Assessing the Impact of Aspects on Exception Flows: An Exploratory Study

Roberta Coelho[1,2], Awais Rashid[2], Alessandro Garcia[2], Fabiano Ferrari[2], Nélio Cacho[2], Uirá Kulesza[3,4], Arndt von Staa[1], Carlos Lucena [1]

[1] Computer Science Department – Pontifical Catholic University of Rio de Janeiro, Brazil
[2] Computing Department, Lancaster University, Lancaster, UK
[3] CITI/DI/FCT - New University of Lisbon, Portugal
[4] Recife Center for Advanced Studies and Systems, Brazil
{roberta, arndt,lucena}@inf.puc-rio.br {marash,garciaa, ferrari.f, n.cacho}@comp.lancs.ac.ak
uira@di.fct.unl.pt

Abstract. Exception handling mechanisms are intended to support the development of robust software. However, the implementation of such mechanisms with aspect-oriented (AO) programming might lead to error-prone scenarios. As aspects extend or replace existing functionality at specific join points in the code execution, aspects' behavior may bring new exceptions, which can flow through the program execution in unexpected ways. This paper presents a systematic study that assesses the error proneness of AOP mechanisms on exception flows of evolving programs. The analysis was based on the object-oriented and the aspect-oriented versions of three medium-sized systems from different application domains. Our findings show that exception handling code in AO systems is error-prone, since all versions analyzed presented an increase in the number of *uncaught* exceptions and exceptions caught by the wrong handler. The causes of such problems are characterized and presented as a catalogue of bug patterns.

Keywords: Exception handling, aspect-oriented programs, static analysis, empirical study, uncaught exceptions, obsolete handler, unintended handler.

1. Introduction

Exception handling mechanisms aim at improving software modularity and system robustness by promoting explicit separation between normal and error handling code. It allows the system to detect errors and respond to them correspondingly, through the execution of recovery code encapsulated into handlers. The importance of exception handling mechanisms is attested by the fact they are realized in many mainstream programming languages, such as Java, C++ and C#.

The goal of Aspect-Oriented Programming (AOP) [41] is to modularize concerns that crosscut the primary decomposition of a system (e.g., functions, classes, components) through a new abstraction called aspect. Aspects use specific constructs to perform invasive modifications of programs [1], and include additional behavior at specific points in the code. AOP is being exploited to improve the modularity of exception handling and other equally-important crosscutting concerns, such as transaction management [31], distribution [31], and certain design patterns [13, 15]. According to some studies [5, 6, 7, 9, 20, 31], AOP has succeeded in improving the modular treatment of several exception handling scenarios. However, it is recognized that flexible programming mechanisms (e.g., inheritance and polymorphism [24]) might have negative effects on exception handling. Hence, while the invasiveness of aspect composition mechanisms may bring a realm of possibilities to software design, often allowing for more stable crosscutting designs [14, 25, 9], they might be useless for practical purposes if they make the exception handling code error prone. Aspectual refinements of base behavior can either improve abnormal behavior robustness or adversely contribute to typical problems of poorly designed error handling code, such as exception subsumption [29] and unintended handler action [24, 29].

Unfortunately, there is 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 [8, 18, 19, 27]. In addition, most of the empirical studies of AOP do not go beyond the discussion of modularity gains and pitfalls obtained when aspects are applied to exception handling [5, 6, 7] and other crosscutting concerns [9, 14, 26, 31]. For instance, these studies do not account for the consequences bearing with new exceptions and handlers that come along with the aspects' added functionality.

This paper reports a first systematic study that quantitatively assesses the error proneness of aspect composition mechanisms on exception flows of programs. The evaluation was based on an exception flow analysis tool (developed in this work) and code inspection of exception behaviors in Java and AspectJ [33] implementations of two industrial software systems – Health Watcher [14, 31] and Mobile Photo [9] – and one open-source project – JHotDraw [16][1]. For the first two systems more than one release was examined. Overall, this corresponds to 10 system releases, 41.1 KLOC of Java source code of which around 4.1 KLOC are dedicated to exception handling, and 39 KLOC lines of AspectJ source code, of which around 3.2 KLOC are dedicated to exception handling. These systems are representatives of different application domains and exhibit heterogeneous exception handling strategies. Some negative outcomes were consistently detected through the analyzed releases using AOP, such as:

- higher evidence of *uncaught exceptions* [17] when aspect advices act as exception *handlers*, thereby leading to unpredictable system crashes [34]; and
- a multitude of *exception subsumptions* [29]*,* some of them leading to *unintended handlers* [24], .i.e, exceptions that are thrown by aspects and unexpectedly caught by existing handlers in the base code;

The causes of such increases were investigated, and are presented in the form of a bug pattern catalogue related to the exception handling code. During this study we implemented an exception flow analysis tool for Java and AspectJ programs, which was very useful when finding and characterizing these bugs. The contributions of this study are as follows:

- It performs the first systematic analysis which aims at investigating how aspects affect the exception flows of programs.
- It introduces a set of bug-patterns related to the exception handling code of AO programs that were characterized based on the data empirically collected.
- It presents an exception flow analysis tool for Java and AspectJ programs, which was developed to support the analysis.

The contributions of this work allow for: (i) developers of robust aspect-oriented applications to make more informed decisions in the presence of evolving exception flows, and (ii) designers of AOP languages and static analysis tools to consider pushing the boundaries of existing mechanisms to make AOP more robust and resilient to changes. The remainder of this paper is organized as follows. Section 2 describes basic concepts

[1] The source code of all systems used in this study is available on the website http://www.inf.puc-rio.br/~roberta/aop_exceptions.

associated with exception handling in AO programs. Section 3 defines the hypotheses and configuration of our exploratory study, the target applications and the evaluation procedures. Section 4 reports our analysis of the empirical data collected in this study. Section 5 presents a bug catalogue for exception handling code in AO systems based on the bug patterns that actually happened in each investigated system, and Section 6 provides further discussions and lessons learned. Section 7 describes the related work. Finally, Section 8 presents our conclusions and directions for future work. Due to space limitations, throughout this article we assume that the reader is familiar with AOP terminology (i.e., aspect, join point, pointcut, and advice) and the syntax of AspectJ's main constructs.

2. Characterizing the Exception Handling Mechanism in AO Programs

In order to support the reasoning about exception flows in AO programs we present the main concepts of an exception-handling mechanism and correlate each element with the constructs available in most AO languages. An exception handling mechanism is comprised of four main concepts: the exception, the exception signaler, the exception handler, and the exception model that defines how signalers and handlers are bound [12].

Exception Raising. An exception is raised by an element - method or method-like construct, e.g., advice - when an abnormal state is detected. In most languages an exception is usually assumed as an error, and represents an abnormal computation state. Whenever an exception is raised inside an element that cannot handle it, it is signaled to the element's caller. The exception signaler is the element that detects the abnormal state and raises the exception. Thus, in AO programs the signaler can be either a method or an advice. In Figure 1, the advice a1 detects an abnormal condition and raises the exception EX. Since this advice intercepts the method mA, the exception EX comes with the additional behavior included into the affected method.

Exception Handling. The exception handler is the code invoked in response to a raised exception. It can be attached to protected regions (e.g. methods, classes and blocks of code) or specific exceptions [16]. Handlers are responsible for performing the recovery actions necessary to bring the system back to a normal state and, whenever this is not possible, to log the exception and abort the system in an expectedly safe way. In AO programs, a handler can be defined in either a method or an advice. Specific types of advice (e.g., around and after [6]) have the ability to handle the exceptions thrown by the methods they advise.

Handler Binding. In many languages, the search for the handler to deal with a raised exception occurs along the dynamic invocation chain. This is claimed to increase software reusability, since the invoker of an operation can handle it in a wider context [16, 24]. In AO programs the handler of one exception can be present: (i) in one of the methods in the dynamic call chain of the signaler; or (ii) in an aspect that advises any of the methods in the signaler's call chain. Figure 1 depicts one scenario in which one advice (a1) signals the EX exception, and the other advice (a2) is responsible for handling EX, i.e. a2 intercepts one of the methods in the dynamic call chain and handles this exception.

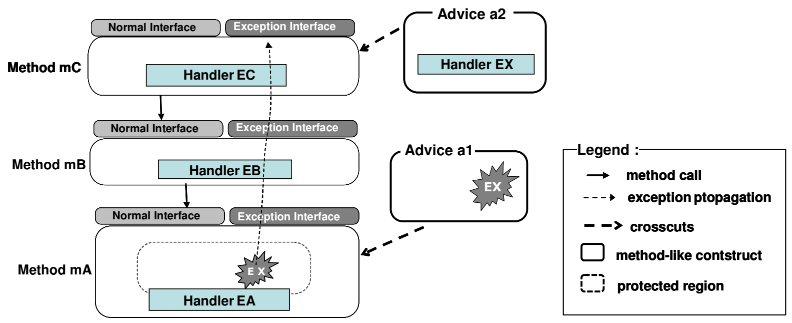

Fig. 1. Exception propagation.

An *exception path* is a path in a program call graph that links the signaler and the handler of an exception. Notice that if there is no a handler for a specific exception, the exception path starts from the signaler and finishes at the program entrance point. In Figure 1, the exception path of EX is <a1→mA→mB→mC→a2>. Therefore, the *exception flow* comprises three main moments: the exception signaling, the exception flow through the elements of a system, and the moment in which the exception is handled or leaves the bounds of the system without being handled, thus becoming an uncaught exception.

