On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study

Phil Greenwood[1], Thiago Bartolomei[2], Eduardo Figueiredo[1], Marcos Dosea[3], Alessandro Garcia[1], Nelio Cacho[1], Cláudio Sant'Anna[1,4], Sergio Soares[5], Paulo Borba[3], Uirá Kulesza[4], and Awais Rashid[1]

[1] Lancaster University, United Kingdom
{p.greenwood,e.figueiredo,n.cacho,a.garcia}@lancaster.ac.uk
awais@comp.lancs.ac.uk
[2] Kiel University of Applied Sciences, Germany
thiago.bartolomei@gmail.com
[3] Federal University of Pernambuco, Brazil
{mbd2,phmb}@cin.ufpe.br
[4] Pontifical Catholic University of Rio, Brazil
{claudio,uira}@les.inf.puc-rio.br
[5] Pernambuco State University, Brazil
sergio@dsc.upe.br

Abstract. Although one of the main promises of aspect-oriented (AO) programming techniques is to promote better software changeability than object-oriented (OO) techniques, there is no empirical evidence on their efficacy to prolong design stability in realistic development scenarios. For instance, no investigation has been performed on the effectiveness of AO decompositions to sustain overall system modularity and minimize manifestation of ripple-effects in the presence of heterogeneous changes. This paper reports a quantitative case study that evolves a real-life application to assess various facets of design stability of OO and AO implementations. Our evaluation focused upon a number of system changes that are typically performed during software maintenance tasks. They ranged from successive re-factorings to more broadly-scoped software increments relative to both crosscutting and non-crosscutting concerns. The study included an analysis of the application in terms of modularity, change propagation, concern interaction, identification of ripple-effects and adherence to well-known design principles.

1 Introduction

Design stability [1-4] encompasses the sustenance of system modularity properties and the absence of ripple-effects in the presence of change. Development of stable designs has increasingly been a deep challenge to software engineers due to the high volatility of systemic concerns and their dependencies [5]. Contemporary developers often need to embrace a plethora of unexpected changes on driving design concerns, ranging from simple perfective modifications and re-factorings to more architecturally-relevant system increments. In fact, incremental development has been established as the de-facto practice in realistic software development in order to

E. Ernst (Ed.): ECOOP 2007, LNCS 4609, pp. 176–200, 2007.

progressively cope with such evolving system concerns [1, 3, 5]. Some recent industrial case studies have demonstrated that around 50% of object-oriented (OO) code is altered between two releases, and 68% of change requests are accepted and implemented [1, 3].

It has been empirically observed that design stability is directly dependent on the underlying decomposition mechanisms [4-6]. For instance, certain studies have detected that the versatility of multiple inheritance is one of the main causes of ripple-effects in OO systems [6]. Interestingly, we are in an age where emerging aspect-oriented (AO) programming techniques [7] are targeted at improving software maintainability. The proponents of aspect-oriented programming argue that superior modularity and changeability of crosscutting concerns are obtained through the use of new composition mechanisms, such as pointcut-advice and inter-type declarations. It is often claimed that such AO mechanisms support enhanced incremental development [5, 6, 8], and avoid early design degradation [9].

However, there is no empirical evidence AO decompositions promote superior design stability in realistic evolutionary software development [1], especially when experiencing changes of a diverse nature. Most of the existing case-studies in the literature have reported on the positive and negative impacts of aspect-oriented programming in the upfront modularization of conventional crosscutting concerns such as: persistence [9-11], distribution [11], exception handling [12, 13], and design patterns [14-16]. More fundamentally, these evaluations do not compare the stability of AO and OO decompositions while applying heterogeneous types of changes to both crosscutting and non-crosscutting concerns. As a result, there is no empirical knowledge on how aspect-oriented programming contributes to the reduction of ripple-effects and design principle violations in incremental development scenarios.

This paper presents a case-study that quantitatively assesses the stability of OO and AO design versions of a real-life Web-based information system, called Health Watcher (HW). The OO version was implemented in Java, while the AO versions were implemented in AspectJ [17] and CaesarJ [8]. Our evaluation focused upon ten releases of the HW system, which underwent a number of typical maintenance tasks, including: re-factorings, functionality increments, extensions of abstract modules, and more complex system evolutions. Some of the crosscutting concerns were "aspectized" from the first release, while others were modularized as new HW versions were released. Very often, additive and subtractive modifications required the alteration of how two or more concerns were inter-related. The design stability of OO and AO versions was evaluated according to conventional suites of modularity and change impact metrics. Such measures allowed us to analyze the extent to which the OO and AO implementations were vulnerable to ripple-effects, and exhibited symptoms of violations of fundamental design properties, such as the narrow interfaces and the Open-Closed principle [18].

The main outcomes of our analysis *in favour* of AO designs were:

(1) the concerns aspectized upfront tended to show superior modularity stability in the AO designs; changes tended to be confined to the target module and only minor fragility scenarios were observed in the aspect interfaces;

(2) AO solutions required less intrusive modification (e.g. changes to existing operations and lines of code) even when the change focused on a non-crosscutting concern;

(3) aspectual decompositions have demonstrated superior satisfaction of the Open-Closed principle [18] in most of the maintenance scenarios;
(4) both OO and AO implementations have exhibited a significant stability of high-level design structures; however, architectural ripple effects (i.e. changes to architectural elements) were observed when persistence-related exceptions needed to be introduced in the OO design;

Alternatively, the main findings *against* aspectual decompositions were:

(1) significant incidence of violation of pivotal design principles – such as narrow interfaces and low coupling – were detected in evolutionary scenarios involving classical design patterns, such as the Command and State design patterns [19];
(2) although invasive modification was more frequent in the OO solution, the AO modifications tended to propagate to seemingly unrelated modules;
(3) in general the aspectization of exception handling has shown no improvement; in addition, certain design degeneration was observed due to the creation of artificial method signatures in order to expose contextual information to the aspectized exception handlers.

The remainder of the paper is structured as follows: Section 2 describes the experimental settings and justifies the decisions made to ensure the study is valid. Section 3 describes the Health Watcher system used as the base for this study and also describes the changes applied. The results gathered from applying modularity metrics are discussed in Section 4. Section 5 discusses how the changes propagate within each paradigm. The evolution of the concern interactions are discussed in Section 6. Other related discussions of the results are conducted in Section 7. Finally, Sections 8 and 9 conclude this paper by discussing related work and summarizing this paper's findings.

2 Experimental Settings

This section describes the configuration of our study including the choice of the target application (Section 2.1), and the evaluation methodology (Section 2.2).

2.1 Target System Selection

The first major decision that had to be made in our investigation was the selection of the target application. The chosen system is a typical Web-based information system, called Health Watcher (HW) [11]. The main purpose of the HW system is to allow citizens to register complaints regarding health issues. This system was selected because it met a number of relevant criteria for our intended evaluation. First, it is a real and non-trivial system with existing Java and AspectJ implementations (each around 4000 lines of code). The HW system is particularly rich in the kinds of non-crosscutting and crosscutting concerns present in its design. HW also involves a number of recurring concerns and technologies common in day-to-day software development, such as GUI, persistence, concurrency, RMI, Servlets and JDBC. Second, each HW design and implementation choice for both OO and AO solutions has been extensively discussed and evolved in a controlled manner. Both the OO and

AO designs (Sections 3.1 and 3.2) of the HW system were developed with modularity and changeability principles as main driving design criteria. Third, qualitative and quantitative studies of the HW system have been recently conducted [9, 11] allowing us to correlate our results with these previous studies. Finally, the first HW release of the Java implementation was deployed in March 2001, since then a number of incremental and perfective changes have been addressed in posterior HW releases; it has allowed us to observe typical types of changes in such an application domain.

2.2 Study Phases and Change Scenarios

The study was divided into three major phases: *(1)* the development and alignment of HW versions, *(2)* the implementation of change scenarios, and *(3)* the assessment of the three versions (developed in phase 1) and the successive releases (delivered in phase 2). In the first phase, we prepared the base versions of the OO and AO implementations of the HW system. The OO solution was already available and implemented in Java. Two AO implementations were assessed: one based on a pre-existing AspectJ version (implemented after the original OO implementation) [11, 17], and a newly created CaesarJ [8] version for this study. Both the Java and AspectJ implementation have been successfully used in other studies [9, 11], and so provided a solid foundation for this study. An independent post-graduate student was responsible for implementing the CaesarJ version by re-factoring the Java implementation. We would like to highlight that this study has not targeted the comparison of the two AO programming languages (i.e. AspectJ and CaesarJ). On the contrary, the objective of using more than one particular language was to allow us to yield broader conclusions that are agnostic to specific AO language features.

