Analyzing Exception Flows of Aspect-Oriented Programs*

Roberta Coelho[1], Arndt von Staa[1] (PhD supervisor), Awais Rashid[2] (PhD co-supervisor)

[1] Computer Science Department, PUC-Rio, Brazil

[2] Computing Department, Lancaster University, Lancaster, UK

`{roberta,arndt}@inf.puc-rio.br, awais@comp.lancs.ac.uk`

Abstract. *The goal of exception handling mechanisms is to make programs more reliable and robust. However, the integration of exception handling mechanisms with aspect-oriented languages raises unique issues. This paper presents a PhD work whose goal was to investigate such issues. The main contributions of this work were the following: (i) to perform the first exploratory study aiming at assessing the error proneness of AOP mechanisms on exception flows of programs; (ii) the development of SAFE (Static Analysis for the Flow of Exceptions), an exception-flow analysis tool for AspectJ programs; (iii) the identification of a set of bug patterns on the exception handling code of AO systems; and (iii) the definition of an approach that enables the developer to find faults on the exception handling code statically.*

1. Introduction

Among the existing techniques for building robust systems, *exception handling* is a widely used mechanism for structuring the error recovery code. It allows the system to detect errors and respond to them correspondingly through the execution of recovery code encapsulated into *exception handlers*. The importance of exception handling mechanisms is attested by its use on the development of several large scale systems [Romanovsky et al 2001] and the fact that many exception handling specific constructs (e.g., `try-catch` blocks) are embedded in mainstream programming languages such as Ada, Java, and C#.

The past decade has seen the rise of Aspect-Oriented Programming (AOP) [Kiczales 1996] techniques as a means to modularize *crosscutting concerns*, such as persistence, distribution, security, transaction management, volatile business rules, certain design patterns and *exception handling* [Castor Filho et al. 2007] in some situations. Since then AOP has been increasingly used in the development of *large scale systems*, most notably the JBoss Application Server, the IBM Websphere, and SORIAN (a Hospital Information System developed by Siemens). Moreover, a number of industrial-strength aspect-oriented programming languages and frameworks have already been released (e.g., AspectJ, JBoss AOP, and Spring AOP).

AOP provides an abstraction called aspect to modularize *crosscutting concerns*. One of its constructs, called *advice*, defines an additional behavior that can be included at specific points in the *program execution* (e.g., *before* the execution of a method, or

* This thesis can be found at: http://www.inf.puc-rio.br/~roberta/ctd2009

after, or even *before* and *after* at the same time). This new construct has the ability to change the *program control flow* - even preventing the original body of a method to be executed. It is recognized that flexible programming mechanisms (e.g., inheritance and polymorphism [Miller and Anand 1997]) might have negative effects on *exception handling*. Hence, while the invasiveness of aspect composition mechanisms may bring a realm of possibilities to software design, they might render less useful if they make the *exception handling* code error prone. In other words, does the code dedicated to improve the system robustness become *itself* a source of program failures, threatening the system robustness?

Unfortunately, there was no systematic evaluation of the positive and negative effects of AOP on the robustness of exception handling code. Existing research in the literature has been limited to analyze the impact of aspects on the *normal control flow* [Rashid and Chitchyan 2003, Hannemann and Kiczales 2002, Garcia et al. 2005]. In addition, most of the empirical studies of AOP do not go beyond the discussion of modularity gains and pitfalls obtained when using aspects to modularize crosscutting concerns [Soares et al. 2006, Greenwood et al. 2007, Figueiredo et al 2008]. For instance, these studies do not account for the consequences bearing with new *exceptions* and *handlers* that come along with the aspects' added functionality. Thus, any practitioner considering using AOP in a project, or any researcher on aspect oriented software development must consider the fundamental questions:

- What is the effect of aspects on the exceptional flow of programs?
- What pitfalls should be avoided when implementing the exception handling code of AO programs?
- How can one know the *exceptions* arising as a result of aspectual compositions?
- And finally, how can one detect and fix violations on the exception handling code of AO programs?

These research questions motivated the PhD work presented in this paper which is organised as follows. Section 2 presents the main contributions of this work. Section 3 details each contribution. Section 4 presents works that are directly related to our own. Finally, Section 5 presents our concluding remarks. Due to space limitation, we assume throughout this article that the reader is familiar with AOP terminology (i.e., aspect, join point, pointcut, and advice) and the AspectJ language.