Exception Interfaces [24]: The caller of a method needs to know which exceptions may cross the boundary of the called one. In this way, the caller will be able to prepare the code beforehand for the exceptional conditions that may happen during system execution. For this reason, some languages provide constructs to associate to a method's signature a list of exceptions that this method may throw. Besides providing information for the callers of such method, this information can be checked at compile time to verify whether handlers were defined for each specified exception. This list of exceptions is defined by Miller and Tripathi [24] as a method's *exception interface*. Ideally, the exception interface should provide complete and precise information for the method user. However, they are most often neither complete nor precise [4], because languages such as Java provide mechanisms to bypass this mechanism. This is achieved by throwing a specific kind of exception, called *unchecked exception*, which does not require any declaration on the method signature. For convenience, in this paper we split this concept of exception interface into two categories:

(i) the explicit exception interfaces, which are part of the method (or method-like construct) signature and explicitly declare the list of exceptions; and

(ii) the complete (de facto) exception interfaces, which capture all the exceptions signaled by a method, including the implicit (unchecked) ones not specified in the method signatures.

In the rest of this paper, unless it is explicitly mentioned, we use the expression "exception interface" to refer to a complete (de facto) exception interface. Although both the normal interface (i.e. method signature) and the exception interface of a method can evolve along a software life cycle, the impact of such a change on the system varies significantly. When a method signature varies, it affects the system locally, i.e. only the method callers are directly affected. On the other hand, the removal or inclusion of new exceptions in an exception interface may impact the system as a whole, since the exception handlers can be anywhere in the code. As depicted in Figure 1, an aspect can add behavior to a method without changing the normal interface of that method. However, the additional behavior may raise new kinds of exceptions, hence impacting the exceptional interfaces.

Exception Types and Exception Subsumption. Object-oriented languages usually support the classification of exceptions into exception-type hierarchies. The exception interface is therefore composed by the *exception types* that can be thrown by a method. Each handler is associated with an *exception type*, which specifies its handling capabilities - which exceptions it can handle. The representation of exceptions in type hierarchies allows type *subsumption* [29] to occur: when an object of a subtype can be assigned to a variable declared to be of its supertype, the subtype is said to be subsumed in the supertype. When an exception is signaled, it can be subsumed into the type associated to a handler, if the exception type associated to the handler (i.e., *the hander type*) is a supertype of the exception type being caught.

3. Evaluation Procedures

This section describes our study configuration in terms of its goals and hypotheses, the criteria used for the target systems selection (Section 3.1), methodology employed to conduct the exceptional code analyses (Section 3.2), and the actual execution of our study (Section 3.3). The goal of this case study is to evaluate the impact of AOP on exception flows of AspectJ programs, comparing them with their Java counterparts. The investigation relies on determining, in multiple Java and AspectJ versions, which exception-handling bug patterns (Section 5) are typically introduced in their original and subsequent releases. The analyzed error-prone scenarios vary from uncaught to *unintended handler actions*.

The OO and AO versions of three applications have been compared in order to observe the positive and negative effects caused by aspects on their exception flows. Specific procedures were undertaken in order to distinguish AOP liabilities for exception handling implementation from well-known intrinsic impairments of OO mechanisms on exception handling [24]. These procedures were important to detect whether and which AO mechanisms are likely to lead to unexpected and error-prone scenarios involving exception handling. As a result, the null hypothesis (H0) for this study states that there is no difference in robustness of exception handling code in Java and AspectJ versions of the same system. The alternative hypothesis (H1) is that the impact of aspects on exception flows of programs can lead to more program flaws associated with exception flow.

3.1 Target Systems

One major decision that had to be made for our investigation was the selection of the target applications. We have selected three medium-sized systems to which there was a Java version and an AspectJ version available. Each of them is a representative of different application domains and heterogeneous realistic ways of incorporating exception handling into software systems being developed incrementally. The target systems were: Health Watcher [14, 31] (HW), Mobile Photo [9] (MP) and JHotDraw [16, 21] (JHD). The HW system [14, 31] is a Web-based application that allows citizens to register complaints regarding health issues in public institutions. MP is a software product line that manipulates photo, music and video on mobile devices. JHotdraw framework [16] is an open-source project that encompasses a two-dimensional graphics framework for structured drawing editors. It comprises a Java swing and an applet interface. In our study, we focused on the Java Swing version of the JHotdraw. Moreover, such systems exhibit a number of crosscutting concerns in addition to exception handling. Table 1 lists the crosscutting concerns that were implemented as aspects in the AO versions of each system.

Table 1. Target Systems description

System	Description and Crosscutting Concerns
Health Watcher (HW)	*Version 1* : concurrency control, persistence (partially) and exception handling (partially).
	Version 9: concurrency control, transaction management, design patterns (Observer, Factory and Command), persistence (partially) and exception handling (partially).
Mobile Photo (MP)	*Version 4*: exception handling and some functional requirements comprising photo manipulation, such as to sort a list of photos, to choose the favorites, and to copy photo.
	Version 6: exception handling and some functional requirements comprising the manipulation of different kinds of media (i.e., photos and audio files), such as: to sort a list of medias, to choose the favorites, and to copy a media and sending SMS).
AJHotDraw (HD)	*Version 1* : persistence concern, design policies contract enforcement and undo command.

Table 2. Code characteristics per system.

	Health Watcher V1		Health Watcher V9		Mobile Photo V4		Mobile Photo V6		HotDraw	
Number of:	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**
Lines of code	6080	5742	8825	7838	2540	3098	1571	1859	21027	21123
Lines of code for exception handling	1167	854	1889	1242	474	424	356	296	320	341
Classes	88	90	132	129	46	49	30	29	288	279
Aspects	0	11	0	24	0	14	0	10	0	31
`try` blocks	131	118	233	173	49	40	36	24	60	61
`catch` blocks	285	177	481	266	69	60	52	38	67	72
`throw` clauses	227	182	334	229	21	18	20	17	52	56
`try` blocks inside classes	131	108	233	161	49	21	36	9	60	61
`catch` blocks inside classes	285	164	481	252	69	28	52	16	67	72
`throw` clauses inside classes	227	176	334	219	21	4	20	4	52	51
`try` blocks inside aspects	n/a	10	n/a	12	n/a	19	n/a	15	n/a	0
`catch` blocks inside aspects	n/a	13	n/a	14	n/a	32	n/a	22	n/a	0
`throw` clauses inside aspects	n/a	6	n/a	10	n/a	14	n/a	13	n/a	5
`after` advices	n/a	4	n/a	22	n/a	30	n/a	15	n/a	15
`around` advices	n/a	5	n/a	6	n/a	21	n/a	17	n/a	18
`before` advice	n/a	3	n/a	4	n/a	5	n/a	2	n/a	15

Heterogeneous, Non-Trivial Policies for Exception Handling. The target systems were also selected because they met a number of relevant additional criteria for our intended evaluation (Section 3). First, they are non-trivial software projects and particularly rich in the ways exception handling is related to other crosscutting and non-crosscutting concerns. For instance, we could find most of the typical categories of exception handlers in terms of their structure as documented in [7], including nested exception handlers and context-affecting handlers. Second, the behavior of exception handlers also significantly varied in terms of their purpose [4], ranging from error logging to application-specific recovery actions (e.g., rollback). Third, each of these systems contains a considerable amount of code dedicated to exception handling within both aspects and classes as detailed in Table 2.

Presence of Different Aspects in Incrementally-Developed Programs. Finally, AOP was applied in different ways through the system releases: (i) aspects were used to extract non-exception-handling concerns in JHotDraw, and all exception handlers are defined in the base code, (ii) aspects were used to modularize various crosscutting concerns in the Mobile Photo product line, including exception handling apart from the original release, and (iii) aspects were used to partially implement error handling in Health Watcher, where other behaviors were also aspectized. Good AOP practices were applied to structure such systems as stated in [9, 14, 31, 21]. Similar to Java releases, all the AspectJ releases were implemented and changed by developers with around three years of experience in AO design and programming. In fact, HW and MP systems have been used in the context of other empirical studies focusing on the assessment and comparison of their Java and AspectJ implementations in terms of modularity and stability [9, 14]. Alignments of Java and AspectJ versions have been undertaken in order to guarantee that both were implementing the same normal and exceptional functionalities.

3.2 Static Analysis of Exception Flow

The analysis of the exception flow can easily become unfeasible if done manually [28, 29]. In order to discover which exceptions can be thrown by a method, due to the use of unchecked exceptions, the developer needs to recursively analyze each method that can be called from such method. Moreover, when libraries are used, the developer needs to rely on their documentation, which is most often neither precise nor complete [4].

Current exception flow analysis tools [10, 11, 28] do not support AOP constructs. Even the tools which operate on Java bytecode level [11] cannot be used in a straightforward fashion. They do not interpret the aspect-related code included on the bytecode after the weaving process of AspectJ. Hence, we developed a static analysis tool to derive exception flow graphs for AspectJ programs and support our investigation on determining flaws associated with exception flows. This tool is based on the Soot framework for bytecode analysis and transformation [32] and is composed of two main modules: the Exception Path Finder and the Exception Path Miner. Both components are described next, and more detailed information can be found at the companion website [3].

Exception Path Finder. This component uses Spark, one of the call graph builders provided by Soot. Spark is a field-sensitive, flow-insensitive and context-insensitive points-to analysis [32], also used by other static analysis tools [10, 11]. The Exception Path Finder generates the exception paths for all checked and unchecked exceptions, explicitly thrown by the application or implicitly thrown (e.g., via library method) by aspects and classes. It associates each exception path with information regarding its treatment. For instance, whether the exception was uncaught, caught by subsumption or caught by the same exception type. In this study we are assuming that only one exception is thrown at a time – the same assumption considered in [10, 11].

