
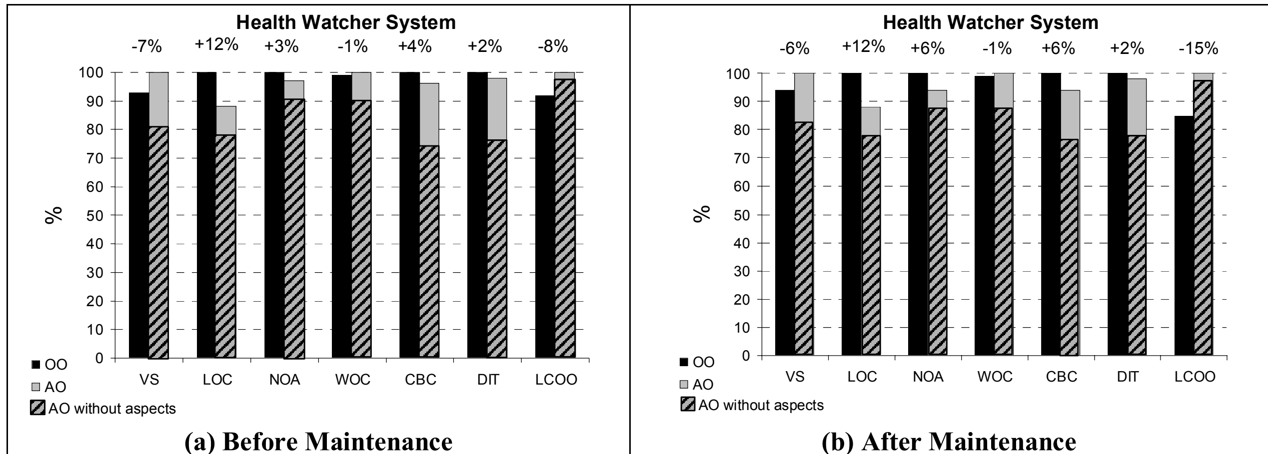


Figure 2. Separation of Concerns Metrics



(a) Before Maintenance

(b) After Maintenance

Figure 3. Coupling, Cohesion and Size Metrics

The AO implementation of HealthWatcher also exhibits better results with respect to LOC. Figures 3a shows that the AO implementation was 12% superior in the absolute value. This result confirms that the AO implementation has succeeded to capture common and redundant code of scattered and tangled concerns in the OO implementation. Thus, this quantitative study reinforces the observations of Soares et al [16] related to the LOC gains in the AspectJ implementation.

Figures 3a shows the superiority of the OO implementation with respect to the cohesion metric (LCOO). The OO solution was 8% superior in the absolute value. The production of components with a lower cohesion was a side effect in the AO implementation. In all layers of the AO version, there is at least one aspect that contributes to decrease the cohesion of the system, i.e. increase the value of LCOO. The lack of cohesion in these aspects occurs because each aspect is meant to encapsulate crosscutting behavior applied to different components. However, these behaviors cannot be directly related to each other, producing high LCOO values.

Finally, the WOC (weighted operations per component) measures also exhibit a similar absolute value for both OO and AO solutions (Figure 3a). Although the AO implementation presents a lot of new aspects with their respective new advice and methods which contribute to the growth of the WOC metric values, the implementation of some classes and interfaces existing in the OO version are simplified in the AO implementation. The distribution implementation, for example, replaces a set of repetitive classes and interfaces from the Distributed Adapters pattern in the OO version [1] by a set of aspects in the AO version. This design helps to reduce the WOC absolute value in the AO implementation.

As we can see in Figure 3b, there are few changes in the absolute values of both OO and AO versions after the implementation of new case cases in the maintenance phase. The percentage differences are very similar to the values computed for the original system versions. For some metrics, such as NOA and CBC, we can also perceive some reduction in the percentage difference. The results demonstrate the superiority of the AO implementation for many of the metrics used in our study even in the presence of maintenance activities. Also the higher number of components (VS metric) in the AspectJ implementation should not be viewed as a negative factor, since it is only an evidence of increased modularity of the system concerns. We provide further discussions about the cohesion issue in Section 6.3.

5. Results: The Layer Viewpoint

This section presents analysis of the metrics data by considering the main system layers: GUI, Distribution, Business, and Data. This analysis is important to understand the impact that the separation of concerns provided by the AO implementation has brought to each layer with respect to coupling, cohesion, and size attributes. The analysis also allows to quantify which layers in the OO and AO solutions have exhibited better results with respect to the metrics. Due to space limitation, we do not present all the data and graphics of the absolute values for each of the metrics considering the system layers. They can be found in [19].