2. PhD Main Contributions

This PhD work aimed at providing answers to the relevant research questions presented above. Hence, the main contributions of the PhD work are as follows:

(i) it performs the first systematic study which aims at investigating how aspects affect on the exception flows of programs [Coelho et al 2008a].

(ii) it introduces a set of bug-patterns related to the exception handling code of AO programs that were characterised based on the data empirically collected [Coelho et al 2008b].

(iii) it implements an *exception flow analysis tool* for AspectJ programs. This tool performs a *static analysis* of the woven *bytecode* of AspectJ programs in order to find the exceptions that may flow from aspects (*checked* and *unchecked*) and how such exceptions flow on the base code (i.e., *exception paths*) [Coelho et al 2008c, Coelho et al 2009].

(iv) it proposes an approach based on static analysis whose goal is to help developers: (i) to understand the exceptional flow of AO programs, (ii) to describe exception handling contracts, and (iii) to automatically check them [Coelho et al. 2008c, Coelho et al. 2009].

The contributions of this PhD work have a broader impact on the *current research on AOP*, and the on how faults on the *exception handling* code can be detected. They allow for: the designers of AO languages to consider pushing the boundaries to make AOP more robust and resilient to exception handling faults; the ones proposing *AO refactoring methods* to consider the exceptional flow when suggesting the refactoring of specific concerns to aspects; the AOP developers of robust aspect-oriented applications to make more informed decisions in the presence of exception flows. Last but not least, the idea of statically checking the exception handling code through the definition and automatic check of *exception handling contracts* can be applied to different programming languages besides AspectJ and Java (as presented here).

3. How were the Research Questions Tackled?

3.1. Statically Discovering the Exception Flows of Programs

In order to detect the *exceptions* arising as a result of aspectual compositions we developed a static analysis tool called SAFE (*Static Analysis for the Flow of Exceptions*). Such a tool is needed because the analysis of the *flow of exceptions* inside a system can easily become unfeasible if performed manually [Robillard and Murphy 2003]. In order to discover which exceptions can be thrown by a method (or method like construct such as an advice), due to the use of *unchecked exceptions*, the developer needs to recursively analyse each method call. Moreover, when method calls are part of a library, the developer needs to rely on the library documentation which most often is neither precise nor complete [Thomas 2002, Cabral and Marques 2007].

SAFE is an *interprocedural exception flow analysis* tool for Java and AspectJ programs[1]. The *exception flow analysis* performed by SAFE is a dataflow analysis, resembling the analysis based on *def-use* pairs, but instead of running on the control flow graphs, it runs on the program call graph. SAFE runs on an extension of the program call graph (PCG) whose nodes contain additional information comprising the statements where exceptions may be thrown (`throw` clause) or handled (by an enclosing `try-catch` block). The *exception-flow analysis* tool traverses such program representation and calculates (i) the actual exceptions that might arise at every method and advice and (ii) the *exception paths* of each exception (i.e., the path in the program call graph that links the exception signaler and its handler). The SAFE tool is based on Soot framework for static analysis of *bytecode*. It uses the field-sensitive, flow-

[1] The existing exception flow analysis tools for Java programs [Robillard and Murphy 2003, Fu and Ryder 2007] do not support AspectJ constructs. Even the tool that operates on Java bytecode level [Fu and Ryder 2007] cannot be used in a straightforward fashion, since it cannot interpret the effects on bytecode after AspectJ weaving process.

insensitive and context-insensitive points-to analysis provided by Soot to build the PCG.We used this tool on a collection of Java and AspectJ applications (see Section 3.2), to: (i) support both local and global reasoning of the exception-handling structure of a system; (ii) automatically check the *exception handling contracts* (See Section 3.4); and to (iii) discover the new dependencies that may arise between aspects and classes on exceptional scenarios (e.g., an advice may throw an exception that is handled by a method in a class).

3.2. Assessing the Impact of Aspects on Exception Flows: The Exploratory Study

In this PhD work we performed the first systematic study that quantitatively assessed the error proneness of aspect composition mechanisms on the *exceptional flow of programs*. The evaluation was based on the *SAFE tool*, and on code inspections of the exception behaviors of Java and AspectJ implementations of three real applications: Health Watcher (HW), Mobile Photo (MP) and JHotDraw (HD). Overall, this corresponded to 10 system releases (two releases of HW and MP were evaluated), 41.1 KLOC of Java source code of which approximately 4.1 KLOC were dedicated to exception handling, and 39 KLOC lines of AspectJ source code of which around 3.2 KLOC were dedicated to exception handling. In this study we analyzed both OO and AO versions of the same systems because we wanted to check whether AO constructs (AspectJ) increase/decrease the number of defects in the exception handling code.