Exception Path Miner. This component classifies each exception path according to its signaler (i.e., class method, aspect advice, intertype or declare soft constructs) and handler. Such classification helps the developer to discover the new dependencies that arise between aspects and classes on exceptional scenarios. For instance, an exception can be thrown by an aspectual module and captured by a class or vice-versa. These different dependencies represent seeds to manual inspections whose goal is to evaluate the error proneness of the abnormal code in AO systems.

3.2.2 Inspection of Exception Handlers

The classification of the handler action for each exception path was based on a complementary manual inspection. It consisted of examining the code of each handler associated with exception paths found by the exception flow analysis tool (Section 3.2.1). Such manual inspections were also targeted at: discovering the causes for uncaught exceptions and exception *subsumptions*. It enabled us to systematically discover *bug hazards* associated with Java and AspectJ modules on the exception handling code. A bug hazard [2] is a circumstance that increases the chance of a bug to be present in the software. For instance, type coercion in C++ is a bug hazard because it depends on complex rules and declarations that may not be visible when working on a class. Each handler action was classified according to one of the categories presented in Table 3.

Table 3. Categories of handler actions and corresponding descriptions.

Category	Description
swallowing	The handler is empty.
logging	Some information related to the exceptional scenario is logged.
customised message	A message describing the failure is presented to the user.
show exception message	The exception message attribute (exception.getMessage()) is presented to the user.
application specific action	An specific action is performed (e.g., rollback).
incorrect user message	A message that is not related to the failure that happened is presented to the user.
new exception	A new exception is created and thrown.
wrap	The original exception or any information associated to it is used to construct a new exception which is thrown.
convert to soft	The exception is converted into a SoftException. This action is specific to AspectJ programs and happens when the delcare soft construct is used.
framework default action	To avoid uncaught exceptions some application frameworks such as java.swing, define catch classes that handle any exception that was not caught by the application and performs a default action (e.g. kill the thread which threw the exception.).
uncaught	No handler caught the exception.

3.3 Study Operation

This study was undertaken from March 2007 to November 2007. During this period target systems were selected and the static analysis tool was implemented and executed for each target system. It was followed by the manual inspection of every exception path. The *Exception Path Finder* was used to generate the exception flow graph for every exception occurrence. Then the *Exception Path Miner* classified each exception path according to its signaler and handler (see Table 4). We discarded a few unchecked exceptions[2] that can be thrown by JVM in almost every program statement execution (e.g., `IllegalMonitorStateException`) and are not normally handled inside the system. The same filter was adopted by Cabral and Marques [4] in an empirical study of exception handling code in object-oriented systems. This filtering was performed on the static analysis. Then we manually inspected each one of the 2.901 exception paths presented in Table 4. The goal of this inspection was threefold: (i) to discover what caused uncaught exceptions and exception subsumptions; (ii) to specify the handler action of each exception path, and (iii) to determine the *bug hazards* associated with AspectJ constructs on certain exception handling scenarios.

4. Analysis of Exception Flows and Handler Actions

This section presents the results for each of the study stages. First, it presents evaluation of the data collected via the exception flow analysis tool (Section 4.1). The following discussion focuses on the information collected during the manual inspections of each exception path (Section 4.2). Our goal in providing such a fine-grained data analysis is to

[2] The discarded exceptions were the exceptions thrown by bytecode operations (NullPointerException, IllegalMonitorStateException, ArrayIndexOutOfBoundsException, ArrayStoreException, NegativeArray SizeException, ClassCastException, ArithmeticException) and exceptions specific to the AspectJ (NoAspectBoundException). Since such exceptions may be thrown by almost every operation, including those could generate too much information which could compromise the usability of the exception analysis.

enable a detailed understanding of how aspects typically affected positively or negatively the robustness of exception handling in each target system and its different releases.

4.1 Empirical Data

Table 4 presents the number of *exception paths* identified by the exception flow analysis tool (Section 3.2.1). It presents the tally of exception paths per target system structured according to a "*Signaler-Handler*" relation. The element responsible for signaling the exception can be either a class or an aspect. When the exception is signaled by an aspect, it is signaled by one of its internal operations: an advice, a method defined as intertype declaration, or a `declare soft` construct[3]. An exception occurrence can be caught in two basic ways. It can be caught by a *specialized handler* when the `catch` argument has the same type of the caught exception type. Alternatively, it can be caught by *subsumption* when the `catch` argument is a supertype of the exception being caught. It is also possible that the exception is not handled by the application and remains *uncaught*. This happens when there is no system's handler defined for the exception type in the exception flow.

Table 4. Classification of exception paths per target system.

	Health Watcher V1		Health Watcher V9		Mobile Photo V4		Mobile Photo V6		HotDraw	
	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO
Signaler: Class										
Uncaught	5	9	9	0	0	0	0	0	124	112
Handler on Class										
Specialized Handler	196	132	277	119	53	26	63	13	64	5
Subsumption	43	26	47	21	13	0	9	0	316	143
Handler on Aspect										
Specialized Handler	n/a	8	n/a	8	n/a	7	n/a	2	n/a	0
Subsumption	n/a	4	n/a	40	n/a	0	n/a	0	n/a	0
Signaler: Aspect										
Construct: Advice										
Uncaught	n/a	2	n/a	27	n/a	5	n/a	16	n/a	0
Handler on Class										
Specialized Handler	n/a	0	n/a	0	n/a	2	n/a	0	n/a	0
Subsumption	n/a	3	n/a	2	n/a	1	n/a	3	n/a	84
Handler on Aspect										
Specialized Handler	n/a	21	n/a	60	n/a	18	n/a	8	n/a	0
Subsumption	n/a	98	n/a	181	n/a	0	n/a	2	n/a	0
Construct: Declare Soft										
Uncaught	n/a	32	n/a	1	n/a	42	n/a	40	n/a	0
Handler on Class										
Specialized Handler	n/a	0	n/a	0	n/a	0	n/a	0	n/a	0
Subsumption	n/a	46	n/a	47	n/a	1	n/a	1	n/a	36
Handler on Aspect										
Specialized Handler	n/a	0	n/a	63	n/a	0	n/a	0	n/a	0
Subsumption	n/a	0	n/a	20	n/a	0	n/a	0	n/a	0
Construct: Intertype										
Uncaught	n/a	0	n/a	0	n/a	0	n/a	0	n/a	24
Handler on Class										
Specialized Handler	n/a	0	n/a	0	n/a	0	n/a	0	n/a	0
Subsumption	n/a	0	n/a	0	n/a	0	n/a	0	n/a	121
Handler on Aspect										
Specialized Handler	n/a	0	n/a	0	n/a	0	n/a	0	n/a	0
Subsumption	n/a	0	n/a	0	n/a	0	n/a	0	n/a	0

The next subsections analyze the *exception paths* presented in Table 4 in detail. First, Section 4.1.1 contrasts the occurrence of *subsumptions* and *uncaught* exceptions in Java

[3] `declare soft` is an AspectJ specific construct. It is associated to a pointcut and wraps any exception thrown on specific join points in a `SoftException`, and re-throws it.

and AspectJ versions of each target system. Section 4.1.2 determines the relation between certain aspect elements (as exception signalers) and higher or lower incidences of *uncaught* exceptions and *subsumptions*. Section 4.1.3 focuses the analysis on how exceptions thrown by aspects are typically treated in the target systems.

4.1.1 The Impact of Aspects on How Exceptions are Handled

A recurring question to AO software programmers is whether it is harmful to aspectize certain behaviors in existing OO decompositions in the presence of exceptional conditions. Hence, our first analysis focused on observing how aspects affected the robustness of the original exception handling policies of the Java versions. Figure 2 illustrates the total number of exception paths on which exceptions (i) remained uncaught exceptions, (ii) were caught by *subsumption*, or (iii) by specialized handlers in each of the target systems.

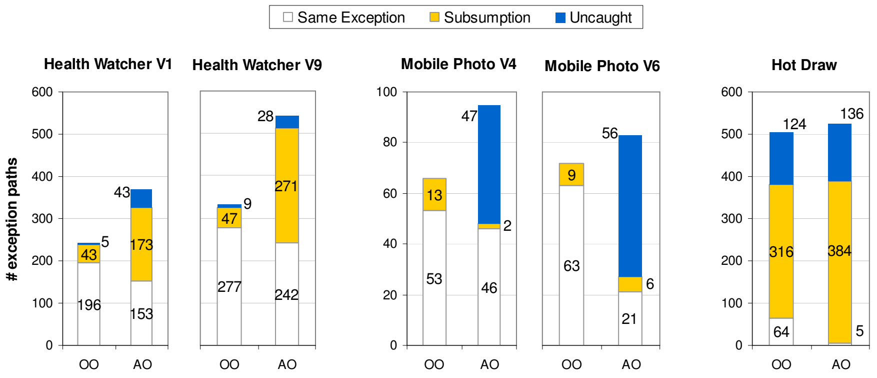

Fig. 2: Uncaught exceptions, *subsumptions*, and specialized handlers per system.

Figure 2 shows a significant increase in the overall number of exception paths. Also significant is the increase in *uncaught exceptions* and *subsumptions* for the AO versions of all the three systems. This increase is a sign that the robustness of exception handling policies in AspectJ releases was affected and sometimes degraded when compared to their Java equivalents. Of course, the absolute number of exception paths is expected to vary due to design modifications, such as aspectual refactorings. However, the number of uncaught exceptions and *subsumptions* ideally should be equivalent between the Java and AspectJ implementations of a same system, since experimental procedures were undertaken to assure that both versions implemented the same functionalities (Section 3).

