An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs

Fabiano Ferrari[1], Rachel Burrows[2,3], Otávio Lemos[4], Alessandro Garcia[2], Eduardo Figueiredo[3], Nelio Cacho[5], Frederico Lopes[6], Nathalia Temudo[7], Liana Silva[7], Sergio Soares[8], Awais Rashid[3], Paulo Masiero[1], Thais Batista[6], José Maldonado[1]

[1] Computer Systems Department, University of São Paulo – USP, São Carlos, Brazil
[2] Informatics Department, Pontifical Catholic University of Rio de Janeiro – PUC-Rio, Rio de Janeiro, Brazil
[3] Computing Department, Lancaster University, Lancaster, United Kingdom
[4] Department of Science and Technology, Federal University of São Paulo – UNIFESP, S.J. Campos, Brazil
[5] School of Science and Technology, Federal University of Rio Grande do Norte – UFRN, Natal, Brazil
[6] Computer Science Department, Federal University of Rio Grande do Norte – UFRN, Natal, Brazil
[7] Department of Computing and Systems, University of Pernambuco – UPE, Recife, Brazil
[8] Informatics Centre, Federal University of Pernambuco – UFPE, Recife, Brazil

{ferrari,masiero,jcmaldon}@icmc.usp.br, {rachel.burrows,e.figueiredo,marash}@comp.lancs.ac.uk, otavio.lemos@unifesp.br
afgarcia@inf.puc-rio.br, neliocacho@ect.ufrn.br, {nmt,lsos}@dsc.upe.br, scbs@cin.ufpe.br, {fred.lopes,thais}@ufrnet.br

ABSTRACT

This paper presents the results of an exploratory study on the fault-proneness of aspect-oriented programs. We analysed the faults collected from three evolving aspect-oriented systems, all from different application domains. The analysis develops from two different angles. Firstly, we measured the impact of the obliviousness property on the fault-proneness of the evaluated systems. The results show that 40% of reported faults were due to the lack of awareness among base code and aspects. The second analysis regarded the fault-proneness of the main aspect-oriented programming (AOP) mechanisms, namely pointcuts, advices and intertype declarations. The results indicate that these mechanisms present similar fault-proneness when we consider both the overall system and concern-specific implementations. Our findings are reinforced by means of statistical tests. In general, this result contradicts the common intuition stating that the use of pointcut languages is the main source of faults in AOP.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – Diagnostics, Code Inspections; D.3.3 [**Programming Languages**]: Language Constructs and Features.

General Terms

Measurement, Experimentation, Languages, Verification.

Keywords

Aspect-oriented programming, fault-proneness, software testing.

1. INTRODUCTION

With software development becoming increasingly incremental, programmers should be aware of contemporary implementation mechanisms that are fault-prone in the presence of changes. In particular, the recent adoption of aspect-oriented programming (AOP) languages and frameworks, such as AspectJ [32] and Spring [23], requires a better understanding of the fault causes in AOP. AOP [25] is a programming technique that aims to improve the modularisation and robust implementation of concerns that cut across multiple software modules. Classical examples of crosscutting concerns usually implemented as aspects are logging, exception handling, concurrency, and certain design patterns.

The modularisation of crosscutting concerns in AOP is achieved through a complementary set of programming mechanisms, such as pointcut, advice, and intertype declaration (ITD) [32]. In addition, a basic property associated with AOP is *obliviousness*. This property implies that the developers of core functionality need not be aware of, anticipate or design code to be advised by aspects [15]. Obliviousness is also influenced by quantification, i.e. the ability to declaratively select sets of points via pointcuts in the execution of a program. The more expressive a pointcut language is, the more support for obliviousness it provides [31].

New AOP models, frameworks and language extensions are constantly emerging, in some cases leveraging different degrees of obliviousness [17, 20, 26, 35]. While some researchers have been optimistic about the benefits of AOP [26], others have shown scepticism [1, 2, 28]. For example, previous research has indicated that the use of certain AOP mechanisms can violate module encapsulation [1] and even introduce new types of faults [2]. In particular, some researchers claim these faults are likely to be amplified in the presence of evolutionary changes [24]. However, the empirical knowledge about the actual impact of AOP on fault-proneness remains very limited even for core programming mechanisms, such as pointcuts and intertype declarations.

Concerned with these issues, we present a first exploratory analysis on the fault-proneness of AOP properties and mechanisms. Our analysis develops in terms of the three following questions: (1) How does obliviousness influence the presence of faults in aspect-oriented (AO) programs? (2) What is the impact of specific AOP mechanisms on fault-proneness? and (3) Whether and how do certain characteristics of concern implementations correlate with the introduction of faults? To the best of our knowledge, this is the first initiative that systematically

ICSE'10, May 2–8, 2010, Cape Town, South Africa.

tackles these questions based on the exploratory analysis of real-world AO systems. Previous research has mainly focused on defining fault taxonomies and testing approaches based on AOP main concepts and current technologies [2, 3, 4, 10, 12, 34, 36]. However, we can only find limited empirical evidence on how such faults occur in practice.

To achieve our goals, we selected three AO systems from different application domains. We analysed fault-prone AOP mechanisms by means of various testing and static analysis. Throughout the study, faults were reported by two means: (1) by developers during system development and evolution; then afterwards (2) by independent testers who reported post-releases faults. The main findings of our study are:

- Obliviousness has been a controversial software property since the early research in AOP [28, 31]. Our analysis confirms that the lack of awareness between base and aspectual modules tends to lead to incorrect implementations. Despite the modern support provided by AOP-specific IDEs, uncertainty about module interactions still remains in the presence of aspects within the system.
- Interestingly, our findings contradict the common intuition stating that the use of pointcut languages is the main source of faults in AOP [10, 12, 29]. The main AOP mechanisms currently available in AspectJ-like languages – namely, pointcuts, advices and intertype declarations (ITDs) – present similar fault-proneness when the overall system is considered.
- The numbers of internal AOP mechanisms showed to be good fault indicators when considering the sets of modules within each concern[1] implementation separately. In this case the number of faults associated with a concern was directly proportional to the number of AOP mechanisms used to implement that concern.

In addition, the gathered results in our study support these findings with statistical significance. The remainder of this paper is organised as follows: Section 2 summarises the related research. Section 3 describes the study configuration. It includes our research hypotheses and a description of the target systems and evaluation procedures. Following, Section 4 brings the collected data. This data is analysed and discussed in Section 5. Section 6 discusses the limitations of this work. Finally, Section 7 presents our conclusion and future research.

2. RELATED WORK

The difficulty of performing fault-proneness evaluation in AOP is mainly twofold: (i) there are not yet several releases and documented faults for AO software projects available for analyses, and (ii) AOP has introduced new properties, such as obliviousness and quantification, and a wide range of mechanisms often used to implement different categories of crosscutting concerns. Previous research was limited to evaluate the benefits and drawbacks of aspects from different angles, such as design stability [13, 16] and system robustness [5, 9, 14]. However, the fault-proneness of AO programs is not yet a well-understood phenomenon.