This study was performed according to the process defined by [Wohlin et al. 2000]. The hypotheses of the study were the following: (i) **the null hypothesis (H0)** was that there was no difference on the robustness of exception handling code in Java and AspectJ versions of the same system; and (ii) **the alternative hypothesis (H1)** was that the impact of aspects on exception flows of programs can lead to more program flaws associated with the exception flow. Figure 1 presents some of the study results.

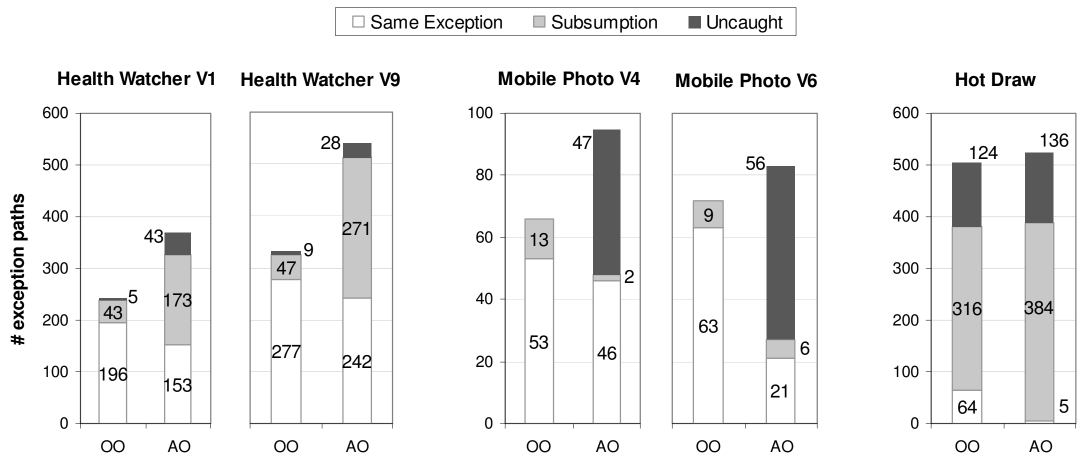


Figure 1. Uncaught exceptions, subsumptions, and specialized handlers in AO and OO versions.

Figure 1 illustrates the number of *exception paths* found in each target system. We can observe a significant increase in the number of *uncaught* exceptions in every AO implementation. This means that the number of exceptions thrown inside the system, which will not be handled and will transparently propagate back to the program entry point (causing the Java virtual machine to terminate), will be higher in the AO implementations of each target system. An additional observation was a decrease in the number of exceptions caught by specialized handlers (i.e., exceptions caught by handlers whose type is the same of the exception type being caught) and a corresponding increase in the number of *exception subsumptions* (i.e., exceptions caught by handlers whose type is a super-type of the exception type being caught). The *exception subsumption* may represent an instance of the *Unintended Handler Action* problem - a scenario in which the exception is mistakenly caught by a handler in the base code (e.g. a `catch Exception` or a `catch Throwable` clause). In order to get a more fine-grained view of how exceptions were handled we manually inspected the exception handling code of every target system. The code inspection revealed a set of *recurring program anomalies* in the exception handling code of AspectJ programs which caused exceptions to become *uncaught* or to be caught by "wrong" handlers.

3.3. Identifying Exception Handling Bug Patterns in AO Programs

We organized the *recurring program anomalies* found during the study into a catalogue of *bug patterns* (presented at [Coelho et al. 2008b]) related to the exception handling code of AO systems. These bugs came mainly from three sources: (i) bugs related to aspects that signal exceptions - such as the *Solo Aspect Signaler* bug pattern – which characterizes a scenario on which no handler is defined for the exceptions raised by the aspects; (ii) bugs related to *exception handling aspects* (i.e., aspects defined to handle exceptions [Castor Filho et al. 2007]) - such as the *Late Binding Aspect Handler* a bug that occurs when the aspect intercepts a point in the base code where the exception had already been caught by another handler; and (iii) misuses of the AspectJ specific construct that wraps any checked exception into an unchecked exception (i.e, `declare soft`). We have observed that these bug patterns have the potential of negatively affecting the robustness of a system. Therefore, there is a need for both: building verification tools and techniques tailored to improve the reliability of the exception handling code in aspect-oriented programs and improving the design of exception handling mechanisms in AO programming languages. Next section describes our effort to address this first need.

