Is it Difficult to Test Aspect-Oriented Software? Preliminary Empirical Evidence based on Functional Tests

Thiago Gaspar Levin[1], **Fabiano Cutigi Ferrari**[1]

[1]Computing Department
Federal University of São Carlos (UFSCar) – São Carlos – SP – Brazil

{thiago.levin, fabiano}@dc.ufscar.br

Abstract. *Several studies have investigated the impact of particular programming paradigms on testing. However, few studies consider a cross-paradigm context, in which the testability of a program when it migrates from one paradigm to another. In this study we analyse the difficulty to adapt tests designed for object-oriented (OO) systems to equivalent aspect-oriented (AO) implementations and vice versa. Initially, we apply functional testing to groups of OO and AO applications. Fixing one paradigm as a baseline, we then adapt the test sets to make them executable in equivalent implementations in the other paradigm. We collect churn-related metrics to evaluate the effort to reuse the existing test sets. The results support the analysis and conclusions that, overall, favour the OO paradigm.*

1. Introduction

Object Oriented Programming (OOP) provides mechanisms to isolate data on how they are handled. Specific concepts of this paradigm such as inheritance, encapsulation and polymorphism bring benefits ranging from design and coding to reuse and maintenance [3]. Nevertheless, OOP does not allow the clear separation of certain concerns of the software, typically related to non-functional requirements. Such concerns, called crosscutting concerns, tend to be spread over or tangled with various modules of the system.

Since the late 90's, Aspect-Oriented Programming (AOP) [10] has become a possible solution for better modularising crosscutting concerns. AOP provides mechanisms to encapsulate crosscutting behaviour, which is later combined with the remaining system functionalities through the use of mechanisms such as pointcut, advice and introductions (also known as intertype declarations).

Even after almost two decades of the AOP dissemination, it is still adopted with caution by developers and researchers, as observed in a study that dates from 2009 [15]. From our experience and observations, when adopted, AOP is applied to refactor existing object-oriented (OO) systems to achieve better modularisation of crosscutting concerns. Examples of AOP applied in this context can be found in work undertaken by van Deursen et al. [22], Mortensen et al. [14], Ferrari et al. [7] and Alves et al. [1], not limited to particular technologies such as Java and AspectJ.

Given this focus on software refactoring using AOP, reusing existing test sets becomes a natural choice. As far as possible, test sets should be run on revised implementations (including aspects) for validation purposes. However, as observed by Ferrari et al. [8], the literature on AOP-related testing research endorses the idea of testing aspect-oriented (AO) software as being more difficult than testing – for instance – OO software. In their work, Ferrari et al. summarises their experience on developing customised testing approaches for AO programs and list a variety of challenging factors AOP poses for the construction of test models and the analysis of test requirements.

In spite of previous research on testing AO programs, to the best of our knowledge, no previous work has addressed the difficulties of reusing existing test sets for software that migrates from pure OO code to an AO implementation. With this context in mind, in this paper we describe the results of a study that measures the effort, in terms of churn-related metrics, to adapt existing test sets to AO code that results from refactoring tasks. Furthermore, for a comparison basis, we explore the other way around, in an hypothetical scenario of refactoring AO applications to transform them in pure OO implementations. For both cases, the target languages are Java and AspectJ, for which we evaluate the necessary changes in test code to make it aligned with the target implementation, be it AO or OO. The Section 3 of this paper describes the study configuration, which includes the research goal, the OO/AO systems we tested, the experimental design and the metrics we collected. Section 4 details the results and the respective analysis. Related work is summarised in Section 5 and conclusions are presented in Section 6. Section 2 starts by providing some basic background on source code metrics and functional testing, which is the testing technique that has driven the definition of the test sets.

2. Background

Source code metrics: According to Nagappan and Ball [17], "*software systems evolve over time due to changes in requirements, optimisation of code, fixes for security and reliability bugs etc*". Indeed, Schneidewind [20] applied several metrics across releases of the software and within releases to analyse the changes defining maintenance stability criteria and developing and applying trend metrics for stability evaluation. Some metrics defined by the author address: (1) maintenance actions (*e.g.* KLOC Change to the Code); (2) reliability (*e.g.* Mean Time to Failure, Total Failures, Remaining Failures, and Time to Next Failure); and (3) test effort (*e.g.* Total Test Time).