Figure 3 shows the percentage of occurrence for each category of handler action. We can observe that the relative number of uncaught exceptions also increased in every system, and so did the relative number of *subsumptions*. In some target systems, this increase was significant. In the Mobile Photo V6, for example, the number of uncaught exceptions represent 67.5% of the exceptions signaled on the system. In the Health Watcher V9, the percentage of exceptions caught by *subsumption* increased from 14.1% in OO version to 50.1% in the AO version. This significant increase amplifies the risk of unpredictable system crashes in AspectJ systems, caused by either uncaught exceptions or inappropriate exception handling via *subsumptions*. Correspondingly, there was a decrease in the percentage of exceptions handled by specialized handlers in all AO implementations. When the handler knows exactly which exception is caught, it can take an appropriate recovery action or show a more precise message to the user. However, this was not the typical case in the AO implementations of the investigated systems.

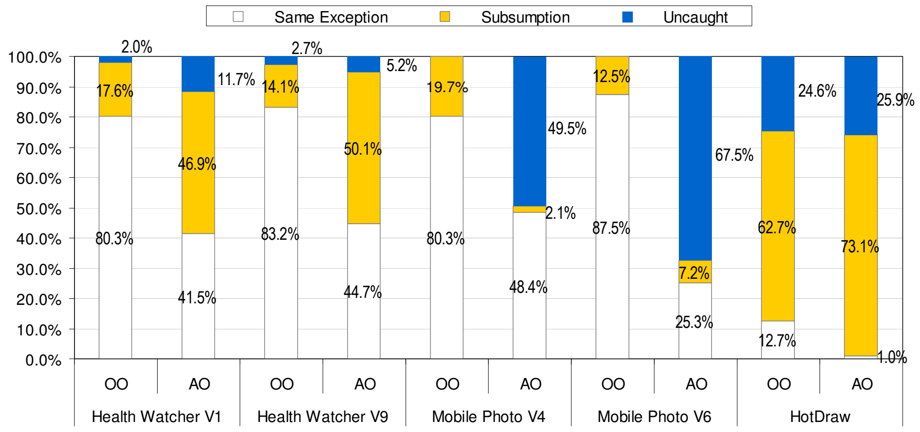

Fig. 3: Percentage of uncaught exceptions, *subsumptions*, and specialized handlers.

4.1.2 The Blame for Uncaught Exceptions and Subsumptions

After discovering that the number of *uncaught* exceptions and *subsumptions* has significantly increased in the AO implementations (Section 4.1.1), we continued our analysis, looking for the main causes of such discrepancies between AO and OO versions. The intuition here is that most of these exceptions were signaled by the aspects in the three target systems. Figure 4 presents charts that confirm this intuition; they show the participation of the exceptions signaled by aspects in the entire number of uncaught exceptions and *subsumptions* per system.

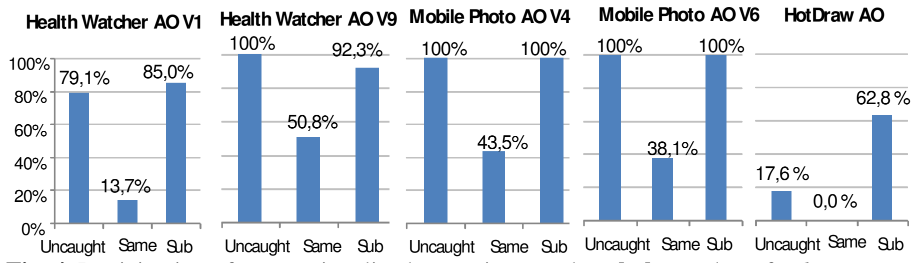

Fig. 4. Participation of aspect-signalized exceptions on the **whole** number of *subsumption, uncaught* and specifically-handled exceptions **per system**.

In both AO versions of Health Watcher and Mobile Photo, the aspects were responsible for signaling most of the uncaught exceptions and those ones caught by *subsumption*. In Mobile Photo V4 and V6, for example, aspects were responsible for 100% of the uncaught exceptions found in this system. This means that no base class in this system signaled an exception that became uncaught. In the AO version of JHotDraw, the aspects were responsible for signaling only 17.6% of the uncaught exceptions, and the aspects participation on the number of exceptions caught by subsumption was high (62.8%). This is explained by the fact that the exception policy of the HotDraw OO was already based on exception subsumption (see Figure 2), thus the exceptions signaled by aspects were handled in the same way.

4.1.3 Are All Exceptions Signaled by Aspects becoming Uncaught or Caught by Subsumption?

Figure 5 gives a more detailed view of what is happening with all exceptions signaled by aspects. We can observe that not all exceptions signaled from aspects become uncaught or are caught by *subsumption*. In HealthWatcher AO V9, for example, only 7% of the exceptions signaled by aspects became uncaught, but they represented 100% of the uncaught exceptions reported to this system (see Figure 3). On the other hand, in the AO versions of the MobilePhoto, the percentage of exceptions signaled by aspects that became uncaught is high (68.1% and 80%). As discussed in the next section, this system was the one that had the exception handling concern aspectized.

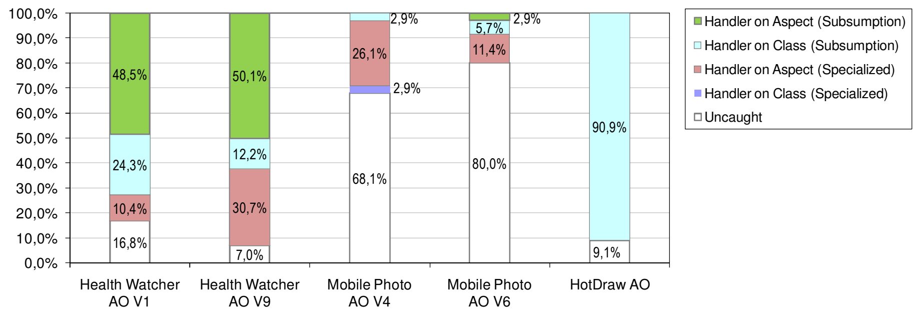

Fig.5. Handler type of exceptions thrown by aspects.

In Figure 5, the exceptions caught by *subsumption* on handlers coded inside classes characterize a *potential fault*. They may represent scenarios in which the exception signaled by an aspect is mistakenly handled by an existing handler in the base code. Another interesting thing to notice in Figure 5 is the increase in the percentage number of exceptions signaled by aspects and handled by specialized handlers from versions 1 to 9 of Health Watcher AO. It illustrates that exceptions signaled by an aspect can be adequately handled.

4.2 Detailed Inspection

In order to obtain a more fine-grained view of how exceptions were handled in AO and OO versions of the same system, we manually inspected all the 2,901 exception paths presented in Table 2. Each exception path was classified according to the action taken on its handler – following the classification presented in Section 3.2.2. Table 5 illustrates the number of each type of handler action per target system and the ratio (%) between the number of the handler action in the AO version and the corresponding value in the OO version of the same system. The ratio is expressed as the quotient of former divided by the latter.

As mentioned before, the total number of exception paths mostly increased in AO versions. During the manual inspections we discovered there were two causes for that: (i) if one exception is not caught inside a specific method (e.g., due to a fault on an aspect that acts as handler) this exception will continue to flow on the call chain, generating new exception paths; and (ii) specific design modifications bring new elements to the call graph and consequently lead to more exception paths. Figure 6 illustrates the handler actions per target system. Overall, it confirms the findings of previous sections based on the tool outputs: the aspects used to implement the crosscutting functionalities tend to violate the exception policies previously adopted in each system. Subsequent subsections elaborate further on the data in Figure 6 and explain the causes behind AspectJ inferiority.

Table 5. Classification of *exception paths* according to their handler action.

	Health Watcher V1			Health Watcher V9			Mobile Photo V4			Mobile Photo V6			HotDraw		
Handler Action	**OO**	**AO**	**Ratio, %**	**OO**	**AO**	**Ratio, %**	**OO**	**AO**	**Ratio, %**	**OO**	**AO**	**Ratio, %**	**OO**	**AO**	**Ratio, %**
swallowing	5	7	140.0	5	7	140.0	0	0	--	0	0	--	3	3	100.0
logging	7	1	14.3	12	10	83.3	14	6	42.9	41	13	31.7	4	11	275.0
customised message	12	43	358.3	20	73	365.0	13	4	30.8	0	0	--	0	0	--
show exception message	43	32	74.4	39	100	256.4	0	0	--	7	1	14.3	291	285	97.9
application specific action	115	121	105.2	169	160	94.7	3	5	166.7	0	0	--	8	0	0.0
incorrect user message	17	53	311.8	16	43	268.8	0	0	--	0	0	--	0	0	--
new exception	3	3	100.0	3	3	100.0	0	3	--	1	2	200.0	0	0	--
wrap	37	38	102.7	60	65	108.3	36	0	--	23	0	0.0	0	0	--
convert to soft	0	40	--	0	100	--	0	37	--	0	13	--	0	8	--
framework default action	0	0	--	0	0	--	0	0	--	0	0	--	74	82	110.8
uncaught	5	43	860.0	9	28	311.1	0	47	--	0	56	--	124	136	109.7
TOTAL	244	381	156.1	333	589	176.9	66	102	154.5	72	85	118.1	504	525	104.2

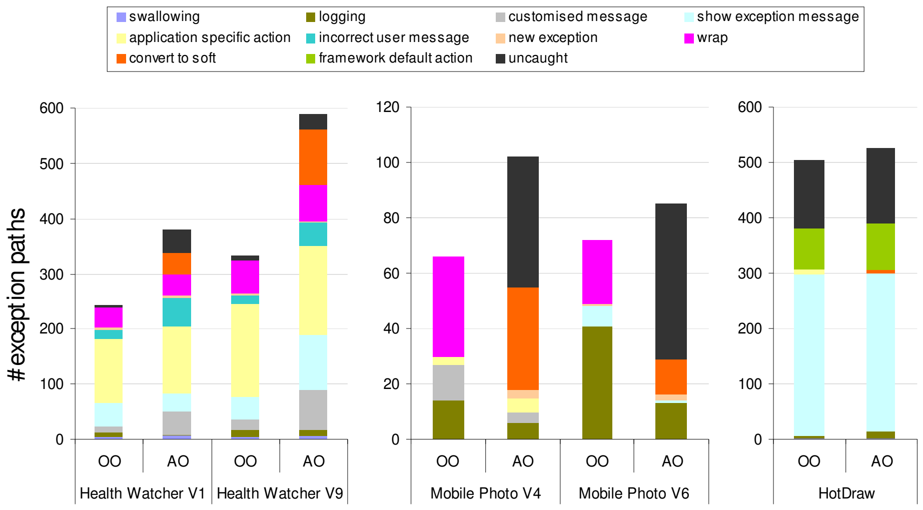

Fig. 6. The handler actions in the *exception paths* of each target system.

