Assessing the Impact of Aspects on Design By Contract Effort: A Quantitative Study

Henrique Rebêlo\(^1\)  Ricardo Lima\(^1\)  Uirá Kulesza\(^2\)  Cláudio Sant’Anna\(^3\)  Roberta Coelho\(^2\)  Alexandre Mota\(^1\)  Mário Ribeiro\(^1,4\)  César Oliveira\(^1\)

\(^1\)Federal University of Pernambuco, PE, Brazil
\{hemr, rmfl, acm, mmr3, calo\}@cin.ufpe.br
\(^2\)Federal University of Rio Grande do Norte, RN, Brazil
\{uir, roberta\}@dimap.ufrn.br
\(^3\)Federal University of Bahia, BA, Brazil
santanna@dcc.ufba.br
\(^4\)Federal University of Alagoas, AL, Brazil
marcioribeiro@ic.ufal.br

Abstract

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 well-understood modularity attributes. This paper reports a quantitative case study of the adequacy of aspects for modularizing design by contract concern. The study consisted of refactoring a real-life application so that the code responsible for implementing the contract enforcement strategies was moved to aspects. Our analysis was driven by fundamental modularity attributes, such as separation of concerns, coupling, and size. We have found that AO techniques improved separation of concerns between the design by contract code and base application code. However, contradicting the general intuition, the AO version of the system did not present significant gains regarding four classical size metrics we employed.

1 Introduction

Design by Contract (DbC), originally conceived by Meyer [12], is a technique for developing and improving functional software correctness. The key mechanism in DbC is the use of the so-called “contracts”. A contract formally specify 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. However, 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 [7]. 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 [1, 15].

It is assumed that the contracts of a system is de-facto a crosscutting concern that can be better modularized by the use of aspect-orientation [8, 10, 11]. Recent studies [10, 8, 1, 15, 16] have shown that object-oriented 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 [10] 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 as-
assessments. Quantitative evaluation 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 size, when used for modularizing non-trivial homogeneous and heterogeneous design by contract code.

This paper complements Lippert and Lopes work [10] by performing quantitative assessments of OO and AO implementations for invariants and pre- and postconditions of a real-life web-based information system, called Health Watcher (HW) [6]. The OO version was implemented in Java, whereas the AO version was implemented in AspectJ. Our evaluation focused upon on well-known modularity attributes such as separation of concerns, coupling, and size [17, 5]. We have found that the AO solution improved the separation of design by contract concern of the HW system. Moreover, the use of aspects have exhibited significant reuse of DbC features such as preconditions. However, the AO implementation of HW has not presented significant gains regarding four classical size metrics.

This paper is structured as follows: Section 2 describes our experimental settings and justifies the decisions made to ensure the study is valid. The results gathered from applying the modularity metrics are discussed in Section 3. Section 4 analyzes the obtained results and points some constraints on the validity of our study. Finally, Sections 5 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 moved design by contract code to aspects. The choice of the target system is discussed in Section 2.2. In addition, the metrics used in the assessment process (Section 2.3), and our assessment procedures (Section 2.4) are described.

2.1 Aspectizing Design By Contract

Our study focused on the placement of contracts. We moved 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 operations. As such, the methods calls JC.requires, JC.ensures, and invariant denotes pre- and postconditions, and invariants of the target system, respectively.

We used the Extract Fragment to Advice [13] refactoring to move contracts to aspects. Figure 1 illustrates this 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. Due to space constraints, we do not show how we extracted other DbC features to advice.

2.2 Target System Selection

The first major decision 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) [6]. 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 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 [6]. Second, the original implementation of HW is composed by eleven use cases that are detailed described by an available requirements document, which is essential to understand its main functionalities [6]. Third, other qualitative and quantitative studies of the HW system have been recently conducted [9, 4, 6, 3], and so provided a solid foundation for this study.