To date, there is limited empirical knowledge of the impact of AOP on software correctness. In spite of that, we present pieces of work that we believe are mostly related to our own. They are basically distributed in two categories discussed in the following:

Fault taxonomies and fault-proneness of AO programs: A number of fault taxonomies and bug patterns for AO programs can be found in the recent literature [2, 4, 9, 12, 36]. Alexander et al. [2] proposed the first study on AOP-specific faults. Their main contribution is a characterisation of possible sources of faults that are specific to AO programs. Based on these sources, Alexander et al. also defined a high-level fault taxonomy for AO software, mainly focusing on features of AspectJ-like languages. Further fault taxonomies were built on top of it [4, 36]. They either include new fault types or refine the already existing ones. However, none of them have been empirically evaluated to date. Ferrari et al. [12] summarised these and other taxonomies in a broader fault categorisation for AO software. The study presented in this paper uses such categorisation for classifying the AOP-related faults found in the target systems.

So far, we have identified a single attempt to evaluate the fault-proneness of AOP programs. Coelho et al. [9] present an exploratory study of the robustness of AspectJ-based systems. The study analyses the flow of exceptions in five medium-sized OO systems from different domains and their AO counterparts, whose exception-handling code have been aspectised. The results show that in all AO systems there was an increase in the number of uncaught exceptions (i.e. exceptions that cross the system boundaries without being handled) and also in the number of unexpected handler actions. Differently from our work, Coelho et al. only focus their analysis on exception handling mechanisms, while we focus on AOP mechanisms in general.

Fault-proneness of crosscutting concerns: Eaddy et al. [11] performed an empirical study which provides evidence suggesting that crosscutting concerns cause defects. The authors examined the concerns of three medium-sized open-source Java projects and found that the more scattered the implementation of a concern is, the more likely it is to have defects. Their findings recommend the use of AOP techniques to modularise crosscutting concerns in order to reduce the number of faults potentially caused by them. However, Eaddy et al. have not investigated the fault-proneness of AOP mechanisms as we do in this study. In our study, we also analyse crosscutting concerns, although from a different angle. We aim to evaluate the impact of the specific characteristics of AO concern implementations on the system fault-proneness. We also highlight that we do not contrast AO implementations with OO counterparts in order to find out which approach results in more or less faults related to crosscutting concerns.

3. STUDY SETTING

This section describes our study configuration. Section 3.1 presents our goals and hypotheses. Section 3.2 provides an overview of the target systems. Section 3.3 explains the evaluation procedures we applied to each selected system and the associated tooling support.

3.1 Goal Statement and Research Hypotheses

Our objective is to evaluate the fault-proneness of AOP properties and mechanisms when they are applied to evolving programs. We are particularly interested in observing the underlying reasons of the introduction of faults. First, we analyse whether a key property of AOP, obliviousness, facilitates the emergence of faults under software evolution conditions. Moreover, we aim at analysing how specific characteristics of concerns being *aspectised* impact on the fault-proneness of AO programs. For example, we

[1] From hereafter, we use the term "concerns" to refer to crosscutting concerns in general, as defined by Kiczales et al. [25].

investigate how the internal implementation details of a concern, such as lines of code and use of specific AOP mechanisms, are correlated with the presence of faults. Based on these goals, we defined two research hypotheses. For each of them, the null and the alternative hypotheses are as follows:

Hypothesis 1 (H1)

- H1-0: Obliviousness does not exert impact on the fault-proneness of evolving AO programs;
- H1-1: Obliviousness exerts impact on the fault-proneness of evolving AO programs;

Hypothesis 2 (H2)

- H2-0: There is no difference among the fault-proneness of the main AOP mechanisms;
- H2-1: There are differences among the fault-proneness of the main AOP mechanisms;

To achieve our goals, we needed to apply a number of evaluation procedures. Our analysis embraced 12 releases of three AO systems from different application domains. Such systems include a wide range of heterogeneous concerns implemented as aspects. The systems and evaluation procedures are following described.

3.2 The Target Systems

The three medium-sized applications used in this study are from significantly different application domains. The first one, called iBATIS [22], is a Java-based open source framework for object-relational data mapping. It was originally developed in 2002 and over 60 releases are available at SourceForge.net[2] and Apache.org[3] repositories. The second application is HealthWatcher (HW) [16, 27], a typical Java web-based information system. HW was first released in 2001 and allows citizens to register complaints about health issues. The third evaluated system is a software product line for mobile devices, called MobileMedia (MM) [13]. MM was originally developed in 2005 to allow users to manipulate image files in different mobile devices. It has then evolved to support the manipulation of additional media files, such as videos and MP3 files.

Every AO release of a given system has an OO counterpart. In particular, iBATIS had its AO releases derived from OO builds available at SourceForge.net, which were used as baselines for implementation alignment. HW and MM, on the other hand, have evolved based on a sequence of planned changes, and both OO and AO versions of given release were developed concurrently [13, 16]. All releases have experienced exhaustive assessment procedures – code revision and testing (Section 3.3) – by independent developers in order to achieve functionalities well aligned with the original Java system.

From hereafter, we refer to the AO versions[4] of the target systems by their simple names or abbreviations, i.e. iBATIS, HW and MM. Four releases of iBATIS were considered in our evaluation, namely iBATIS[5] 01, 01.3, 01.5 and 02. We also analysed four HW releases – HW 01, 04, 07 and 10 – and four MM releases – MM 01, 02, 03 and 06. These releases were chosen because they encompass a wide range of different fine-grained and coarse-grained changes, such as refactorings and functionality increments or removals. Table 1 shows some general characteristics of the three target systems. For more information about each of them, the reader may refer to the respective placeholder websites or to previous reports of these systems [13, 16, 22].

Table 1. Key characteristics of the target applications.

	iBATIS	Health Watcher	MobileMedia
Application Type	data mapper framework	health vigillance application	product line for mobile data
Code Availability	Java/AspectJ	Java/AspectJ	Java/AspectJ
# of Releases	60 / 4	10 / 10	10 / 10
Selected Releases	4	4	4
Avg. KLOC	11	6	3
Avg. # of Aspects (only AspectJ)	46	23	10
Evaluation Procedure	testing	testing	interference analysis

We selected iBATIS as the main subject of illustrative examples in this paper in order to promote coherence in the discussions. This is the most complex target system from which we derived the largest data set (e.g. number of faults) and on which we mostly draw our analyses. However, we also refer to examples of the other systems in order to highlight recurring observations across the three systems. The highest number of faults in iBATIS was expected. The already-stable implementation of the other two systems yielded less fault-related data than iBATIS. Their implementations are more mature as they have been originally released for four years or more and underwent more corrective and perfective changes. HW and MM have also been targeted by a number of previous studies focusing in other equally-important quality attributes [9, 13, 14, 16, 27]. Therefore, as the AO implementations have different degrees of stability, the results originated from these systems will provide support for drawing more general findings.