4.2.1 Health Watcher

In the AO versions of Health Watcher, there was an increase in the number of exception paths classified as `incorrect user message` (see Table 5), in relation to the corresponding OO versions. It means that there were exception paths in such systems in which a message not related to the exception that really happened was presented to the user. This characterizes the problem known as *Unintended Handler Action,* when an exception is handled by mistake by an existing handler. The causes of such failures were diverse: (i) mistakes on the pointcut expressions of exception handling aspects in both versions; (ii) in version 9, an aspect defined to handle exceptions intercepted a point in the code in which the exception was already caught; (iii) aspects signaled exceptions and no handler was defined for such exceptions in both versions; and (iv) the wrong use of the `declare soft` statement. Each of these causes entails a bug pattern in AspectJ that will be discussed in Section 5. In the version 1, all softened exceptions became uncaught (categories `convert to soft` and `uncaught` respectively), because the `declare soft` statement was not used correctly (see *Handler Mismatch* in Section 5.3). In version 9, the misuse of the `declare soft` statement was fixed but some exceptions remained uncaught or unintended, handled by a `catch` block on the base code that presented an `incorrect user message`.

4.2.2 Mobile Photo

In all AO versions of Mobile Photo there was a significant increase in the number of uncaught exceptions. This application defined many exception handler aspects. Due to mistakes on pointcut expressions and a limitation on the use of `declare soft` many exceptions became uncaught. Differently from the exception handling policy defined in the Health Watcher system, in Mobile Photo there were no "catch all" clauses on the View layer to prevent exceptions - not handled by the handler defined for it - from becoming uncaught.

4.2.3 HotDraw

The target system that presented the lesser impact on the exception policy was the JHotDraw system. The reason is twofold. First, the exception policy in OO version was poorly defined, which is visible thanks to the expressive number of uncaught exceptions and subsumptions (Figure 2). Second, the AO version of the JHotDraw system was built upon a well defined set of refactoring steps [21], and most of the aspects of AJHotDraw were composed by intertype declarations. These refactorings moved specific methods from classes to aspects, such as the methods related to persistence and undo concerns. The `catch` statements for exceptions thrown by the refactored methods were not affected in the AO version, i.e. they remained in the same places on the base code. This explains why most of the exceptions signaled by aspects were caught by base code classes (Figure 5). However, even this system presented potential faults in the exception handling code (Section 5).

5. Characterizing Exception-Handling Bug Patterns in AspectJ

Bug patterns [2] are recurring correlations between signaled errors and underlying bugs in a program. They are related to design anti-patterns, but bug patterns are typical sources of faults at source code level. The manual inspection of the exception handling code related to the exception paths reported by the tool allowed us to identify several exception-handling bug patterns. These patterns can be classified into three categories. First, the use of *aspects as handlers* led to some scenarios in which the `catch` clauses were moved to aspects, the so-called exception handling aspects. However, these aspects did not catch the exceptions they were intended to handle. Second, the application of *aspects as signalers* often implied aspects signaling exceptions for which no handler was defined. Such exceptions flew through the system and became uncaught exceptions or were caught by an existing handler in the code (usually by *subsumption*). Third, the use of `declare soft` construct was often problematic: due to its complex semantics, almost all developers performed similar mistakes when using this construct in almost all the analyzed software releases.

In some cases, we observed that the use of `declare soft` in combination with after throwing advice generated a bytecode in which the after throwing advice were not included, what represents a bug in the AspectJ weaver. Table 6 summarizes the bug pattern distribution in relation to the analyzed systems. The next sections describe the bug patterns shown in this table. For each of them, we provide a description, but due to space constraints, only some examples based on code snippets are provided; code examples for all the bug patterns can be found on the companion website [3].

Table 6. Distribution of the bug patterns per system.

	Health Watcher AO		Mobile Photo AO		HotDraw AO
Bug patterns	**V1**	**V9**	**V4**	**V6**	**V1**
Aspects as Handlers					
Inactive Aspect handler	✓	✓	✓	✓	
Late Binding Aspect Handler		✓			
Obsolete Handler in the Base Code					✓
Aspects as Signalers					
Solo Signaler Aspect	✓	✓			✓
Unstable Exception Interfaces	✓	✓	✓	✓	
Exception Softening					
Handler Mismatch.	✓				
Solo Declare Soft Statement.			✓	✓	
Unchecked Exception Cause					✓
The Precedence Dilemma			✓	✓	

5.1 Advice as Exception Handlers

The role of aspects as handlers can be classified into two: (1) the aspect can handle its own internal exceptions; and (2) and it can handle external exceptions thrown by other aspects or classes. Aspects can be used to modularize the handlers of external exceptions relative to other crosscutting concerns implemented as aspects. The latter occurred in both Health Watcher and Mobile Photo systems. It can also be used to modularize part of exception handling from the base code (as in Mobile Photo). Such exception handling aspects are implemented using `around` and `after throwing` advice. The first two bug patterns presented next are related to aspects that act as external exception handlers, the last one is related to aspects implementing internal handlers.

Inactive Aspect Handler. This kind of fault happens when an Aspect Handler does not handle the exception that it was intended to handle. The cause is a faulty pointcut expression. Such a fault prevents the handler from advising the join point in which an exception should be handled. This exception either becomes *uncaught* (Section 5.2) or is mistakenly caught by an existing handler (*unintended handler action* discussed in Section 5.2). Instances of this bug pattern were detected in Health Watcher and Mobile Photo systems as exception handling was not aspectized in HotDraw. The typical reasons for this bug pattern are the fragility of the pointcut language, usually based on naming conventions, and the number of different and very specific join points to be intercepted by the handler aspects.

Late Binding Aspect Handler. This bug pattern occurred in Health Watcher V9. The concurrency control was implemented within an aspect, which throws the `TransactionException` exception. A specific handler aspect – called `EHAspect` - was defined to handle this exception and although the pointcut expression was correctly specified, the handler intercepted a point in which the exception was already caught beforehand by a "catch all clause" on the base code. This problem is difficult to diagnose because the current IDEs will indicate to the developer of the `EHAspect` that the join points in the code (where the exception should be caught) are correctly intercepted. This explains why this fault remained until version V9 of the HW system. Moreover, even if there is no "catch all" clause between signaler and aspect handler during development, such a clause can be added in a maintenance task. If the handler was defined in the base code and it was a checked exception, the compiler would warn the developer that the handler was inactive. Figure 7 (a) presents a schematic view of this problem. In this figure, the advice a1 adds a new functionality to method mA. This additional functionality comes along with a new exception EX, which flows through the advised method call chain until it is handled. Another advice was defined to handle the exception (advice a2), which intercepts a point on the base code were the exception EX should be handled (method mC). We can observe from this schematic view that the exception EX was caught by a catch clause defined on method mB and, as a consequence, EX could not reach the point in the code where it should be handled by advice a2.

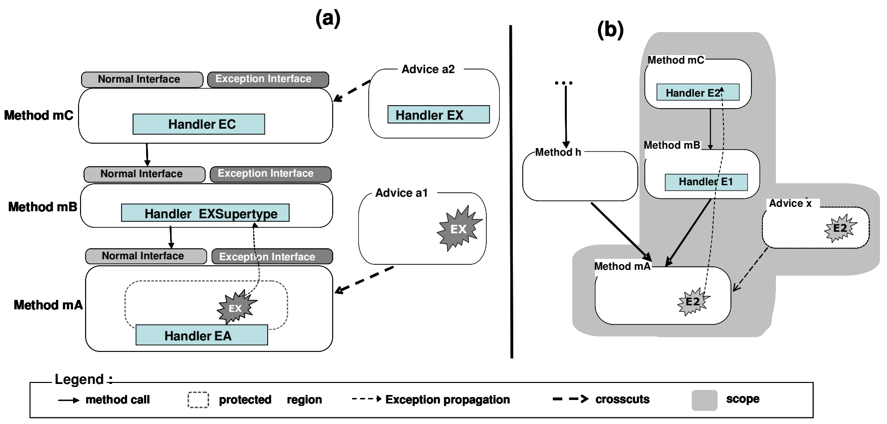

Fig.7. Schematic view of Bug Patterns - (a) Late Binding Handler, and (b) Unstable Exception Interfaces.