All three base HW versions were verified according to a number of *alignment rules* in order to assure that coding styles and implemented functionality were exactly the same. Moreover, the implementations followed the same design decisions in that best practices were applied in all implementations to ensure a high degree of modularity and reusability. This alignment and validation exercise was performed by an independent post-doc researcher. A number of test-cases were exhaustively used for all the releases of the Java, AspectJ and CaesarJ versions to ease the alignment process. These alignment procedures assure that the comparison between versions is equitable and fair. Inevitably, some minor re-factorings in the three versions had to be performed when misalignments were observed at the composition-level, interface-level, module-level or even LOC-level. When these misalignments were discovered the implementers for that particular version (in the case of the Java and AspectJ versions this was the original HW developers) were notified and instructed to correct the implementation accordingly.

The second phase involved the implementation of nine changes (Section 3.3) in all three HW versions (available from [20]). Each change involved: *(i)* the design and implementation of new modules to be included or existing modules to be removed from the system, *(ii)* the use of language mechanisms to compose such new modules with existing ones, and *(iii)* if necessary, changes to the modules already present in the previous HW release. Also, for each change, we needed to again ensure that the Java, AspectJ and CaesarJ implementations were aligned and so the alignment process described above was applied to each subsequent version of HW developed.

The goal of the third phase was to compare the design stability of AO and OO designs. In order to support a multi-dimensional data analysis, the assessment phase was further decomposed in four main stages. The first two stages (Section 4) are aimed at examining the overall maintenance effects in fundamental modularity properties through the HW releases. The third stage (Section 5) evaluates the three implementations from the perspective of change propagation. The last stage (Section 6) focuses on assessing design stability in terms of how the implementation of concern "boundaries" and their dependencies have evolved through the HW releases. Section 7 also discusses architecture-level design stability. Traditional metrics suites were used in all the assessment stages, and will be discussed in the respective sections. Although design stability is discussed in terms of these metrics, the metric measurements are a direct reflection of design changes, i.e. any variations in the metric values are evidence of changes that occurred in the code structure.

3 Health Watcher System

Both the OO and AO architectural designs of the HW system are mainly determined by the conjunctive application of both client-server and layered architectural styles [21]. The original architecture aimed at modularizing user interface, distribution, business rules, and data management concerns. Most of them are layers in both the OO and AO architectures; the only exception is the distribution concern that has been aspectized and is no longer a layer in the AO architectural design. All the corresponding OO and AO design structures that realize such driving design concerns have been successfully reused in several applications [3, 10, 11]. Fig. 1 and Fig. 2 present representative slices of both OO and AO architecture designs.

3.1 The Object-Oriented Design

In the HW system, complaints are registered, updated and queried through a Web client implemented using Java Servlets [22], represented by the components in the view layer. Accesses to the HW services are made through the *IFacade* interface, which is implemented by the *HealthWatcherFacade*. This facade works as a portal to access the business collections, such as *ComplaintRecord* and *EmployeeRecord*. Records access the data layer using interfaces, like *IComplaintRep*, which decouple the business logic from the specific type of data management in use. For example, Fig. 1 shows a *ComplaintRep* class that implements a repository for a database.

This structure prevents some code tangling, because it clearly separates some main concerns in layers. However, tangling is not completely avoided [11]. For example, the *HealthWatcherFacade* implements several concerns, including transaction management (persistence) and distribution. One possibility is the use of adapters [19] to take care of transaction functionality, but at a high price, since developers must maintain both the facade and the adapter.

The design of the OO HW system also fails to completely prevent code scattering. For example, business components implement the distribution concern; this comes from the need to implement the *Serializable* interface to allow them to be sent across a

network. Furthermore, almost all components must deal with the exception handling concern. However, despite not completely separating concerns, the layered architecture gives some support to adaptability. For instance, in the system configuration used in our experiment, instead of RMI, one could use EJBs.

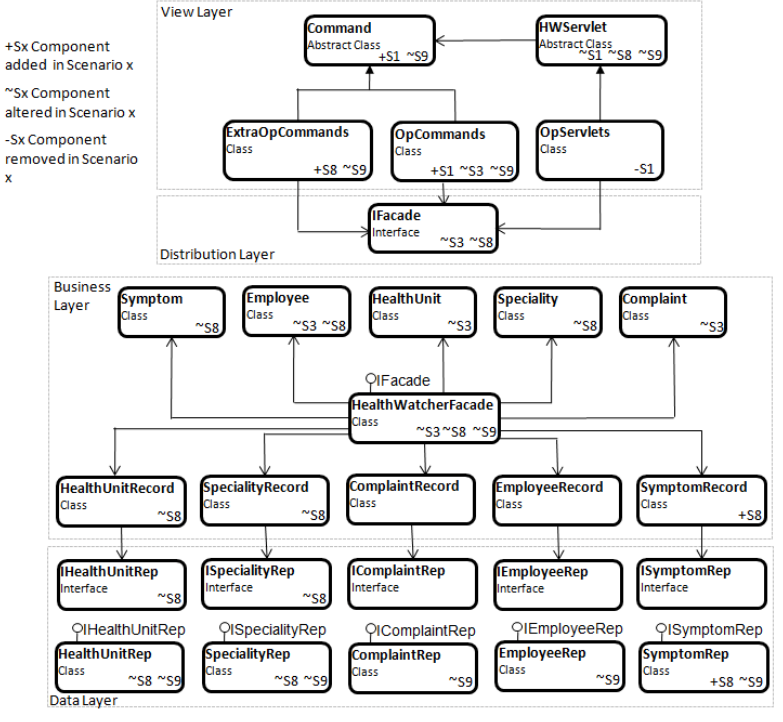

Fig. 1. OO HW architecture. A sub-set of scenarios and the modules they affect are marked.

3.2 The Aspect-Oriented Design

Fig. 2 displays a diagram that is essentially the same for both AO implementations (AspectJ and CaesarJ). The AO versions modularize some concerns that were tangled and scattered in the OO decomposition counterpart. Basically, in the first AO release of the HW system, crosscutting elements relative to distribution, persistence, and concurrency control concerns were modularized as aspects.

For instance, the concurrency control concern was removed from the layers and encapsulated in two aspects, namely *HWManagedSync* and *HWTimestamp*; each of these deals with a specific facet of concurrency control. Timestamp is a technique used to avoid object inconsistency. This problem can occur when two copies of an object are retrieved by different requests before one of them can update its version. The technique uses a timestamp field to avoid object updating if there is a newer version of it stored in the persistence mechanism.

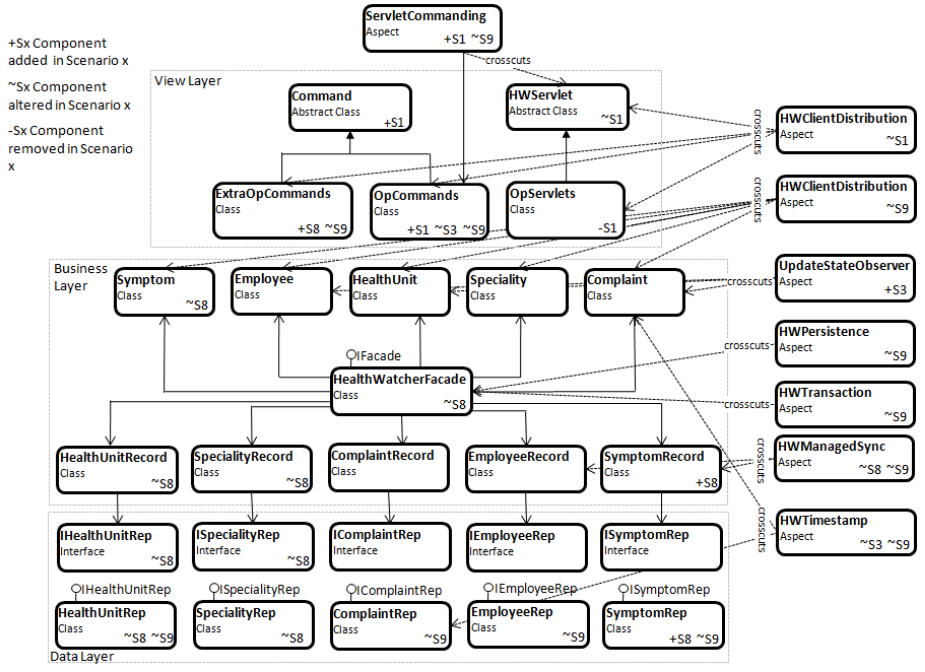

Fig. 2. AO HW architecture. A sub-set of scenarios and the modules they affect are marked.