3.3 The Evaluation Procedures

We followed different approaches to evaluate each target system. The evaluation procedures were defined according to the system characteristics and information available at the moment this study started. In short, we aimed at identifying as many faults as possible given time and resource constraints, while systematically avoiding bias while evaluating the three systems.

3.3.1 iBATIS Evaluation

Evaluation strategy: The iBATIS system has experienced two testing phases: pre-release and post-release testing. Pre-release testing aimed at producing defect-free code to be committed to a CVS repository. The test sets provided with the original OO implementations were used as baselines in this phase. Any abnormal behaviour when regressively testing the AO version of a given release was investigated. When a fault was uncovered, it should be documented in an appropriate detailed report (Section 3.3.4). Post-release testing, on the other hand, aimed at assessing the implementation through enhancement of the original test sets. The enhanced tests were executed against both OO and AO versions of a given release. A fault should *only* be reported if it was noticed in the AO version but *not* in the OO counterpart. This procedure ensured that only faults introduced during the aspectisation process would be reported for further analysis.

Tooling support: For test case design and execution, we used JUnit[6] and GroboUtils[7], a JUnit extension that enables multi-

[2] http:// sourceforge.net/projects/ibatisdb/files/ (03/02/2010)

[3] http://archive.apache.org/dist/ibatis/binaries/ibatis.java/ (03/02/2010)

[4] References to OO counterparts will be made explicit throughout the text.

[5] Such releases correspond to the original builds #150, #174, #203 and #243 found in SourceForge.net, respectively.

[6] http://www.junit.org/ (03/02/2010)

[7] http://groboutils.sourceforge.net/ (03/02/2010)

threaded tests. To measure test coverage, we used Cobertura[8], a tool that allows for fast code instrumentation and test execution.

3.3.2 HW Evaluation

Evaluation strategy: HW was tested in a single phase in our study as initial testing was already performed during its development time. Such initial tests involved people who were unaware of evaluations that would be further performed. Hence, the testing of the HW system was extended in this study to cover the assessment phase, thereby improving the degree of test coverage. Differently from iBATIS, however, no original test set was made available. Thus a full test set was built from scratch based on the system specification and code documentation. In order to reduce test effort and avoid systematic bias during test creation, test cases were automatically generated with adequate tool support. As well as for iBATIS, a fault should *only* be reported if it was noticed in the AO version but *not* in the OO counterpart, and all uncovered faults were similarly reported.

Tooling support: We used CodePro[9], an Eclipse plugin for automatic JUnit test case generation for Java programs. We also used Mockrunner[10], a lightweight framework for testing web-based applications. It extends JUnit and provides the necessary facilities to test the servlets implemented within the HW system. Finally, we used Cobertura to measure the test coverage.

3.3.3 MM Evaluation

Evaluation strategy: As well as HW, MM has not been developed with awareness of further fault-based evaluation. Moreover, post-release tests during system evolution and maintenance only revealed faults related to robustness (e.g. data input validation), however not necessarily being related to AOP mechanisms. Despite this, we have evaluated MM using the Composition Integrity Framework (CIF) [6]. CIF helped us identify problems in aspect interactions established either between aspects and base code or among multiple aspects. Since MM is a software product line and includes mandatory, optional and alternative features, CIF was applied in varied configurations of MM. In doing so, we were able to derive a set of faults that resulted from a broad range of aspect-oriented compositions.

Tooling support: We used the CIF framework, which performs static analysis of join point shadows. CIF is able to: (i) report the join point shadows that are involved in a specific composition, and (ii) report aspect interactions which are not governed by an explicit dependency, e.g. via the use of the declare precedence statement.

3.3.4 Fault Reporting

Every fault identified either during development (iBATIS only), during the assessment phase (iBATIS and HW), or during static analysis (MM) was documented in a customised report form. During the assessment or static analysis phases, the testers provided information as much as possible, with special attention to the test case(s) that revealed the fault (if applicable) and the fault symptom. In addition, the tester provided some hints about the fault location. Then, the reports were forwarded to the original developers, who were in charge of concluding the fault documentation.

3.3.5 Fault Classification

After the fault documentation step, each fault was classified according to a fault taxonomy for AO software proposed in our previous research [12]. This taxonomy includes four high-level categories of faults that comprise the core mechanisms of AOP: (1) pointcuts; (2) introductions (or intertype declarations – ITDs) and other `declare`-like structures; (3) advices; and (4) the base program. In order to systematically classify every fault, it was took into account the fault origin and not only its side-effects. The classification based on the first three categories above was straightforward. For instance, it was relatively trivial to identify mismatching pointcuts, misuse of `declare`-like structures or wrong advice types. However, base program-related faults required extended analysis and reasoning about them. For example, base code changes that result in broken pointcuts should be classified as base program-related although their side-effects might be unmatched join points in the code. Section 5 analyses the impact of each fault category on the overall fault distribution considering all targeted systems.

4. DATA COLLECTION

This section presents the results obtained for each target system. Section 4.1 describes the results of test execution in iBATIS and HW releases, and the number of aspect interaction problems identified in the MM configurations. Section 4.2 presents the fault distribution per fault category and the fault distribution per concern.

4.1 Test Execution Results

Figure 1 shows the test execution results for iBATIS (on the left-hand side) and HW (on the right-hand side). The iBATIS original test sets comprise 100, 103, 108 and 130 test cases for each release, respectively. The final, improved test sets include 246 test cases for iBATIS 01, 01.3 and 01.5, and 256 tests for release 02. For a given release, new test cases were either mined from the successive releases in the SourceForge repository or designed from scratch. According to Figure 1, the number of successful test cases increased across the releases, what might mean code enhancement. However, as discussed in the next sections, this not necessarily means reduction in number of faults.

HW test sets were entirely designed for this study, with support from CodePro. Additionally, a few tests were manually implemented based on functional requirements. In total, 925, 981, 998 and 998 tests were generated for releases 01, 04, 07 and 10, respectively. We can observe in Figure 1 that the number of tests that failed plus the tests that raised exceptions (labelled as *error* in the legend) increased across the releases. This suggests that the number of faults increased during the system evolution. However, as we will see in Section 4.2, this not necessarily means larger number of faults in successive releases.

Figure 2 presents the coverage achieved for each release of iBATIS and HW. For iBATIS in particular, the figure shows the coverage obtained with the original and enhanced test sets. In spite of HW test sets being significantly larger than the iBATIS ones, the yielded coverage in HW is lower than in iBATIS. This is due to automatically-generated redundant tests that exercise common parts of the code. Nevertheless, for both systems, we focused on the creation of tests that exercise parts of the code affected by the aspectual behaviour. Following this strategy, we were able to uncover faults not yet revealed in previous system evaluations.