2.3 The Metrics

In our study, a suite of metrics for separation of concerns (SoC), coupling, and size [17, 5] were selected to evaluate the OO and AO implementation versions of the HW system. This suite was adapted from classic OO metrics [2] to be applied to the AO paradigm. In addition, the chosen metrics have already been used in several empirical case studies [5, 9, 4, 6, 3]. For all the employed metrics, a lower value implies better results. Table 1 summarizes each metric used in this case study, and associates it with the relevant modularity attribute.

Separation of Concerns (SoC) metrics measure the degree to which a single concern (design by contract in our
study) affects the system. The coupling metric CBC indicates the degree of dependency between components. Excessive coupling is not desirable, since it is detrimental to modular design. Size metrics are important to evaluate the complexity and different perspectives of the final system. In this way, the metric group includes metrics for both general system attributes (e.g., Number of Lines of Code) and quantities 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 refactored versions of a particular target system. For further details about SoC, CBC, and size metrics, refer to [2, 17, 5].

### 2.4 Assessment Procedures

The main goal of this empirical case study is to answer how the HW system behaves regarding design by contract modularity when implemented with AO techniques. To this end, our study was divided into three major phases: (i) the implementation of the design by contract concern and alignment of the original HW according to its requirements document; (ii) the refactoring of the design by contract concern (developed in phase i) of HW to aspects, and (iii) the assessment of the two versions (the OO and AO versions developed in phases i and ii, respectively) of HW system.

In the first phase, we implemented the design by contract concern for the OO solution of the HW system, which is already available and implemented in Java. As aforementioned, HW comprises several classical crosscutting concerns, but no existing quantitative work have explored the design by contract one. We analyzed the entire available requirements document of the HW system to understand its functionalities and involved actors. This was fundamental to apply required preconditions, postcondition, and invariants for all the HW use cases. The implementation comprehends both homogeneous and heterogeneous contracts for the HW use cases. We found some inconsistencies 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.

The second phase involved the refactoring of the design by contract crosscutting concern of HW system to aspects. After extracting all the contracts to aspects, we looked for reuse opportunities and eliminated identical contracts already moved to AspectJ advice. Basically, we implemented contracts in the aspects using before and after advice. Eventually, when we have old expressions, which refers to both pre- and post-state of a method execution, we used around advice. Further details on how to instrument old expressions with around advice, refer to [15, 16].

The goal of the third phase was to compare in a quantitative way the OO and AO versions of the HW system performed in the previous phases. In the measurement process, the data was partially gathered by the AJATO measurement tool [1]. It supports some metrics: LOC, NOA, NOO. Additionally, we used the AOP metrics tool [2] to collect CBC, LCOO, and VS. Eventually, we collected the SoC metrics (CDC, CDO, CDLOC) [17, 5] manually.

The data collection of SoC metrics (CDC, CDO and CDLOC) was preceded by what the metrics’ authors call as “shadowing” process. In this process, the code implementing DbC was identified and shadowed in every class, interface and aspect. The metrics were, then, manually computed based on the shadowed code. The complete description of the gathered data, measurement tools, and shadowed code is available at [14]

