International Journal of Software Engineering
and Knowledge Engineering
Vol. 23, No. 7 (2013) 913–941
© World Scientific Publishing Company
DOI: 10.1142/S0218194013500265

World Scientific
www.worldscientific.com

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

QUANTIFYING THE EFFECTS OF ASPECTUAL DECOMPOSITIONS ON DESIGN BY CONTRACT MODULARIZATION: A MAINTENANCE STUDY

HENRIQUE REBÊLO[*,‖], RICARDO LIMA[*,**], UIRÁ KULESZA[†,§§],
MÁRCIO RIBEIRO[‡,‖‖], YUANFANG CAI[§,***], ROBERTA COELHO[†,¶¶],
CLÁUDIO SANT'ANNA[¶,†††] and ALEXANDRE MOTA[*,††]

[*]*Federal University of Pernambuco, PE, Brazil*

[†]*Federal University of Rio Grande do Norte, RN, Brazil*

[‡]*Federal University of Alagoas, AL, Brazil*

[§]*Drexel University, PA, USA*

[¶]*Federal University of Bahia, BA, Brazil*

[‖]*hemr@cin.ufpe.br*
[**]*rmfl@cin.ufpe.br*
[††]*acm@cin.ufpe.br*
[§§]*uira@dimap.ufrn.br*
[¶¶]*roberta@dimap.ufrn.br*
[‖‖]*marcio@ic.ufal.br*
[***]*yfcai@cs.drexel.edu*
[†††]*santanna@dcc.ufba.br*

Received 3 February 2012
Revised 14 January 2013
Accepted 28 February 2013

Although it is assumed that the implementation of design by contract is better modularized by means of aspect-oriented (AO) programming, there is no empirical evidence on the effectiveness of AO for modularizing non-trivial design by contract code in realistic development scenarios. This paper reports a quantitative and qualitative case study that evolves a real-life application to assess various facets of the adequacy of aspects for modularizing the design by contract concern. Our evaluation focused upon a number of system changes that are typically performed during software maintenance tasks. The study was driven by an analysis of fundamental modularity attributes, such as separation of concerns, coupling, conciseness, and change propagation. We have found that AO techniques improved separation of concerns and the design stability between the design by contract code and base application code throughout the development scenarios. However, contradicting the general intuition, the AO versions of the system did not present significant gains regarding four classical size metrics we employed.

Keywords: Design by Contract; aspect-oriented programming; refactoring; maintenance study.

1. Introduction

Design by Contract (DbC), originally conceived by Meyer [32], is a technique for developing and improving functional software correctness and quality [44, 3, 28, 27]. The key mechanism in DbC is the use of the so-called "contracts". A contract formally specifies an agreement between a client and its suppliers. Client classes must satisfy the supplier class conditions before calling one of its methods. When these conditions are satisfied, the supplier class must guarantee certain properties, which constitute the supplier class's obligations. For instance, when a client breaks a condition (client violation), a runtime error occurs. The use of such pre- and postconditions and invariants to specify software contracts dates back to Hoare's 1969 paper on formal verification [21]. The novelty with DbC is to make these contracts executable. This is useful for isolating errors during debugging, and for validating contracts that are used as documentation or for increasing code reliability and correctness [3, 42].

It is often claimed that the contracts of a system is de-facto a crosscutting concern that can be better modularized by the use of aspect-orientation (AO) [23, 30, 31]. Recent studies [30, 23, 3, 42, 38, 40] have shown that object-oriented (OO) abstractions are not able to modularize the main features of design by contract methodology, such as invariants and pre- and postconditions, and tend to lead to programs with poor modularity (scattered and tangled DbC code).

To the best of our knowledge, Lippert and Lopes [30] conducted the most well-known systematic study that explicitly investigated the use of AO to implement classical design by contract features such as pre- and postconditions of a large OO framework, called JWAM. Among other things, they compared the contracted Java and AspectJ implementations of such OO framework. According to their findings, the AspectJ implementation improved the modularity of design by contract concern. Also, they argue that the use of AO drastically reduced the number of contracts (e.g. precondition) and lines of code (LOC). However, the authors presented their findings in terms of a qualitative assessment. Quantitative evaluation in their study consisted solely of counting LOC. Hence, there is no empirical evidence that AO techniques promote a superior solution in well-understood modularity attributes such as separation of concerns, coupling, and conciseness, when used for modularizing non-trivial homogeneous and heterogeneous design by contract code. Moreover, they have not analyzed the scalability of AO for implementing design by contract concern in the presence of widely-scoped design changes. Hence, the understanding of the effects of AO decompositions to implement DbC concern on key maintainability-related software attributes are still deep challenges to software engineers.

This paper presents a systematic case study that quantitatively and qualitatively assesses the positive and negative impacts of AO techniques for implementing the classical design by contract features (pre- and postconditions, and invariants) on a number of design changes applied on a real-life web-based information system, called

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

Health Watcher (HW) [20][a]. The OO versions were implemented in Java, whereas the AO versions were implemented in AspectJ. The design stability evaluation of the Java and AspectJ versions were based on two conventional metric suites for modularity attributes [45, 14] (such as separation of concerns, coupling, and conciseness) and change impact [53]. Hence, our investigation complements in two senses the existing well-known qualitative empirical study on the use of AO for implementing design by contract concern performed by Lippert and Lopes work [30].

The main outcomes of our analysis in favour of AO implementations of design by contract concern were:

(i) The design by contract concern tended to show superior modularity stability in the AO designs; changes tended to be confined with the same modules and fewer operations were added due to the higher reuse achieved by AO techniques;
(ii) The "aspectization" of design by contract drastically reduced the number of design by contract features, specially the number of invariants;
(iii) AO solutions required less intrusive modifications (e.g. changes to existing operations and lines of code) of the design by contract concern throughout the applied maintenance scenarios;
(iv) Aspectual decompositions have demonstrated superior satisfaction of the Open-Closed principle [33] in all the maintenance scenarios;
(v) In all the maintenance scenarios, the use of aspects led to improvements in coupling in the implementation of design by contract features, such as preconditions.

Alternatively, the main findings against aspectual decompositions were:

(i) The use of size measures was helpful to show that the AO implementations of design by contract have not presented significant gains regarding four classical size metrics, for example, the AO implementations of DbC increased the number of lines of code and the number of operations of the studied target system throughout the maintenance scenarios;
(ii) Although invasive modification was more frequent in the OO solutions, the AO modifications tended to propagate to unrelated modules when performing OO refactoring changes;
(iii) In general the aspectization of design by contract has shown no improvement when dealing with heterogenous contracts.

This paper is structured as follows: Section 2 describes our experimental settings and justifies the decisions made to ensure the study validity. 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 the modularity metrics are discussed in Sec. 4. Section 5 discusses how the changes propagate within each paradigm.

[a] The work described in this article is the revised and extend version of a paper presented at SEKE 2011 [41].

Section 6 analyzes the obtained results and points some constraints on the validity of our study. Section 7 discusses related work. Finally, Sec. 8 concludes this paper by summarizing this paper's findings.

2. Experimental Settings

This section describes the configuration of our study. Section 2.1 briefly exemplifies and explains how we have implemented the design by contract concern using aspect-oriented decompositions. Section 2.2 discusses the choice of the target system. Finally, Sec. 2.3 describes our assessment procedures.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

2.1. *Aspectizing design by contract*

In order to enable the design by contract implementation and modularization, this work considers two implementation techniques: standard object-orientation features with Java and AO [24] decompositions. We chose AspectJ [23] to modularize DbC features with AO because it is the most consolidated AO language. Besides, our goal was to assess the suitability of core AO mechanisms for handling DbC modularization rather than other emerging AO mechanisms available in programming languages such as CaesarJ [34]. Java assertions, on the other hand, is a well-known technique for contract enforcement of Java code. Since our study explicitly distinguishes between pre- and postconditions, and invariants, we encapsulate Java assertions in three kinds of Java contract methods: JC.requires for precondition enforcement, JC.ensures for postcondition enforcement, and JC.invariant for invariant enforcement. Lippert and Lopes [30] use the same strategy to deal with Java contracts, except that they just consider pre- and postconditions.

Our study focused on the placement of contracts. We refactored all the JC.requires, JC.ensures, and JC.invariant calls in the selected portions of the selected target system to aspects. These methods are declared in the JC class which encapsulate all the Java contract (assertion) operations. We used the Extract Fragment to Advice [35] refactoring to move contracts to aspects. Figure 1 illustrates these mechanics. It shows a trivial example of aspectization of preconditions using a **before** advice. Note that since the two methods of the class C have the same precondition α, we were able to refactor it to single advice, hence exploring reuse opportunities. Likewise, we modularize the postcondition β in a single **after returning** advice (see Fig. 2). We use **after returning** advice since the postcondition should be established just after the normal termination of a method. In this paper, we do not consider exceptional postconditions [28, 42]. Regarding invariants, we use both **before** and **after returning** advice for modularizing invariant constraints (see Fig. 3). According to the semantics of invariants [28, 42], they should be established just after the normal termination of a constructor's execution and before and after execution of every instance method of a particular class. The left hand side of Fig. 3 illustrates how scattered and tangled a Java implementation of an invariant constraint becomes.