For the maintenance actions and evolutions that are the major purposes of this study, there is a metric called Code Churn [17]. Code churn is generally used to predict the defect density in software systems and it is easily collected from a system change history. Usually this kind of metric is used to compare system versions to estimate how many lines were added, changed and removed. There are other variations of this metric that can be used to collect more results as number of test cases and test requirements that will be shown in Section 3.

Functional Testing: also known as black-box testing, functional testing is a technique that relies on the software specification to derive test requirements [16].

Software inputs are provided and the produced output is evaluated to verify that it is in accordance with specified objectives. the verification is done through statements that compare the expected results with the actually obtained output. To partition the input data, a set of test selection criteria have been defined as follows:

- **Equivalence Partitioning:** aims to support the definition of a subset of all possible inputs of a program, achieving a finite amount of data, and hence leading to a feasible testing activity. According to Myers et al. [16], an equivalence class is a set of valid or invalid states for input conditions.
- **Boundary-Value Analysis:** extends Equivalence Partitioning criterion by selecting values at the upper and lower boundaries of each equivalence class, both for inputs and outputs [16].
- **Systematic Functional Testing (SFT):** combines Equivalence Partitioning and Boundary-Value Analysis [11]. The goal of this technique is associate the benefits of functional testing (independent implementation) and greater code coverage on test. The test set must include at least two test cases that cover each equivalence class and one test case to cover each boundary value. This minimises problems of coincidental correctness.

Table 1 shows an example of a set of test requirements for a Chess game used in this study according to the Equivalence Partitioning and Boundary-Value Analysis criteria. In this example, test requirements are defined for the Bishop piece according to its valid and invalid moves. "Check" and Error messages are displayed depending on the current move attempt. For instance, a valid class regards a move to a free square in a diagonal direction in relation to the current square (C1 class in the example). In another situation, if the desired square is occupied, the Bishop goes back to its original position (I1 class in the example). The boundary values will be between one to seven positions diagonally, where seven squares is the limit of board for Bishop piece.

Table 1: Equivalence classes and boundary values for a Bishop move (Chess)

`Chess: Bishop`			
Input Condition	Valid Classes	Invalid Classes	Boundary Values
Bishop: move	(C1) square is free, bishop goes any number of squares diagonally but may not leap over the pieces	(I1) square is occupied, bishop goes back to the same position diagonally	(B1) Bishop goes 1 position diagonally
Bishop: Capture move	(C2) any position diagonally (if enemy occupies that square)		(B2) Bishop goes 7 positions diagonally
Output Condition	Valid Classes	Invalid Classes	
Opponent piece is captured	(O1) the captured piece if removed; bishop occupies its position		
"check" message	(O2) "check" message is shown when Bishop is able to capture the King piece in the next move		
"error" message	(O3) "error" message is shown when Bishop moves to an invalid position either going ahead or capturing an opponent piece		

Similar analysis of specifications and definition of test requirements were done for all applications used in this study. Details of the applications are given in the next section.

3. Study Setup

3.1. Study Goal

The goal of this study is to investigate the difficulty of testing AO and OO programs, in particular when one migrates from one paradigm to the other and vice versa. More specifically, given two equivalent programs (one OO and another AO), we intend to analyse the effort to adapt a test suites from one paradigm to the other. To keep the study goal aligned with the consensus from the testing research community, as we contextualised in Section 1, the assumption is that testing AO programs is more difficult than testing OO programs. Section 4.3 analyses our findings with respect to this.

Based on the context described in Section 1, software developers, for different reasons, may decide to refactor a system, say an OO program, using AO programming constructs. In doing so, they will need to regressively test the new version using the existing test suite, and hence some modifications in test-related code are necessary. Note that modifications may also be required when developers decide to migrate an AO system to an OO version. This study investigates the difficulty for performing such modifications, considering equivalent implementations in both paradigms. We hereafter consider two implementation equivalent when they encompass the same set of functionalities, irrespective of internal implementation details (*i.e.* paradigm-specific code).

The evaluation presented in this paper is based on the functional technique by applying the systematic functional testing criterion (SFT). We next describe the target OO and

AO systems, infrastructure, the experimental design and the metrics collected to support our analysis.