Obsolete (or Outdated) Handler in the Base Code. When an aspect handles or softens (Section 3.4.3) an exception previously thrown by an application method, the handler associated with this exception on the base code will become obsolete. The reason is that the exception handled by it can no longer be signaled. In this study, four exceptions handled by aspects generated obsolete handlers. Notice that an obsolete handler may lead to the consequences presented by Miller and Tripathi [24].

5.2 Aspects as Exception Signalers

During the manual inspections we found potential faults that can occur when aspects signal exceptions. They are detailed below.

Solo Signaler Aspect. *Solo Signalers* are the aspects that signal an exception and no handler is bound to it. Such an aspect may lead to the same failures caused by the Inactive Aspect Handler defined in the previous section: an *uncaught exception* or an *Unintended Handler Action*. The *Unintended Handler Action* [24] is usually characterized by the exception signaled by an aspect being handled by *subsumption* via classes.

Unstable Exception Interface. In this study we observed that aspects had the ability of *destabilizing the exception interface* of the advised methods. Every time a static or dynamic scope is used and the advice may signal an exception, the exception interface of the method will vary according to the scope in which a method is called. As a consequence, the same method could raise a different set of exceptions, even when the method arguments were the same, depending on the static (e.g., which class called it) or dynamic (information on the execution stack) scopes. The next code snippet, extracted from the AJHotDraw implementation, exemplifies an *unstable exception interface.*

```
pointcut commandExecuteCheckView(AbstractCommand command): this(command)

        && execution(void AbstractCommand+.execute())
        && !within(*..DrawApplication.*) && !within(*..CTXWindowMenu.*)
         && !within(*..WindowMenu.*) && !within(*..JavaDrawApp.*);

       before(AbstractCommand command) : commandExecuteCheckView(command) {
              if (command.view() == null) {
                throw new JHotDrawRuntimeException("execute should NOT be
                        getting called when view() == null");
              }
       }
```

In this example, the `execute()`method will throw a `JHotDrawRuntimeException` if it is called from a method that is not defined on the classes specified on the pointcut expression (`DrawApplication, CTXWindowMenu, WindowMenu and JavaDrawApp`). As a consequence, the same method will have different behaviors depending on the scope it is called. When the exceptions that can be thrown from a method vary according to the scope it is executed, we say that such method contains an *unstable exception interface*.

Figure 7 (b) presents a schematic view of this problem. In this figure, the *advice x* adds a new functionality to *method mA* only when such method is called from *method mC* (i.e., the pointcut expression contains a dynamic scope delimiter). Therefore, this additional functionality, and the new exception E2 that comes with it, will not be part of method mA when it is called from another method such as *method h.* As a consequence, when the *method mA* is called from *method mC* it may throw E2 exception – and a handler should be defined for it. On the other hand, if it is called from *method h*, it will not throw the exception E2 (even if the method arguments are the same as the one passed on the previous scope) since *advice x* does not affect the *method mA* in this scope.

5.3 Softening Exceptions

In AspectJ an advice can only throw a checked exception if all intercepted methods can signal it (i.e. declaring it on their `throws` clause). In other words, concerning checked exceptions, an advice should follow a rule similar to the "*Exception Conformance*" rule [28] applied during inheritance, when methods are overridden. As a result an advice can only throw a checked exception if it is thrown by every intercepted method. To bypass this restriction, AspectJ offers the `declare soft` statement, which converts (wraps) a given checked exception (in a specific scope) into a specialized unchecked exception, named `SoftException`. The syntax is: `declare soft : <someException> : <scope>`. The `scope` is specified by a pointcut that selects the join points in which the `someException` exception will be wrapped. AspectJ is the only AO language that provides a `declare soft` construct. As detailed in Section 6.4, in Spring AOP and JBoss AOP, advices are allowed to throw any kind of exception, either checked or unchecked. It is possible because their weavers convert the exception interface of every advised method to allow every kind of exception to flow from it – including a `Throwable` in its `throws` clause. This section presents some bug patterns and also potential error-prone scenarios on the exception handling code when the `declare soft` statement is used.

Solo Declare Soft Statement. According to the AspectJ documentation [33], every time an exception is softened by an aspect, the developer should implement another aspect that will be responsible for handling the softened exception. However, this solution is very fragile. It is up to the programmer to define a new aspect to handle the exception that was softened, and no message is shown at compile time to warn the programmer in case s/he forgets to define this aspect handler. In Health Watcher and Mobile Photo, exceptions were softened and no handler was defined for them. This led to uncaught exceptions and *unintended handler actions* - exceptions caught by *subsumption* on the base code.

Unchecked Exception Cause. When a checked exception is softened, it is wrapped in a `SoftException` object. As mentioned before, in Java-like languages the type of an exception is used to make the binding between an exception and its handler. Thus, when wrapping an exception, we are also wrapping useful information in order to provide a fine-grained action for each exception. To overcome this limitation, at every point that needs to handle a softened exception, one should catch the `SoftException` and unwrap it (through its `getCause()` method) in order to compare its cause with every possible exception that may potentially be thrown inside the handler's context. Such "wrapping" solution is documented as one of the exception handling anti-patterns [22].

Handler Mismatch. Some exceptions were softened in one of the Health Watcher versions. However, handlers were defined for the exceptions' primitive types (i.e. types before being wrapped in a `SoftException`). This *Handler Mismatch* implies that almost all exceptions signaled by aspect implementations became uncaught or were caught by unintended handlers. The code snippet bellow, extracted from Health Watcher, illustrates this problem. The `HWTransactionManagement` aspect softens the exception, and the `HWTransactionExceptionHandler` aspect tries to capture the primitive exception (i.e., a `TransactionException` exception). This bug pattern illustrates an emergent property of a particular combination of aspects woven into the base program.

```
public aspect HWTransactionManagement {
    ...
    declare soft: TransactionException:
                  call(void IPersistenceMechanism.beginTransaction())…;
}

public aspect HWTransactionExceptionHandler {
     void around(HttpServletResponse response) :
          execution(* HWServlet+.doGet(HttpServletRequest,
       HttpServletResponse)) && args(.., response) {
        try {  proceed(response); }
        catch (TransactionException e) { ... }
    }
 }
```

The Precedence Dilemma. This problem occurs when an `after throwing` advice is used in combination with the `declare soft` statement for a specific pointcut. Only the code related to the `declare soft` is included in the bytecode. Since both constructions work by converting one exception into another, the weaver cannot decide which one should happen first and as a consequence includes on the bytecode only the code relative to the `declare soft` statement. This bug in the language implementation generates a `SoftException` exception that will not be adequately caught.

6. Discussions and Study Constraints

This section provides further discussion of issues and lessons learned while performing this exploratory study.

6.1 Exception Handling vs. AOP Properties

The goal of exception handling mechanisms is to make programs more reliable and robust. However, we could observe that some properties of AOP may conflict with characteristics of exception mechanisms. In this study we observed that *quantification* and *obliviousness* properties pose specific pitfalls to the design of exception handling code. We explain and discuss these pitfalls in the following.

Quantification Property. Aspects have the ability to perform invasive modifications at specific join points in the program execution where a property holds – an ability also known as the *quantification property* [35]. AspectJ supports quantification via pointcuts and advice. Pointcuts are in general specified in terms of two kinds of pointcut designators: `call` and `execution`. They intercept the call and execution of methods, respectively. On exception-aware systems, such designators may cause different impact in the exceptional interfaces of methods. While the `execution` pointcut affects the exceptional interface of the advised methods themselves, the call advice affects the exceptional interface of the advised method's caller. Such impact can also be influenced by static and dynamic scopes associated with the pointcuts. Static scopes such as `within` and `withincode` delimit the classes or packages on which the aspects will inject a new behavior. Yet, dynamic scope constructs (i.e., `cflow` and `cflowbelow`) allow aspects to affect (or not) a specific point in the code depending on the information available on the runtime execution stack.

The main consequence of the quantification property on exception-aware AO systems was that the *exception interfaces* of methods can vary depending on where the method was called, even when the method arguments were the same. Therefore, the same method of a class could raise a different set of exceptions depending on which object called it or on some information on the execution stack (in case of `cflow` and `cflowbelow`, for example). These *unstable exception interfaces* cannot happen in OO programs since the set of exceptions thrown by a method cannot vary according to the scope where it is executed – provided that the arguments are the same. We observed in our study that in scenarios in which methods presented such *unstable exceptional interfaces*, the exceptions signaled on specific scopes by the advised method often became uncaught or were erroneously handled by an existing handler on the base code (*Unintended Handler Action* bug pattern discussed in Section 5). A possible reason is that it is more difficult for the method's user to prepare the base code to handle the exceptions that will be thrown depending on the dynamic or static scope it is executed.

Obliviousness property. The *obliviousness property* [35], which was believed to be a fundamental property for aspect-oriented programming, states that programmers of the base code do not need to be aware of the aspects which will affect it. It means that programmers do not need to prepare the base code to be affected by the aspects [35]. However, since there are no mechanisms to protect the base code from the exceptions that will flow from aspects, a new exception signaled by the aspect may flow through the system, if no handler is defined for it. This exception may become uncaught and terminate the system in an unpredictable way. Even in cases when a handler aspect is defined for each aspect that can throw an exception (as implemented in the AO versions of Health Watcher), there is no guarantee that the exception thrown by an aspect will be handled by the handler aspect defined to it. Such exceptions may be prematurely caught by a handler on the base code, as illustrated on the bug pattern *Late Binding Aspect Handler* (Section 5.1). Moreover, AspectJ and other existing AO languages allow the invasive modifications caused by aspects to happen dynamically. Although this mechanism opens a new realm of possibilities in software development, it hinders the task of preparing the base code of the exceptions that can be thrown from aspects. During system execution, it is difficult to anticipate whether any unintended handler action or uncaught exception will be caused by the aspects.