[8] http://cobertura.sourceforge.net/ (03/02/2010)

[9] http://www.instantiations.com/ (03/02/2010)

[10] http://mockrunner.sourceforge.net/ (03/02/2010)

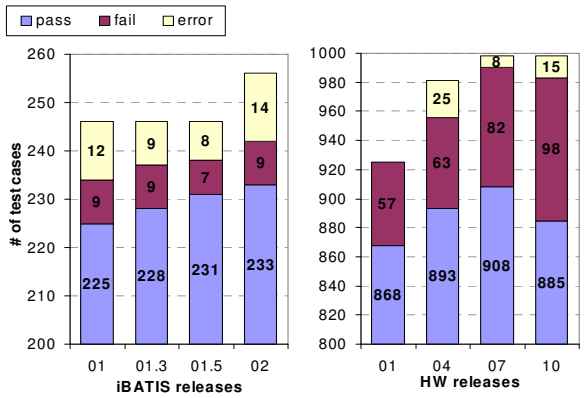

Figure 1. Test case execution in iBATIS (left) and HW (right) releases.

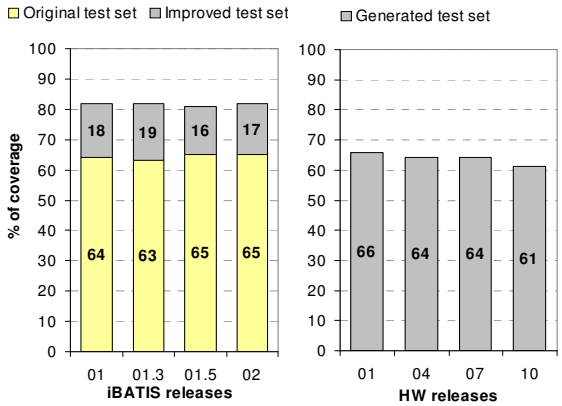

Figure 2. Test coverage in iBATIS (left) and HW (right) releases.

As described in Section 3.3, the MM system was evaluated through static analysis using the CIF framework [6]. This activity yielded a list of potential faults related to aspect interactions regarding advices that share common join points. In MM, such list included 16 potential faults for the varied configurations of the four evaluated releases. Further analysis enabled us to identify and classify the real faults. The results are presented in the next section.

4.2 Fault Distribution

This section summarises all faults we identified along our study. Faults are grouped per category (Section 4.2.1) and per concern (Section 4.2.2). The fault categorisation is in accordance to the fault taxonomy for AO programs [12].

4.2.1 Fault Distribution per Fault Category

Tables 2 and 3 respectively present the number of reported faults for iBATIS and for all the systems. The reported faults are grouped per category. The lower number of faults uncovered for HW and MM can be explained by their size – they are smaller than iBATIS – and by these systems having experienced only post-release evaluation. A total of 83 faults in iBATIS (20.8 faults on average per release) and 104 considering all systems have been documented. In-depth analyses of the reported faults drive the discussion presented in Section 5.

.

Table 2. Fault distribution per category in iBATIS.

Fault Category	iBATIS releases				Total	Average
	01	01.3	01.5	02		
Pointcut-related	10	2	1	5	**18**	4.5
ITD-related	5	2	1	6	**14**	3.5
Advice-related	6	4	1	4	**15**	3.8
Base-program related	2	1	10	23	**36**	9.0
Total	**23**	**9**	**13**	**38**	**83**	**20.8**

Table 3. Fault distribution per category in all systems.

Fault Category	iBATIS	MM	HW	Total
Pointcut-related	18	1	0	19
ITD-related	14	4	6	24
Advice-related	15	4	4	23
Base-program related	36	0	2	38
Total	**83**	**9**	**12**	104

4.2.2 Fault Distribution per Concern in iBATIS

Table 5 presents the fault distribution per concern in iBATIS. To analyse this distribution, we considered only the iBATIS data set because it contains the largest amount of faults. Moreover, iBATIS is the only system where faults appear distributed over all the aspectised concerns. Data sets collected for the other two applications, on the other hand, were limited to post-release evaluations only and are not considered in this section. The concerns listed in Table 5 are briefly described in Table 4. Notice that some concerns are aspectised only in releases 01.5 and 02. Section 5.2 discusses how certain implementation characteristics of a concern (e.g. required AOP mechanisms) may impact on the fault-proneness of that specific concern.

Table 4. Concerns implemented with aspects in iBATIS.

Concern	Description	Release
Concurrency	Ensures multiple activities and requests could be executed within the framework in a consistent manner.	all
Type Mapping	Deals with the mapping of data into different formats. E.g. when data is retrieved and stored in the database, the application checks to see if the data content is not null before proceeding with the transaction.	all
Design Patterns	Subset of the Gang-of-Four design patterns such as Singleton, Observer, Adapter, and Strategy	all
Error Context	ErrorContext object stores data about executing activities. This data is used and sometimes printed as an event trace in the event of an exception.	all
Exception Handling	Mechanisms that deal with to an erroneous execution flows (In Java this includes try/catch/throws and finally clauses).	all
Connection, Session & Transaction	Detected as three separate concerns, regards mechanisms that allow for database access and control. E.g. transaction managers and SQL query runners.	01.5 and 02

Table 5. Fault distribution per concern in iBATIS.

Concern	iBATIS releases				Total	Average
	01	01.3	01.5	02		
CC - Concurrency	0	0	0	1	**1**	0.25
TM - Type Mapping	2	0	0	1	**3**	0.75
DP - Design Patterns	2	0	0	0	**2**	0.5
EC - Error Context	13	5	1	2	**21**	5.25
EH - Exception Handling	6	4	2	16	**28**	7
CN - Connection	--	--	7	10	**17**	8.5
SS - Session	--	--	3	3	**6**	3
TR - Transaction	--	--	0	3	**3**	1.5
OT - Others	0	0	0	2	**2**	0.5
Total	**23**	**9**	**13**	**38**	**83**	**20.8**

5. DATA ANALYSIS AND DISCUSSION

This section performs exploratory and statistical analyses of the measures presented in Section 4. We aim at identifying AOP properties and mechanisms that tend to yield faulty implementations. Section 5.1 evaluates the H1 hypothesis, i.e. how the obliviousness property impacts the correctness of AO programs in the presence of code evolution. Section 5.2 evaluates the H2 hypothesis in order to identify the fault-proneness of specific AOP mechanisms. We evaluate H2 from two points of view: considering the overall system implementation, and the implementations of specific concerns. Both analyses for H2 are supported by statistical tests.