3.2. Target Applications and Tools

In this study we selected small- and medium-sized applications for which we fully created SFT-adequate test suites. Table 2 includes general information about the target applications. In such table, "DP" means design patterns[1] and refers to applications implemented by Hannemann and Kiczales [9], "#C" is the number of classes and "#A" is the number of aspects (aspects are only available for AO implementations). Note that these applications were selected because each of them had both implementations (OO and AO) developed by third-party researchers and were either available for download or had their full source code listed in the original reports (references can be found in Table 2). The specifications to support the definition of test requirements were either documented in the original reports or were inferred after the analysis the applications' source code.

Table 2: Target applications that were used in the study and their description

Application name	Description	Total LOC	#C	#C
		OO / AO		#A
1. AbstractFactory (DP)	Creates the initial GUI that allows the user to choose a factory and generate a new GUI with the elements that the respective factory provides [9]	90 / 97	4	4 / 1
2. Boolean	Testing boolean formulas with terms AND, OR, XOR, NOT and variables [2, 19]	301 / 316	12	10 / 2
3. Bridge (DP)	Decouple an abstraction from its implementation so that the two can vary independently [9]	76 / 82	6	6 / 1
4. Chess	Chess game containing GUI [1]	1155 / 945	13	13 / 1
5. Interpreter (DP)	This system implements an interpreter for a language of boolean expressions [9]	118 / 126	8	8 / 1
6. VendingMachine	VendingMachine consists in an application for a vending machine into which the customer inserts coins in order to get drinks [12]	209 / 245	9	9 / 1
7. Question Database	Facilitate the management, reuse and improving collection of questions of evidence prepared by teachers [6]	6447 / 6479	27	27 / 5
8. ATM-log	Manager application of the bank account [1]	496 / 519	12	11 / 1
9. ChainOfResponsability (DP)	This system implements an GUI interface based in design pattern ChainOfResponsability [9]	96 / 150	5	5 / 2
10. Flyweight (DP)	This system show on the screen a message with characters in upper or lower case according with the parameters [9]	44 / 61	4	4 / 2
11. Memento (DP)	This system records a value in a point of execution [9]	29 / 64	2	3 / 2
12. ShopSystem	Simplified e-commerce system [2]	360 / 381	10	8 / 8
13. Telecom	This system calculates and reports the charges and duration of phone calls (local and long distance calls) [21]	186 / 197	8	8 / 2

We used the following tools for the test environment:

- IDE Eclipse version 4.3.1 (Kepler Service Release 1)[2];
- JUnit Framework version 3.7.2[3];
- AJDT Eclipse plugin version 2.2.4[4];
- Metrics Eclipse plugin to collect LOC metrics[5]; and
- Tool Meld Diff Viewer - version 1.3.0[6].

[1]Note that 6 out of 23 design pattern implementations were randomly selected.
[2]http://eclipse.org/ide/ – last access 21/06/2014
[3]http://junit.org/ – last access 21/06/2014
[4]https://projects.eclipse.org/projects/tools.ajdt – last access 22/06/2014
[5]http://metrics.sourceforge.net/ – last access 22/06/2014
[6]http://meldmerge.org/ – last access 22/06/2014

3.3. Experimental Design and Procedures

For experimental design, some independent variables can be defined so that they can be manipulated or controlled in experimental processes [23]. For this, the following independent variables are defined:

- Programming paradigm for program implementation;
- Testing tools used;
- Technique and testing criterion used;
- Size and complexity of programs;
- Testers' ability on application of criterion.

After the definition of these independent variables, we structured the experimental design by defining a test selection criterion as a baseline and defining the order of testing considering the target applications.

Selection of a testing criterion: The test selection criterion used was Systematic Functional Testing (SFT), which is classified as a functional-based criterion. SFT subsumes other traditional functional-based criteria, namely Equivalence Partitioning and Boundary-Value Analysis [11].

Order of testing: Figures 1, 2 and 3 illustrate how we split the target applications intro two groups in order to apply the SFT criterion. As an initial note, for Question Database, which happens to be the largest application in the set, we only tested concerns (functional or non-functional) that are implemented with aspects in the AO version. Given that functional-based testing relies on the software specification, we derived the test requirements for Question Database based on this narrowed set of concerns. Obviously, such concerns are also present in the OO implementation since both implementation are equivalent as previously stated in Section 3.1.