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Metrics</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separation of Concerns (SoC)</td>
<td>Concern Diffusion over Components (CDC)</td>
<td>Number of classes and aspects that contribute to the implementation of a concern [5].</td>
</tr>
<tr>
<td></td>
<td>Concern Diffusion over Operations (CDO)</td>
<td>Number of methods and advice that contribute to a concern’s implementation [5].</td>
</tr>
<tr>
<td></td>
<td>Concern Diffusion over LOC (CDLOC)</td>
<td>Counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a “concern switch” [5].</td>
</tr>
<tr>
<td>Coupling</td>
<td>Coupling Between Components (CBC)</td>
<td>Number of classes and aspects declaring methods or fields that may be called or accessed by other components [2].</td>
</tr>
<tr>
<td>Size</td>
<td>Lines of Code (LOC)</td>
<td>Number of lines of code [2].</td>
</tr>
<tr>
<td></td>
<td>Design By Contract Lines of Code (DbCLOC)</td>
<td>Number of lines of code that are relative to DbC.</td>
</tr>
<tr>
<td></td>
<td>Number of Preconditions (NOPre)</td>
<td>Number of preconditions of each class or aspect.</td>
</tr>
<tr>
<td></td>
<td>Number of Postconditions (NOPo)</td>
<td>Number of postconditions of each class or aspect.</td>
</tr>
<tr>
<td></td>
<td>Number of Invariants (NOI)</td>
<td>Number of invariants of each class or aspect.</td>
</tr>
<tr>
<td></td>
<td>Number of Attributes (NOA)</td>
<td>Number of attributes of each class or aspect [2].</td>
</tr>
<tr>
<td></td>
<td>Number of Operations (NOO)</td>
<td>Number of methods and advice of each class or aspect [2].</td>
</tr>
<tr>
<td></td>
<td>Vocabulary Size (VS)</td>
<td>Number of components of the system [2].</td>
</tr>
</tbody>
</table>

1http://www.teccomm.les.inf.puc-rio.br/emagno/ajato/
2http://aopmetrics.tigris.org/
### 3 Study Results

This section presents the results of the measurement process. The data have been collected based on the set of defined metrics (Table 1). We present the results by means of tables. Rows labelled “Diff.” indicate the percentage difference between the original and refactored versions of the HW. A positive value means that the OO version fared better, whereas a negative value indicates that the AO version exhibited better results.

#### 3.1 Quantifying Separation of Concerns

Table 2 shows the obtained results of the separation of concern metrics. The “Diff.” row shows significant differences in favor of the AO implementation in terms of the Concern Diffusion over Components (CDC). This divergence is a direct consequence of the adopted strategy for creating new DbC aspects in HW system. For example, we created a new aspect whose sole responsibility was to implement all the non-null input parameters required to fulfill the HW requirements. This classical contract checking denotes an example of homogenous contracts of the HW system. In cases when we had heterogeneous contracts, we create one aspect per layer to encapsulate the HW contracts. This scenario contributed to a better result in favor of the AO implementation regarding the CDC metric. The DBC concern is spread over 51 components (classes or interfaces) in the OO version, whereas in the AO solution it was only 12 components (in which 11 are aspects). This led us to a percentage reduction of 76.47% in favor of AO version.

Still regarding the AO solution, the DBC concern fared better for the CDO metric. It is scattered over 297 in the OO solution against to only 196 operations relative to the AO solution. As a result, we had a 34% percentage of reduction in favor to AO solution. Finally, the Concern Diffusion over LOC (CDLOC) was the metric where the AO implementation of HW system performed better against its counterpart in OO implementation. This implies that the design by contract concern is more tangled in the OO solution than in the AO implementation. The OO solution presents 1193 “concern switches” over the system code, whereas the AO solution had no occurrence of concern switches. This means that the DbC concern was completely untangled, localized, and isolated with aspects.

#### 3.2 Quantifying Coupling and Size

Table 3 shows the obtained results for the coupling and size metrics. 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 6.14% percentage reduction in favor of AO solution (see the CBC column in Table 3).

Contradicting the general intuition that aspects make programs smaller [10, 8] due to reuse, the OO version and its counterpart in AO did not present significant gains in relation to the four classical metrics: VS, NOO, LOC, and NOA. For instance, the Vocabulary Size (VS) grew as expected with 11% more components (classes + aspects) due to the introduction of design by contract aspects. Thus, AO version involved 100 components (with 11 created aspects), whereas the OO implementation included only 89 components to comprise the same functionalities. Moreover, 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 26.59% more method-like definitions in the AO version. In the HW system, the difference of the number of LOC was only 1.92% in favor to AO solution. Hence, even with significant reuse of design by contract code as we discuss next, the aspect code for realizing the DbC concern requires a lot of extra idioms which led to extra effort during implementation.