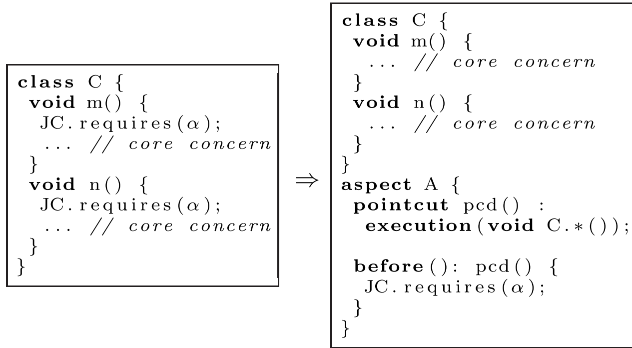

Fig. 1. Refactoring precondition code to aspects.

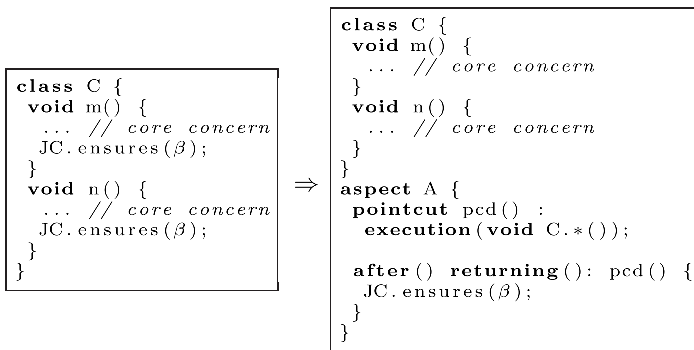

Fig. 2. Refactoring postcondition code to aspects.

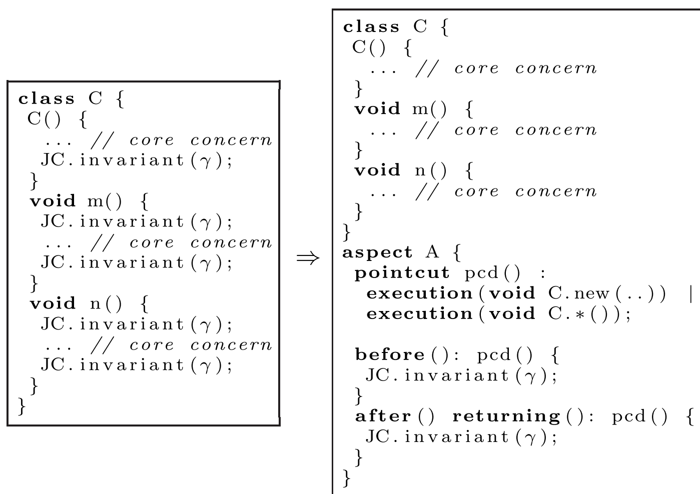

Fig. 3. Refactoring invariant code to aspects.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

2.2. *Target system selection*

The first major decision, that we had in our investigation, was the selection of the target system. The chosen system is a real web-based information system, called Health Watcher (HW) [20]. The main purpose of the HW system is to allow citizens to register complaints regarding health issues. This system was selected because it addresses a number of relevant criteria for our intended evaluation. First, it is a real and non-trivial system with available OO and AO implementations with a number of recurring concerns and technologies common in day-to-day software development, such as GUI, persistence, concurrency, RMI, Servlets and JDBC [20]. Second, the original implementation of HW is composed by eleven use cases that are detailed by an available requirements document, which is essential to understand its main functionalities [20]. Third, other qualitative and quantitative studies of the HW system have been recently conducted [26, 15, 20, 8], and so provided a solid foundation for this study.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

2.3. *Assessment procedures*

The main goal of this empirical case study is to answer how a system behaves regarding design by contract modularity when implemented with AO techniques. To this end, we analyzed the impact of DbC modularity through a set of software modularity, change propagation, and pluggability attributes as main driving design criteria.

The study was divided into three major phases: (i) implementation and alignment of the HW design by contract concern according to its requirements document; (ii) implementation of change scenarios, and (iii) quantitative assessment of the OO and AO versions of the HW system along with their change scenarios.

Development of the HW Base Release. In the first phase, we implemented the design by contract concern for the OO and AO base release of the HW system, which is already available and implemented in Java and AspectJ [20], respectively. As mentioned above, HW comprises several classical crosscutting concerns, but no existing quantitative work have explored the design by contract one. Hence, we analyzed the, entire available, requirements document of the HW system to understand its functionalities and involved actors. This analysis was fundamental to recognize the HW's design constraints in terms of pre-, postconditions, and invariants [20]. The implementation comprehends both homogeneous and heterogeneous contracts for the HW use cases. We found some inconsistences of the original HW implementation by its validation with contracts. Since this task is out of scope, we just mention that we made an alignment (fixing the found bugs) of the HW implementation to fulfil its requirements. Moreover, a number of test cases were exhaustively used for all the used releases of the Java and AspectJ versions of the HW system. This assure that the comparison conducted between the object-oriented (OO) and aspect-oriented (AO) versions was fair.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

HW Change Scenarios. The second phase involved the implementation of four changes (Sec. 3.3) in the original HW system [20]. Each change involved: (i) the design improvement through refactoring and design patterns [2, 29, 18], (ii) the design and implementation of new functionalities (comprising six new user cases) to be included, (iii) the removal of one use case from the system, and (iv) the complete removal of the design by contract concern.

HW DbC Modularity Assessment. The goal of the third phase was to compare in a quantitative way the design by contract modularity of AO and OO implementations of HW system. In order to support a multi-dimensional data analysis, the assessment phase was further decomposed in three main stages. The first two stages (Sec. 4.2) are aimed at examining the overall maintenance effects (regarding DbC concern) in fundamental modularity attributes through the HW used releases. The last stage (Sec. 5) evaluates the OO and AO implementations from the perspective of change propagation and pluggability. Traditional metrics were used in all the assessment stages, and will be discussed in the respective sections. All measurement results are available from [39].

3. Health Watcher System

This section illustrates and describes the OO and AO architectural designs of the HW system. Section 3.1 briefly describes the HW OO design. The AO design of HW is discussed in Sec. 3.2. In addition, we discuss the selected change scenarios applied to HW in Sec. 3.3.

3.1. *OO architectural design*

The OO version of the HW system is implemented using the Java programming language. The Layer architectural pattern [4] is used to structure the system classes into three main layers: GUI (Graphical User Interface), Business, and Data. Figure 4 presents a partial class diagram of the OO implementation. It illustrates the main architectural elements. For instance, 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 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 [2, 29, 18] are used in the design of the HW layers to achieve a reusable and maintainable implementation.

Figure 4 presents other details about the HW system such as:

- the Distribution concern which is responsible for making distributed the system services provided by the Business layer;
- implementation of concurrency control mechanisms in business and data classes;
- implementation of the design by contract concern in all the system layers.

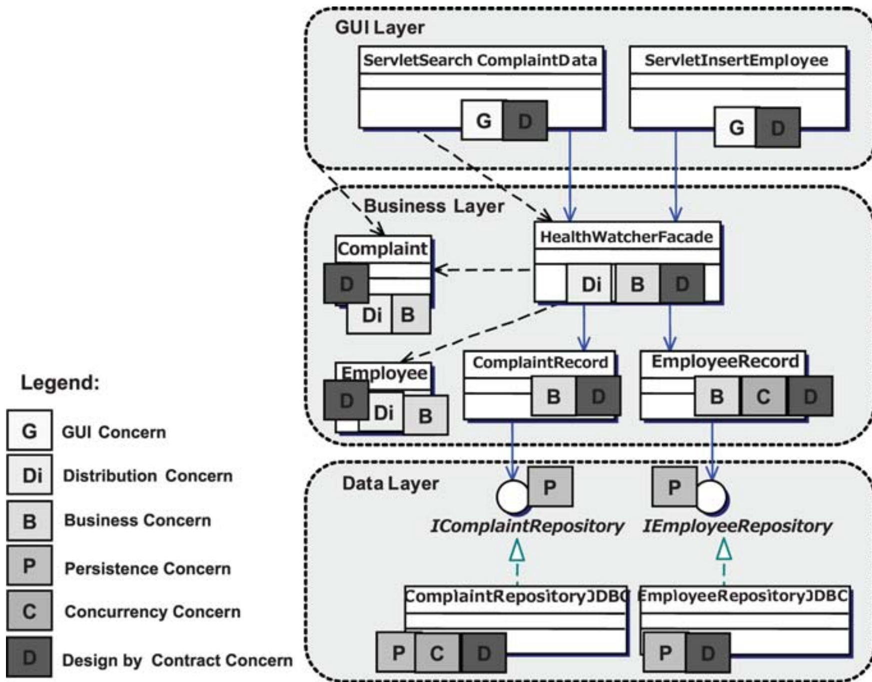

Fig. 4. Health Watcher Object-Oriented Design.