3.3 Change Scenarios

The selected changes to be applied to the HW system are varied in terms of the types of modifications performed. Some of them add new functionality, some improve or replace functionality, and others improve the system structure for better reuse or modularity. The purpose was to expose the OO and AO implementations to distinct maintenance and evolution tasks that are recurring in incremental software development. These changes originated from a variety of sources: the experiences of the original developers of HW including changes they would like to implement (that are actually necessary) and changes from previous empirical studies [9, 12]. The remaining changes were created by the students and researchers involved in this study, where certain extensions and improvements that could be applied were identified. This ensured a variety of sources of changes was used and so would not artificially bias one paradigm. Before the changes were applied, the original developers of HW were consulted to confirm whether these changes were valid.

This wide range of modifications provide an indication as to whether one paradigm supports better design stability for certain types of change. The changes were implemented using the best possible practices in the given paradigm to ensure a fair comparison. Each of the scenarios is summarized in Table 1, a sub-set of the scenarios and the modules that they affect in the OO and AO paradigm are marked on Fig. 1 and Fig. 2 according to whether a component is added, removed or modified during a particular scenario.

Table 1. Summary of the scenarios used in the study

Scenario	Change	Impact
1	Factor out multiple Servlets to improve extensibility.	View Layer
2	Ensure the complaint state cannot be updated once closed to protect complaints from multiple updates.	View/Business Layers
3	Encapsulate update operations to improve maintainability using common software engineering practices.	Business/View Layers
4	Improve the encapsulation of the distribution concern for better reuse and customization.	View/Distribution/ Business Layers
5	Generalize the persistence mechanism to improve reuse and extensibility.	Business/Data Layers
6	Remove dependencies on Servlet response and request objects to ease the process of adding new GUIs.	View Layer
7	Generalize distribution mechanism to improve reuse and extensibility.	Business/View/ Distribution Layers
8	New functionality added to support querying of more data types.	Business/Data/View Layers
9	Modularize exception handling and include more effective error recovery behaviour into handlers.	Business/Data/View Layers

4 Modularity Analysis

We have described earlier how the assessment procedures were organized in four stages (Section 2.2). This section presents the results for the first two stages, where we primarily analyze the initial modularity of each OO and AO solutions (Section 4.1). Then we analyze their stability throughout the implemented changes (Section 4.2). Both the first and second assessment phases are supported by a metrics suite that quantified four fundamental modularity attributes, namely separation of concerns (SoC), coupling, cohesion, and conciseness [23]. Such metrics were chosen because they have already been used in several experimental studies and proven to be effective quality indicators (e.g. [9, 12, 14, 15, 24]).

The metrics for coupling, cohesion, and size were defined based on classic OO metrics [25]; the original metrics definitions were extended to be applied in a paradigm-independent way, supporting the generation of comparable results. Also, this suite introduces three new metrics for quantifying SoC. They measure the degree to which a single concern in the system maps to: (i) the design *components* (i.e. classes and aspects) – based on the CDC metric (Concern Diffusion over Components), (ii) *operations* (i.e. methods and advice) – based on the CDO metric (Concern Diffusion over Operations), and (iii) *lines of code* – based on the CDLOC metric (Concern Diffusion over Lines of Code). The majority of these metrics can be collected automatically by applying the appropriate tool [26, 27].

The separation of concern metrics had to be calculated manually. This involved a post-graduate student (not involved in the implementation phase of the study) 'shadowing' the code to identify which segment of code contributed to which concern. In circumstances when it was not clear which concern the segment contributed to, discussions between all people involved in the implementation took place to reach a common agreement. For all the employed metrics, a lower value implies a better result. Detailed discussions about the metrics are out of the scope of this work and appear elsewhere [14, 23].

4.1 Quantifying Initial Modularity

This stage evaluates the modularity of the base versions in order to have an overall understanding of the modularity attributes of the first release of each OO and AO implementation. Hence, it can support our further analysis on how the maintenance changes affected the degree of modularity initially obtained in the Java, AspectJ and CaesarJ implementations. Instead of analysing each individual metric result, we provide a general view of the meanings behind the results. Fig. 3 presents the modularity results for SoC, coupling, cohesion, and size in the base version. The *concurrency* concern is representative of all the analysed crosscutting concerns which included: concurrency, persistence, distribution and exception handling. Furthermore, other architectural and design elements were analysed using the SoC metrics that are discussed later, these elements included: view (GUI) layer, business layer and various design pattern. The metrics raw data can be found at [20].

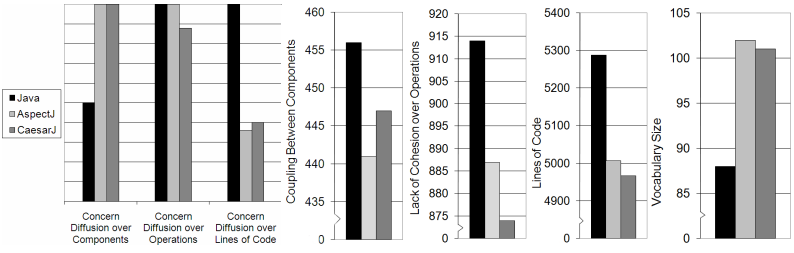

Fig. 3. Relative SoC metric values for concurrency (CDC, CDO and CDLOC), and absolute values for coupling, cohesion, lines of code and vocabulary size for the base version of Health Watcher

A careful analysis of the measures (Fig. 3) determines that the AO implementations offer superior modularity in these initial versions. Even though the base AO implementations have more modules (measured by the Vocabulary Size metric which counts the number of components) and a higher Concern Diffusion over Components (CDC), this does not automatically mean that separation of concerns is worse. It is due to the fact that additional aspectual modules have been created for the purpose of isolating certain crosscutting concerns (Section 3.2). When other values are analysed together, such as Concern Diffusion over Lines of Code (CDLOC) and over Operations (CDO), Coupling between Components (CBC), Lack of Cohesion over Operations (LCOO), it becomes clear that AO implementations create more cohesive and self-contained modules that are less coupled to each other. They create dedicated components for each concern, but they also require fewer Lines of Code (LOC), since they reduce the duplicated code. The subsequent sections will analyse how these modularity properties alter due to the application of change scenarios (Section 3.3).

4.2 Quantifying Modularity Stability

After producing an overview of the modularity attributes in the base versions we proceeded with the analysis regarding the stability of these attributes. In the following we present the most significant results for each modularity attribute.

The variations in the modularity metrics are traced back to the implementations in order to determine the causes of instability. Generally, any variation to their values is considered undesirable. Often the variations are unavoidable, particularly when a certain module is the focus of an implementation change. Thus, we categorise the variation in the values as either being unavoidable or negative. Unavoidable variations occur when a component that is directly related to the affected concern is altered. For example, if a scenario targets the GUI concern, then variations in the metric values for the view related modules are generally expected and unavoidable. However, if unrelated modules are affected, then these should be considered as being negative variation. As a result, the approach with the most stable design is the one which minimises the number of negative variations.

Concern Measures. The analyzed crosscutting concerns included concurrency, distribution, persistence, and exception handling. These were selected because they were the main modularization target either at the initial HW design decomposition (Sections 3.1 and 3.2) or subject to change when applying the scenarios described in Section 3.3. For the same reasons, other non-crosscutting concerns were also analyzed including the view and business layers of the HW architecture. Finally, the modularity of concerns relative to some key design patterns adopted during the system HW evolutions – such as the Command and State design patterns [19] – were analyzed.

From the analysis of SoC metric results, three distinct groups of concerns naturally emerged, with respect to which type of modularization paradigm presents superior stability. The following set of figures illustrates the relative variations between versions in the Java, AspectJ and Caesar implementations. When no bar for a particular implementation is present, this means no variation in that particular measure was observed for the corresponding scenario. We do not distinguish in the graphics the difference between an increase in a metric value and a drop in a metric value.