Initially, we defined two groups of applications, each one including OO and AO implementations of six programs plus two concerns of the Question Database system[7]. This is illustrated in Figure 1.

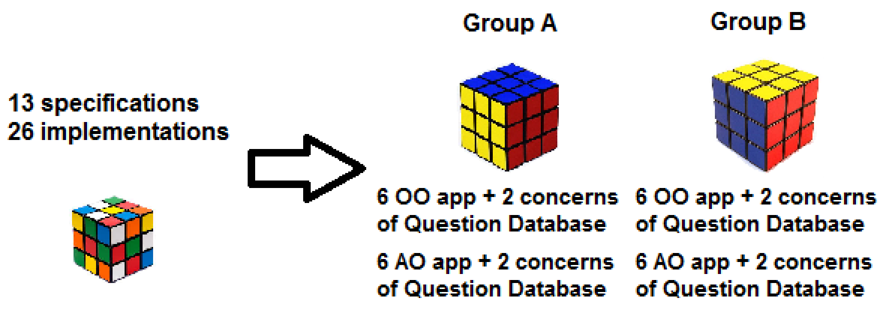
Group A
Group B
13 specifications
26 implementations
6 OO app + 2 concerns of Question Database
6 AO app + 2 concerns of Question Database
6 OO app + 2 concerns of Question Database
6 AO app + 2 concerns of Question Database

Figure 1: Applications were divided in two groups.

In Group-A, we created SFT-adequate test sets for the OO implementation (*i.e.* test sets written purely in Java – see Figure 2). Next, we modified the test cases to make them executable in the AO equivalent implementations (Figure 3). On the other way around, in Group-B we firstly created test sets for the AO implementations, then adapted such test sets to the OO counterparts.

[7]Security (controls the active time of the mouse movement and the keyboard key) and Auditing (registers logs of performed operations) concerns in Group-A and Exception Handling and Connection Control concerns in Group-B.

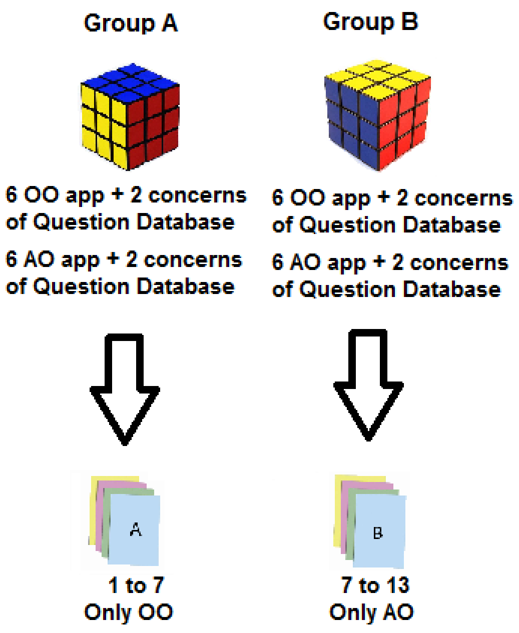
Group A
Group B
6 OO app + 2 concerns of Question Database
6 AO app + 2 concerns of Question Database
6 OO app + 2 concerns of Question Database
6 AO app + 2 concerns of Question Database
A
B
1 to 7 Only OO
7 to 13 Only AO

Figure 2: Set tests were created for each group within a particular paradigm.

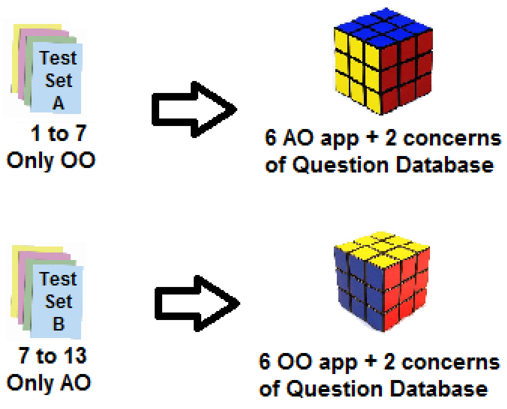
Test Set A
1 to 7 Only OO
6 AO app + 2 concerns of Question Database
Test Set B
7 to 13 Only AO
6 OO app + 2 concerns of Question Database

Figure 3: Set tests were adapted to the counterpart implementations.