The design of the OO HW system fails to completely modularize the design by contract concern (focus of our work). Even though the HW system is organized in layers and uses several design patterns, the realization of DbC is completely scattered and crosscuts the entire HW layers.

3.2. *AO architectural design*

The AO version of the HW system was implemented using AspectJ [23]. The design followed the same Layer architectural pattern [4] to structure the system into three layers. Unlike previous works [26, 20], we modularized only the design by contract concern (not considered by such works), since it is our main concern of interest.

Figure 5 shows the design of the AO HW system version. A UML stereotype ≪aspect≫ is used to represent the DbC aspects of the system. Moreover, UML dependency relationships with the ≪crosscuts≫ stereotype indicate that an aspect is introducing behavior in the system classes. As we can see in Fig. 5, different aspects modularize the design by contract crosscutting concern extracted from the OO implementation. Hence, we were able to isolate the DbC concern as aspects. Besides, the physical separation, the main benefits and drawbacks to use aspect decompositions to modularize contracts are discussed in Secs. 4 and 5.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

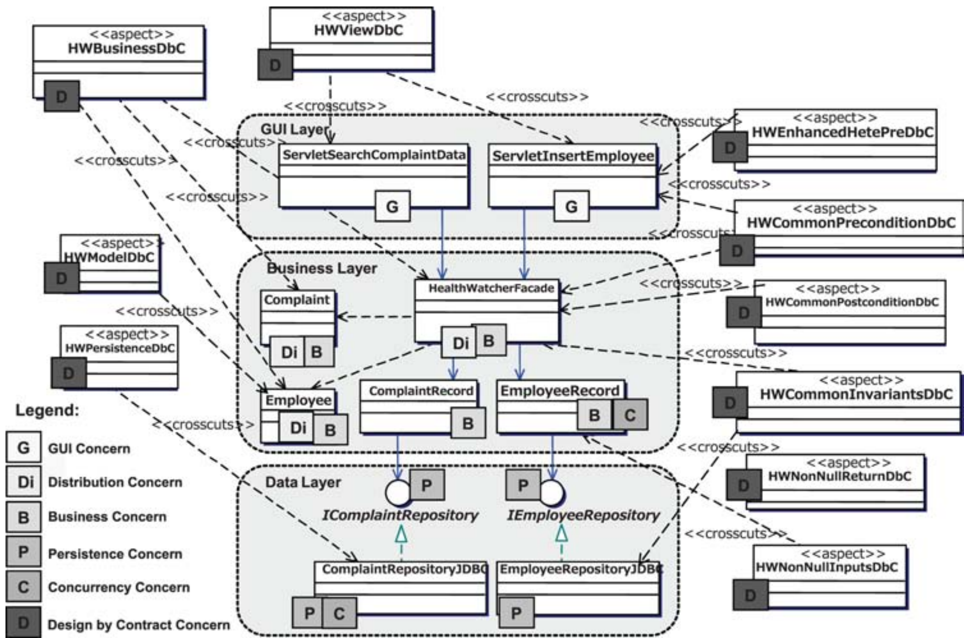

Fig. 5. Health Watcher Aspect-Oriented Design.

3.3. *Change scenarios*

As mentioned in Sec. 2.3, in the second phase of our investigation, we evolved both OO and AO implementations of the HW base release according to a set of change scenarios. Table 1 summarizes changes made in each release. The scenarios comprise different types of changes involving refactoring and improvement, addition of six new functional use cases (which represent typical operations encountered in the maintenance of information systems), removal of one functional use case, and the complete removal of the design by contract concern. Table 1 also presents which type of change each release encompassed. The purpose of these changes is to exercise the implementation of the design by contract features boundaries and, so, assess the overall modularity of contracts in the presence of maintenance and evolution tasks that are recurring in incremental software development.

4. Modularity Analysis

We have described how the assessment phase was organized in three stages (Sec. 2.3). This section presents the results for the first two stages, where we analyze the initial modularity of each OO and AO solutions of the HW base version (Sec. 4.1). Then we analyze their stability throughout the change scenarios (Sec. 4.2). We used a metrics suite (Table 2 summarizes each metric used in this case study) that quantified three fundamental modularity attributes, namely separation

Table 1. Summary of change scenarios in Health Watcher.

Release	Descriptions	Type of change
R1	Addition of two Adapter patterns and two Factory pattens to improve the distribution, data, and view concerns	Refactoring and improvement
R2	Addition of six new functional use cases: InsertMedicalSpecialty, and UpdateMedicalSpecialty and InsertHealthUnit and InsertSymptom and UpdateSymptom	Addition of functionality
R3	Removal of one use case: SearchSpecialtiesByHealthUnit	Removal of functionality
R4	Removal of the design by contract concern	Removal of the design by contract concern

os concerns (SoC), coupling, and conciseness [45, 14]. Such metrics were chosen because they have already been used in several experimental studies and proved to be effective maintainability indicators [19, 26, 15, 20, 8, 12].

The metrics for coupling and size were defined based on classic OO metrics [7]; the original metrics definitions were extended to be applied in a paradigm-independent

Table 2. The metrics suite.

Attributes	Metrics	Definitions
Separation of Concerns (SoC)	Concern Diffusion over Components (CDC)	Number of classes and aspects that contribute to the implementation of a concern [19].
	Concern Diffusion over Operations (CDO)	Number of methods and advice that contribute to a concern's implementation [19].
	Concern Diffusion over LOC (CDLOC)	Counts the number of transition points for each concern in the LOC. Transition points are points in the code where there is a "concern switch" [19].
Coupling	Coupling Between Components (CBC)	Number of classes and aspects declaring methods or fields called or accessed by other components [7].
Size	Lines of Code (LOC)	Number of lines of code [7].
	Design By Contract Lines of Code (DbCLOC)	Number of lines of code that are relative to DbC.
	Number of Preconditions (NOPre)	Number of preconditions of each class or aspect.
	Number of Postconditions (NOPo)	Number of postconditions of each class or aspect.
	Number of Invariants (NOI)	Number of invariants of each class or aspect.
	Number of Attributes (NOA)	Number of attributes of each class or aspect [7].
	Number of Operations (NOO)	Number of methods and advice of each class or aspect [7].
	Vocabulary Size (VS)	Number of components of the system [7].

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

way, supporting the generation of comparable results. The size metric group also includes metrics for both general system attributes (e.g. Number of Lines of Code) and metrics that are specific to design by contract such as Number of Preconditions (NOPre). The size metrics related to DbC are useful to quantify reuse of design by contract code in existing systems. In addition, this suite introduces three new metrics for quantifying SoC [45, 14]. They measure the degree to which a single concern (design by contract, in the case of this study) in the system maps to: (i) components (i.e. classes and aspects) — based on the metric Concern Diffusion over Components (CDC), (ii) operations (i.e. methods and advice) — based on the metric Concern Diffusion over Operations (CDO), and (iii) lines of code — based on the metric Concern Diffusion over Lines of Code (CDLOC). The majority of these metrics can be collected automatically by applying an existing measurement tool [13]. Additionally, we used the AOP metrics tool [1] to collect the coupling (CBC) metric.

The SoC metrics require the manual "shadowing" of the code (i.e. identifying which segment of code contributes to the DbC concern such as pre- and post-conditions). Although the mapping of DbC features to the source code is not completely automated, it is facilitated with tool support [43]. For all the employed metrics, a lower value implies a better result. Detailed discussions about the metrics appear elsewhere [45, 19, 14]. The complete description of the gathered data, measurement tools, and shadowed code is also available at [41].

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 implementations of HW system. Instead of analyzing each individual metric result, we provide a general view of the meanings behind the results. Figure 6 presents the modularity results for SoC, coupling, and size in the base version regarding design by contract crosscutting concern.