3.4. Statically Detecting Exception Handling Faults

Based on the *fault model* discovered during the empirical study, we developed an approach that is capable of *statically detecting violations in the exception handling code* of AspectJ systems. We wanted to detect exception handling faults *statically*, instead of through *testing* as is usual for *fault models*, due to three main reasons. First, the early detection and prevention of faults is less costly [Boehm 1981, Bruntink and Deursen 2006]. Secondly, testing exception handling is inherently difficult – as the root causes that invoke the exception handling code are often difficult to simulate. Last but not least, the large number of different exceptions that can happen at runtime may lead to a *test-case explosion* problem.

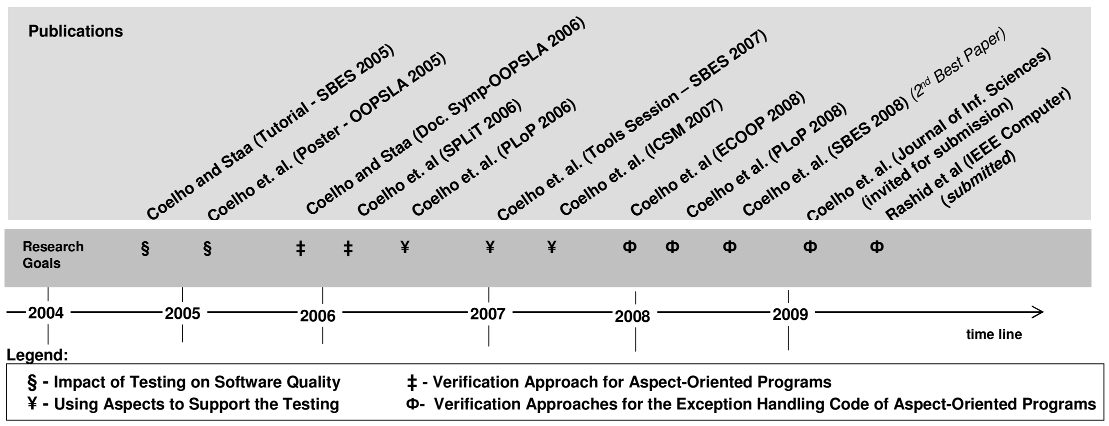
The proposed verification approach is based on two main steps (i) the definition and (ii) automatic checking of *exception handling contracts*. According to it the developer specifies the *exception handling contracts* of the system (i.e., the elements responsible for handling specific exceptions thrown inside the system). Such *exception handling contracts* are then automatically checked by the SAFE tool during the exception flow analysis. As a result the developer can discover *statically* (at *compile time*) which exceptions remain *uncaught* or are caught by the "wrong" handler at *runtime* - without the need of creating test cases to exercise every exceptional situation. In order to evaluate the usefulness of this approach to detect violations on the exception handling code, we applied it on the three real-world applications developed in both Java and AspectJ presented in Section 3.2. Our empirical results show that our approach mined 2.901 *exception-paths*. 886 *exception-paths* mined from the Java applications, from which 138 represented occurrences of *uncaught exceptions*. 2.015 *exception-paths* mined from the AspectJ systems, from which it detected 310 occurrences of *uncaught exceptions*, 44 occurrences of exceptions thrown by classes that were mistakenly caught by handlers defined on aspects, and 345 occurrences of exceptions thrown by aspects that were mistakenly caught by handlers defined on classes [Coelho 2008d].

4. Related Work

Works have been proposed aiming at characterizing the new kinds of faults in AO programs [Ceccato et al. 2005, Bækken and Alexander 2006, Zhang and Zhao 2007]; however, such works neither tackled the potential problems related to the exception handling code (shown in Section 3.3), nor conducted observational studies to provide evidences of the proposed bug patterns (shown in Section 3.2). Moreover, the techniques and tools proposed so far to check the reliability of aspect-oriented code mainly focus on: test-input generation [Xie and Zhao 2006]; definition of test criteria [Lemos et al. 2007]; test selection approaches [Xie et al. 2006]; and mutation testing [Anbalagan and Xie 2008]. Such techniques have been limited to analyze the impact of aspects on the *normal control of programs;* and as a consequence do not propose ways of checking the reliability of the exception-handling code of AO systems (shown in Section 3.4).