For the statistical tests performed along section 5.2, we used the R language and environment[11]. We use the *Pearson's chi-square* test to check whether or not there is a statistically significant difference between the fault counts. We assume the commonly used confidence level of 95% (that is, p-value threshold = 0.05). The *Spearman's rank correlation* test is used to check how the fault counts correlate with AOP mechanism counts. This test is used because in our analysis the used metrics (i.e. number of faults) are nonparametric. For evaluating the results of the correlation tests, we adopted the Hopkins criteria to judge the goodness of a correlation coefficient [21]: < 0.1 means trivial, 0.1-0.3 means minor, 0.3-0.5 means moderate, 0.5-0.7 means large, 0.7-0.9 means very large, and 0.9-1 means almost perfect.

5.1 H1: The Impact of Obliviousness

Obliviousness plays a central role in AOP, but there is little empirical knowledge on how this property actually affects the fault-proneness under usual development settings. We then analysed its impact on the fault-proneness of AO systems from two viewpoints: (i) obliviousness and software evolution; and (ii) a categorisation of obliviousness listed by Sullivan et al. [31]. The results are following presented.

5.1.1 Obliviousness and Software Evolution

Considering the fault distribution per fault category (Tables 2 and 3) for iBATIS, the total number of faults related to the base code was 36, what is at least twice as large as any other number of faults within the other three categories. From these, 27 faults were caused by either perfective or evolutionary changes within the base code, what led pointcuts to break. They represent the largest number amongst all fault types reported for the iBATIS system, which in turn corresponds to 33% of the total number of faults.

This problem, usually referred to as the *fragile pointcuts* problem [29], is closely related to the quantification and obliviousness models implemented in AspectJ-like languages. Changing a program requires a review of the crosscut enumerations (i.e. the pointcuts) which conflicts with the idea of programs being oblivious to the aspects applied to them [18]. We found that this problem is magnified in realistic development scenarios as the one observed in iBATIS. Several developers worked in parallel in the iBATIS project, each of them refactoring and evolving different crosscutting concerns into aspects. This means that obliviousness was present not only between base code and aspects. The aspect implementations were oblivious to each other as well. For example, an aspect that advises a set of join points might not be aware of other aspects inserting behaviour into the same join points, hence rising the risk of either misbehaviour or pointcut mismatching in the event of any code change. Partial aid currently provided by AOP IDEs increases the developers' awareness of the aspect effects in the base code. However, uncertainty about how aspects indirectly interfere in the base code still remains.

We noticed this problem occurred mainly in the iBATIS system as faults were reported during both development and assessment phases (Section 3.3.4). Evolving some functionality necessarily required fixing faults identified in the existing base or aspect code. On the other hand, HW and MM implementations were more stable and were extensively evaluated in previous research [9, 13, 14, 16, 27]. For example, since HW had been first released, a number of incremental and perfective changes took place [16], what resulted in both base and crosscutting code being more stable than in iBATIS code. Due to these refinements, HW and MM had proportionally fewer faults in this category, most likely due to the robustness of the code. Nonetheless, none of the three systems have been evaluated in terms of fault-proneness, as presented in this paper.

We further analysed the MM system this using the CIF framework [6], and the results reinforce our findings. The majority of faults here were sourced from areas of code where aspect interactions occurred at runtime. For example, in 45% of cases (4 out of 9), faults were caused due to missing `declare precedence` statements. These faults resulted in arbitrary execution order of advices that share the same join point. Obliviousness was clearly the main reason for the introduction of faults in these cases, where aspects were successively introduced along the development cycles. Considering the same scenario in Java, the developers were naturally enforced to make an explicit design decision on the order of respective pieces of behaviour within a method.

5.1.2 Obliviousness Categories

We classified all documented faults according to the four categories of obliviousness listed by Sullivan et al. [31]. The summary of this categorisation is presented in Table 6. The goal was to gather further evidence about the impact of obliviousness on the correctness of the evaluated systems. The obliviousness categories represent different types of information hiding. We focused on two types of obliviousness that are relevant for the purposes of this analysis, briefly described as follows: (i) *Language-level obliviousness* is present when there is no local notation in the code about aspect behaviour that may be inserted at this point; and (ii) *Feature obliviousness*, which is present when the base code developer is unaware of the features or the in-depth semantics of an aspect that is advising the base code.

Even though it is impossible to be entirely sure of the true causes of faults, we followed a set of guidelines to decide if the collected faults were likely to have been caused by language-level and/or feature obliviousness. In short, when behaviour is inserted at join points via advice, we can claim that language-level obliviousness

[11] http://www.r-project.org/ (03/02/2010)

is present. This is because there is no explicit call or notation of this extra behaviour within the base code. We categorised a fault as caused by language-level obliviousness if the fault was likely to be avoided in case such an explicit local notation was present. Now, let us consider a base code developer who has followed specific design rules to expose certain join points or create hooks for an aspect developer without full knowledge of the implementation details of this aspect. In this case, language-level obliviousness is not present, but feature obliviousness is. We classify a fault as being related to feature obliviousness if, in order for the fault to have been avoided, further attention would need to be given to the aspect semantics.

Table 6. Faults associated with obliviousness.

System	Obliviousness Category					
	Language	Feature	Both	Language only	Feature only	Total
iBATIS	31	4	4	27	0	31
HW	0	3	0	0	3	3
MM	8	4	4	4	0	8
Total			8	31	3	42

The results of this categorisation show that most of the faults related to obliviousness were categorised as language-level. They represent around 70% of all base program-related faults (i.e. 26 out of 38 – see Total column in Table 3). In regard to faults related to feature obliviousness, they were mostly found in cases where aspects either directly interact within the same module or share common join points. This indicates that evolving code that is oblivious to aspects has varying impacts on the fault-proneness of the system. This problem was mainly observed in iBATIS, in which faults have been reported during the evolution of the releases. MM and HW, on the other hand, experienced only post-release tests. Nevertheless, 11 out of 21 faults revealed for HW and MM were assigned to obliviousness at either language-level, feature or both (see Total column in Table 6).

To conclude, analysing the impact of obliviousness on the fault-proneness of the evaluated systems provided us with evidences that support the H1 alternative hypothesis (H1-1). That is, "*Obliviousness exerts impact on the fault-proneness of evolving AO programs*". In our study, a large amount of faults (40%) could be directly associated with the base code being oblivious to aspects. Their majority was observed in the iBATIS evolution. Many faults observed were also related to aspects being oblivious to other aspects. Missing `declare precedence` statements were responsible for 45% of the faults found in MM. Considering all faults (Table 3), 11% were categorised as feature obliviousness-related, although a much larger proportion were related to language-level (i.e. 38%). This result is interesting because it might further reinforce the motivation for AOP models based on explicit class-aspect interfaces, such as XPIs [17] and EJPs [20].

5.2 H2: Fault-proneness of AOP Mechanisms

There is often an assumption that the use of pointcut languages is the main source of faults in aspect-oriented programs [10, 12, 29]. However, there is limited understanding of the real magnitude of pointcut faults with respect to other AOP mechanisms. The analysis of our second hypothesis is drawn in terms of the fault categorisation presented in Section 4. The null hypothesis (H2-0) states that there is no significant difference amongst the fault-proneness of core AOP mechanisms.