Concerns with Superior AO Design Stability. Most of the concerns that present a widely-scoped crosscutting nature have presented superior design stability when implemented using AO techniques. This includes concurrency, persistence and distribution, with the sole exclusion of exception handling (discussed later). All of these concerns were modularized into aspects from the first AspectJ and CaesarJ implementations. Fig. 4 presents the metrics results for the concurrency concern, the representative of how concerns in this group have behaved. In general, the concern diffusion over components (CDC) metric is less affected on AO implementations as the initial modules seem to cope well with newly introduced scenarios and the changes are localized in these modules.

The concern diffusion over operations (CDO) metric also presents a very superior result for the AO implementations. This difference largely comes from the quantification properties in AO, where the use of declare statements and regular expressions in pointcuts eliminates the need for some operations. For example, in the specific case of concurrency, the timestamp behaviour affects CDO in the OO implementation (particularly in Scenario 2), but has less influence on the values for AO version, where it is implemented with advice and intertype declarations (or bindings in CaesarJ). The results for concern diffusion over lines of code (CDLOC) provide evidence that AO implementations present better stability.

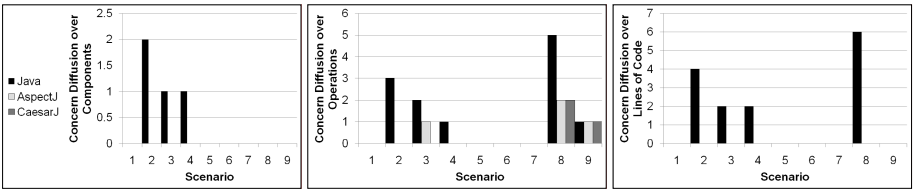

Fig. 4. Changes in the SoC metrics for the concurrency concern

Concerns with Superior OO Design Stability. Concerns that are widely scoped but that do not have a crosscutting nature, such as view and business; have presented slightly superior design stability in the OO implementation. Fig. 5 presents the metrics results for the view concern. In general, this type of concern is already well modularised by OO decompositions and this modularisation is stable throughout the changes. AO implementations do not visibly modify this OO modularisation, but the transfer of some functionality affects the stability of these concerns. Using the view results as an example, we observe that the concerns are spread over the same number of more components (CDC) in AO. That is because the AO implementations not only suffer from the same unavoidable changes as OO, but also may present some negative changes, such as new pointcuts having to be introduced to aspects related to the view layer. In contrast, the OO implementation will have the necessary functionality inserted directly to the view layer and so will not impact the CDC metric. The number of operations per concern (CDO) also shows the same general trend. The concern diffusion over lines of code (CDLOC) results provide additional evidence of the superiority of OO decompositions for these concerns, because the AO implementations are more instable with respect to how certain concerns interact with each other (Section 6).

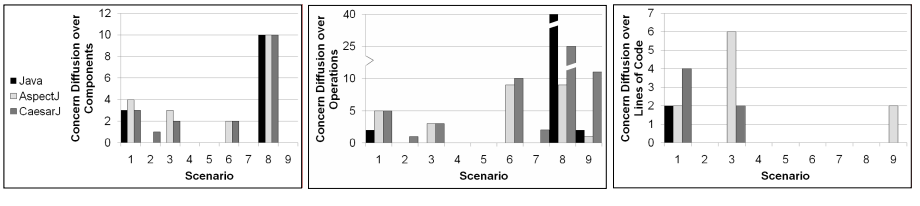

Fig. 5. Changes in the SoC metrics for the view layer

Concerns with no Stability Superiority Observed. Some concerns have not presented explicit superiority in either of the paradigms. These include the exception handling (EH) concern and also the more localized concerns, such as the Command and State patterns [19]. As can be observed in Fig. 6, both paradigms experience similar high degrees of variation in all the metrics for the EH concern. This similarity occurs for two main reasons. First, the EH concern was not explicitly modularised in the AO implementations until the last scenario (Section 3.3). Before that, the exception handlers were just aspectized when they related to other concerns that were also aspectized. For example, when concurrency was aspectized, its handlers were also

removed from the base code and encapsulated in a new module. Second, AO implementations were not able to promote modularization gains in terms of this concern even in Scenario 9. The general problem with using AO to encapsulate EH is that it usually involves negative changes through the creation of artificial operations in order to expose context information for the aspectized exceptional behaviour [12].

When analysing the results for the more localized concerns we were confronted with very distinct and irregular patterns. It was not possible to generalize about their stability, because specific characteristics of the concerns and their applications play a major role on the results. For example, the State pattern was very stable and similar for all metrics in both paradigms. However, the Command pattern was stable through several scenarios but was largely affected by the last two scenarios, in a similar proportion on AO and OO. We conclude that, due to the narrow scope of these concerns, their stability largely depends on the localization of the change.

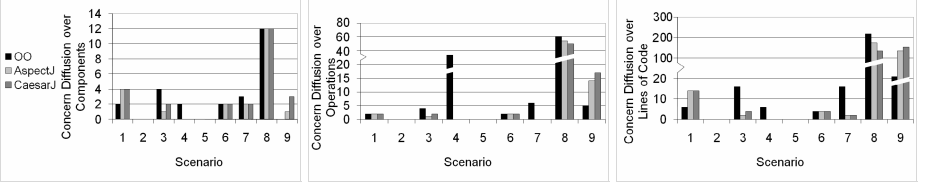

Fig. 6. Changes in the SoC metrics for the exception handling concern

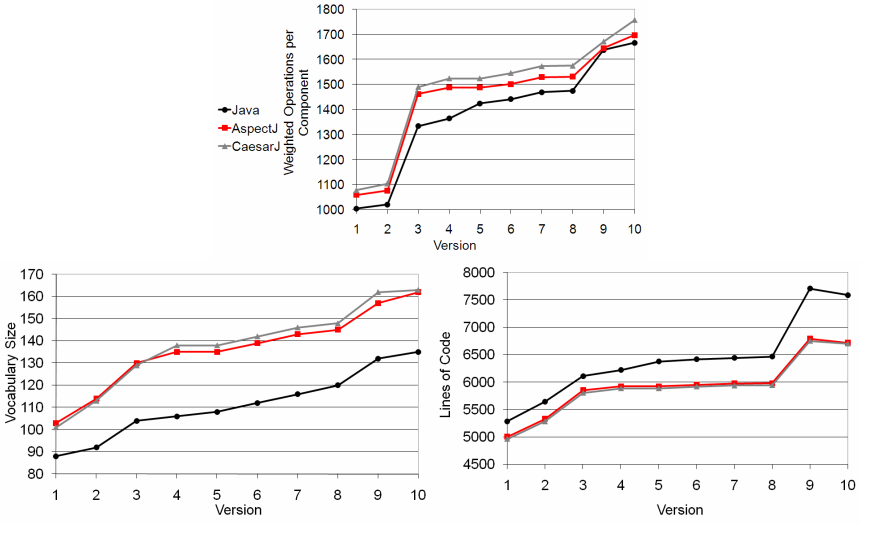

Fig. 7. Size metric variation through the 10 versions

Size Measures. Both OO and AO implementations present very similar stability with respect to size measures. Fig. 7 shows the absolute results for Weighted Operations per Component (WOC – i.e. complexity of operations based on number of parameters) and

Vocabulary Size (VS – i.e. number of components) through the versions. The VS curves show that AO implementations tend to be worse, and this difference gets larger, rendering AO less stable regarding the number of components needed to modularize the concerns. The Lines of Code (LOC) curves, present an inverted result, where AO implementations scale better, due to being able to avoid repeated code.

The WOC curves show that AO implementations are usually more complex, as they introduce pointcuts and advice that are counted as operations. In general the curves are very similar. However, AO performs better for functional changes (i.e. Scenario 9), due to better quantification, but poorly in the exception handling scenario as new advice and artificial methods must be created for each handler. As a general conclusion for the size attribute, AO is more stable with respect to LOC and for functional changes, while OO is better in VS and in exception handling.

Coupling and Cohesion Measures. Due to the strong synergy between coupling and cohesion, we will discuss these two attributes together. The Depth of Inheritance Tree (DIT) metrics are largely the same and do not bring any interesting insights. In this section we will refer to Coupling Between Components (CBC) and Lack of Cohesion over Operations (LCOO) metrics as coupling and cohesion.