The application of the SoC metrics was useful to quantify how effective was the separation of the design by contract concern in the OO and AO implementations of the HW system (Fig. 6). Hence, a careful analysis of the measures determines that

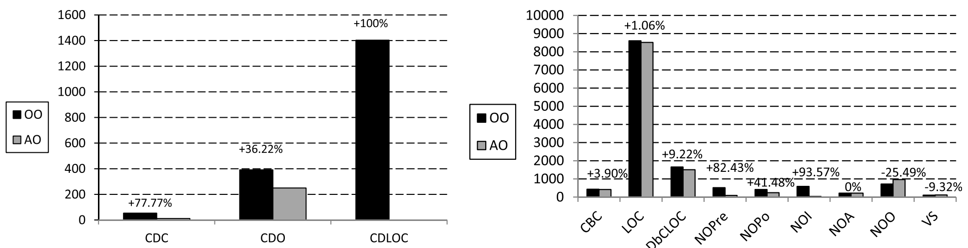

Fig. 6. Relative SoC, coupling, and size metrics values for design by contract of the base versions of Health Watcher.

the AO implementations offer superior modularity (for all the SoC metrics: CDC, CDO, and CDLOC) in these initial HW versions.

Regarding coupling, as observed, there is a small difference in favor of the AO implementation of HW system. Aspects reduced the coupling between system classes by removing the DbC-related code from them. However, the aspects still need to reference and, thus, are coupled to classes on which they introduce the DbC behavior. Hence, we had only 3.90% percentage reduction in favor of AO solution (see the CBC metric in Fig. 6).

Contradicting the general intuition that aspects make programs smaller due to reuse [30, 23, 48, 26], the OO version and its counterpart in AO did not present significant gains in relation to the four classical metrics: Vocabulary Size (VS), Number of Operations (NOO), Lines of Code (LOC), and Number of Attributes (NOA). The VS metric in Fig. 6 shows that the AO implementation needed to define 9.32% more components (classes + aspects) than the OO version. This difference is justified by the presence of several new aspects in the AO implementation of the HW system. Such aspects are used to modularize the design by contract concern which is completely tangled and scattered in all the system layers.

The Number of Operations (NOO) grew significantly in the AO version due to the modularization of DbC with new mechanisms such as advice. As a result, we had 25.49% more method-like definitions in the AO version. In the HW system, the difference of the number of LOC was only 1.06% in favor to AO solution. Hence, even with significant reuse of design by contract code (illustrated by the specific DbC size metrics NOPre, NOPos, and NOI), the aspect code used for realizing the DbC concern requires a lot of extra idioms which led to extra effort during implementation. This finding, contradicts the Lipert and Lopes study [30], on which they had a reduction of more than 50% in DbC LOC due to reuse. While HW system has homogeneous contracts that can be significantly reused, some heterogenous contracts can be harmful to the final LOC due to the poor reuse and the extra aspect code needed to "aspectize" the design by contract concern.

Finally, we had no difference between the two versions in relation to the Number of Attributes (NOA) and do not bring any interesting insights. The subsequent sections will analyze how these modularity properties alter due to the application of change scenarios (Sec. 3.3).

4.2. *Quantifying modularity of change scenarios*

After producing an overview of the modularity attributes in the base versions, we proceeded with the analysis regarding the stability of modularity attributes in the presence of change scenarios. Generally, any variation to their values is considered undesirable and indicate instability. However, variations are unavoidable, particularly when a certain module is the focus of an implementation change. Hence, variation in the values can be unavoidable or negative. Unavoidable variations occur when a component that is directly related to the affected concern is modified. For example, if a

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

scenario targets the DbC concern (which is the focus of this work), then variations in the metric for the related modules are generally expected and unavoidable. Since, the DbC concern crosscuts the entire layer of the HW system among several modules, let us say that in a particular development phase we are interested on the contracted modules of the GUI concern. Hence, the change propagation of adding, removing, or changing a contract in the view related modules are expected and also unavoidable. However, if unrelated modules are affected, these changes should be considered as being a negative variation. In summary, the approach with most stable design is the one which minimizes the number of negative variations.

In relation to the change scenarios (discussed in Sec. 3.3), it is important to stress that we do not consider the change scenario denoted by the HW release 4. Such scenario is useful to assess in a qualitative and quantitative way the pluggability of the design by contract concern (Sec. 5.3). Hence, since the last HW release does not contain the DbC concern anymore, it does not make sense do quantify the SoC, coupling, and size of this version. In the following we present the most significant results for each modularity attribute regarding the other HW change scenarios. (All the absolute values are available in [41].)

Separation of Concerns. Figure 7 presents the metrics results for the design by contract concern regarding separation of concerns. It shows that the widely-scoped crosscutting nature of DbC have presented superior design stability when implemented using AO techniques. 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) metrics also presents a very superior design stability for the AO implementations. This divergence largely comes from the quantification properties in AO, where the use of existing pointcut

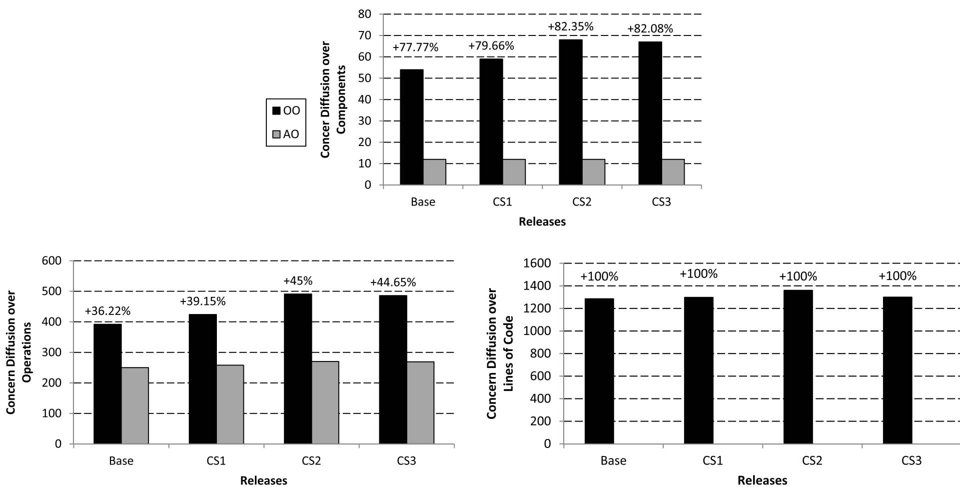

Fig. 7. Changes in the SoC metrics for the design by contract concern.

declarations eliminates the need for declaring operations (advice in this case). Even though the evolution scenario (denoted by the HW release 2) extends the system with six new use cases, quite few aspect-like methods were introduced. Hence, the AO design remained stable even when new functionalities are added. The results for concern diffusion over lines of code (CDLOC) provide additional evidence that AO implementations present better stability.

Coupling and Size Measures. Both OO and AO implementations present very similar stability with respect to Coupling Between Components (CBC) and size metrics. Figure 8 illustrates the curves results for coupling and size metrics of Health Watcher. The CBC graphic shows that the AO implementations fared better, but in a not very significant way. It happens mainly because, although many aspects reduce the coupling of system classes by modularizing the design by contract crosscutting concern, they still reference the classes in which they introduce some behavior. Despite the small difference between OO and AO regarding coupling, the AO implementation tend to be better while applying the changing scenarios. This happens because during the maintenance activities performed by the changing scenarios, it was observed that no new aspects were created and the existing ones provide few heterogenous DbC features. As a result, even though the VS graphic shows that the OO implementations performed better than AO, the difference between them remained constant (due to zero new aspects). Hence, the increase in the VS metric is justified by the introduction of aspects to modularize the DbC concern. The same reason justifies the worse results of AO implementations in relation to the Number of Operations (NOO) metric. The main difference is that this metric varies in each release due to some heterogenous contracts and changes scenarios (e.g. adding/removing functionalities).

Last but not least, the Lines of Code (LOC) values present not significant gains for AO implementations. It is often claimed that AO solutions tend to have less LOC

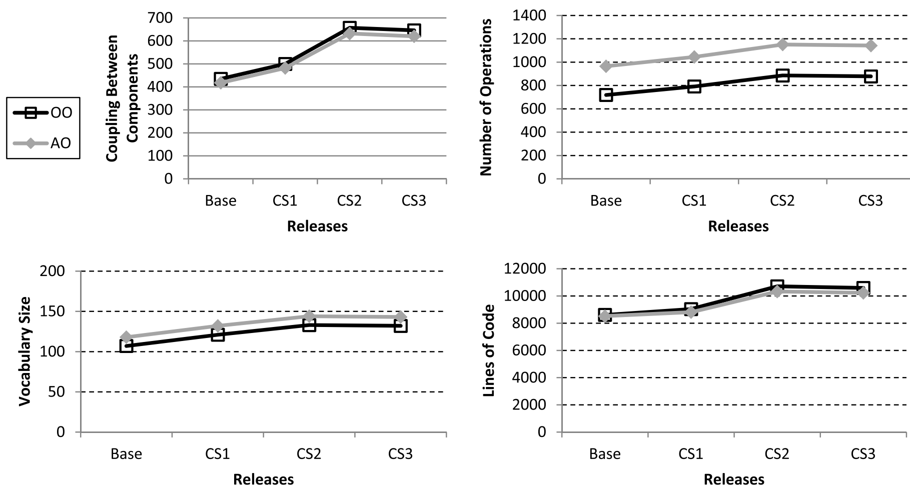

Fig. 8. Coupling and size metrics variation through the analyzed 4 Health Watcher releases.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

