INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Exceptions and Aspects: The Devil is in the Details

Fernando Castor Filho *Nelio Cacho*
Eduardo Figueiredo *Raquel Marnhao*
Alessandro Garcia *Cecília Mary F. Rubira*

Technical Report - IC-06-08 - Relatório Técnico

May - 2006 - Maio

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

Exceptions and Aspects: The Devil is in the Details

Fernando Castor Filho [1], Nelio Cacho [2], Eduardo Figueiredo [3], Raquel Maranhão [1], and Alessandro Garcia [2] Cecília Mary F. Rubira [1]

[1] Institute of Computing, State University of Campinas, Brazil
{fernando,cmrubira}@ic.unicamp.br, raquelmaranhao@gmail.com
[2] Computing Department, Lancaster University, UK
{ncacho,garciaa}@comp.lancs.ac.uk
[3] Informatics Department, Pontifical Catholic University of Rio de Janeiro, Brazil
emagno@inf.puc-rio.br

Submitted to ACM Symposium on Foundations of Software Engineering, 2006

Abstract. One of the fundamental motivations for employing exception handling in the development of robust applications is to lexically separate error handling code so that it can be independently reused and modified. It is usually assumed that the implementation of exception handling can be better modularized by the use of aspect-oriented programming (AOP). However, the trade-offs involved in using AOP with this goal are not yet well-understood. This paper presents an in-depth investigation of the adequacy of the AspectJ language for modularizing exception handling code. The study consisted in refactoring existing applications so that the code responsible for implementing heterogeneous error handling strategies was moved to separate aspects. We have performed quantitative assessments of four systems - three object-oriented and one aspect-oriented - based on four fundamental quality attributes, namely separation of concerns, coupling, cohesion, and conciseness. Our investigation also included a multi-perspective analysis of the refactored systems, including (i) the reusability of the aspectized error handling code, (ii) the beneficial and harmful aspectization scenarios, and (iii) the scalability of AOP to aspectize exception handling in the presence of other crosscutting concerns.

1 Introduction

Exception handling [12] mechanisms were conceived as a means to improve modularity of programs that have to deal with exceptional situations [4]. Their designs are aimed at promoting an explicit textual separation between normal and abnormal code, in order to support the construction of programs that are more concise, reusable, evolvable, and reliable [4, 12]. Several researchers [6, 17, 25] have explored new programming techniques in order to reap the promised benefits of existing exception handling mechanisms. In spite of this, achieving modular implementations of error handling code is still difficult for software engineers. The main problem is that realistic software systems exhibit very intricate relationships involving the normal-processing code and error recovery concerns. Moreover, exception handling is known to be a global design issue [9, 20] that affects almost all the system modules [17], mostly in an application-specific fashion [1]. It is well-known that exception handling constructs introduce complications in common activities in software development, such as source code analysis and testing [22]. Also, a large part of the system

code is usually devoted to error detection and handling [5, 26], but this part of the code is often the least understood, tested, and documented [5].

Given the broadly-scoped character of exception handling, aspect-oriented programming (AOP) techniques [14] emerge as a natural candidate to promote enhanced modularity, reusability, and conciseness of programs in the presence of exceptions. In fact, it is usually assumed that the exceptional behaviour of a system is a crosscutting concern that can be better modularized by the use of AOP [14, 16, 17]. Some recent research works [15, 17, 23] have investigated the degree to which AOP can improve the separation of concerns relative to some forms of fault tolerance mechanisms.

The most well-known study focusing specifically on exception handling was performed by Lippert and Lopes [17]. The authors had the goal of evaluating if AOP could be used to separate the code responsible for detecting and handling exceptions from the normal application code in a large object-oriented (OO) framework. According to this study,the use of AOP brought several benefits, such as less interference in the program texts and a drastic reduction in the number of lines of code (LOC). However, this first study has not investigated the "aspectization" of application-specific error handling, which is often the case in large-scale software systems. In addition, in spite of the assumption made by many authors that using AOP for separating exceptional code from the normal application code is beneficial, the involved trade-offs are not yet well-understood. For instance, previous investigations have not analyzed whether aspect-oriented (AO) solutions scale well in the presence of complex relationships involving the normal application code and error recovery code. Also, the interaction between exception handling aspects and aspects that implement other concerns still has not been explicitly studied. Hence, some important research questions remain unaddressed:

- Does AOP promote an improvement in well-accepted quality attributes other than separation of concerns, such as coupling cohesion, and size?
- Is exception handling a reusable aspect in real, deployable, software systems?
- When is it beneficial to aspectize exception handling? When it is not?
- How do exception handling aspects affect aspects implementing other concerns?