Fig. 8 displays the graphs for the average cohesion and coupling per component. Observing the curves, AO implementations have generally much more stable values for both coupling and cohesion. It can be seen that most of the scenarios actually improve their values, while OO implementations are affected in different ways. The only major changes for AO occur in version three and the last two scenarios. In the former, the cohesion values are worsen due to hook methods having to be inserted in the base code for the newly added aspects to be executed correctly. In general, AO implementations were much better and more stable with respect to coupling and cohesion attributes due to the generic capabilities of AO techniques.

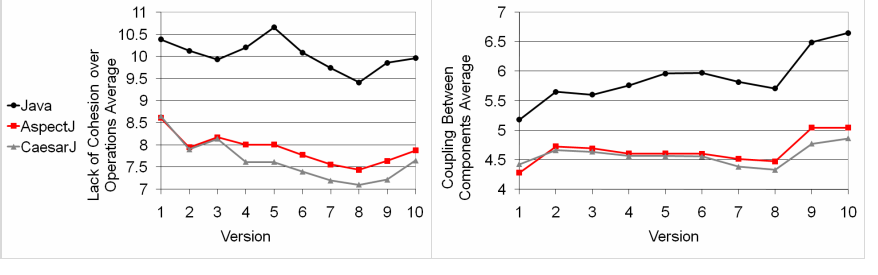

Fig. 8. Average results for cohesion (left) and coupling (right) per component throughout the versions

5 Change Impact Analysis

This section describes the third assessment stage where we quantitatively analyze to what extent each maintenance scenario entails undesirable change propagations in the AO and OO HW implementations.

5.1 Impact of the Change Nature

This phase relies on a suite of typical change impact measures [3, 4] presented in Table 2, such as number of components (aspects/classes) added or changed, number of added or modified LOC, and so forth. The purpose of using these metrics is to quantitatively assess the propagation effects, when applying the various changes, in terms of different levels of abstraction: components, operations, lines of code and relationships. The lower the change impact measures the more stable and resilient the design is to a certain change.

Table 2. Change propagation metrics for Java (J), AspectJ (A) and CaesarJ (C)

		Sc. 1	Sc. 2	Sc. 3	Sc. 4	Sc. 5	Sc. 6	Sc. 7	Sc. 8	Sc. 9
Added Components	**J**	24	12	2	3	4	4	4	12	5
	A	29	16	3	0	4	4	2	12	6
	C	25	16	3	0	4	4	2	12	6
Changed Components	**J**	2	6	14	25	1	24	2	23	46
	A	2	5	4	0	1	26	2	19	52
	C	2	4	5	0	1	25	2	18	21
Added Operations	**J**	4	4	10	0	0	0	0	62	5
	A	2	7	1	0	0	0	0	40	16
	C	2	7	1	0	0	0	0	32	17
Changed Operations	**J**	3	61	15	19	2	25	1	21	94
	A	0	57	3	0	1	25	0	10	94
	C	0	57	6	0	1	25	0	14	99
Added Pointcuts	**A**	5	12	8	0	2	0	2	0	16
	C	5	12	11	0	1	0	0	4	9
Changed	**A**	1	0	0	0	0	1	2	3	1
	C	1	0	0	0	0	1	0	4	1
Added LOC	**J**	72	59	123	294	6	1	0	653	109
	A	4	34	4	0	1	2	0	290	60
	C	8	35	6	0	1	2	0	300	70
Changed LOC	**J**	4	53	12	40	1	107	1	13	177
	A	2	50	0	0	5	110	4	9	141
	C	2	52	5	0	5	110	4	5	187

Adherence to the Open-Closed Principle. AO solutions generally require more new components to implement a change. In comparison the OO implementation require existing components to be modified more extensively to implement the same change. This behaviour is confirmed in the change propagation metrics (see Table 2) whereby much more extensive changes (in terms of added operations and LOC) occur in the OO version. Up to 30% fewer operations and up to 60% fewer LOC are added in the AO implementations throughout the scenarios. This indicates that the AO solutions conform more closely to the Open-Closed principle [18] which states that "software should be open for extension, but closed for modification".

The scenario which demonstrates this difference best is Scenario 3. The purpose of this scenario is to isolate the update method calls, which is implemented in the form of the Observer pattern [19]. This involves modifying a sub-set of the command classes in the view layer by removing the update calls. However, the OO implementation requires further modification, for example a variety of classes within the business layer require modification (marked on Fig. 1) so that the update method

is called when the state of business objects alters. The AO implementation is able to quantify and capture these state changes via pointcuts and so the Observer pattern is able to be applied by introducing new components rather than modifying existing ones (the added modules are marked on Fig. 2).

In order to analyse the results more closely and to identify specific reasons for these changes the results will be split into three categories: the first examines groups of scenarios where the AO versions are superior, the second focuses on scenarios where the AO and OO implementations are comparable and the final category looks at scenarios where the OO version is superior. It is interesting to note that these categories can be mapped to particular types of change.

Propagation of Functional Changes is Superior in AO Designs. When considering the scenarios (1, 3, 8 and 9) where the AO solutions are superior, they require fewer changes to components (in terms of modified LOC, added LOC, etc). It is also clear that these changes are absorbed in other ways. For example, we can see that pointcuts must either be modified or added in these scenarios. Equally, the OO implementation requires new fields/parameters to be added directly to the base classes in order to implement the changes in these scenarios. These additions can be directly absorbed within new aspects. What is not clear from analysing these metrics in isolation is which type of change propagation is more desirable and which paradigm provides the better mechanism for absorbing these changes. When considering the earlier modularity metrics we can conclude that the AO implementation has less impact on these attributes and so AO provides better mechanisms for absorbing these changes. Note that the changes performed in Scenario 4 reflect an attempt to improve the modularity of the distribution concern, as this concern was already well modularised in the AO versions no changes were necessary.

AO and OO are Comparable when Implementing Perfective Changes. In the cases where (Scenarios 5, 6, and 7) the AO and OO implementations have similar change propagation metrics, the modifications are related to improving the design structure of HW in terms of extensibility and reusability. These improvements to the design generally target OO concepts, this can be inferred from the metrics due to the low number of pointcuts added or modified in these scenarios. For example, two of the scenarios (6 and 7) involve splitting a class implementation in two parts. As this structure is repeated in both the AO and OO implementation it is natural that they are both affected in similar ways. This is also reflected in the earlier modularity metrics.

OO Implementation of the State Pattern was Superior. Finally, the OO implementation is clearly superior in Scenario 2, which basically involves the instantiation and inclusion of the State pattern [19]. Although the added state update behaviour is crosscutting, it does require 'hook methods' to be inserted in the base components for advice to be executed at the correct events. This results not only in the same structural changes as the OO implementation (introduction of methods, fields and inheritance relationships) but also in new aspects to be added. In comparison, the OO solution can encapsulate the same behaviour using OO mechanisms and, as a consequence, reduces the need for changes across modules. There are other scenarios (such as Scenario 1) where the suitability of using AO is questionable but due to the fact that multiple concerns are involved (view and distribution), the AO solutions are able to capture these relationships more cleanly and, so, offer improvements over OO.

5.2 Overall Stability Measurements

From these change propagation metrics various attributes can be extracted that can be used to comment on the implementations' stability. One of the most significant results from these stability metrics is the number of unique components modified through the nine scenarios applied. For each paradigm 72 different modules are modified throughout all the scenarios. It may appear at first glance that because this measure is the same the stability is also the same. However, the actual number of potential modules that could be modified has to be taken into account. In the case of the OO design, this is 1113 potential classes (the cumulative figure of all classes in all versions). The AO versions have over 250 more classes/aspects that could be modified. As a result, the fact that the number of modified modules is the same for the AO versions despite this, illustrates that the AO implementations are more stable.

Similarly, when analysing the changes to the LOC that are made to these 72 modules the AO solutions again show more stability. For example, in version 10 of the OO implementation of HW there are in total 5453 LOC of which 32% of these have either been modified or added during the course of the study. This is compared to around 4000 LOC in version 10 of the AO implementations of which only 18-19% have been added or modified.

6 Concern Interaction Analysis

We have presented results concerning the first three assessment phases. This section discusses the last evaluation stage: the scalability of OO and AO solutions from the viewpoint of stability in concern interactions.