5.1. Graphical User Interface (GUI)

The OO implementation of the GUI layer has exhibited better results than the AO solution for many of the metrics considering their absolute values. The AO solution was superior only with respect to the LOC and CBC metrics. The only aspects codified for this

layer address the handling of distribution- and persistence-related exceptions. These exception handling aspects allow to modularize the repetitive code related to distribution and persistence exceptions existing in the code of Java servlets of the GUI layer. This is the reason for the improvement in the LOC and CBC metrics for the AO solution. On the other hand, these aspects are also responsible for the increase in the values collected for the other metrics.

In the maintenance phase, we observed a decrease in the percentage difference between OO and AO versions. It happens because the exception handling aspects did not need to be modified, they already have a generic implementation which address the exception handling related to persistence and distribution for the new servlets introduced. Also, there are new improvements related to the LOC and CBC metrics. In this way, the AO solution for this layer tends to improve when new use cases (servlets) are implemented.

5.2. Distribution

The AO implementation of the Distribution layer has exhibited better results for almost all the coupling and size metrics considering their absolute values. The only exception was the cohesion metric, which will be discussed in Section 6.3. One of the main benefits of the AO solution is to avoid the extensive class hierarchy provided by the OO solution based on the Distributed Adapter pattern [1]. Many classes and interfaces are specified in the OO solution to guarantee the transparent distributed communication between GUI and Business layers. They maintain a lot of repetitive code which contributes to the increase of the size and coupling metrics. In the AO solution, aspects are used: (i) to advise calls to the business facade class in the servlet client classes; and (ii) to redirect these calls to a distributed version of the system. The results showed that the AO solution for the Distribution layer improves considerably not only the SoC metrics but also the other coupling and size attributes. The percentage superiority of the evolved AspectJ version is mostly the same as in the original implementations.

5.3. Business

The OO solution presents better absolute values for most of metrics considering the Business Layer. Only the LOC and NOA metrics exhibit better results for the AO solution. The reduced value for the LOC metric in the AO implementation reflects how effective the aspects have modularized repetitive persistence and concurrency crosscutting concerns encountered in the OO implementation related mainly to: (i) transaction demarcation; and (ii) synchronization protocols. The modularization of these concerns brings the need to have more aspects to address them (VS metric), with their respective new operations (WOC metric) and coupling with other elements (CBC and DIT metrics). Thus, there is a initial cost associated with the separation of these concerns in the Business Layer.

Looking at the same measures after the maintenance activities, we observed an improvement in the metrics for the AO version. The VS, LOC, NOA, WOC and CBC metrics of the AO solution show improvements in the percentage differences compared to the OO solution. It happens because the AO solution reused the general implementation of the transaction and concurrency policies in the new maintenance scenarios, implemented as a set of abstract aspects. However, each use case of the maintenance phase demanded the writing of less lines of code related to the distribution, persistence, and concurrency concerns, which were modularized in the aspects. At the end, more effort was required to change the OO version for most the measures.

5.4. Data

The analysis of the measures obtained for the Data layer for both original and maintenance versions exhibits better results for the OO version. The use of aspects in the AO solution of the Data layer has succeeded in the separation of concurrency and persistence concerns, but it has produced a larger layer in terms of size metrics (VS, LOC, NOA and WOC metrics). The increase of these values was caused by the implementation of concurrency aspects for this layer. The AO implementations present better results only for the CBC metric. The reason for this inferior value in this metric was the implementation decision to use only one exception class to represent both persistence and concurrency exceptions in every data access class.

The metric data collected for the Data layer after the maintenance of the system shows the increase in the percentage differences compared to the values obtained in the original system versions. This increase favors the OO version for the LOC, WOC and CBC metrics. The degradation of the AO version in terms of these metrics is caused by the implementation of the timestamp concurrency policy in the `HWTimestamp` subaspect (Figure 1b). The maintenance scenarios demanded the creation of several AspectJ inter-type constructions in this aspect, which are responsible to introduce methods for the management of timestamps in the different data access classes.

6. Discussions and Lessons Learned

This section presents an overall analysis of the previously observed results on the application of metrics, described in Sections 4 and 5. We present

discussions on the impact of AOP in different maintainability facets of the HealthWatcher system.