Figure 4 exemplifies how a test set for the Chess application was adapted from the OO implementation to the AO counterpart. The modified lines are highlighted in bold.

3.4. Metrics

As discussed in Section 2, there are many kinds of metrics to collect information from software, some of them applicable to source code. In this study, beyond conventional source code metrics (*e.g.* LOC and number of test cases), we focus on churn-related metrics in varied granularities with the aim of comparing test suites developed and adapted to both programming paradigms. We next describe each metric.

- ***Total-LOC***: number of non-commented lines of code in the source files, considering the programs under testing.
- ***#Classes***: number of classes in the program.
- ***#Aspects***: number of aspects in the program.
- **Total Test Case LOC (*Total-LOC-TC*)**: number of non-commented LOC in the test classes (*i.e.* classes that inherit from *JUnit* `TestCase` class).

Example of Test Case - Chess OO application:

```
 1 public void testBishopMovement_nSquaresDiagonal_I1(){
 2         startRow = 7; startColumn = 5;
 3         desRow = 7; desColumn = 3;
 4
 5         assertFalse(bishop.legalMove(startRow, startColumn, desRow,
 6         desColumn, cellMatrix.getPlayerMatrix() ));
 7         assertEquals("Bishop can only move along diagonal lines",
               bishop.strErrorMsg);
 8 }
```

Example of Test Case - Chess AO application:

```
 1 public void testBishopMovement_nSquaresDiagonal_I1(){
 2         startRow = 7; startColumn = 5;
 3         desRow = 5; desColumn = 3;
 4
 5         assertFalse(bishop.legalMove(startRow, startColumn, desRow,
 6         desColumn, cellMatrix.getPlayerMatrix()));
 7         ErrorMsg em = ErrorMsg.aspectOf();
 8         String output = em.getErrorMsg();
 9         assertEquals("Bishop can only move along diagonal lines", output);
10 }
```

Figure 4: Test case example for the Chess application.

- ***Churn-LOC-TC***: number of lines added, modified and removed from a test class considering its previous version and its new version. This metric can be separated into other three submetrics: **Added LOC (*ADD*)** – number of lines added to the new version of a test class; **Modified LOC (*MOD*)** – number of lines changed in the new version of the test class in comparison with its previous version; and **Removed LOC (*REM*)** – number of lines removed from the previous version of a test class to create a new version.
- **Total Test Cases (*Total-TC*)**: number of test cases created for a program.
- **Test Requirements:** the number of equivalence classes and boundary values (in this case, all related to functional testing).

Considering the test case for Chess AO shown in Figure 4, lines 7 and 8 were added in order to extract information that is encapsulated in the `ErrorMsg` aspect. Besides that, line 9 was modified when compared to same test case for Chess OO. In this example, the *ADD* metric accounts for 2 and *MOD* metric account for 1, respectively. We follow similar rationale form *REM* and other metrics collected in our study. The next section presents the results and analysis.

4. Results and Analysis

Before we analyse the results, we describe all data collected with respect to Group-A and Group-B of applications. The description is based on the figures presented in Table 3. The top part of the table – labelled with *Group A: OO – AO* – regards results for the first group of applications (*i.e.*Group-A). As described in Section 3, in this group OO implementations were tested first, then the test sets, if necessary, were adapted to the AO paradigm. In Group-B, on the other hand, AO implementations were firstly tested. Results for Group-B are shown in the bottom part of the table.

The "Total LOC TC" and "Total TC" columns represent, respectively, the *Total-LOC-TC* and the *Total-TC* metrics when full SFT coverage is achieved for each application. The %-related columns represent the percentages of adapted lines of test code (*i.e. ADD*, *MOD* and *REM* metrics) when test cases were migrated from one paradigm to another. We highlight that according to Myers et al. [16], in the Equivalence Partitioning criterion a single test case can be designed to cover more than one valid equivalence class. Therefore, even