This paper presents an in-depth study performed to assess the adequacy of AspectJ [16], a general-purpose aspect-oriented extension to Java, for modularizing exception handling code. The study consisted of refactoring four different applications so that the code responsible for handling exceptions was moved to aspects. Three of these applications were originally written in Java and one was implemented in AspectJ. This study differs from the Lippert and Lopes study for a number of reasons. First, the targets of the study are complete, deployable systems, not reusable infrastructures, like a framework. Hence, the exception handling code also implements non-uniform, complex strategies, making it harder to move handlers to aspects. Second, we employ the metrics suite proposed by Sant'Anna and his coleagues [21] to quantitatively assess attributes such as coupling, conciseness, cohesion, and separation of concerns in both the original and the refactored system. Third, we evaluate how exception handling aspects interact with aspects implementing other concerns. Fourth, we assess the overall quality of both the error handling aspects and the application classes affected by them. Fifth, we do not attempt to move error detection code to aspects.

This paper is organized as follows. The next section 2 describes the setting of our study. The results of the study are presented in Section 3. Section 4 analyzes the obtained results and points some constraints on the validity of our study. Section 5 discusses related work. The last section points directions for future work.

2 Study Setting

This section describes the configuration of our study. Section 2.1 briefly explains the procedure we adopted to move exception handling code to aspects. We assume the reader is familiar with the basic mechanisms of the AspectJ language. Section 2.2 describes the targets of our study. Section 2.3 presents the metrics suite we have used to quantitatively evaluate the original and refactored versions of each system.

2.1 Aspectizing Exception Handling

Our study focused on the handling of exceptions. We moved `try-catch`, `try-catch-finally`, and `try-finally` blocks in the four applications to aspects. Hereafter, we refer to these types of blocks collectively as `try-catch`, or handler, blocks, unless otherwise noted. We use the terms `try` block, `catch` block (or handler), and `finally` block (or clean-up action) to explicitly refer to the homonymous parts of a `try-catch` block.

Handlers in the aspects were implemented as after and around advice. Whenever possible, we used after advice, since they are simpler. After advice are not appropriate, though, for implementing handlers that do not raise an exception because they cannot alter the flow of control of a program. In these cases, around advice were employed. Clean-up actions were implemented as after advice. New advice were created on a per-`try`-block basis, excluding cases where handlers could be reused. In situations where multiple `catch` blocks are associated to a single `try` block, we created a single advice that implements all the `catch` blocks. This helps in decreasing the number of advice and, at the same time, avoids problems related to ordering multiple advice associated to the same join point.

For each target system, we employed a different strategy for organizing exception handling aspects. This approach helped us in understanding how different organizations influence handler reuse. Various approaches are possible. Extreme alternatives include putting all the exception handling code in a single aspect or creating several simple aspects that encapsulate the possible handling strategies for each type of exception. More moderate approaches include creating a handler aspect per class that includes exception handling code or one aspect for each package. Each organization has pros and cons that revolve around the code size vs. modularity trade-off.

We used the Extract Fragment to Advice [18] refactoring to move handlers to aspects. In several occasions, it was necessary to modify the implementation of a method in order to expose join points that aspects could select. Usually, this amounted to extracting new methods whose body is entirely contained within a `try` block. After extracting all the handlers to advice, we looked for reuse opportunities and eliminated identical handlers. Figure 1 shows a trivial example of aspectization of handlers using an around advice.

```
class C{
 void m(){
  try {...}
  catch(E e){...}
 }
}
```

```
class C{
 void m(){
  ... // former body of
 }    // the try block
}
aspect A {
 pointcut pcd :
  execution(void C.m());
 void around(): pcd() {
   try{ proceed(); }
   catch(E e) {...}
 } declare soft:E:pcd();
}
```

Fig. 1. An example of aspectization of exception handlers.