5. Concluding Remarks

In this work we have tackled an issue that was neglected by previous research work and empirical studies on Aspect Oriented Programming: the impact of aspects on the exception flow of programs and its consequences on program robustness. As the main results of our work focused on AspectJ programs, one question to be asked is to what extent our findings can be applied to other AOP languages. We have investigated other AOP technologies (i.e., CaesarJ, JBoss AOP and Spring AOP) and have observed that they follow similar *join point models* as the one of AspectJ. Such *join point models* enable aspects to add new exceptions at specific points in the code, potentially causing the bug patterns discovered here. Moreover, further investigations lead to the conclusion that the threats brought by aspects to the robustness of the exception handling code are not caused by specific constructs but to general properties of aspectual compositions (i.e., *inversion of control*, *quantification*, *obliviousness*) which may conflict the characteristics of exception handling. Such findings are described at [Coelho et al. 2008c, Coelho et al. 2009]. Part of this PhD research was conducted in the context of the European Network of Excellence on AOSD (AOSD-Europe), which included a seven-month research visit to Lancaster University. The PhD work summarized here resulted in a significant number of scientific publications, presented in a *time line* below.


Publications
Coelho and Staa (Tutorial - SBES 2005)
Coelho et. al. (Poster - OOPSLA 2005)
Coelho and Staa (Doc. Symp-OOPSLA 2006)
Coelho et. al (SPLiT 2006)
Coelho et. al. (PLoP 2006)
Coelho et. al. (Tools Session – SBES 2007)
Coelho et. al. (ICSM 2007)
Coelho et. al (ECOOP 2008)
Coelho et al. (PLoP 2008)
Coelho et. al. (SBES 2008) (2nd Best Paper)
Coelho et. al. (Journal of Inf. Sciences) (invited for submission)
Rashid et al (IEEE Computer) (submitted)
Research Goals
§ § ‡ ‡ ¥ ¥ ¥ Φ Φ Φ Φ Φ
2004
2005
2006
2007
2008
2009
time line
Legend:
§ - Impact of Testing on Software Quality
¥ - Using Aspects to Support the Testing
‡ - Verification Approach for Aspect-Oriented Programs
Φ- Verification Approaches for the Exception Handling Code of Aspect-Oriented Programs

Additional Publications – *not included on the References Section*
Rashid, A., Meunier, R., Cottenier, T., Greenwood, P., Coelho, R., Chitchyan, R., Suedholt, M., Joosen, W., Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe, IEEE Computer (submitted)
Coelho, R.; Cirilo E., Kulesza, U., Staa, A.v., Rashid, A., Lucena, C., JAT: A Test Automation Framework for Multi-Agent Systems, International Conference on Software Maintenance (ICSM 2007), 2007.
Coelho, R., Cirilo E., Kulesza, U., Staa, A.v., Rashid, A.v., Lucena, C., JAT Framework: Creating JUnit-Style Tests for Multi-Agent Systems, Tools Session – Brazilian Symposium on Software Engineering (SBES 2007), 2007.
Coelho, R., Dantas, A., Kulesza, U., Staa, A.v., Cirne, W., Lucena, C. The Monitor Aspect Pattern, Pattern Languages of Programming Design (PLoP 2006) in conjunction with OOPSLA 2006, 2006.
Coelho, R., Alves, A., Kulesza, U., Costa, A., Staa, A.v., Lucena, C., Borba, P. On Testing Crosscutting Features Using Extension Join Points, 3rd Workshop on Product Line Testing (SPLiT 2006).
Coelho, R., Staa, A.v., Using Interfaces to Support the Testing of Crosscutting Features. In: Doctoral Symposium of Object- Oriented Programming, Systems, Languages and Applications (OOPSLA 2006), 2006.
Coelho, R.; Staa, A.v.; Tutorial: Software Testing, SBES 2005, Uberlândia, 2005.
Coelho, R., Kulesza, U., Staa, A.v., . Improving Architecture Testability with Patterns. In: Object- Oriented Programming, Systems, Languages and Applications (OOPSLA 2005), 2005.

Figure 2. Research works organized in a timeline.

References

Anbalagan. P, Xie, T., (2008) “Automated generation of pointcut mutants for testing pointcuts in AspectJ programs”. ISSRE, 2008, p. 239-248.

Baekken, J.; Alexander, R., “A Candidate Fault Model for AspectJ Pointcuts”. ISSRE 2006, 2006, p.169-178.

Boehm. B. W. (1981), “Software Engineering Economics”. Prentice-Hall, 1981.

Bruntink, M, Deursen, A., Tourwé, T. (2006) “Discovering faults in idiom-based exception handling”. ICSE 2006, p. 242-251.