The analysis of the data gathered based on the modularity and change impact metrics (Sections 4 and 5) makes it evident that most of the concerns involved in HW are scattered and tangled with each other over the system classes. For example, the class *HealthWatcherFacade* implements a business layer facade (Fig. 1), but also incorporates code for distribution- and persistence-specific functionalities. The implementation of certain system concerns and the way they interact through the system modular decomposition change as the system evolves. As a consequence, this section discusses how the concern interactions changed over time in the three HW implementations. The goal is to observe how changes relative to a specific concern "traversed the boundaries" of other concern implementations and/or generated new undesirable inter-concern dependencies.

6.1 Concern-Interaction Categories

We have performed an analysis of AO and OO design stability when there are interactions involving two or more concerns. In order to support such an analysis, we have observed different categories of interactions involving the analyzed concerns. In the context of the HW system, there are different ways in which the concerns interact with each other: invocation-based interaction, interlacing, and overlapping. Our classification of concern interactions is based on how the concern realizations share elements in the implementation artefacts, which have been defined and exploited in previous studies [12, 15].

Invocation-based Interactions. This is the simplest form of composition. The invocation-based interaction occurs when a component realizing the first concern is connected to a component realizing another concern based on one or more method calls. An example of this form of interaction is when the business facade class calls persistence methods to invoke transaction management in the OO HW version.

Interlacing. In this case, implementations of two concerns, C1 and C2, have one or more components in common [12, 15]. However, C1 and C2 are implemented by different sets of methods, attributes and statements in the shared classes. In other words, the involved concerns have common participant components, but there is no common element implementing both. We can identify interlacing either in the component level or in the method level. Both component- and method-level interlacing produce concern tangling but at different levels of abstraction. Alternatively, in the method-level interlacing the implementations of concerns C1 and C2 have one or more methods in common. However, different pieces of code in these methods are dedicated to implement both concerns. An example of this interaction category is the concurrency control implemented in some methods of data management classes of the OO HW version. In these methods there is exception handling code that is not related to the concurrency control concern.

Overlapping. The implementations of concerns C1 and C2 share one or more statements, attributes, methods, and components. This dependency style is different from interlacing because here the shared elements contribute to both concerns rather than being disjoint. Depending on the kind of elements participating in the interaction, it can be classified as component overlapping, operation overlapping, or attribute overlapping. The update methods are examples of operation overlapping. They are used to guarantee that in a distributed environment the modified data in one (client) machine is reflected to another (server) machine and that persistence data modified in memory is also reflected to the persistence mechanism.

6.2 Concern Interactions in the Base Versions

We explore the categories of concern interactions (previous subsection) in the base version in order to provide a general idea of how each pair of concerns depends on each other. This first analysis allows us to track the stability of these dependences through the software scenarios. We have analyzed all the 15 possible pair-wise concerns of the three HW implementations (Java, AspectJ and CaesarJ) and we have found many similarities between them. The results regarding the types of interactions are essentially the same for AspectJ and CaesarJ; therefore, we discuss them as AO issues. Our first observation is that, in general, concern boundaries are wider in the OO implementation, i.e. the concerns interact in more components in OO than in AO. Although the AO compositions usually present fewer components in the concern boundaries, these components usually present more intricate interactions. Fig. 9 presents two illustrative instances of concern interactions, concurrency with distribution and concurrency with persistence. The left side of the pie chart (Fig. 9) shows that 5 components present interlacing between the concurrency and distribution concerns in the OO version, while only 3 components present interlacing of these concerns in the AO version. In this interaction, it is not clear that components in the

concern boundaries of the AO design have stronger interaction. On the other hand, in the concurrency with persistence interaction (Fig. 9, right side) the same number of operations (17) realizes the interaction between these two concerns. However, AO composition has more methods with overlapping of concerns (6 against 2). Furthermore, the AO version has fewer components with interlacing which also means a stronger dependency of concurrency with persistence in these components.

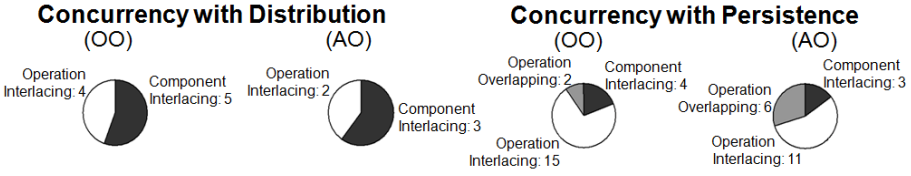

Fig. 9. Pie charts of interactions between concurrency and distribution concerns (left) and concurrency and persistence concerns (right) in the base versions

6.3 Scalability of Aspectual Decompositions

In this section we focus on the stability of each pair of concerns through the implementations. We verify for each concern the number of components it shares with other concerns. This kind of analysis supports assessment of concern modularization and stability because it shows whether the inter-concern coupling drops with the software evolution or not. Table 3 shows the obtained results for each pair-wise concern in terms of the number of components in the boundaries. Due to space limitations, this table presents only the results for the base version and Scenarios 3, 6 and 9 for all the 15 compositions. We have selected these versions because (i) they group together each 3 sequences of evolution and (ii) they represent scenarios with more extensive changes, including architectural ones.

The results in Table 3 show that, in general, concern interactions are more stable in the AO implementations through all scenarios. Only one composition, concurrency with view (see Table 3, line 9), presents stability in favour of OO. Alternatively, 5 compositions in AspectJ (and 4 in CaesarJ) present stable results: business with concurrency (line 1), concurrency with distribution (line 6), concurrency with EH (line 7), concurrency with persistence (line 8), and persistence with view (line 15). This last one is not totally stable in CaesarJ, but a minor variation occurs between the Scenarios 3 and 6. Besides, all the other AO compositions involving at least one crosscutting concern show an almost stable behaviour, such as business with persistence, distribution with view and business with distribution. This stability of AO solutions is a result of the better separation of crosscutting concerns. For instance, concurrency is far more stable in all of its compositions in AO solutions than in OO ones, which is explained by its better modularization (Section 4.2).

Similarly to concurrency, the concern interaction analysis summarized in Table 3 also shows that the distribution concern is far better separated in the AO versions. All compositions of distribution (lines 2, 6, 10, 11, and 12 of Table 3) have fewer shared components in AspectJ and CaesarJ. Although the introduction of the Adapter pattern

Table 3. Number of shared components of the pair-wise concerns through the HW implementations in Java (J), AspectJ (A) and CaesarJ (C)

Concerns Interaction		Base			Scenario 3			Scenario 6			Scenario 9		
		J	A	C	J	A	C	J	A	CJ	J	A	C
1	**Bus + Conc**	3	5	5	4	5	5	5	5	5	6	5	5
2	**Bus + Dist**	29	10	18	32	8	19	22	8	19	29	11	23
3	**Bus + EH**	26	30	29	28	30	29	29	30	29	38	41	40
4	**Bus + Per**	23	12	12	24	10	10	24	10	10	34	12	12
5	**Bus + View**	18	19	19	17	21	20	17	21	20	25	29	28
6	**Conc + Dist**	5	3	3	7	3	3	8	3	3	8	3	3
7	**Conc + EH**	7	11	11	9	11	11	10	11	11	11	11	11
8	**Conc + Per**	4	3	3	6	3	3	6	3	3	6	3	3
9	**Conc + View**	0	2	2	0	3	2	0	3	2	0	2	1
10	**Dist + EH**	40	32	32	44	29	29	33	29	29	37	31	32
11	**Dist + Per**	28	16	16	32	13	13	21	14	14	27	15	15
12	**Dist + View**	16	5	5	16	2	2	4	2	2	6	2	2
13	**EH + Per**	40	32	31	42	30	29	41	30	29	48	34	32
14	**EH + View**	20	24	24	23	28	28	23	28	28	32	38	39
15	**Per + View**	17	5	5	15	5	5	15	5	4	22	5	4

(Scenario 5) improves the separation of the distribution concern in the Java version and it also drops interaction of this concern, this improvement is not enough to make it superior to the AO implementations. In spite of the superiority of AO compositions of crosscutting concerns, OO is no worse in the interaction of non-crosscutting concerns. In fact, in most of compositions involving either business or view, OO is comparable to AO implementations or even better. For instance, the OO composition of concurrency with view has no shared classes in the base version and this situation remains stable in the following scenarios. This result is not surprising, as OO decomposition is suitable to modularize non-crosscutting concerns.