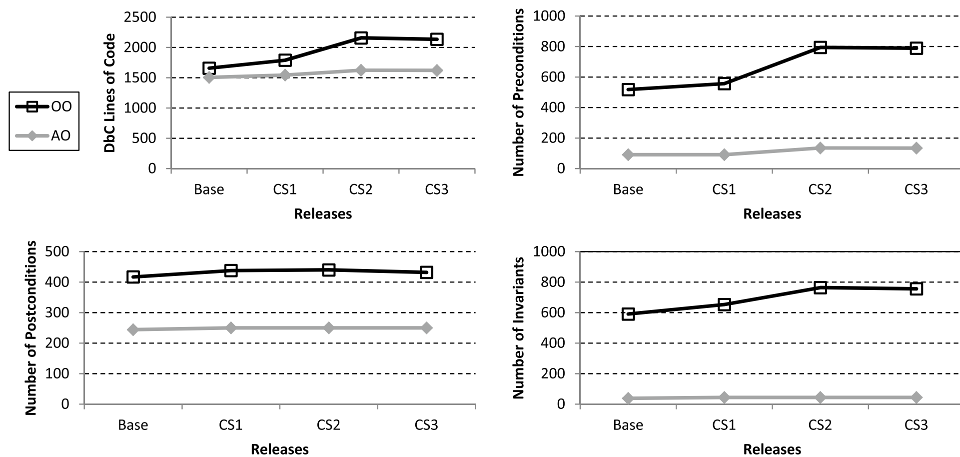

Fig. 9. Specific size metrics for quantifying design by contract reuse.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

due to reuse [30, 23, 48, 26]. However, in the context of design by contract implementations we do not see this effect. Even with higher reuse performed by the use of aspects (discussed in the following), the heterogenous contracts and the extra code used to expose the intercepted join points as well as their context information tend to increase the lines of code related to contract concern.

Size Measures for Design by Contract Concern. Figure 9 presents the results related to design by contract reuse. Observing the curves, AO implementations have much more stable values for all the DbC-related size metrics. The overall DbC Lines of Code (DbCLOC) shows that the AO implementations tend to be better after maintenance activities. This metrics take into account not only the code of pre- and postconditions, and invariants, but all the DbC related code involved for the realization of such concern. In the case of aspects, we mention all the code related to expose the intercepted join points and their context values. Due to this extra code, the gains of AO implementations were minimized against OO ones. However, when analyzing the reuse of DbC features, we can observe that the AO implementations fare better than OO implementations regarding pre- and postconditions, and invariants reuse. Also, the preconditions and invariants presents the higher reuse of the overall DbC code. The main reason is due to heterogenous contracts being more present in postconditions. Despite this, we had a mean of 42.44% of reuse of postconditions. In summary, the design by contract concern tend to be very stable when implemented with AO in all change scenarios analyzed.

5. Change Impact Analysis

Section 2.3 described how the assessment phase were organized in three stages. This section presents the last stage where we quantitatively analyze to what extent each maintenance scenario entails change propagation in the OO and AO HW

implementations. This phase relies on a suite of typical change impact measures [53], such as number of components (classes and aspects) added or changed, number of added or modified lines of code (LOC), and so forth. The purpose of using these metrics is to quantitatively assess the propagation effects when introducing or changing a specific feature, in terms of different granularities: components, operations, and LOC. Besides, the suite includes metrics to assess the changes in *pointcuts* and mainly related to DbC feature declarations (e.g. preconditions). The lower the change impact measures the more stable the design is to a certain change.

It is important to note that this change propagation metrics is related exclusively to measure the impact of the design by contract concern in the overall HW system. For instance, once a component is added to a change scenario, it is only considered if it contains DbC-related code; otherwise it is discarded from the assessment.

Table 3 shows the change propagation in the Health Watcher design as it evolves through the change scenarios (Table 1). In order to analyze the results more closely and to identify specific reasons for the applied changes, we classified the scenarios into three groups: adherence of Open-Closed principle [33] (Sec. 5.1), comparison when implementing a perfective change (Sec. 5.2), and quantifying DbC pluggability (Sec. 5.3).

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

Table 3. Measures of change propagation in Health Watcher.

Releases			CS1	CS2	CS3	CS4
Components	Added	OO	7	9	0	0
		AO	0	0	0	0
	Removed	OO	1	0	1	1
		AO	0	0	0	12
	Changed	OO	1	15	5	67
		AO	9	3	4	0
Operations	Added	OO	54	60	0	0
		AO	11	12	0	0
	Removed	OO	18	0	5	0
		AO	2	0	1	269
	Changed	OO	1	8	1	482
		AO	1	0	0	0
DbC LOC	Added	OO	217	335	0	0
		AO	70	162	0	0
	Removed	OO	82	0	22	2135
		AO	11	10	8	1621
	Changed	OO	0	0	1	0
		AO	44	0	0	0
DbC features	Added	OO	217	335	0	0
		AO	14	54	0	0
	Removed	OO	82	0	21	1977
		AO	0	10	1	428
	Changed	OO	0	0	1	0
		AO	0	0	0	0
PCs	Added	AO	7	18	0	0
	Removed	AO	0	0	1	282
	Changed	AO	11	3	3	0

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

5.1. *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 behavior is confirmed in the change propagation metrics (Table 3) whereby much more extensive changes (in terms of added operations and LOC) occur in the OO implementations. Up to 80% fewer operations and up to 51.54% fewer LOC (note that this LOC is only related to DbC features such as preconditions) are added in the AO implementations throughout the scenarios. This indicates that the AO solutions conform more closely to the Open-Closed principle [33] which states that "software should be open for extension, but closed for modification".

The change scenario which best illustrates this difference is denoted by the HW release 2 (CS2). The purpose of this scenario is to add six new contracted use cases to the HW system. This involves modifying a sub-set of the HW classes in all layers (which is acceptable to perform evolution). However, the OO implementation requires further modification to add features related to the design by contract concern. The AO implementation is able to quantify and capture these changes involving DbC features via pointcuts rather than modifying existing ones in the base code. Moreover, still regarding change scenario 2, the AO solution is still superior in terms of added (DbC) LOC.

5.2. *Comparison between OO and AO when implementing perfective changes*

When considering refactorings and improvements to the overall design of the HW system (the addition of two adapter patterns [2] and two factory patterns [18]) in terms of extensibility and reusability, the AO solution fared better than its counterpart in OO. Observing the change propagation values of the CS1 (change scenario 1), we can note that in the OO version we needed to add much more DbC features to new components; whereas in AO version, we just reused the existing ones. One crucial evidence of the benefits of the AO implementation is related to the number of added DbC features. We had to add 217 features in OO against only 14 in AO version. This indicates an evidence that the AO solution tend to be much more stable in relation to propagation changes involving perfective changes than OO solution. Also, this shows that, in contrast to the OO solution, the existing aspects could be easily expanded to incorporate new features.

5.3. *Quantifying design by contract pluggability*

Finally, the AO implementations are clearly superior in the change scenarios CS3 and CS4, which basically involves the partial and total removal of the design by contract concern. The change scenario CS3 is responsible for removing a use case in the HW system and its DbC-related code. The change scenario CS4 goes beyond

where it completely removes the design by contract concern of the final production HW system. We consider that a design by contract code is completely removed from the system when we exclude or comment the lines of code related to its realization on classes or aspects. Hence, as observable in Table 3, the effort to unplug or plug a DbC feature such as preconditions is much easier in the AO implementation of HW system. We need less changes in components, and also fewer lines of code need to be eliminated to partial or complete remove the DbC concern from the HW system. Even though an aspect declaration has a lot of pointcuts, advice, and DbC-related features, the effort to completely unplug the design by contract concern in practice is to remove 11 aspects from the HW build and remove the class which encapsulate all the DbC operations; whereas in the OO version, we need to make 67 invasive changes to OO classes to remove several lines of code related to DbC.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

6. Discussions and Lessons Learned

This section presents an overall analysis of the previously observed results on the application of the modularity and change propagation metrics, described in Secs. 4 and 5. We present discussions on the effects of AO decomposition on DbC modularization in different maintainability facets of the Health Watcher system. Furthermore, we discuss the constraints on the validity of our empirical case study.

6.1. *Observing ripple effects*

A further analysis performed in this study was related to identify possible ripple-effects caused by changes that propagate between unrelated modules. As explained, if a change targets the HW Servlets, it would be expected to have changes localized to the view layer. Any other particular change that is propagated to other concerns (other layers) is considered a negative change.