6.2 Representativeness

We have investigated other AOP technologies such as: CaesarJ [23], JBoss AOP and Spring AOP. Basically, they follow the same join point model as AspectJ, which allows an aspect to add or modify behavior on join points, potentially adding new exceptions. Table 7 summarizes our analysis regarding exception throwing and handling mechanisms available in such technologies, which was mainly based on available documentation.

Table 7: EH constructs in different AO programming languages.

	declare soft	advice can signal		advice types that act as external handlers				moments of actuation		pointcut scope			
										static		dynamic	
		checked	unchecked	handler-like	after throwing	after-all-like	around	call	execution	within-like	withincode-like	cflow-like	cflowbelow-like
AspectJ	yes	partially	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
CaesarJ	no	partially	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
JBoss AOP	no	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	yes
Spring AOP	no	yes	yes	no	yes	yes	yes	no	yes	yes	no	no	no

According to Table 7, only AspectJ provides a syntactic element to explicitly soften checked exceptions (2nd column). Thus, the bug patterns related to this construct (Section 5.3) are peculiar to AspectJ. Depending on the nature of exceptions that may be thrown by advice, all languages allow advice to throw runtime exceptions (4th column). In AspectJ and CaesarJ, an advice can only throw a checked exception if "every" intercepted method can throw it (declaring it on its throws clause) (3rd column). In CaesarJ, only around advice signature may throw checked exceptions. In Spring AOP and JBoss AOP languages, advice may throw checked exceptions, no matter the exceptions that can be signaled by the advised methods[4]. All languages allow the definition of pointcut scopes (11th to 14th columns), and allow the advice to intercept a method at both calls and executions (9th and 10th columns), consequently facilitating the occurrence of *unstable exceptional interfaces.* Therefore all bug patterns associated with *Advice as Signalers* (Section 5.2) may occur on systems developed in such languages. Finally, all languages allow the definition of aspects that may handle exceptions thrown by another aspect of the base code (5th to 7th columns). As a consequence, all bug patterns associated with *Advice as Handlers* (Section 5.1) can also be found on systems developed in these languages.

[4] It is possible because the exception interface of every advised method is modified to allow any kind of exception to flow from it (`throws Throwable` defines the exception interface of the intercepted methods)

6.3 Study Constraints

The main benefit of an exploratory study such as this one is that it allows the effect of a new programming method to be assessed in realistic situations [42]. One may argue that evaluating the AO and OO versions in a sample of 10 releases for three different systems is a limiting factor. The needed characteristics for the target systems (i.e., medium-sized systems to which there was a Java version and an AspectJ version available) and study based on manual code inspections (a very time-consuming task) restricted the number of subjects evaluated in the study. Given such restrictions, we feel that our set is representative as it includes significant, varied policies and aspectization processes for exception handling (Section 3.1). Another factor that might influence the study results *against* aspectual decompositions could be the developers' expertise on AOP and AspectJ. However, as mentioned before (Section 3.1) all the target systems developers had significant experience in AOP and AspectJ constructs. Moreover, the fact that the AO version of each target system was developed after the OO version, could also impact in the study results, acting *in favor* or *against* AO solutions. However, most AO systems developed so far are derived from an OO version, to which AO *refactorings* [21] are typically applied. Therefore, the threats to validity in this study are not much different than the ones imposed on the other empirical studies with similar goals [6, 9, 13, 14].

6.4 Additional Lessons Learned

AO Refactoring Strategies in Exception-Aware Systems. Many AO systems nowadays are generated from an OO version in which some crosscutting concerns are detected and AO Refactoring techniques are used to convert some crosscutting concerns into aspects. Such AO Refactoring techniques should account for the consequences of aspects on the exception flow of programs. The catalogue of bug patterns presented in this study can be used by such techniques to prevent some avoidable bugs when refactoring a system.

Software Maintainability. Since it is very hard to define at the beginning of a project which exceptions should be dealt with inside the system [30], the exception handling code is often modified along the system development and maintenance tasks. As a consequence, some bugs avoided during AO refactoring, such as the *Late Binding Aspect Handler* (Section 5.1), may be included during a maintenance task - breaking an existing exception handling policy. The exception handling policy comprises a set of design rules that defines the system elements responsible for signaling, handling and re-throwing the exceptions; and the system dependability relies on the conformance to such rules. Reasoning about the exceptional control path, looking for potential-faults on the exception handling code, can quickly become unfeasible if carried out manually [28]. Thus, developers need tools to support them in (i) understanding the impact of aspect weaving on the existing exception handling policy, and (ii) finding bugs on the exceptional handling code along maintenance tasks.

Finding Bugs on Exception Handling Code of AO Programs. Testing exception handling code is inherently difficult [36] due to the huge number of possible exceptional conditions to simulate in a system and the difficulty associated to the simulation of most scenarios. Hence, a valuable strategy for finding faults on the exception handling code can be to *statically* look for them [36]. The exception flow analysis tool developed in our work can detect some failures (e.g., uncaught exceptions), and support the manual inspections whose goal is to find out the cause of the failure (e.g., *bug diagnosing*[5]). Our tool could be extended in order to automatically detect some of the bug patterns described in this work. A similar strategy was adopted by Bruntink et al [36] to find faults on idiom-based exception handling code.

New Interactions between Aspects and Classes. The works presented so far on the interactions between aspects and classes focus on the normal control flow and on information extracted from data-flow analysis. In this study we could observe that new kinds of interaction, between aspects and classes, emerged from the exceptional scenarios (e.g., one class catches one exception thrown by an aspect). Such *Signaler-Handler* relationships between the elements of an AO system can be used as a coupling metric that exists between these elements on exceptional scenarios. We are currently refining the categorization of the *Signaler-Handler* relationships derived from this study.

7. Related Work

Since the effects of AO composition mechanisms on the flow of exceptions on a system are still not well understood, we conducted an empirical study in order to discover these effects and their extent in AO systems. In this section, we present works we believe are directly related to our own, distributed in four categories: (i) static analysis tools; (ii) AOP and exception handling; (iii) experimental studies on exception handling code; and (iv) AO fault models and bug patterns.

Static Analysis Tools: Robillard and Murphy [29] developed a tool called Jex that analyzes the flow of exceptions in Java Programs. Based on java source code this tool performs dataflow analysis in order to find the propagation paths of checked and unchecked exception types. Jo et al. [17] present a set-based static analysis of Java programs that estimates their exception flows. This analysis is used to detect too general or unnecessary exception specifications and handlers. Fu et al. [10] developed a static analysis tool, built upon Soot framework for bytecode analysis, and Spark a call graph builder provided by Soot that generates a call graph of a higher precision compared to the works mentioned previously. This static analysis tool generates the exception paths to every exception thrown on the system. Fu et al. [11] extended their tool in order to compute chains. An exception chain is a combination of semantically-related exception paths. Our tool is similar to the previous one [10], but it works on top of AspectJ code.

AOP and Exception Handling: Lippert and Lopes [20] applied aspect constructs on a large OO framework, called JWAM, to modularize the exception handling code. In their experiment, they obtained a large reduction in the amount of exception handling code present in the application – from 11% of the total code in the OO version to 2.9% in the AO version. Castor Filho et al. [6, 7] performed a similar study but their work reports that the reuse of exception handlers is not straightforward as advocated beforehand by Lippert and Lopes [20]. Instead, it depends on a set of factors such as: the type of exceptions being handled; what the handler does; the amount of contextual information needed; what the method raising the exception returns; and what the throws clause actually specifies. Our study differs from its predecessors since it does not aim at aspectizing exception handling constructs. Actually, we aim at providing a better understanding on how programmers write

[5] The Bug fixing is a less complex problem after the bug was effectively diagnosed.

exception handling code in AspectJ, and identifying possible flaws in the usage of aspects in the presence of exceptional scenarios.

Experimental Studies on Exception Handling Code: Bruno and Cabral [4] performed a quantitative study in which they examined source code samples of 32 different applications, both for Java and .NET. The goal of their study was to identify how exceptions were handled in different categories of systems. They examined the exception handlers and the respective actions taken on them. As a result of this analysis, they observed that the action handlers were very simple (e.g., logging and present a message to the user). However, Bruno and Cabral did not consider the exception paths of each system. As a consequence, they did not take into account the number of *uncaught* exceptions, and the number of exceptions treated by each handler. In our work, we performed an empirical study of how AOP constructs may influence on the way the exceptions are treated on the system.

AO Fault Models and Bug Patterns: Alexander et al. [37] proposed a candidate fault model that includes a set of fault types mostly related to AspectJ features. However, none of them is related to the exceptional scenarios. This fault model was later extended by Cecatto et al. [38], who characterized faults related to “incorrect changes in exceptional control flow.” These faults may occur when an aspect signals an exception which can triggers the execution of a catch statement, either in the aspect itself or in the base program. They also argue that signaled exceptions, when declared as soft, may imply the execution of different branches in the aspectized code. Bækken [39] presents a fine-grained fault model for pointcuts and advice in AspectJ programs. Although Bækken does not describe faults related to exceptional scenarios, he discuss how control and data flows are influenced by exception throwing in order to establish necessary and collectively sufficient conditions for a fault to produce a failure. Ferrari et al. [43] summarized all the previously identified fault types and included three new ones, which were all grouped according to the AO features they are related to. In addition, Ferrari et al. proposed a set of mutation operators to model instances of most of identified fault types, including some related to exception handling code. However, none of these authors detail the consequences of possible faults nor assessed the fault density in the context of real systems. Regarding bug patterns in AO programs, Zhang and Zhao [40] presented a set of general bug patterns for AO programs based on the AspectJ language. The authors stated that a bug pattern is a “*recurring relationship between potential bugs and explicit errors in a program.*” However, the authors did not conduct any observational study that could provide evidences of presence of the proposed bug patterns. The bug patterns we present in this paper are specifically related to exception handling code in AO software and are based on recurring faults found throughout a fine-grained analysis of a set of AO applications.