Method signatures (`throws` clauses) and the raising of exceptions (`throw` statements) were not taken into account in this study because these elements are related to exception detection. It is usually argued that exception detection is too strongly coupled to the normal application code to be adequately modularized [1].

2.2 Our Case Studies

Four different applications were refactored in our study, three of them OO and one of them AO. Hereafter we call them "target systems". We believe that these applications are representative of how exception handling is typically used to deal with errors in real software development efforts for several reasons. First, these systems were selected mainly because they include a large number of exception handlers that implement diverse exception handling strategies that range from trivial to sophisticated. Second, they encompass different characteristics, diverse domains, and involve the use of distinct real-world software technologies. Finally, they present heterogeneous crosscutting relationships involving the normal code, the handler code, the clean-up actions, and other crosscutting concerns. The rest of this subsection describes the four targets systems of the study.

The original implementation of the three first systems was implemented in Java and, afterwards all the exception handling code was refactored to aspects. Telestrada is a traveler information system being developed for a Brazilian national highway administrator. For our study, we have selected some self-contained packages of one of its subsystems comprising approximately 3350 LOC (excluding comments and blank lines) and more than 200 classes and interfaces. Java Pet Store[1] is a demo for the Java Platform, Enterprise Edition[2] (Java EE). The system uses various technologies based on the Java EE platform and is representative of existing e-commerce applications. Its implementation comprises approximately 17500 LOC and 330 classes and interfaces. The third target system is the CVS Core Plugin, part of the basic distribution of the Eclipse[3] platform. The implementation of the plugin

[1] http://java.sun.com/developer/releases/petstore/
[2] http://java.sun.com/j2ee
[3] http://www.eclipse.org

comprises approximately 170 classes and interfaces and approximately 20000 LOC. It is the target system with the most complicated exception handling scenarios.

Health Watcher was the only system originally implemented in AspectJ. It is a web-based information system that was developed for the healthcare bureau of the city of Recife, Brazil. The original system version involved the aspectization of distribution, persistence, and concurrency control concerns. Furthermore, the system includes some very simple exception handling aspects whose handling strategy consists in printing error messages in the user's web browser. The implementation of Health Watcher comprises 6630 LOC and 134 components (36 aspects and 98 classes and interfaces). The refactoring of Health Watcher consisted in moving exception handling code from classes to aspects. Moreover, we also moved exception handling code from aspects related to other concerns to aspects dedicated exclusively to exception handling.

2.3 Metrics Suite

The quantitative assessment was based on the application of a metrics suite to both the original and refactored version of the target systems. This suite includes metrics for separation of concerns, coupling, cohesion, and size [21] to evaluate both original and refactored implementations and have already been used in various different experimental studies [2, 3, 10, 11]. The coupling, cohesion, and size metrics are extensions of traditional and OO metrics in order to be applied in a paradigm-independent way, and support the generation of comparable results between Java and AspectJ solutions. Additionally, the metrics suite introduces three new metrics for quantifying separation of concerns. They measure the degree to which a single concern (exception handling, in our study) in the system maps to the design components (classes and aspects), operations (methods and advice), and lines of code. For all the employed metrics, a lower value implies a better result. Table 1 presents a brief definition of each metric, and associates them with the attributes measured by each one. Detailed descriptions of the metrics appear elsewhere [21].

3 Study Results

This section presents the results of the measurement process. The data have been collected based on the set of defined metrics (Section 2.3). The presentation is broken in three parts. Section 3.1 presents the results for the separation of concerns metrics. Section 3.2 presents the results for the coupling and cohesion metrics. Section 3.3 presents the results for the size metrics.

We present the results by means of tables that put side-by-side the values of the metrics for the original and refactored versions of each target system. We break the results for the three OO target systems in two parts, in order to make it clear the contribution of classes and aspects to the value of each metric. For the AO application (Health Watcher), we break the results in three parts, in order to make it clear the contribution of classes, exception handling aspects, and aspects related to other concerns to the value of each metric. Hereafter, we use the term "class" to refer to both classes and interfaces. Rows labelled "Diff." indicate the percentual difference between the original and refactored verions of each system, relative