Table 3: Metrics collected from Test Sets

`Group A: OO - OA`

Application Name	Total	%.	Churn LOC TC						Test Requirements		
	LOC	size	ADD	%ADD	MOD	%MOD	REM	%REM	Equiv.	Bound.	Total
	TC	diff.							classes	values	TC
AbstractFactoryOO	20								4	1	4
AbstractFactoryOA	20	0 %	0	0%	0	0%	0	0%			
BooleanOO	29								7	3	7
BooleanOA	37	+ 27.58 %	8	27.58%	1	3.44%	0	0%			
BridgeOO	76								16	2	16
BridgeOA	76	0 %	0	0%	0	0%	0	0%			
ChessOO	281								28	13	39
ChessOA	302	+ 7.47 %	21	7.47%	8	2.84%	0	0%			
InterpreterOO	47								48	2	48
InterpreterOA	47	0 %	0	0%	0	0%	0	0%			
VendingMachineOO	41								9	10	10
VendingMachineOA	43	+ 4.87 %	2	4.87%	5	12.19%	0	0%			
QuestionDatabaseOO	47								5	3	8
QuestionDatabaseOA	47	0 %	0	0%	6	12.76%	0	0%			
Average		**+ 5.70 %**		**5.70%**		**4.46%**		**0%**			

`Group B: OA - OO`

Application Name	Total	%.	Churn LOC TC						Test Requirements		
	LOC	size	ADD	%ADD	MOD	%MOD	REM	%REM	Equiv.	Bound.	Total
	TC	diff.							classes	values	TC
ATM-logOA	111								9	5	15
ATM-logOO	111	0 %	0	0%	4	3.6%	0	0%			
ChainOfResponsabilityOA	108								6	0	6
ChainOfResponsabilityOO	96	- 11.11 %	0	0%	18	16.66%	12	11.11%			
FlyweightOA	36								4	4	4
FlyweightOO	36	0 %	2	5.55%	4	11.11%	2	5.55 %			
MementoOA	31								2	2	3
MementoOO	31	0 %	0	0%	8	25.8%	0	0%			
ShopSystemOA	256								22	35	30
ShopSystemOO	256	0 %	0	0%	0	0%	0	0%			
TelecomOA	257								12	16	23
TelecomOO	244	- 5.05 %	0	0%	15	5.83%	13	5.05%			
QuestionDatabaseOA	50								5	5	6
QuestionDatabaseOO	54	+ 8 %	4	8%	2	4%	0	0%			
Average		**- 1.16 %**		**1.93%**		**9.57%**		**3.10%**			

though the SFT criterion requires two test cases for each equivalence class, the number of test cases (*Total-TC* metric) not necessarily has to be as twice as large the sum of equivalence classes and boundary values, as the reader can notice in Table 3.

Note that each application is listed twice in Table 3. Despite this, churn-related measures are available for only one implementation. For example, for Group-A, metrics values are available only for AO implementations, since the adaptations were applied to OO test code. Likewise, for Group-B, churn-related measures are available exclusively for OO test code.

4.1. Group-A Results

In Group-A, test sets of three applications did not require any modification to conform to the AO implementation. These applications are AbstractFactory, Bridge and Interpreter.

Substantial differences in *Total-LOC-TC* values can be observed for Boolean and Chess. From OO to AO, we observe an increase of 27.58 % and 7.47 % for these systems, respectively. Although these increases do not reflect all types of code changes (*e.g.* related to *MOD*), they indicate a trend for more lines of code in AO-specific test sets.

Regarding churn-related values, Boolean, VendingMachine and Question Database showed expressive values considering particular metrics. For instance, for Boolean 8 LOC were added to the test code (*i.e.* 27.58%) from OO to AO implementation. For VendingMachine, 12.19% of test code was modified (see *MOD* column), and for Question Database *MOD* shows 12.76% of changes in test code.

Overall, Group-A results reveal that, on average, adapting OO test sets to AO implementations required additions of 5.70%, modifications in 4.46% of test code lines, with no code removal in any application. This results show that migrating OO-based test cases to the other paradigm (*i.e.* AO) requires more code increment than changes and removals.

4.2. Group-B Results

In Group-B, only the ShopSystem test set did not require modifications to become aligned with the OO implementation. Similarly to Group-A, results for Group-B show expressive differences for *Total-LOC-TC* values in two applications: ChainOfResponsability and Telecom. Following a contrary trend, however, we observe a decrease of 11.11 % and 5.05 % in *Total-LOC-TC*, respectively.