6.1. Aspects Reuse and Modifiability

We have observed that the presence of reusable aspects brought some benefits when the system was modified. In the AO version of the HW system, there is a set of abstract and reusable aspects related to the persistence, concurrency, and error handling concerns. These aspects have contributed to the decrease in the lines of code of the final system. These benefits can be observed in the complete analysis of the system with the decrease of 12% in LOC for both AO original and maintenance versions. The specific analysis of the GUI and Business layers (Sections 5.1 and 5.3) also showed how these reusable aspects contribute to amortize the initial cost of these aspects in several metrics (such as WOC, DIT, and CBC) along the system maintenance.

The positive values of SoC metrics and the existence of several reusable aspects also contribute to facilitate the maintenance of the AO implementation. An additional interesting observation is that more components (classes and aspects) were needed to be modified in the AO version, because it requires changing both the classes along the layers to implement the use case functionality and the aspects implementing the crosscutting issues.

6.2. Stability of Aspectual Modules

In our study, the existing aspects from the original version were not modified in the maintenance scenarios. Hence, this study seems to confirm the natural intuition that the additional design effort to “aspectize” the crosscutting behaviors is compensated by the reduced effort spent in maintenance scenarios involving the target system. It happened because the stability of the base layered architecture is preserved more in the AO version when addressing the new use cases. In our investigation, the layered architecture of the Java systems has been somewhat degraded at the implementation level with the introduction of the new functionalities. The concern associated with each layer is progressively diffused over the other layers during the maintenance process, as it is evidenced in the SoC metrics (e.g. see Figure 2).

6.3. Cohesiveness

As illustrated in the previous sections, cohesion seems a major problem in the AO implementations for all the system layers. In fact, cohesion is always a polemic issue as we have identified similar problems in previous studies [3, 6, 8, 9] in which the implementation of the aspects did not contain internal fields. In this sense, we believe that the cohesion measures are not directly conclusive as the LCOO metric (Table 1) focuses on a specific cohesion dimension; it counts the explicit relations between internal component fields and operations. While looking at Figure 3 and discarding the influence of aspects on the overall cohesion measures, the OO implementation is still superior even after the changes have been introduced (Figure 3b). As a consequence, more empirical studies using different cohesion metrics need to be performed in order to infer more broad conclusions. On the other hand, the SoC metrics used in our study can also assess the cohesion dimension related to specific system concerns. The CDLOC metric, for example, shows how different pieces of code of the system are directly related to a specific concern.

6.4. Scalability of AOP

In order to analyze the scalability of both OO and AO versions in the maintenance phase when referring to distribution, persistence and concurrency, we have used the collected values for the SoC metrics. We considered a solution as scalable if the evolution of the implementation did not impact a number of modules that is higher than the number of modules affected in the original implementation. Comparing the results obtained in the original and maintenance versions, we can observe that the AO version was much more scalable than the OO solution. The increase in the percentage differences between both versions (Fig. 2) demonstrates how the AO solution has required fewer changes in modules than the OO when referring to distribution, persistence and concurrency concerns. For example, the CDC values shows that the OO solution required changes in more components (8 additional classes for distribution, 4 for persistence and 7 for concurrency). On the other hand, the AO solution did not require changes in additional classes or aspects for the distribution and concurrency concerns, and required only the additional `SymptomRepositoryRDMBS` class for the persistence concern. This class, however, was introduced only after maintenance activities.

6.5. Study Constraints

Someone could argue that we have not assessed all the possible internal software attributes affecting the system maintainability. However, on the basis of prior research on empirical software engineering, we were able to identify four relevant attributes that seem underlie most of the quantitative case studies: coupling, cohesion, size, and SoC. Practitioners and researchers can add other assessment elements to customize the criteria to particular settings and further case studies. In addition, as discussed in Section 3.1, we have decided to focus on the metrics previously described because they have already been proved to be useful as effective quality indicators in several case

studies [3, 6, 8, 9, 10]. Also, strictly speaking, the scope of our experience is indeed limited to the system chosen this study, and the Java and AspectJ languages.