Attributes	Metrics	Definitions
Separation of Concerns	Concern Diffusion over Components	Counts the number of components that contribute to the implementation of a concern and other components which access them.
	Concern Diffusion over Operations	Counts the number of methods and advice which contribute to a concern's implementation plus the number of other methods and advice accessing them.
	Concern Diffusion over LOC	Counts the number of transition points for each concern through the LOC. Transition points are points in the code where there is a "concern switch".
Coupling	Coupling Between Components	Counts the number of components declaring methods or fields that may be called or accessed by other components.
	Depth of Inheritance Tree	Counts how far down in the inheritance hierarchy a class or aspect is declared.
Cohesion	Lack of Cohesion in Operations	Measures the lack of cohesion of a class or an aspect in terms of the amount of method and advice pairs that do not access the same field.
Size	Lines of Code (LOC)	Counts the lines of code.
	Number of Attributes	Counts the number of fields of each class or aspect.
	Number of Operations	Counts the number of methods and advice of each class or aspect.
	Vocabulary Size	Counts the number of components (classes, interfaces, and aspects) of the system.

Table 1. Metrics Suite

to each metric. A positive value means that the original version fared better, whereas a negative value indicates that the refactored version exhibited better results.

3.1 Separation of Concerns Measures

Table 2 shows the obtained results for the three separation of concerns metrics. In general, the refactored versions of the target systems performed better than the original versions. In the refactored versions, all the code related to exception handling that was not automatically generated was moved to aspects. We did not consider automatically generated code because this code does typically not need to be maintained by developers. Among the target systems, only the Java PetStore includes automatically generated code, produced by the Java EE compiler.

Even though the measures of Concern Diffusion over Components diverged strongly amongst the four target systems, it is clear that the refactored solutions fared better. This divergence is a direct consequence of the adopted strategy for creating new handler aspects in each target system. In Telestrada, for complex classes with 7 or more `catch` blocks, we created a new aspect whose sole responsibility is to implement the handlers for that class. Furthermore, each package includes an aspect that modularizes exception handling code for simpler classes. In the Java Pet Store a single exception handling aspect was created per package. In Health Watcher, new exception handling aspects were created for each concern. For the CVS Plugin, the programmer was left free to create aspects as he deemed necessary. Later, we reorganized the aspects in the same way as Java Pet Store (one aspect per package). The results in this section refer to the first organization. It is important to stress, though, that the difference between the two was insignificant. The results for Concern Diffusion over Components in Table 2 reflect these design choices. Telestrada is the system whose refactored version achieved the worst results (a reduction of 18.2%), since a great number of exception handling aspects were created. The refactored Java Pet Store achieved

Application		*Concern Diffusion over Components*		*Concern Diffusion over Operations*		*Concern Diffusion over LOC*	
		Original	Refactored	Original	Refactored	Original	Refactored
Telestrada	*Classes*	22	0	42	0	208	0
	Aspects	-	18	-	44	-	0
	Total	**22**	**18**	**42**	**44**	**208**	**0**
	Diff.	**-18.18%**		**+4.76%**		**-100%**	
Java Pet Store	*Classes*	110	20	256	21	1168	84
	Aspects	-	37	-	179	-	0
	Total	**110**	**57**	**256**	**200**	**1168**	**84**
	Diff.	**-48.18%**		**-21.88%**		**-92.81%**	
Eclipse CVS Core Plugin	*Classes*	59	0	236	0	1118	0
	Aspects	-	4	-	180	-	0
	Total	**59**	**4**	**236**	**180**	**1118**	**0**
	Diff.	**-93.22%**		**-23.73%**		**-100%**	
Health Watcher	*Classes*	35	0	115	0	488	0
	EH Aspects	5	10	9	70	0	0
	Other Aspects	7	0	12	0	48	0
	Total	**47**	**10**	**136**	**70**	**536**	**0**
	Diff.	**-78.72%**		**-48.53%**		**-100%**	

Table 2. Separation of Concerns Metrics.

a middle ground between Telestrada and Health Watcher, with a reduction of 48.18%. In the refactored Health Wathcer, the small number of exception handling aspects (10) represented a reduction of 78.7% in the value of the metric. Finally, for the CVS Plugin, only 4 exception handling aspects were created, resulting in a reduction of 93.22%.