With respect to churn-related metrics, ChainOfResponsability, Flyweight and Memento showed significant values considering particular metrics. For instance, ChainOfResponsability test code was 16.66% modified, while 11.11% of such code was removed. Flyweight test set had 11.11% of its code modified and 5.55% removed, and Memento test code was the most modified of Group-B (25.8%).

Overall, for Group-B, more modifications and removals were needed than for Group-A. On average, test code was 9.57% modified and 3.10% removed to conform with OO implementations, while only 1.93% lines were added to the test code. This represents some evidence about the higher difficulty to migrate tests from AO applications OO counterparts. An analysis of results for both groups are next presented.

4.3. Analysis

Based on the results obtained for both groups of applications, we can highlight the following findings:

Less code is written for testing OO programs: Despite the belief in code size reduction with improved modularisation of crosscutting concerns in AO programs, *Total-LOC-TC* for OO implementations is mostly equal or lower than the values obtained for AO counterparts (the only exception is Question Database AO in Group-B). The analysis of test cases for AO applications reveals that more specific code is needed for test cases of AO systems, specially to expose context information to build *JUnit* assertions. Figure 4 shows an example: while in the OO implementation contents of error messages can be extracted directly from the `bishop` object, in the AO code such contents are encapsulated within the `ErrorMsg` aspect, so AspectJ-specific instructions are required to expose error-related information.

Test code for OO programs conforms better with the Open-Closed principle: such principle states that "*software should be open for extension, but closed for modification*" [13]. Observing the results shown in Table 3, test code for OO applications conforms to this principle more than test code for AO applications. This is due to the higher number of changes required by test sets of Group-B to become executable in OO implementations, when compared to test sets of Group-A.

Test code for OO programs is more reusable: even though the increments to test code in Group-A were higher than the increments observed in Group-B (*ADD* averages of 5.70%

and 1.93%, respectively), *MOD* and *REM* averages indicate that the interventions in existing code is more recurring in test sets for AO systems in order to adapt them to OO implementations. Around 10% of test code in Group-B required some kind of modification and 3% had to be removed to be run on OO applications. These numbers fall to 4.5% and 0%, respectively, in Group-A.

5. Related Work and Limitations

With respect to test evaluation across paradigms, Prado et al. [18] and Campanha et al. [4] compared procedural and OO programming using the same set of programs from the data structures domain. Differently from us, they focus on structural and mutation testing, respectively. Their results show that there is no significant difference in cost and strength of structural tests between the target paradigms. For mutation testing, both cost and the strength is higher in programs implemented in the procedural paradigm than in the OO paradigm.

Ceccato et al. [5] discussed the difficulties for testing AO programs in contrast with OO programs. According to them, it would be easier to test AO programs if aspects could be tested in isolation, since in OO applications have crosscutting behaviour spread across several system modules. The authors proposed a testing strategy to integrate aspects and base code incrementally, however they did not report any kind of evaluation of their strategy.

Ferrari et al. [8] discuss difficulties for testing AO programs based on practical experience on structural and mutation testing. Regarding code coverage in structural testing, they report no huge effort is necessary when AO-specific criteria is applied; in the worst case, less than 25% of test cases were added to original unit-based tests. However, the analysis of the basic model for the creation of test cases is not trivial due to the complex interactions between aspects and the base code. With respect to mutation testing, preliminary results show that non-trivial faults can be simulated using AO-specific mutation operators. To reveal such faults, on average 5% of additional test cases are necessary starting from SFT-adequate test sets. Note that in both cases, no comparison regarding different programming paradigms is provided as we did in this paper.

The main limitations of this work regard (i) the number and size of target systems, and (ii) the process of developing test sets. We needed to identify applications with equivalent OO and AO systems to compose Groups A and B. To avoid working with low quality code, we selected only applications that were previously reported in AOP-related literature, mainly in testing-related work. Given this scenario, only the source code of a small number of applications was available. In regard to the development of SFT-adequate test sets, two practitioners interchangeably developed tests for applications of Group-A and Group-B. We believe this task distribution helped to alleviate a possible "privilege" of one of the paradigms in terms of knowledge gain – and consequent quality improvement of test code – as long as tests were developed.

6. Conclusion and Future Work

This paper investigated the difficulty of reusing test sets across OO and AO equivalent software implementations. As an exploratory study, we developed functional-adequate test sets for two groups of applications, being the first purely OO (developed in Java) and the second containing AO-specific AspectJ code.