### Table 2. Separation of Concerns Metrics.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Before Refactoring</th>
<th>After Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Classes</td>
<td>Aspects</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>CDC</td>
<td>51</td>
<td>-</td>
</tr>
<tr>
<td>CDO</td>
<td>297</td>
<td>-</td>
</tr>
<tr>
<td>CDLOC</td>
<td>51</td>
<td>-</td>
</tr>
<tr>
<td>Diff</td>
<td>-76.47%</td>
<td>-34%</td>
</tr>
</tbody>
</table>

### Table 3. Coupling and Size Metrics.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Before Refactoring</th>
<th>After Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CBC</td>
<td>CDLOC</td>
</tr>
<tr>
<td>Classes</td>
<td>342</td>
<td>1272</td>
</tr>
<tr>
<td>Aspects</td>
<td>467</td>
<td>346</td>
</tr>
<tr>
<td>NOPre</td>
<td>383</td>
<td>183</td>
</tr>
<tr>
<td>NOPs</td>
<td>183</td>
<td>530</td>
</tr>
<tr>
<td>NOI</td>
<td>383</td>
<td>183</td>
</tr>
<tr>
<td>NOA</td>
<td>383</td>
<td>183</td>
</tr>
<tr>
<td>NOO</td>
<td>383</td>
<td>183</td>
</tr>
<tr>
<td>VS</td>
<td>89</td>
<td>89</td>
</tr>
<tr>
<td>Diff</td>
<td>-6.14%</td>
<td>-50%</td>
</tr>
</tbody>
</table>

OO solution against to only 196 operations relative to the AO solution. As a result, we had a 34% percentage of reduction in favor to AO solution. Finally, the Concern Diffusion over LOC (CDLOC) was the metric where the AO implementation of HW system performed better against its counterpart in OO implementation. This implies that the design by contract concern is more tangled in the OO solution than in the AO implementation. The OO solution presents 1193 “concern switches” over the system code, whereas the AO solution had no occurrence of concern switches. This means that the DbC concern was completely untangled, localized, and isolated with aspects.
This finding, contradicts the Lipert and Lopes study [10], on which they had a reduction of more than 50% in LOC due to reuse. This happens because while HW system has homogeneous contracts that can be significantly reused, some heterogeneous contracts can be harmful to the final LOC due to the poor reuse of such heterogeneous contracts 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).

As aforementioned, in order to quantify the reuse of the design by contract code, we employed specific metrics to DbC concern: DbCLOC, NOPre, NOPo, NOI. The Number of LOC relative to DbC (DbCLOC) was the only specific size metric in which the benefits by the AO solution was less than 10% (7.54%). This happens because the DbC aspects used to modularize the HW contracts (e.g., invariants) have significant extra code (such as pointcuts to intercept the join points involved in the DbC concern realization and advice contextual information) to cope with the crosscutting concern modularization.

The remaining three DbC size metrics performed significantly better in favor to AO solution. For instance, the Number of Invariants (NOI) exhibited a reuse of 90.60%. This is understandable since an invariant implementation in Java needs to be invoked several times to fulfill its semantics. An invariant holds after every constructor execution and just before and after every instance method execution of a class [15]. With AO and AspectJ, we can modularize an invariant with two advice reducing the so scattered implementation of invariant calls to only 2 occurrences [15].

In relation to pre- and postconditions, we had a higher reuse of preconditions in the AO implementation (83.08%). Regarding postconditions, the AO solution fared better with a reuse of 50% against its counterpart in OO implementation. In fact, these numbers can substantially vary depending the degree of homogeneous and heterogeneous pre- and postconditions that can appear in a particular system. In the HW system, we observed that the postconditions are less reusable than preconditions, due to postconditions present more heterogeneous contracts.

## 4 Discussion

This section makes a qualitative analysis of the obtained results (Section 3). Furthermore, we discuss the constraints on the validity of our empirical case study.