As mentioned earlier and also presented in Table 3, CaesarJ has very similar results to AspectJ. However, there are some exceptions which present significant difference, such as the business with distribution composition. In this composition the CaesarJ solution is worse than AspectJ, but better than Java. The difference between AspectJ and CaesarJ is due to their distinct composition mechanisms. The AspectJ solution uses declare parents construct to introduce the *Serializable* interface into classes, but CaesarJ does not provide such a construct. Therefore, this piece of code which is part of the distribution concern remains scattered in the CaesarJ implementation which contributes to the high interaction between business and distribution.

7 Discussions

7.1 Observing Ripple Effects

A further analysis performed was centred on identifying ripple-effects caused by changes that propagate between unrelated components. For example, if a change targets the Servlets then it would be hoped that these changes would be localised to the view layer. Any change which targets a particular concern and is propagated to other concerns is considered a negative change. Typically scenarios which target a

crosscutting concern (i.e. Scenarios 5 and 7 which target persistence and distribution) the AO paradigm is able to localise these changes to specific components. In comparison the change has to span multiple layers/concerns in the OO version.

In scenarios which typically target the layers of the HW system the OO implementation performs better or comparable to the AO implementations. This is interesting as the AO implementation has the same core behaviour (in terms of layers) so the changes should propagate in a similar manner. As in the OO version, the AO core layers require modification but changes also propagate to other (seemingly) unrelated concerns. For example, Scenarios 1 and 6 specifically target the view layer; the OO implementation is able to contain these changes entirely within this layer. However, the AO versions require other additional concerns to be modified such as distribution and pattern implementations but not at the expense of modularity. Although the OO implementation also has these dependencies, the increased tangling prevents the modifications from spreading to multiple components but do affect more operations and LOC. A similar ripple-effect occurred between the persistence and concurrency concerns within Scenario 3, whereby the AO implementations required modification of the timestamp behaviour but the OO implementation did not due to the differences in the levels of concern locality (Section 4.2).

However, within the same scenario (Scenario 3) and Scenario 5 the OO implementation demonstrates a significant weakness that occurs in the majority of the scenarios. This is the high fragility [2, 28] of the *HealthWatcherFacade* class (see Fig. 1) which is caused by the extremely high tangling of concerns within this class. Although this may not seem a bad property due to the fact that it appears to reduce the ripple-effects it does increase the complexity and reduces the modularity.

Generally, we can conclude that ripple-effects occur in both the AO and OO versions. The improved SoC within the AO versions causes the changes to propagate to more unrelated components, making the changes less obvious as unexpected components are affected. In turn the OO version requires more extensive changes within each affected component making these changes more obvious. These differences lead to a notion of 'deep' and 'wide' ripple-effects. The AO ripple-effects tend to go 'deeper' in that the changes propagate to unrelated components. In comparison the OO ripple-effects tend to go 'wider' in that the ripple-effects tend to more extensively affect the modified components.

This notion is illustrated in Scenario 8, which focuses on modifying the business, view and data layers. The changes in the OO version tend to be located in each of these layers (marked on Fig. 1), however, the AO version requires additional behaviour to be added in seemingly unrelated components (marked on Fig. 2). The concurrency concern within OO implementation requires a concurrency manager to be inserted directly into the new classes added (hence being "wider"). In comparison the AO versions needs extra logic to determine which class is currently being accessed and then delegate to the appropriate concurrency manager and is "deeper" by affecting an unobvious component.

It would appear that wider ripple-effects were less problematic due to the changes having to be made being more obvious. However, when taking into account the earlier modularity metrics it is clear that if these ripple-effects could be identified or limited then the AO paradigm would be superior. Pointcut fragility [29] is the significant factor that contributes to these AO ripple-effects. For example, in Scenarios 6 and 8

the re-factorings and component additions that take place invalidate pointcuts that are used to apply the Observer pattern. This results in unintuitive changes having to be made. This trade-off that must be accounted for; future AO techniques should take this pointcut fragility into account and allow more expressive and semantic-based pointcuts to be specified.

7.2 Architecture Stability Analysis

Our policy to analyse design stability at the architecture level followed the same principle and relied on similar measures as in the implementation level. The analysis at the architecture level was supported by a modularity suite of metrics based on measures for SoC, component coupling and cohesion, and interface complexity. However, now the metrics are defined in terms of architecture-level abstractions, such as components, interfaces and operations [30]. First, we compared the modularity of the OO and AO HW architectures in the base version. For SoC, we assessed the scattering of the concerns over the design elements and the architecture description. The considered concerns during the analysis were: distribution, persistence, concurrency, exception handling, view, and business.

The most significant difference between the two solutions was related to the persistence concern. The persistence concern was present in many more architectural elements in the OO solution than in the AO solution. For instance, in the OO architecture more components (5 vs. 2), more interfaces (22 vs. 9) and more operations (154 vs. 45) have the persistence concern. The reason for this phenomenon is that, in the OO architecture, the persistence-specific exceptional events are propagated from components in the data layer to components in the view layer. Therefore, these persistence exception events are handled by almost all interfaces between the data, business, distribution, view layers. In the AO architecture, these exceptions are caught and treated via the persistence aspect earlier in the interfaces of the data layer and do not propagate through the other interfaces and respective layers.

After applying the metrics to the base version, we applied the metrics for both OO and AO architectures in the other versions of the HW system. The goal was to analyse the impact of the evolution changes in the architecture modularity. The scenario which changes impact most in the architecture was Scenario 8. This occurred because this scenario demanded the addition of a number of operations in the interfaces between each connected pair of layers. Note that Scenario 8 impacts the boundaries of every layer in Fig. 1 and Fig. 2. For instance, it affected *IFacade*, *HealthWatcherFacade* and *IHealthUnitRep* interfaces in the distribution, business and data layers respectively. In fact, the measures showed that the persistence concern is more stable in the AO architecture. As the persistence concern is not well modularized in the OO architecture (as stated earlier), every operation added in Scenario 8 had to address the persistence concern. Each new operation had to consider the persistence-specific exceptional events. The concern metrics highlighted that the number of operations containing the persistence concern in the OO architecture increased 38 (from 154 in the base version to 192) in the version produced after applying Scenario 8. In comparison, the increase that occurred in the AO architecture was just one operation. This result confirmed the previous results for

the implementation level that Scenario 8 favours the AO implementation of the persistence concern due to the quantification mechanisms.

8 Related Work and Study Constraints

8.1 Related Work

The current body of empirical knowledge on AO techniques cannot explain the influence of compelling aspect interactions on long-term design stability. Lopes and Bajracharya [31] presented an analysis of modularity in AO design using the theory of modular design developed by Baldwin and Clark [32]. They have studied the design evolution of a Web Service application where they observed the effects of applying aspect modularisations using Design Structure Matrix and Net Option Value. It was an interesting first experiment that observed that there was added value in introducing aspects into an already modularized design. It contributes to an earlier work by Sullivan [33] by: (i) providing a realistic and modern example, and (ii) the analysis of effects of AOP on the value of the overall design. However, this study involved only the application of only two kinds of aspects: logging and authentication.

There is little related work focusing either on the quantitative assessment of AO solutions in general, or on the empirical investigation about the design stability of AO decompositions. Substantial empirical evidence is missing even for crosscutting concerns that software engineers face every day, such as persistence, distribution, and exception handling. There are several case studies in the literature involving the "aspectization" of such pervasive crosscutting concerns [10-12]. However, these studies mainly focus on the investigation on how the use of AO abstractions supports the separation of those concerns. They do not analyze other effects and stringent quality indicators in the resulting AO systems. Furthermore, they do not quantify the benefits and drawbacks of AO techniques in the presence of widely-scoped changes. A number of empirical analyses need to be carried out, since certain typical criticisms that AOP has suffered [34] and the initial studies have exhibited some controversies even when aspectizing classical crosscutting concerns, such as transaction management [10, 11], exception handling [12, 13], and design patterns [14-16].

We have previously performed a far-reaching maintenance study [24], but our target was aspects specific to multi-agent systems. In addition, the aspects used in this system have a localized scope and tend to affect a few modules; they do not have a major influence on the architectural design of the system. In addition, the introduced changes were restricted to simple changes in few classes or aspects. Also, we did not evaluate aspect interaction issues. In another study, we evaluated how AspectJ scales to modularize pattern compositions [15]. However, the stability of the aspectized pattern combinations was not assessed. Our previous study documented in [19] evaluated the scalability of the AspectJ implementation by performing some initial changes to the HW system. These changes correspond to a subset of the changes made in Scenario 8. However, this previous study did not examine concern interaction, design stability or the affect on design principles. Nevertheless, the study presented in this paper has confirmed and expanded in scope our previous findings.