This notion is illustrated in the change scenario 2, which focuses on modifying the HW layers to add new use cases. The activity to add constrained modules (contracted classes and methods) is performed by one layer (concern) at a time. For instance, there are situations where the view layer is under modification to adhere the new constrained modules, but the DbC features (such as preconditions) are propagated to other layers such as the business layer. More specifically, in the view layer there is an interface called IFacade, but as a limitation of the OO decomposition, we cannot add any precondition to this component, so we should propagate this change to the modules affected (the classes which implement such interface) by such interface. As a result, we have two classes implementing this interface: the RMI-ServletAdapter in the same view layer and the HealthWatcherFacade class in the business layer. This way, the attempt to add a precondition to a method of the IFacade interface, results in changes in two components, one of them localized in another unrelated layer (the business layer where the HealthWatcherFacade class is

declared). In the same sense, while reasoning about the correctness of a method at the RMIServletAdapter class, if we change a precondition in this class, we might change the related precondition in the HealthWatcherFacade class as well. Thus, these result in negative changes caused by OO ripple-effects. These negative changes can be avoided in AO implementation since AspectJ-like languages can instrument an interface and its hierarchy.

However, within the change scenario 1, which focuses on refactoring and improvement by the addition of design patterns [2, 18], such as adapter [2] and abstract factory [18], we can observe in Table 3 that the AO solution also presents some ripple-effects. We had to change 9 existing components (aspects), whereas only one class in the OO version had to be changed. Pointcut fragility [49, 22, 37] is the significant factor that contributes to these AO ripple-effects. The object-oriented refactorings [17] invalidate pointcuts that are used to apply to the existing DbC concern. This results in unintuitive changes having to be made. This is one of the trade-off that must be considered while adopting AO techniques. Future AO techniques could take this pointcut fragility into account and allow more robust pointcut specifications, thus avoiding undesirable ripple-effects. Once this AO ripple-effect is handled with refactoring of AO code to expose the refactored join points in the OO version, we can observe that the remaining change propagation results fared better for AO solution (as explained in Sec. 5.2).

Generally, we can conclude that ripple-effects occur in both OO and AO implementations in practice. The modularized design by contract concern within the AO versions can be broken by OO refactorings resulting in less obvious changes to unrelated components. In turn the OO versions can also require unrelated changes to other modules (e.g. business layer) while changing, for example, a precondition localized at the HW's view layer.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

6.2. *Design by contract aspects reuse versus lines of code*

We have observed that the presence of reusable pointcuts brought some benefits when the HW system was modified. In the AO versions of the HW system, there are sets of reusable pointcut declarations related to the design by contract concern. These aspects have drastically contributed to the decrease in the overall number of DbC features of each final system after the application of the respective change scenario. These benefits can be observed in the complete analysis of all the change scenarios of the HW system. As a result, we had a reuse of invariants in each AO implementation higher than 90%, which give us an overall mean of 93.81% of reuse against OO decompositions when modularizing the design by contract concern.

Despite the fact we had significant gains of SoC metrics and reuse of DbC features in favor of the AO implementations of HW system, we found out that reusing contracts in some cases can be more difficult than usually advertised [30]. Contracts reuse depends directly on their types (e.g. postconditions) and mainly if such contracts is homogeneous or heterogenous [9]. Since an invariant crosscuts several

methods in a single class, it is naturally more reusable than pre- and postconditions that are relative to one or few particular methods. Hence, the reuse of pre- and postconditions are directly related to their homogeneity [9]. In other words, if several constrained methods present an intersection of common contracts, their reusability can be improved. As an example, similarly to Lippert and Lopes [30], we found that several methods in HW present the following homogeneous postcondition: JC. ensures(result != null). This postcondition states that every method using this contract must return an object that is non-null. The same situation also occurred for preconditions on input object parameters.

In the HW system we observed that the reuse of postconditions was quite low (a mean of 50%) when compared with preconditions (more than 80%) and invariants (93.81%). This scenario happens due to postconditions in HW being more heterogenous than preconditions or invariants. With this finding, we can conclude that the more heterogenous is a contract, its reuse with AO programming is minimized. We discuss beneficial and harmful scenarios of DbC aspectization in Sec. 6.4.

Lippert and Lopes [30] discuss that by using AO decompositions they could reduce more than 50% of the total design by contract LOC due to the reuse. However, contradicting this general intuition that aspects make programs considerably smaller, we found that despite the higher reuse of DbC concern (as previously discussed) with the AO versions of HW system, the gains in terms of the overall system LOC was only 2.65% and 18% considering exclusively the LOC of DbC concern. This was a directly consequence of heterogenous contracts and the extra LOC used to intercept the constrained join points, the contextual information, etc. [23].

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

6.3. *Stability and scalability of the design by contract concern*

One of the most significant results from the applied stability metrics is the number of unique AO components modified through the four scenarios applied. Since the base version, no new aspects were added to the HW system. Every change was inserted on existing design by contract aspects. Another point to highlight was the reuse achieved by the AO solution. Quite few DbC features were added to existing aspects due to quantification properties which enabled the higher reusability performed by the AO solution (also highlighted in the previous Sec. 6.2).

Similarly, when analyzing the change scenarios in terms of LOC, the AO solution again showed more stability in comparison to OO. This happens even when the aspect code needs to cope with more extra code[b] to expose the intercepted join points. The overall reduction of LOC in favor of AO was only 24%, but the reusability achieved by the quantification properties made this few reduction worthwhile when adding, for example, new preconditions and invariants.

[b] We mean by extra code the code used to expose the *join points* using *pointcuts* and the behavior written within the *advice*.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

In order to analyze the scalability of both OO and AO implementations in the change scenarios when referring to design by contract concern, we have used the collected values of both SoC and change propagation metrics. We consider a solution as scalable if the evolution of its implementations did not impact a number of modules that is higher than the number of modules affected before the changes. Comparing the obtained results in the base and change versions of the HW, we observed that the AO implementations were much more scalable than the OO ones. For instance, the CDC values shows that the OO implementations required changes in more components than AO implementations.

6.4. *Representative scenarios*

This section presents some examples of scenarios that were identified during the process of refactoring Health Watcher. These scenarios represent recurring situations in which a developer would have to deal with if faced with the task of modularizing design by contract code using aspects. Each scenario template consists of two sides: (i) left-hand side includes some design by contract code from the OO version of Health Watcher, and (ii) right-hand side which includes a code snippet from the AO version showing the result of moving design by contract to an aspect. We have identified some scenarios in Health Watcher and classified each one as beneficial or harmful, according to the quality of the AO implementation (based on the metrics), when compared to the OO one.

A Beneficial Scenario. Figure 10 shows a scenario where the use of aspects is advantageous. In this scenario, in the left-hand side of the template, the three marked preconditions are used to check the nullity of the input parameters of the three methods illustrated in the class Employee (e.g. JC.requires(name ! = null) for the method setName). Therefore, it is straightforward to implement these homogenous preconditions with a single **before** advice in the AO version (as illustrated in the right-hand side of the refactoring template).

It is easy to note that the effect of the scenario presented in Fig. 10 on the values of some of the employed metrics is either positive, negative or neutral. For instance, the separation of concerns metrics tend to be better for the AO version after refactoring, but only for the CDLOC metric (since the tangling was removed). The CDC and CDO metrics in this case are neutral due to the concern is realized by always one component (class or aspect) and one operation (method or advice). The number of preconditions shows that we decreased in 2 in the AO implementation (only 1 precondition after refactoring) in contrast to the same 3 preconditions in the OO version. Furthermore, the AO implementation has fewer couplings than in OO version, because it does not refer to the class JC anymore. Finally, the measures for the size metrics are worse in the AO version, even though we have reuse of preconditions (3 references in OO to only 1 in AO). This refactoring code clearly exemplify several findings we had with the entire AO versions of Health Watcher system.

```
public class Employee {
 ...

 public void setName(String name) {
  JC.requires(name != null, this);
   ...
 }
 public void setLogin(String login) {
  JC.requires(login != null, this);
   ...
 }
 public void setPassword(String pwd) {
  JC.requires(pwd != null, this);
   ...
 }
}
```

⇓

```
public class Employee {
 ...

 public void setName(String name) {
   ...
 }
 public void setLogin(String login) {
   ...
 }
 public void setPassword(String pwd) {
   ...
 }
}
aspect HWNonNullInputsDbC {
 pointcut nonNullInputMeth():
  execution(void Employee.set*(String));

 before (Object obj, final Object param):
  nonNullInputMeth() && this(obj) && args(param){
  JC.requires(param != null, obj);
 }
}
```

Fig. 10. A scenario where aspectization is beneficial.