### 4.1 Empirical findings

Our empirical case study confirms some of the findings of the qualitative study conducted by Lippert and Lopes (LL) [10], which claims that the design by contract concern is better modularized with AO programming.

Despite the fact we had significant gains of SoC metrics in favor of the AO implementation of HW system, we found out that reusing contracts can be more difficult than usually advertised [10]. Contracts reuse depends directly on their types (e.g., postconditions) and mainly if such contracts are homogeneous or heterogenous.

For instance, since the nature of an invariant crosscuts several methods in a single class, it is naturally more reusable than pre- and postconditions that are relative to particular methods. However, pre- and postconditions can present significant reuse depends on their contracts. In other words, if several constrained methods present an intersection of common contracts, their reusability can be improved. As an example, similarly to LL [10], 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 (50%) when compared with preconditions (83.08%) and invariants (90.60%). This scenario happens due to postconditions in HW being more heterogeneous than preconditions or invariants. We found more reusability opportunities for heterogeneous preconditions than the heterogeneous postconditions. With this finding, we can conclude that the more heterogenous is a contract, its reuse with AO programming is minimized.

Another important finding of our study is related to the program size after refactoring to aspects. LL [10] discuss that by using AO programming 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 with the AO version of HW system, the gains in terms of the overall system LOC was only 1.92% and only 7.54% considering exclusively the LOC of DbC concern. This was a direct consequence for looking to reusability opportunities for heterogenous contracts. During modularization, we take into account the intercepted join points, contextual information and so forth.

### 4.2 Study Constraints

**System.** Although it can be argued that using a single system for such a study is a limiting factor, we claim that the HW system is representative in terms of the non-trivial applied contracts. The HW system is good candidate for empirical studies due to have a lot of documentation and resources available [6]. Naturally it is desirable to involve more systems and more approaches.

**Metrics.** The applicability, usefulness, and representative of the set of the metrics used in this study can be ques-
tioned. However, due to the nature of the study and the fact that separation of concerns is 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 to better assess the SoC involving DbC. In addition, the SoC metrics described in Section 2.3 have already been proved to be effective quality indicators in several case studies [5, 9, 4, 6, 3].

Languages. In addition, the scope of our experience is limited to Java and AspectJ languages. In relation to design by contract features, our experience only considered the implementation of pre-, postconditions, and invariants.

5 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. This study was the first to include a quantitative analysis of design by contract implementations using well-understood and experimented modularity metrics such as separation of concerns, coupling, and size.

From this analysis we have discovered a number of interesting outcomes. Firstly, the use of aspect-orientation to modularize design by contract improved the separation of concerns when compared to its counterpart in object-orientation. Secondly, the use of aspects tended to present a significant reuse of design by contract features, specially invariants, which we had a reuse more than 90%.

Furthermore, even with the high reuse achieved by AO programming when modularizing DbC, we found out that the aspectization of crosscutting concern such as DbC does not necessarily makes a program drastically smaller than the non-modular one with OO. Aspect code involves much more than just encapsulate a call to a precondition. It is responsible to prepare the crosscutting behavior by intercept all the join points in a system, expose contextual information and so forth. The overall conclusion regarding design by contract modularity is that aspects achieve higher reuse than OO decompositions when handling homogenous contracts, however, not always with large code reduction.

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 aspectitize design by contract code. Moreover, we intend to conduct a scalability study to analyze how aspects scale up when the number of contracts grows.

Acknowledgements

This work is partially supported by INES, funded by CNPq and FACEPE, under grants 573964/2008-4 and APQ-1037-1.03/08. Ricardo Lima is also supported by CNPq under grant No. 314539/2009-3. Henrique Rebêlo is also supported by FACEPE under grant No. IBPG-1664-1.03/08. Cláudio Sant’Anna is also supported by CNPq under grant No. 480374/2009-0.

References


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 at [14].