5.2.1 Analysing the overall fault distribution

Initially, we analyse the values presented in Table 2. Examination of this data indicates that there is a similarity among the total number of faults found in iBATIS, considering the first three categories (pointcut-, advice- and ITD-related). These results suggest these mechanisms present similar fault-proneness; that is, none of them stands out with respect to the number of faults. To further analyse this observation statistically, we first applied a Pearson's chi-square test. This test checks the probability of sample data coming from a population with a specific distribution. If we reach a probability (p-value) higher than, say, 0.05, we can assert with 95% confidence level that there is no reason to reject the hypothesis that the observed data fits the given distribution. In our case, at a confidence level of 95%, the result confirms the uniformity of the fault frequencies among each fault category: the p-value is evaluated to 0.7584, significantly higher than 0.05. This is easy to see since fault counts were 18, 14, and 15; very close to the fitted model where each category is expected to have approximately the same number of faults (15.67 in this case). We then applied the chi-square test to the overall fault set, considering all target systems (i.e. the total of 104 faults presented in Table 3). Again, assuming the confidence level of 95%, the result corroborates the previous finding, i.e. the faults are uniformly distributed over the three main AOP mechanisms, with p-value being evaluated to 0.7275, again significantly higher than 0.05.

Contradicting the conventional wisdom, we have first evidence that supports the H2 null hypothesis (H2-0) that "*there is no difference among the fault-proneness of the main AOP mechanisms*". This is also an interesting result as many researchers have strictly focused on improving the design of pointcut languages (e.g. providing support for more expressive pointcuts [5, 18, 20]). Less attention has been given to support more robust programming with other classical AOP mechanisms. The next section presents a more refined analysis that brings additional evidence on the fault-proneness of such mechanisms from a different point of view.

5.2.2 Analysing the fault distribution per concern

The analysis of fault distribution per fault category only took into account the overall number of faults per category. This section provides a more refined analysis about fault-proneness of AOP mechanisms. For this, we considered certain internal details in the implementation of each concern. We performed a correlation analysis to gather empirical evidence of a cause-effect relationship between the number of AOP mechanisms and defects. This is motivated by the fact that AOP mechanisms may have individual impact on the fault-proneness of a module, a cluster of modules (e.g. modules that implement a given concern) or the full system. For this analysis, we considered only the set of faults identified in iBATIS, since it includes representatives distributed over all the aspectised concerns (Table 4). Moreover, we also considered base program-related faults in order to measure the impact of AOP mechanisms in the system as a whole.

Initially, we applied the Spearman's rank correlation to check how the overall number of AOP mechanisms (pointcuts, advices and ITDs) per release in iBATIS (Table 7) correlates with the number of faults in the same release. The results are presented in Table 8. The correlation is generally low, considering all AOP mechanisms, thus contradicting the results that regard fault distribution per fault category (Section 5.2.1).

Table 7. Number of AOP mechanisms and faults in iBATIS.

iBATIS release	Pointcuts	Advices	ITDs	Faults
01	97	94	79	23
01.3	121	118	86	9
01.5	244	240	238	13
02	244	238	237	38

However, while performing such an analysis based in internal properties of the systems, we should take into account concern-specific characteristics. This is due to the nature of concern implementations, which usually require subsets of AOP mechanisms to be used together. For example, aspectising an exception handler usually requires three coding structures in AspectJ: a pointcut expression, a `declare soft` statement and an advice. In other words, we can investigate whether the number of pointcuts, advices and ITDs (including `declare`-like mechanisms) implemented for the purposes of a concern impact on the number of faults it presents.

Table 8. Correlation between number of faults and number of AOP mechanisms in iBATIS releases.

Metric	Coefficient	P-value
POINTCUTS	0.2108185	0.7892000
ADVICES	0.0000000	1.0000000
DECLARATIONS	0.0000000	1.0000000

Hence, we checked how the number of AOP mechanisms used to implement a concern correlates with the fault distribution per concern. We measured the maximum and the average number of each AOP mechanism per concern across all iBATIS releases. Note that, for a given concern, we considered all modules (aspects and classes) that were involved in its implementation. We applied again the Spearman's rank correlation to the total and average number of faults per concern across the releases. We compared these numbers against the maximum and average number of advices, pointcuts, and ITDs per concern across releases. With such an analysis we can observe whether and how the usage frequency of each AOP mechanism seems to impact on the fault distribution per concern. We used the maximum and average number of mechanisms across releases because they might repeat from release to release. That is, the same pointcut implemented in an exception handling aspect in one release may be present in the same aspect in another release

We also chose two metrics typically applied to OO and AO programs in order to compare the results obtained in this analysis. These metrics are lines of code (LOC) and weighted operations per module (WOM) [7]. WOM adapts the original weighted methods per class (WMC) metric [8] to count methods inside classes as well as aspect operations (i.e. advices, methods and intertype methods). These metrics have been reported as good fault-proneness indicators in studies that comprised OO programs [19, 30]. Again, we considered their maximum and average values across releases for the clusters of modules required for the implementation of each concern.

Table 9 shows the statistics for the AOP mechanisms, LOC and WOM metrics in iBATIS. We again adopted the confidence level of 95%. Tables 10 and 11 present the results of the Spearman's correlation rank run against: (i) the maximum and average number of mechanisms across releases, and (ii) the total and average number of faults per concern across releases. The values and results comprising LOC and WOM metrics are also presented in these tables.

Note that the correlation coefficient is very large for all correlations that take into account the maximum and the average number of pointcuts and advices. In fact, we can observe that the correlation between the maximum number of pointcuts and advice and the average number of faults per concern is significant. We observed a 99% level of confidence (Table 10). For ITDs, the correlation coefficient varies between 0.5 and 0.6, what means moderate-to-large correlation on average if we consider a confidence level to 85%.

Table 10. Correlation with average number of faults per concern/release.

Metric	Coefficient	P-value
MAX-POINTCUTS	0.8809524**	0.0072420
MAX-ADVICES	0.8742672**	0.0045120
MAX-ITDs	0.5509081	0.1570000
MAX-LOC	0.1904762	0.6646000
MAX-WOM	0.1666667	0.7033000
AVG-POINTCUTS	0.8571429*	0.0107100
AVG-ADVICES	0.8571429*	0.0107100
AVG-ITDs	0.5714286	0.1511000
AVG-LOC	0.1904762	0.6646000
AVG-WOM	0.1666667	0.7033000

** correlation is significant at the 0.01 level

* correlation is significant at the 0.05 level

Differently from results of previous studies comprising OO programs [19, 30], LOC and WOM metrics show non-significant correlation with both average and maximum number of faults in our study. These results indicate that when we consider the set of modules involved in AO implementations of crosscutting concerns, the internal number of AOP-specific mechanisms (i.e. pointcuts, advices and ITDs) are good fault-proneness indicators. Moreover, they seem to be better indicators than the traditional LOC and WOM metrics.