A Harmful Scenario. Figure 11 shows a scenario where the aspectization of the design by contract code brings more harm than good. The left-hand side of the template illustrates three marked postconditions that should be enforced when calling the constructor of class Address. Since these postconditions refer to specific attributes of the class Address, there is no benefit with its aspectization. After refactoring (right-hand side of the template), we still have the same three postconditions. Since no reuse is possible due to the heterogeneity nature of the postconditions, the number of LOC also grows and tend to be even worse when compared with the previous scenario. The application of SoC metrics to this example is neutral, except by the CDLOC metric as in the discussed beneficial scenario. In the end, the overall effect of this refactoring was only to increase the complexity of the design by contract concern. The only benefit achieved by this example was the physical separation of concerns, but without reuse and quantification properties of AspectJ-like languages [23].

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

```
public class Address {
 ...

 public Address(...) {
  ...
  JC.ensures(this.street != null, this);
  JC.ensures(!this.street.equals(""), this);
  JC.ensures(!this.street.equals(" "), this);
 }
 ...
}
```

⇓

```
public class Address {
 ...

 public Address(...) {
  ...
 }
 ...
}
aspect HWModelDesignByContract {
 pointcut addressNew():
  execution(Address.new(..));

 after (Address obj) returning ():
  addressNew() && this(obj) {
  JC.ensures(obj.street != null, obj);
  JC.ensures(!obj.street.equals(""), obj);
  JC.ensures(!obj.street.equals(" "), obj);
 }
}
```

Fig. 11. A scenario where aspectization is harmful.

6.5. *Study constraints*

In what follows, we present the threats to validity of our study.

External Validity. First, there is a threat to external validity that we used only one system, which is difficult to draw general conclusions. The HW system is a good candidate for empirical studies due to the good documentation and resources available [20]. Even though the Health Watcher system is representative in terms of the applied contracts and scenarios of changes, we can have systems that the modularization of contracts by aspects is not worthwhile as explained by harmful scenarios in the previous section. Hence, a reader is encouraged to analyze the benefits and drawbacks of the "aspectization" of a particular system. So, it is desirable to involve more systems and more approaches to complement this study. Second, the scope of our experience is limited to Java and AspectJ languages. With respect to design by contract features, our experience only considered the implementation of pre-, post-conditions, and invariants. Our results may potentially generalize to other OO and AO languages and design by contract features, though that requires further analyses.

Internal Validity. The authors implemented the contracts in the HW system, which is a threat to internal validity. However, we minimize this threat since we followed the HW requirements documentation, which contains information with

respect to pre- and postconditions. Hence, we did not infer any contract (e.g. a precondition). We implemented all contracts according to the documentation [20]. As explained, this was one of the main reasons to use the Health Watcher system in this study.

Conclusion Validity. The applicability, usefulness, and representative of the set of the metrics used in this study can be questioned. However, due to the nature of the study and the fact that separation of concerns and change propagation are central to this study, the design by contract crosscutting concern was naturally the one which varied most. Hence, we used a set of metrics related to separation of concerns and change propagation to better assess the design stability of the DbC code throughout the changes. It is important to note that the multi-dimensional analysis conducted by this work allowed us to grasp which measurement outliers were significant and those which were not. In fact, when concluding from the results we have considered all the gathered data and never relied upon one single piece of data from this set. In addition, the SoC metrics described in Sec. 4 have already been proved to be effective quality indicators in several case studies [19, 26, 15, 20, 8].

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

7. Related Work

The current body of empirical knowledge on AO explain how the use of aspect decompositions supports the separation of classical crosscutting concerns such as distribution [48, 47], persistence [36, 47], concurrency [25, 47], and design by contract [23, 3, 11, 42, 38, 40]. However, such works do not analyze other effects and stringent quality indicators in the resulting AO systems. In addition, they do not quantify the positive and negative effects of AO techniques in the presence of widely-scoped changes.

As a result, a number of quantitative empirical assessments have been carried out to compare OO and AO designs, such as exception handling [15, 16, 8], design patterns [19, 5], use cases [10], and other crosscutting concerns [26, 20]. These works [26, 20] also analyze quantitatively the scalability of AO by implementing several diverse changes. In addition to the traditional quantitative modularity metric suites [45, 19, 26, 20, 14] such as separation of concerns, coupling, cohesion, and conciseness, researchers have proposed new quantitative modularity measures [50, 51, 6, 46].

Sullivan *et al.* proposed the application of net option value (NOV) [50, 51] analysis to measure software modularity. The idea is that a module creates value in the form of options: one has the right but not the obligation to replace a module with a new/better version. The more likely a module is subject to change and the more independent it is, the higher option value can be generated. Their analysis is based on design structure matrix (DSM) models where both design and environmental conditions, such as requirements, are uniformly modeled as design variables. Cai *et al.* [6] proposed modularity vector so that design evolution can be simulated and the impact of changes can be predicted based on the variations of NOV values and other measurements. As a simplified variation of NOV analysis, Sethi *et al.* [46] proposed

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

new modularity measurements, such as design volatility, concern scopes and independence level, based on DSM models there both concerns and designs are uniformly modeled. These measurements were used to quantitatively assess which programming paradigm, AO, versus OO, is more stable under given changes and which one can generate higher option values from design level.

However, none of this body of quantitative studies consider design by contract concern as we do in this work. In this context, the most well-known study focusing on the interplay between design by contract and AO was performed by Lippert and Lopes [30]. They used AO techniques to modularize design by contract features such as pre- and postconditions in a large OO framework, called JWAN. Also, they attempted to identify situations where it was easy to aspectize design by contract code. However, it has some shortcomings that hinder its results to be extrapolated to the development of real-like software systems. First, the target of the study was a system where the design by contract is only homogeneous (not application-specific). However, contracted systems can contains heterogeneous DbC (application-specific) such those we found in the Health watcher [20]. Second, their evaluation only considered pre- and postconditions, whereas we also include invariants in our analysis. Third, their overall assessment was performed only in terms of pluggability and incremental development. Quantitative evaluation was performed only in terms of number of LOC. The use of LOC in isolation is usually the target of severe criticisms [52]. For instance, in the context of the Lippert and Lopes study, the use of LOC as the sole metric resulted in a narrow view of the effects of the aspectization of design by contract on the program quality. They portrayed that the AO decompositions as very superior to OO decompositions regarding system size.

8. Concluding Remarks

In this paper, we presented an empirical case study to assess various facets of design by contract modularity of object-oriented and aspect-oriented implementations of a real-life system to empirically understand their positive and negative effects through design changes. This study was the first to include a quantitative and qualitative analysis of the aspectization of design by contract concern and also with an analysis of the implementations regarding modularity and change propagation.

From this analysis we have discovered a number of interesting outcomes. Firstly, the AO implementations tend to have a more stable design when implementing the design by contract crosscutting concern. Furthermore, changes tended to be much less intrusive and more simplistic in the AO implementations. This indicates that aspectual decompositions are superior especially when considering the Open-Closed principle. In certain circumstances aspectual decompositions tended to propagate to unrelated components due to ripple-effects caused by OO refactorings. In addition, even with higher reuse, AO implementations tended to present no significant gains regarding system and design by contract size in relation to OO decompositions as usually advertised by the literature. The overall conclusion regarding design by

contract modularity is that aspect decompositions tended to be much better than OO in relation to several realistic maintenance scenarios leading to a more stable design by contract implementation.

One of the most immediate future work is to derive a predictive model for using aspects to implement design by contract, based on our experience of this study. Hence, developers may recognize the situations where it is advantageous to aspectize design by contract code. In addition, we intend to augment our change scenarios in order to contemplate more kinds and combination changes. Furthermore, we also intend to investigate the modularization of design by contract concern in other kinds of system domains such as Software Product Lines.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

Acknowledgments

This work is partially supported by INES, funded by CNPq and FACEPE, under Grants 573964/2008-4 and APQ-1037-1.03/08. Henrique Rebêlo is also supported by FACEPE under Grant No. IBPG-1664-1.03/08. Ricardo Lima is also supported by CNPq under Grant No. 314539/2009-3. The work of Yuanfang Cai is supported by the National Science Foundation under Grants CCF-0916891 and DUE-0837665.

Appendix A. Online Appendix

We invite researchers to replicate our case study. Source code of the OO and AO versions of the HW system, used measurement tools, shadowed code, and our results are available in [39].

References

1. Aop metrics tool. Available from: http://aopmetrics.tigris.org/.
2. V. Alves and P. Borba, Distributed adapters pattern: A design pattern for object-oriented distributed applications, in *Proceedings of the 1st Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP '01*, 2001.
3. L. C. Briand, W. J. Dzidek and Y. Labiche, Instrumenting contracts with aspect-oriented programming to increase observability and support debugging, in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, Washington, DC, USA, 2005, IEEE Computer Society, pp. 687–690.
4. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns* (John Wiley & Sons, New York, 1996).
5. N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista and C. Lucena, Composing design patterns: A scalability study of aspect-oriented programming, in *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD '06*, New York, NY, USA, ACM, 2006, pp. 109–121.
6. Y. Cai, S. Huynh and T. Xie, A framework and tool supports for testing modularity of software design, in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, New York, NY, USA, ACM, 2007, pp. 441–444.
7. S. R. Chidamber and C. F. Kemerer, A metrics suite for object oriented design, *IEEE Trans. Softw. Eng.* **20** (1994) 476–493.