8. Concluding Remarks

This paper presented a quantitative study to evaluate the impact of aspects on the exception control flow of programs. We selected a set of three systems that were implemented in Java and AspectJ. For two of these systems two different releases were investigated. After that, we compared all versions of the systems in terms of the number of *uncaught* exceptions, exceptions caught by *subsumption*, and exceptions caught with specialized handlers. In all the AspectJ versions, we observed an increase in the number of uncaught exceptions and a decrease in the number of exceptions caught with specialized handlers. Such increase was less significant in AJHotdraw due to the fact that it was built through a well defined set of refactoring steps [21], and most of the aspects are composed by intertype declarations. We performed systematic code inspection of each exception path to find out what caused such negative discrepancies in AspectJ releases. The bug patterns identified came from three sources: aspects acting as *handlers*, aspects as *exception signalers*, and misuses of the `declare soft` construct. This paper also presents a catalogue of bug patterns that characterizes a set of recurring program anomalies found on the exception handling code of AspectJ programs. Our findings indicate that mechanisms of AO languages negatively affect the robustness of exception-aware software systems. As a result, there is a need for both improving the design of exception handling mechanisms in AO programming languages and building static analysis tools and testing techniques tailored to improve the reliability of the error handling code in AO programs. We are currently working on an extension of AspectJ [5] to improve modularity and robustness of exception handling. We are also currently evolving our exception flow analysis tool to support automatic finding of the bug patterns catalogue in this paper.

Acknowledgements: This research was partially sponsored by: CAPES (grants No. 3548-06-6 and 653-07-1);FAPERJ (grant No. E-26/100.061/06);FAPESP (grant No. 05-55403-6);EC Grant AOSD-Europe-European Network of Excellence on AOSD (IST-2-004349);EC Grant AMPLE - Aspect-Oriented, Model-Driven Product Line Engineering (IST-33710);and LatinAOSD/CNPq-Prosul project.

References

1. Aldrich, J.: Open Modules: Modular Reasoning about Advice. In: Proc. of ECOOP’05. LNCS, vol. 3586, Springer (2005) 144-168.
2. Allen, E.: Bug patterns in Java, 2nd edt., Apress (2002)
3. Assessing the Impact of Aspects on Exception Flows: An Empirical Study. Website: http://www.inf.puc-rio.br/~roberta/aop_exceptions
4. Cabral, B., Marques, P.: Exception Handling: A Field Study in Java and .NET. In: Proc. of ECOOP’07. LNCS, vol. 4609, Springer (2007) 151–175
5. Cacho, N.; Castor Filho, F.; Garcia, A.; Figueiredo, E.: EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming. In: Proc. of AOSD’08, (2008).
6. Castor Filho, F., Cacho, N., Figueiredo, E., Maranhão, R., Garcia, A., Rubira, C., Exceptions and Aspects: The Devil is in the Details. In: 13th ACM SIGSOFT (2006)
7. Castor Filho, F., Garcia, A., Rubira, C.: Extracting Error Handling to Aspects: A Cookbook. In: ICSM’07 (2007)
8. Clifton, C.; Leavens, G. T.: Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In: Workshop on Foundations of Aspect Languages (2002).
9. Figueiredo, E.; et al.: Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: Proc. of ICSE’08, (2008).
10. Fu, C.; Milanova, A.; Ryder, B. G.; Wonnacott, D.: Robustness Testing of Java Server Applications. In: IEEE Trans. Software Engineering, vol. 31 (4), (2005) 292-311.
11. Fu, C.; Ryder, B. G.: Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In: ICSE’07, ACM Press, (2007) 230-239.
12. Garcia, A.; et al.: A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. In: Journal of Systems and Software, Elsevier, vol. 59 (6), (2001) 197–222.
13. Garcia, A.; Sant'Anna, C.; Figueiredo, E.; Kulesza, U.; Lucena, C. J. P.; von Staa, A.: Modularizing Design Patterns with Aspects: A Quantitative Study. In: AOSD’05, (2005) 3-14.
14. Greenwood, P.; et al.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: ECOOP’07. LNCS, vol. 4609, Springer (2007) 176–200.
15. Hannemann, J.; Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: OOPSLA’02, ACM Press (2002) 161–173.

16. JHotDraw as Open-Source Project. Online: http://www.jhotdraw.org/, accessed 19/12/2007.
17. Jo, J.; Chang, B.; Yi, K.; Choe, K.: An Uncaught Exception Analysis for Java. In: Journal of Systems and Software, vol. 72 (1), (2004) 59-69.
18. Katz, S.: Aspect Categories and Classes of Temporal Properties. Trans. on Aspect-Oriented Software Development. LNCS, vol. 3380, Springer (2006), 106-134.
19. Krishnamurthi, S.; Fisler, K.; Greenberg, M.: Verifying Aspect Advice Modularly. In: FSE'04, ACM Press (2004) 137-146.
20. Lippert, M.; Lopes, C.: A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In: Proc. of ICSE'00, ACM Press, (2000) 418-427.
21. Marin, M.; Moonen, L.; van Deursen, A.: An Integrated Crosscutting Concern Migration Strategy and its Application to JHotDraw. In: SCAM'07, IEEE Comp. Soc. (2007) 101-110.
22. McCune, T.: Exception Handling Antipatterns. Online: http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html, accessed 19/12/2007, (2006).
23. Mezini, M.; Ostermann, K.: Conquering Aspects with Caesar. . In: AOSD'03, (2003) 90–99.
24. Miller, R.; Tripathi, A.: Issues with Exception Handling in Object-Oriented Systems. In: ECOOP'97. LNCS, vol. 1241, Springer (1997) 85–103.
25. Molesini, A.; Garcia, A.; Chavez, C.; Batista, T.: On the Quantitative Analysis of Architecture Stability in Aspectual Decompositions. In: WICSA'08, (2008).
26. Rashid, A.; Chitchyan, R.: Persistence as an Aspect. In: AOSD'03, (2003) 120-129.
27. Rinard, M.; Salcianu, A.; Bugrara, S.: A Classification System and Analysis for Aspect-Oriented Programs. In: FSE'04, ACM Pres (2004) 147–158.
28. Robillard, M.; Murphy, G.: Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems. In: ACM Trans. Softw. Eng. Methodol., vol.12 (2): (2003) 191-221.
29. Robillard, M.; Murphy. G.: Analyzing Exception Flow in Java Programs. In: ESEC'99, LNCS, vol. 1687, Springer, (1999) 322-337.
30. Robillard, M.; Murphy. G.: Designing Robust Java Programs with Exceptions. In: Proc. of FSE'00. ACM Press (2000) 2-10.
31. Soares, S.; Borba, P.; Laureano, E.: Distribution and Persistence as Aspects. In: Software Practice and Experience, Wiley, vol. 36 (7), (2006) 711-759.
32. The Soot Framework. Online: http://www.sable.mcgill.ca/ soot, accessed 19/12/2007, (2007).
33. The AspectJ Project. Online: http://www.eclipse.org/aspectj/, accessed 19/12/2007 (2007).
34. van Dooren, M.; Steegmans, E.: Combining the Robustness of Checked Exceptions with the Flexibility of Unchecked Exceptions Using Anchored Exception Declarations. In: Proc. of OOPSLA'05, ACM Press (2005) 455–471.
35. Filman, R., Elrad, T., Clarke, S., Aksit, M. Aspect-Oriented Software Development, Addison-Wesley, 2005.
36. Bruntink, M, Deursen, A., Tourwé, T. Discovering faults in idiom-based exception handling. ICSE 2006: 242-251.
37. Alexander, R. T.; Bieman, J. M.; Andrews, A. A.: Towards the Systematic Testing of Aspect-Oriented Programs. Report CS-04-105, Dept. of Computer Science, Colorado State University, Fort Collins/Colorado - USA, 2004.
38. Ceccato, M.; Tonella, P.; Ricca, F.: Is AOP Code Easier or Harder to Test than OOP Code? In: Proc. of WTAOP'05, (2005).
39. Bækken, J. S.: A Fault Model for Pointcuts and Advice in AspectJ Programs. Master's thesis, School of Electrical Engineering and Computer Science, Washington State University, Pullman/WA - USA, (2006).
40. Zhang, S.; Zhao, J.: On Identifying Bug Patterns in Aspect-Oriented Programs. In: Proc. of COMPSAC'07, IEEE Computer Society (2007) 431–438.
41. Kiczales, G.,Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., "Aspect-Oriented Programming", In: ECOOP (1997).
42. Wohlin, C.; Runeson, P.; Host, M.; Ohlsson, M.C.; Regnell, B.; Wesslen, A.Experimentation in Software Engineering - An Introduction. Kluwer, 2000.
43. Ferrari, F. C.; Maldonado, J. C.; Rashid, A.: Mutation Testing for Aspect-Oriented Programs. In: Proc. of ICST'08. IEEE Computer Society (2008).