While performing the analysis presented in this section, we noticed that: (i) concern-specific characteristics define the set of AOP mechanisms that must be used in conjunction to implement such a concern, and (ii) given a specific concern implementation, the usage rate of each mechanism tends to be directly proportional

Table 9. Number of AOP mechanisms, LOC and faults per concern in iBATIS.

	Pointcuts		Advices		ITDs		LOC		WOM		Faults	
	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Total	Avg
Concurrency	7	6.75	5	4.5	3	2.75	1,605	1,435	395	373	1	0.25
Type Mapping	2	2	2	2	3	3	496	496	298	298	3	0.75
Design Patterns	9	6	5	3.5	14	9.5	1,725	1,566	448	391	2	0.5
Error Context	42	26.75	45	29.75	1	0.5	2,109	1,926	450	404	21	5.25
Exception Handling	77	70.25	75	69	82	74.75	4,761	4,490	1,159	994	28	7
Connection	64	64	64	32	61	60.5	901	890	359	358	17	8.5
Session	46	45.5	46	22.75	25	25	331	325	208	203	6	3
Transaction	20	20	20	10	54	53.5	686	684	223	221	3	1.5

to the number of faults associated with that concern, what is supported by the correlation test results. These findings support the H2 null hypothesis (H2-0) because the overall fault distribution per main AOP mechanism showed to be uniform. In addition, the usage rate of each mechanism does not vary independently. That is, it depends on the set of concerns aspectised within a system.

Table 11. Correlation with total number of faults per concern.

Metric	Coefficient	P-value
MAX-POINTCUTS	0.8263621*	0.0114400
MAX-ADVICES	0.8192771*	0.0128300
MAX-ITDs	0.3915663	0.3374000
MAX-LOC	0.3473116	0.3993000
MAX-WOM	0.3473116	0.3993000
AVG-POINTCUTS	0.8024096*	0.0165400
AVG-ADVICES	0.8024096*	0.0165400
AVG-ITDs	0.4191692	0.3013000
AVG-LOC	0.3473116	0.3993000
AVG-WOM	0.3473116	0.3993000

* correlation is significant at the 0.05 level

6. STUDY LIMITATIONS

This section discusses the study limitations based on the four categories of validity threats described by Wohlin et al. [33]. Each category includes a set of possible threats for an empirical study. We identified the items within each category that might threat our study, which are discussed in the following. For each category, we list possible threats and the measures we took to reduce each risk.

Conclusion validity. We identified two categories in this case: (i) *reliability of measures*: subjective decisions were made during the fault classification steps, specially regarding obliviousness levels (Section 5.1); besides, one of the target systems was evaluated with auto-generated test cases (Section 3.3.2), what might have risked the reliability of test results; and (ii) *random heterogeneity of subjects*: evaluated systems came from different application domains. To reduce risk (i), we designed detailed fault report forms and defined a set of guidelines that were followed in order to systematically classify each fault (Section 3.3.5). In regard to the evaluation based on auto-generated tests, this technique has previously yielded relevant results [37, 38], so it should not be seen as a major issue. Regarding risk (ii), although the applications' heterogeneity is considered a threat to the conclusion validity, it helps to promote the external validity of the study.

Internal validity. We detected two possible risks: (i) *ambiguity about direction of causal influence*: the complexity of the aspectised concerns might have made a system release more faulty than the others; and (ii) *history and maturation*: HW and MM systems have been extensively evaluated and continuously improved through the last years, what reduced the number of uncovered faults in such systems. Risk (i) cannot be completely avoided as each functionality differs from the others w.r.t. complexity. However, it was reduced because all systems were developed and revised by experienced programmers. They used implementation guides, design patterns or specific AOP idioms, where applicable. Moreover, systematic regression testing helped developers preserve the semantics of the OO counterparts. In order to reduce risk (ii), we focused our analyses on iBATIS, which consists in the most recent from all target systems and yielded the largest data set to be analysed.

Construct validity. We identified the following construct validity threats: (i) *inadequate operational explanation of constructs*: unclear procedures that should be followed in the event of a fault being uncovered might have biased the results (e.g. should the fault be fixed? How should this fault be classified?); (ii) *confounding constructs and levels of constructs*: different maturity levels of the investigated systems impacted the number of uncovered faults; and (iii) *interaction of testing and treatment*: iBATIS developers were aware of further system evaluation. To reduce risks (i) and (iii), we defined clear procedures and roles that were applied throughout all study steps. In particular, iBATIS development was strongly based on regression testing in order to make only error-free code available in the CVS repository. Although this approach made developers aware of the system evaluation procedures, it was important since it enabled developers to collect data since the early development phases. Risk (ii), on the other hand, could not be avoided due to the few options of medium-sized AO systems currently available for evaluation. Such systems present different levels of maturity, what includes varied fault rates.

External validity. The major risk here is related to the *interaction of setting and treatment*: the evaluated systems might not be representative of the industrial practice. To reduce this risk, we evaluated systems that come from heterogeneous application domains and are implemented with AspectJ, which is one of the representatives in the state of AOP practice. The iBATIS system is a widely-used open source project for object-relational mapping. Even though HW and MM are smaller applications, they are also heavily based on industry-strength technologies. In addition, both systems have been extensively used and evaluated in previous research [9, 13, 14, 16, 27]. To conclude, the characteristics of the selected systems, when contrasted with the state of practice in AO software development, represent a first step towards the generalisation of the achieved results.

7. CONCLUSIONS

This paper presented an exploratory study of the fault-proneness of AOP mechanisms used in the implementation of evolving AO programs. We analysed three systems from different application domains, from which we collected fault-related data upon which we performed our analyses. The results revealed the negative impact of obliviousness on the fault-proneness of programs implemented with AspectJ (H1 hypothesis). More recent methods and languages for AOP can help to ameliorate this problem. Examples of such approaches are aspect-aware interfaces [26], Crosscut Programming Interfaces (XPIs) [17] and Explicit Join Points (EJPs) [20]. Although they reduce the obliviousness among system modules, these approaches help to improve program comprehension by making aspect-base interaction more explicit. In particular, they tend to reduce the language-level obliviousness, which happened to be the category with the largest number of faults in our study.

As far as the H2 hypothesis is concerned, we did not confirm the common intuition that defining pointcuts is the most fault-prone scenario in AOP. There was no AOP mechanism that stood out as the main responsible for the detected faults. We also argue that this correlational study provides a good lead for more probing controlled experiments to investigate this issue further. Nevertheless, recent research on fault taxonomies and testing approaches for AO software has mainly focused on pointcuts as the key bottleneck in AOP [3, 4, 10, 12]. However, the results obtained for the H2 hypothesis motivate more intensive research on the testing support for other AOP mechanisms beyond pointcut expressions, such as intertype declarations and advice.