8. R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. Staa and C. Lucena, Assessing the impact of aspects on exception flows: An exploratory study, in *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP '08)*, 2008, pp. 207–234.
9. A. Colyer, A. Rashid and G. Blair, On the separation of concerns in program families, Technical report COMP-001-2004, Computing Department, Lancaster University, 2004. Available at: http://www.comp.lancs.ac.uk/computing/aose/papers/COMP-001-2004.pdf.
10. F. d'Amorim and P. Borba, Modularity analysis of use case implementations, in *Software Components, Architectures and Reuse (SBCARS), 2010 Fourth Brazilian Symposium*, 2010, pp. 11–20.
11. Y. A. Feldman, O. Barzilay and S. Tyszberowicz, Jose: Aspects for design by contract 80-89, in *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, Washington, DC, USA, 2006. IEEE Computer Society, pp. 80–89.
12. E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho and F. Dantas, Evolving software product lines with aspects: An empirical study on design stability, in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, New York, NY, USA, ACM, 2008, pp. 261–270.
13. E. Figueiredo, A. Garcia and C. Lucena, Ajato: An aspectj assessment tool, in *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP '06)*, 2006.
14. E. Figueiredo, C. Sant'Anna, A. Garcia, T. T. Bartolomei, W. Cazzola and A. Marchetto, On the maintainability of aspect-oriented software: A concern-oriented measurement framework, in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, Washington, DC, USA, IEEE Computer Society, 2008, pp. 183–192.
15. F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia and C. M. F. Rubira, Exceptions and aspects: The devil is in the details, in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14*, New York, NY, USA, ACM, 2006, pp. 152–162.
16. F. C. Filho, A. Garcia and C. M. F. Rubira, Extracting error handling to aspects: A cookbook, in *ICSM*, 2007, pp. 134–143.
17. M. Fowler, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley Longman, Boston, MA, 1999).
18. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Longman, Boston, 1995).
19. A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena and A. von Staa, Modularizing design patterns with aspects: A quantitative study, in *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, New York, NY, USA, ACM, 2005. pp. 3–14.
20. P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant' Anna, S. Soares, P. Borba, U. Kulesza and A. Rashid, On the impact of aspectual decompositions on design stability: An empirical study, in *Proceedings of the 21st European Conference on Object-Oriented Programming*, LNCS, Springer-Verlag, 2007, pp. 176–200.
21. C. A. R. Hoare, An axiomatic basis for computer programming, *Commun. ACM* **12**(10) (1969) 576–580.
22. A. Kellens, K. Mens, J. Brichau and K. Gybels, Managing the evolution of aspect-oriented software with model-based pointcuts, in *ECOOP*, 2006, pp. 501–525.
23. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, Getting started with aspectj, *Commun. ACM* **44** (2001) 59–65.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.

24. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, Aspect-oriented programming, in *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, Vol. 1241, pp. 220–242.
25. J. Kienzle and R. Guerraoui, Aop: Does it make sense? The case of concurrency and failures, in *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)* London, UK, 2002, pp. 37–61.
26. U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa and C. Lucena, Quantifying the effects of aspect-oriented programming: A maintenance study, in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, Washington, DC, USA, 2006, pp. 223–233.
27. Y. L. Traon, B. Baudry and J.-M. Jezequel, Design by contract to improve software vigilance, *IEEE Trans. Softw. Eng.* **32** (2006) 571–586.
28. G. T. Leavens, A. L. Baker and C. Ruby, Preliminary design of JML: A behavioral interface specification language for Java **31**(3) (2006) 1–38.
29. T. Lima, V. Alves, S. Soares and P. Borba, Pdc: Persistent data collections pattern, in *Proceedings of the 1st Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP '01)*, 2001.
30. M. Lippert and C. V. Lopes, A study on exception detection and handling using aspect-oriented programming, in *Proceedings of the 22nd International Conference on Software Engineering, (ICSE '00)*, New York, NY, USA, 2000, pp. 418–427.
31. M. Marin, L. Moonen and A. van Deursen, A classification of crosscutting concerns, in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, Washington, DC, USA, 2005, pp. 673–676.
32. B. Meyer, Applying "design by contract", *Computer* **25**(10) (1992) 40–51.
33. B. Meyer, *Object-Oriented Software Construction* (2nd edn.) (Prentice-Hall, NJ, 1997).
34. M. Mezini and K. Ostermann, Conquering aspects with Caesar, in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, New York, NY, USA, 2003, pp. 90–99.
35. M. P. Monteiro and J. M. Fernandes, Towards a catalog of aspect-oriented refactorings, in *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, New York, NY, USA, 2005, pp. 111–122.
36. A. Rashid and R. Chitchyan, Persistence as an aspect, in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, New York, NY, USA, 2003, pp. 120–129.
37. H. Rebêlo, R. Lima, M. Cornélio, G. T. Leavens, A. Mota and C. Oliveira, Optimizing generated aspect-oriented assertion checking code for JML using program transformations: An empirical study. *Sci. Comput. Program.*, 2012. Accepted to appear. Also available as a TR at: http://www.eecs.ucf.edu/∼leavens/tech-reports/UCF/CS-TR-10-01/TR.pdf.
38. H. Rebêlo, R. Lima, U. Kulesza, R. Coelho, A. Mota, M. Ribeiro and J. E. Araujo, The contract enforcement aspect pattern, in *Proceedings of the 8th Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP '10)*, 2010, pp. 99–114.
39. H. Rebêlo, R. Lima, U. Kulesza, M. Ribeiro, Y. Cai, R. Coelho, C. Sant'Anna and A. Mota, Aspectized contracts and maintenance. Available from: http://cin.ufpe.br/∼hemr/ijseke13.
40. H. Rebêlo, R. Lima and G. T. Leavens, Modular contracts with procedures, annotations, pointcuts and advice, in *SBLP '11: Proceedings of the 2011 Brazilian Symposium on Programming Languages*. Brazilian Computer Society, 2011.

41. H. Rebêlo, R. M. F. Lima, U. Kulesza, C. Sant'Anna, R. Coelho, A. Mota, M. Ribeiro and C. A. L. Oliveira, Assessing the impact of aspects on design by contract effort: A quantitative study, in *SEKE*, 2011, pp. 450–455.
42. H. Rebêlo, S. Soares, R. Lima, L. Ferreira and M. Cornélio, Implementing Java modeling language contracts with aspectj, in *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC '08)*, 2008, pp. 228–233.
43. M. P. Robillard and G. C. Murphy, Representing concerns in source code, *ACM Trans. Softw. Eng. Methodol.* **16** (2007).
44. D. S. Rosenblum, A practical approach to programming with assertions, *IEEE Trans. Softw. Eng.* **21** (1995) 19–31.
45. C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena and A. V. von Staa, On the reuse and maintenance of aspect-oriented software: An assessment framework, in *Proceedings of XVII Brazilian Symposium on Software Engineering*, 2003, pp. 19–34.
46. K. Sethi, Y. Cai, S. Wong, A. Garcia and C. Sant'Anna, From retrospect to prospect: Assessing modularity and stability from software architecture, in *WICSA/ECSA*, 2009, pp. 269–272.
47. S. Soares *et al.*, Distribution and persistence as aspects, *Softw. Pract. Exper.* **36**(7) (2006) 711–759.
48. S. Soares, E. Laureano and P. Borba, Implementing distribution and persistence aspects with aspectj, *SIGPLAN Not.* **37** (2002) 174–190.
49. M. Störzer and C. Koppen, Pcdiff: Attacking the fragile pointcut problem, abstract, in *European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004.
50. K. J. Sullivan, W. G. Griswold, Y. Cai and B. Hallen, The structure and value of modularity in software design, in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-9*, New York, NY, USA, 2001, pp. 99–108.
51. K. J. Sullivan, W. G. Griswold, Y. Cai and B. Hallen, The structure and value of modularity in software design, *SIGSOFT Softw. Eng. Notes* **26** (2001) 99–108.
52. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in Software Engineering: An Introduction* (Kluwer Academic, Norwell, MA, 2000).
53. S. S. Yau and J. S. Collofello, Design stability measures for software maintenance, *IEEE Trans. Softw. Eng.* **11** (1985) 849–856.

Int. J. Soft. Eng. Knowl. Eng. 2013.23:913-941. Downloaded from www.worldscientific.com
by UNIVERSIDADE FEDERAL DA BAHIA on 05/08/14. For personal use only.