7. Related Work

There is little related work focusing either on the quantitative assessment of AO solutions in general, or on the empirical investigation of how AOP scales up in maintenance scenarios. Substantial empirical evidence is missing even for crosscutting concerns that software engineers face every day, such as persistence and distribution. There are several case studies in the literature involving the "aspectization" of such pervasive crosscutting concerns [13, 14, 16, 17]. However, these studies mainly focus on the investigation on how the use of aspect-oriented abstractions supports the separation of those concerns. They do not analyze other effects and stringent quality indicators in the resulting aspect-oriented systems. Even worst, they do not quantify the benefits and drawbacks of AO techniques in the presence of widely-scoped changes. We have previously performed a far-reaching maintenance study [8], but our target was aspects specific to multi-agent systems. These aspects have a localized scope and tend to affect a few modules; they do not have a major influence on the architectural structuring of the system. In addition, the introduced changes were restricted to simple changes in some few classes or aspects.

8. Conclusions

This paper presented a far-reaching study in which we compared AO and OO implementations of a typical web-based information system with respect to primary maintainability attributes. We have found that although the number of operations and components has slightly increased with the use of AOP, various flavors of our study show that the overall quality of the AO system was significantly superior at the system and component levels. The use of AOP required fewer lines of code, helped to achieve an improved separation of concerns, exhibited components with weaker coupling and lower internal complexity. However, a lower cohesion was a side effect in the AO solution mainly because some aspects were not aggregating inter-related behaviors. As a consequence, architectural stability was clearly superior in the AO architectural design of the target system.

Acknowledgements. We would like to thank Sérgio Soares and Paulo Borba for making the HW implementations available. This research was partially sponsored by FAPERJ (grant No. E-26/151.493/2005 and No. E-26/100.061/06), and European Commission Grant IST-2-004349: European Network of Excellence on AOSD (AOSD-Europe).

9. References

[1] V. Alves, P. Borba. Distributed Adapters Pattern: A Design Pattern for Object-Oriented Distributed Applications. Proc. SugarLoafPLoP.01, Rio de Janeiro, October 2001.

[2] F. Buschmann, et al. Pattern-Oriented Software Architecture: A System of Patterns. 1996: Wiley and Sons.

[3] N.Cacho, C. Sant'Anna, E.Figueiredo, A.Garcia, T.Batista, C. Lucena. Composing Design Patterns: A Scalability Study of AOP. Proc. AOSD'06, March 2006.

[4] S. Chidamber, C. Kemerer. A Metrics Suite for OO Design. IEEE Trans. on Soft. Eng.,20-6, 1994, 476-493.

[5] E. Figueiredo, A. Garcia, C. Lucena. AJATO: an AspectJ Assessment Tool. Proc. ECOOP.06, Demo Session, Nantes, France, July 2006.

[6] F. Filho, N. Cacho, R. Ferreira, E. Figueiredo, A. Garcia, C. Rubira. Exceptions and Aspects: The Devil is in the Details. Proc. FSE-14, November 2006.

[7] E. Gamma, et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[8] A. Garcia, et al. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In Software Engineering for Multi-Agent Systems II, Springer, LNCS 2940, 2004.

[9] A. Garcia, et al. Modularizing Design Patterns with Aspects: A Quantitative Study. Proc. AOSD'05, USA, 2005.

[10] I. Godil, H. Jacobsen. Horizontal Decomposition of Prevayler, Proc. CASCON 2005, October, Canada..

[11] G. Kiczales, et al. Aspect-Oriented Programming. Proc. of ECOOP'97, LNCS 1241, Finland, 1997, 220-242.

[12] G. Kiczales, et al. Getting Started with AspectJ. Communications of the ACM. October 2001.

[13] J. Kienzle, R. Guerraoui. AOP: Does it Make Sense? The Case of Concurrency and Failures. Proc. ECOOP'02.

[14] A. Rashid, R. Chitchyan. Persistence as an Aspect. Proc. AOSD'03, USA, 2003.

[15] C. Sant'Anna, et al. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. Proc. Brazilian Symp. on Software Engineering, 2003, 19-34.

[16] S. Soares, et al. "Implementing Distribution and Persistence Aspects with AspectJ". Proc. OOPSLA'02.

[17] S. Soares. An Aspect-Oriented Implementation Method. Doctoral Thesis, Federal Univ. of Pernambuco, 2004.

[18] S. Soares, P. Borba, E. Laureano. Distribution and Persistence as Aspects. Software: Practice & Experience, 2006.

[19] Quantifying the Effects of AOP: A Maintenance Study, www.teccomm.les.inf.puc-rio.br/MaintenanceStudy/index.html.