8.2 Study Constraints

Although it can be argued that using a single system for such a study is a limiting factor we feel that the HW system is representative of modern systems and the scenarios applied are extensive and so reduces the necessity of additional systems. Naturally it is desirable to involve more systems and more approaches.

As stated previously one of the aims of this study was to perform a general comparison of the OO and AO paradigms. This was achieved by re-implementing HW using CaesarJ, however, a similar OO re-implementation was not performed for Java. As such the study could be viewed as a Java vs. AO comparison but we feel this would be unfair. Java is a good representative of OO techniques and re-implementing it would be a wasted exercise. Equally, it would be difficult to reproduce a similar implementation due to the techniques used in the OO implementation i.e. Servlets, etc. Fundamentally for this type of study we require good representatives of AO techniques for the base language. Unfortunately this range does not exist for other OO languages and so limits the benefits of studying other OO languages.

The concerns analysed tended to focus on ones that were more significantly affected by the changes applied in either the AO or OO versions. Due to the nature of the study and the fact that separation of concerns is central to this study the crosscutting concerns were naturally the ones which varied the most. We have also presented results of other non-crosscutting concerns (e.g. view and business layers) to provide a balanced comparison of the AO and OO paradigms.

The applicability and usefulness of some of the specific metrics used in this study has often been questioned such as the cohesion measure. We accept the criticism of such metrics. However, it is important to consider the results gathered from all metrics rather than just one metric in particular. The multi-dimensional analysis allows us to grasp which measurement outliers are significant and which are not. In fact, when drawing conclusions from the results we have considered all the gathered data and never relied upon one single piece of data from this set.

9 Concluding Remarks

The transfer of aspect-oriented technologies to mainstream software development is largely dependent on our ability to empirically understand their positive and negative effects through design changes. Software designs are often the target of unanticipated changes and, as a result, incremental development has been established as the defacto-practice in realistic software development [1, 5, 6]. This study has followed these practices to evolve a real-life application in order to assess various facets of design stability of object-oriented and aspect-oriented implementations. This included the analysis of the implementations modularity, change propagation, concern interaction analysis and identification of architectural ripple-effects.

From this analysis we have discovered a number of interesting outcomes. Firstly, the AO implementations tend to have a more stable design particularly when a change targets a crosscutting concern. Furthermore, changes tended to be much less intrusive and more simplistic to apply in the AO implementations. This indicates that aspectual decompositions are superior especially when considering the Open-Closed principle

[18] (Section 5). Significantly, both OO and AO implementations exhibited high stability in high-level design structures but with a certain architectural ripple-effect occurring within the OO design when persistence-related exceptions had to be introduced. In certain circumstances aspectual decompositions did perform worse. These tended to occur when evolutionary scenarios targeted classical design patterns such as Command and State [19], applying these design patterns violated pivotal design principles, such as narrow interfaces and low coupling. Even though as stated above the AO implementations tended to require less invasive changes, sometimes the modifications propagated to components that were not the direct target of the change scenario. The overall conclusion regarding the design stability and concern interaction analysis (Section 6) is that aspect decomposition narrows the boundaries of concern dependencies, however, with more tight and intricate interactions.

References

1. Larman, C., Basil, V.: Iterative and Incremental Development. A Brief History. IEEE Computer 36(6), 47–56 (2003)
2. Martin, R.: Engineering Notebook: Stability (1997)
3. Mohagheghi, P., Conradi, R.: Using Empirical Studies to Assess Software Development Approaches and Measurement Programs. In: Proc. Workshop on Empirical Software Engineering (WSESE'03), Rome (2003)
4. Yau, S.S., Collofello, J.S.: Design Stability Measures for Software Maintenance. IEEE Transactions on Software Engineering 11(9), 849–856 (1985)
5. Rajlich, V.: Changing the Paradigm of Software Engineering. Communications of the ACM 49(8), 67–70 (2006)
6. Casais, E.: Managing Class Evolution in Object-Oriented Systems. In: Nierstrasz, O., Tsichritzis, D. (eds.) Object-Oriented Software Composition, Prentice Hall, Englewood Cliffs (1995)
7. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Longtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings European Conference on Object-Oriented Programming, Jyväskylä, Finland, Springer, Heidelberg (1997)
8. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. In: 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA (2003)
9. Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., Staa, A., Lucena, C.: Quantifying the Effects of AOP: A Maitenance Study. In: Proc. of 9th Intl. Conference on Software Maintenance (ICSM'06), Philadelphia, USA (2006)
10. Rashid, A., Chitchyan, R.: Persistence as an Aspect. In: 2nd International Conference on Aspect-Oriented Software Development (AOSD), ACM, Boston, Massachusetts (2003)
11. Soares, S., Borba, P., Laureano, E.: Distribution and Persistence as Aspects. Software: Practice and Experience (2006)
12. Filho, F., Cacho, N., Figueiredo, E., Maranhao, R., Garcia, A., Rubira, C.: Exceptions and Aspects: The Devil is in the Details. In: International Conference on Foundations of Software Engineering (2006)
13. Lippert, M., Lopes, C.: A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In: 22nd International Conference on Software Engineering (1999)
14. Garcia, A., Sant'Anna, S., Figueiredo, E., Kuleska, U., Lucena, C., Von Staa, A.: Modularizing Design Patterns with Aspects: A Quantative Study. In: 4th International Conference on Aspect-Oriented Software Development (AOSD), Chicago, USA (2005)

15. Cacho, N., Sant'Anna, C., Figueiredo, E., Garcia, A., Batista, T., Lucena, C.: Composing Design Patterns: A Scalability Study of AOP. In: AOSD, Bonn, Germany (2006)
16. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), ACM, Seattle, Washington (2002)
17. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Getting Started with AspectJ. Communications of the ACM 44(10), 59–65 (2001)
18. Meyer, B.: Object-Oriented Software Construction, 1st edn. Prentice-Hall, Englewood Cliffs (1988)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Pattern, Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, London, UK (1995)
20. Greenwood, P., et al.: Aspect Interaction and Design Stability: An Empirical Study (2007), Available from: http://www.comp.lancs.ac.uk/computing/users/greenwop/ecoop07
21. Buschmann, F., et al.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley and Sons, Chichester (1996)
22. Hunter, J., Crawford, W.: Java Servlet Programming. O'Reilly and Associates Inc. 1998
23. Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., von Staa, A.: On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In: Brazilian Symposium on Software Engineering, Manaus, Brazil (2003)
24. Garcia, A., et al.: Separation of Concerns in Multi-Agent Systems: An Empirical Study. Software Engineering for Multi-Agent Systems, 2(2940) (2004)
25. Chidamber, S., Kemerer, C.: A Metrics Suite for Object-Oriented Design. IEE Transactions on Software Engineering 20(6), 476–493 (1994)
26. Bartolomei, T.T.: MuLaTo (2006), Available from: http://sourceforge.net/projects/mulato
27. Figueiredo, E., Garcia, A., Luena, C.: AJATO: An AspectJ Assessment Tool. In: European Conference on Object-Oriented Programming (ECOOP Demo), France (2006)
28. Colwell, B.: Design Fragility. Computer 37(1), 13–16 (2004)
29. Kellens, A., Mens, K., Brichau, J., Gybels, K.: Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, Springer, Heidelberg (2006)
30. Sant'Anna, C., et al.: On the Quantitative Assessment of Modular Multi-Agent System Architectures. In: NetObjectDays (MASSA) (2006)
31. Lopes, C., Bajracharya, S.: An Analysis of Modularity in Aspect-Oriented Design. In: Proc. Aspect-Oriented Software Development (AOSD), Chicago, USA (2005)
32. Baldwin, C., Clark, K.: Design Rules:The Power of Modularity. Vol 1. MIT Press, Cambridge (2000)
33. Sullivan, K., Griswold, W., Cai, Y., Hallen, B.: The Structure and Value of Modularity in Software Design. In: 8th European Software Engineering Conference, ACM Press, New York (2001)
34. Steimann, F.: The Paradoxical Success of Aspect-Oriented Programming. In: International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) (2006)