Cabral, B., Marques, P. (2007) “Exception Handling: A Field Study in Java and .NET”. ECOOP’07. vol. 4609, Springer (2007), p. 151–175.

Castor Filho,F., Garcia, A., Rubira, C. (2007)“Extracting Error Handling to Aspects: A Cook-book”. ICSM’07, 2007, p.134-143.

Ceccato, M., Tonella, P., Ricca, F. (2005) “Is AOP Code Easier or Harder to Test than OOP Code?”. Workshop on Testing Aspect-Oriented Programs, 2005.

Coelho, R, Awais, R., Garcia, A., Ferrari, F. Cacho, N., Kulesza, U., Staa, A.v., Lucena, C. (2008a) “Assessing the Impact of Aspects on Exception Flows: An Exploratory Study”, ECOOP´08, 2008, p. 207-234.

Coelho, R, Awais, R., Staa, A.v., Noble, J., Kulesza, U., Lucena, C., (2008b), “A Catalogue of Bug Patterns for Exception Handling in Aspect-Oriented Programs”, PLoP’08, 2008.

Coelho, R, Kulesza, U., Rashid, A., Staa, A.v., Lucena, C., (2008c), “Unveiling and. Taming the. Liabilities of Aspect Libraries Reuse”, SBES’08, 2008. *(2nd best paper)*

Coelho, R. (2008d), “Analyzing the Exception Flows of Aspect-Oriented Programs”, PhD Thesis, PUC-Rio, June 2008.

Coelho, R., Kulesza, U., Rashid, A., Staa, A.v., Lucena, C. (2009) “Unveiling and Taming Liabilities of Aspects in the Presence of Exceptions: A Static Analysis Based Approach”, Information Sciences Journal (*invited for submission*).

Figueiredo, E., Cacho, N.; Santanna, C., Monteiro, M.,Kulesza, U., Garcia, A., Soares, , Ferrari, F., Khan, S., Filho, F., Dantas, F. (2008) “Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability”. ICSE’08, 2008, p.261-270.

Fu, C., Ryder, B. G. (2007) “Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications”. ICSE’07, 2007, p.230-239.

Garcia, A., Santanna, C.,Figueiredo, E., Kulesza,U.,Lucena, C., Staa, A.v. (2005) “Modularizing Design Patterns with Aspects:A Quantitative Study”. AOSD’05, p.3-14.

Greenwood, P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., Cacho, N., Santanna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A. (2007) “On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study”. ECOOP’07, 2007, p.176-200.

Hannemann, J., Kiczales, G. (2002) “Design Pattern Implementation in Java and AspectJ”. OOPSLA’02, ACM Press, 2002, p.161-173.

Kiczales, G. Aspect-oriented programming. ACM Computing, 28(154), 1996.

Lemos, O., Vincenzi, A., Maldonado, J, Masiero, P. (2007) “Control and data flow structural testing criteria for aspect-oriented programs”. JSS, 80(6), 2007, p.862-882.

Miller, R., Anand, T. (1997) “Issues with Exception Handling in Object-Oriented Systems”. ECOOP’97, 1997, p.85–103.

Rashid, A., Chitchyan, R. (2003) “Persistence as an Aspect”. AOSD’03, 2003,p.120-129.

Robillard, M., Murphy, G., (2003) “Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems”. ACM Trans. on Sof. Eng. Met., 12(2), 2003, p.191-221.

Romanovsky, A., Dony, C., Knudsen, J. L., and Tripathi, A. (2001) “Advances in Exception Handling Techniques”. Springer Verlag, 2001.

Soares, S., Borba, P., Laureano, E. (2006) “Distribution and Persistence as Aspects”. In: Software: Practice and Experience, Wiley, vol. 36 (7), 2006, p. 711-759.

Thomas, D. (2002) “The Deplorable State of Class Libaries”; JOT, 2002, 1(1); p.21-27.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., and Wesslén, A. (2000) “Experimentation in software engineering: an introduction”. Kluwer Academic Publishers, Boston, 2000.

Zhang, S., Zhao, J. (2007) “On Identifying Bug Patterns in Aspect-Oriented Programs”. COMPSAC, 2007, p.431–438.

Xie, T., Zhao, J. (2006) “A framework and tool supports for generating test inputs of AspectJ programs”. AOSD, 2006, p.190–201.

Xie, T., Zhao, J., Marinov, D.; Notkin, D. (2006) “Detecting Redundant Unit Tests for AspectJ Programs”. ISSRE**,** 2006, p.179-190.