We believe that these study outcomes are helpful in several ways, such as: (i) providing information about harmful AOP mechanisms; (ii) supporting testing processes by pinpointing recurring faulty scenarios; and (iii) enhancing the general understanding towards robust AOP, so that other controlled experiments can be derived in our future research.

8. ACKNOWLEDGMENTS

We would like to thank the iBATIS AO developers Elliackin Figueiredo, Diego Araujo, Marcelo Moura and Mário Monteiro. We also thank Andrew Camilleri for his valuable help while analysing the MobileMedia system with the CIF framework.

The authors received full of partial funding from the following agencies and projects: *Fabiano Ferrari*: FAPESP (grant 05/55403-6), CAPES (grant 0653/07-1) and EC Grant AOSD-Europe (IST-2-004349); *Alessandro Garcia*: FAPERJ (distinguished scientist grant E-26/102.211/2009), CNPq (productivity grant 305526/2009-0 and Universal Project grant 483882/2009-7) and PUC-Rio (productivity grant); *Rachel Burrows*: UK EPSRC grant; *Otávio Lemos*: FAPESP (grant 2008/10300-3); *Sergio Soares*: CNPq (grant 309234/2007-7) and FACEPE (grant APQ-0093-1.03/08); *José Maldonado*: EC Grant QualiPSo (IST-FP6-IP-034763) and CNPq; *Other authors*: CAPES and CNPq, Brazil.

9. REFERENCES

[1] Aldrich, J. 2004. Open Modules: Reconciling Extensibility and Information Hiding. In: SPLAT AOSD'04 Workshop.

[2] Alexander, R. T., et al. 2004. Towards the Systematic Testing of Aspect-Oriented Programs. Report CS-04-105, Colorado State University, Fort Collins-USA.

[3] Anbalagan, P., and Xie, T. 2008. Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs. In: ISSRE'08. 239-248.

[4] Bækken, J. S., and Alexander, R. T. 2006. A Candidate Fault Model for AspectJ Pointcuts. In: ISSRE'06. 169-178.

[5] Cacho, N., Filho, F. C., Garcia, A., and Figueiredo, E. 2008. EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming. In: AOSD'08. 72-83.

[6] Camilleri, A., Coulson, G., Blair, L. 2009. CIF: A Framework for Managing Integrity in Aspect-Oriented Composition. In: TOOLS'09. 16-26.

[7] Ceccato, M., and Tonella, P. 2004 Measuring the Effects of Software Aspectization. In: 1st Workshop on Aspect Reverse Engineering (ARE).

[8] Chidamber, S.R., and Kemerer, C.F. 1994. A Metrics Suite for Object-Oriented Design. IEEE Transactions on Software Engineering 20 (6). 476-493.

[9] Coelho, R., et al. 2008. Assessing the Impact of Aspects on Exception Flows: An Exploratory Study. In: ECOOP'08. (LNCS, vol. 5142). 207-234.

[10] Delamare, R., Baudry, B., Ghosh, S., Le Traon, Y. 2009. A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ. In: ICST'09. 376-385.

[11] Eaddy, M., et al. 2008. Do Crosscutting Concerns Cause Defects? IEEE Transactions on Software Engineering 34 (4). 497-515.

[12] Ferrari, F., Maldonado, J., and Rashid, A. 2008. Mutation Testing for Aspect-Oriented Programs. In: ICST'08. 52-61.

[13] Figueiredo, E., et al. 2008. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: ICSE'08. 261-270.

[14] Filho, F. C., et al. 2006. Exceptions and Aspects: The Devil is in the Details. In: FSE'06. 152-162.

[15] Filman, R. E., and Friedman, D. 2004. Aspect-Oriented Programming is Quantification and Obliviousness. In: Aspect-Oriented Software Development. Addison-Wesley.

[16] Greenwood, P., et al. 2007. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: ECOOP'07 (LNCS, vol.4609). 176-200.

[17] Griswold, W. G., et al. 2006. Modular Software Design with Crosscutting Interfaces. In: IEEE Software 23(1). 51-60.

[18] Gybels, K., and Brichau, J. 2003. Arranging Language Features for More Robust Pattern-Based Crosscuts. In: AOSD'03. 60-69.

[19] Gyimóthy, T., Ferenc, R., and Siket, I. 2005. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. IEEE Transactions on Software Engineering 31 (10). 897-910.

[20] Hoffman, K., and Eugster, P. 2007. Bridging Java and AspectJ through Explicit Join Points. In: PPPJ'07. 63-72.

[21] Hopkins, W. G. 2003. A New View of Statistics. Sport Science, http://www.sportsci.org/resource/stats (01/09/2009)

[22] iBATIS Data Mapper - http://ibatis.apache.org/ (01/09/2009).

[23] Johnson, R., et al. 2007. Spring - Java/J2EE application framework. Ref, Manual Version 2.0.6, Interface21 Ltd.

[24] Kastner, C., Apel, S., and Batory, D. 2007. A Case Study Implementing Features Using AspectJ. In: SPLC'07. 223-232.

[25] Kiczales, G., et al. 1997. Aspect-Oriented Programming. In: ECOOP'97 (LNCS, vol. 1241). 220-242.

[26] Kiczales, G., and Mezini, M. 2005. Aspect-Oriented Programming and Modular Reasoning. In: ICSE'05. 49-58.

[27] Soares, S., Laureano, E., and Borba, P. 2002. Implementing Distribution and Persistence Aspects with AspectJ. In: OOPSLA'02. 174-190.

[28] Steimann, F. 2006. The Paradoxical Success of Aspect-Oriented Programming. In: OOPSLA'06. 481-497.

[29] Stoerzer, M., and Graf, J. 2005. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In: ICSM'05. 653-656.

[30] Subramanyam, R., and Krishnan, M. S. 2003. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. IEEE Transactions on Software Engineering. 29 (4). 297-310.

[31] Sullivan, K., et al. 2005. Information Hiding Interfaces for Aspect-Oriented Design. In: ESEC/FSE'05. 166-175.

[32] The AspectJ Project. http://www.eclipse.org/aspectj/

[33] Wohlin, C., et al. 2000. Experimentation in Software Engineering - An Introduction. Kluwer Academic Publishers.

[34] Xie, T., and Zhao, J. 2006. A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs. In: AOSD'06. 190-201

[35] Huang, S. S., Smaragdakis, Y. 2006. Easy Language Extension with Meta-AspectJ. In: ICSE'06. 865-868

[36] Zhang, S., and Zhao, J. 2007. On Identifying Bug Patterns in Aspect-Oriented Programs. In: COMPSAC'07. 431-438.

[37] Csallner, C., and Smaragdakis, Y. 2005. Check 'n' crash: combining static checking and testing. In: ICSE'05, 422-431.

[38] Harman, M., et. al. 2009. Automated test data generation for aspect-oriented programs. In: AOSD'09. 185-196.