The results are that test code for OO programs requires less modifications to become aligned with AO equivalent implementations. A set of churn-related measures support such conclusion. On average, more modifications were required when test code for AO systems were adapted for OO counterparts. As future work, we plan to explore other testing techniques such as structural end fault-based (mutation) testing. Using the same test sets, we aim to compute the structural and mutant coverage produced by them. Then we can compare, from a point of view of different testing techniques, if the findings will corroborate results presented in this paper.

Acknowledgements

The authors would like to thank the financial support received from CNPq (Universal Grant #485235/2013-7), Federal University of São Carlos (RTN grant) and CAPES.

References

[1] Alves et al., P. (2011). How do programmers learn AOP? An exploratory study of recurring mistakes. In *LA-WASP'11*, pages 37–42.

[2] Bartsch, M. (2007). *Empirical Assessment of Aspect-Oriented Programming and Coupling Measurement in Aspect-Oriented Systems*. PhD thesis, University of Reading.

[3] Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Addison Wesley, 2nd. edition.

[4] Campanha, D. N., Souza, S. R. S., and Maldonado, J. C. (2010). Mutation testing in procedural and object-oriented paradigms: An evaluation of data structure programs. In *SBES'10*, pages 90–99.

[5] Ceccato, M., Tonella, P., and Ricca, F. (2005). Is AOP code easier or harder to test than OOP code? In *WTAOP'05*.

[6] Chagas, J. D. E. and Oliveira, M. V. G. (2009). Object-oriented programming versus aspect-oriented programming. A comparative case study through a bank of questions. Technical report, Federal University of Sergipe, São Cristóvão - Brazil. (in Portuguese).

[7] Ferrari et al., F. C. (2010). An exploratory study of fault-proneness in evolving aspect-oriented programs. In *ICSE'10*, pages 65–74.

[8] Ferrari et al., F. C. (2013). Difficulties for testing aspect-oriented programs: A report based on practical experience on structural and mutation testing. In *LA-WASP'13*, pages 12–17.

[9] Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in Java and AspectJ. In *OOPSLA'02*, pages 161–173.

[10] Kiczales et al., G. (1997). Aspect-oriented programming. In *ECOOP'97*, pages 220–242. Springer-Verlag.

[11] Linkman, S., Vincenzi, A. M. R., and Maldonado, J. C. (2003). An evaluation of systematic functional testing using mutation testing. In *EASE'03*, pages 1–15.

[12] Liu, C.-H. and Chang, C.-W. (2008). A state-based testing approach for aspect-oriented programming. *Journal of Information Science and Engineering*, 24(1):11–31.

[13] Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall.

[14] Mortensen, M., Ghosh, S., and Bieman, J. M. (2008). A test driven approach for aspectualizing legacy software using mock systems. *Information & Software Technology*, 50(7-8):621–640.

[15] Muñoz et al., F. (2009). Inquiring the usage of aspect-oriented programming: An empirical study. In *ICSM'09*, pages 137–146.

[16] Myers et al., G. J. (2004). *The Art of Software Testing*. John Wiley & Sons, 2nd edition.

[17] Nagappan, N. and Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *ICSE'05*, pages 284–292.

[18] Prado, M. P., Souza, S. R. S., and Maldonado, J. C. (2010). Results of a study of characterization and evaluation of structural testing criteria between procedural and OO paradigms. In *ESELAW'10*, pages 90–99 (in Portuguese).

[19] Prechelt, L., Unger, B., Tichy, W. F., Brössler, P., and Votta, L. G. (2001). A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering*, 27(12):1134 – 1144.

[20] Schneidewind, N. F. (1998). An integrated process and product model. In *5th International Software Metrics Symposium*, pages 224–234.

[21] The Eclipse Foundation (2014). AspectJ documentation. Online. `http://www.eclipse.org/aspectj/docs.php` - last access 24/07/2014.

[22] van Deursen, A., Marin, M., and Moonen, L. (2005). A systematic aspect-oriented refactoring and testing strategy, and its application to JHotDraw. TR SEN-R0507, Stichting Centrum voor Wiskundeen Informatica.

[23] Wohlin et al., C. (2000). *Experimentation in Software Engineering: an Introduction*. Kluwer.