The obtained results for Concern Diffusion over Operations were, in general, better for the refactored versions of the target systems. Only the refactored version of Telestrada exhibited worse results, a 4.76% increase (see Section 4.2). The number of operations containing exception handling code in the refactored versions of Java Pet Store and the CVS Plugin was more than 20% lower than in the corresponding original versions. In Health Watcher, handler reuse was exceptionally high. The refactored version exhibited almost 50% less operations that include exception handling code than the original version. Amongst all the target systems, Health Watcher is the one implementing the simplest exception handing strategies. The majority of the exception handlers either log and ignore the exception or log and rethrow it.

Concern Diffusion over LOC was the metric where the refactored systems performed best, when compared to the original ones. The refactored versions of three out of four target systems did not have any concern switches and thus had value 0 for this metric. The only exception was Java Pet Store, because the automatically generated code was not moved to aspects. In spite of that, the measure for the refactored version was still more than 90% lower. Also, the measures of Concern Diffusion over LOC do not seem to be influenced by the size or characteristics of each target system. This can be seen as an indication that AOP scales up well when it comes to promoting separation of exception handlers in the program texts.

Application		*Coupling between Components*		*Depth of Inheritance Tree*		*Lack of Cohesion in Operations*	
		Original	Refactored	Original	Refactored	Original	Refactored
Telestrada	*Classes*	179	142	186	186	408	524
	Aspects	-	39	-	2	-	0
	Total	**179**	**181**	**186**	**188**	**408**	**524**
	Diff.	**+1.12%**		**+1.08%**		**+28.43%**	
Java Pet Store	*Classes*	783	729	245	245	7095	7595
	Aspects	-	65	-	13	-	71
	Total	**783**	**794**	**245**	**258**	**7095**	**7666**
	Diff.	**+1.4%**		**+5.31%**		**+8.05%**	
Eclipse CVS Core Plugin	*Classes*	1481	1412	181	181	18326	19287
	Aspects	-	77	-	4	-	0
	Total	**1481**	**1489**	**181**	**185**	**18236**	**19287**
	Diff.	**+0.54%**		**+2.21%**		**+5.24%**	
Health Watcher	*Classes*	217	197	69	69	766	867
	EH Aspects	5	27	3	3	4	130
	Other Aspects	66	61	17	17	210	210
	Total	**288**	**285**	**89**	**89**	**980**	**1207**
	Diff.	**-1.04%**		**0%**		**+23.16%**	

Table 3. Coupling and Cohesion Metrics.

3.2 Coupling and Cohesion Measures

Table 3 shows the obtained results for the two coupling metrics, Coupling between Components and Depth of Inheritance Tree, and the cohesion metric, Lack of Cohesion in Operations. On the one hand, aspectizing exception handling did not have a strong effect on the coupling metrics. On the other hand, the measure of Lack of Cohesion in Operations for the refactored target systems was much worse than for the original ones.

The original and refactored versions of the target systems had similar results for Depth of Inheritance Tree. For Health Watcher, the measures of the original and refactored versions were identical. In the refactored version of Telestrada, Depth of the Inheritance Tree increased by slightly more than 1%. The increase of 2 in the value of the metric was due to the creation of a new aspect from which two handler aspects inherit. The abstract aspect was created in order to avoid duplicated code. A similar trend was observed for Java Pet Store and the CVS Plugin. However, in the former, the abstract aspect is inherited by a larger number of concrete aspects (5.3% increase in the refactored version).

Amongst all the metrics we employed, Coupling between Components was the least affected by the aspectization of exception handling. For this metric, none of the systems had a difference greater than 1.5% between the original and refactored versions. New couplings were introduced only when exception handling aspects had to capture contextual information from classes. In these cases, at most one new coupling is created per aspect, due to a reference from the aspect to its corresponding class.

Lack of Cohesion in Operations was the metric for which the refactored target systems presented the worst results. The refactored versions of the four target systems performed worse and, in some cases, substantially. For the refactored versions of Health Watcher and

Application		*Lines of Code*		*Number of Attributes*		*Number of Operations*		*Vocabulary Size*	
		Original	Refac.	Original	Refac.	Original	Refac.	Original	Refac.
Telestrada	*Classes*	3352	2885	127	127	423	437	224	224
	Aspects	-	459	-	0	-	44	-	18
	Total	**3352**	**3334**	**127**	**127**	**423**	**481**	**224**	**242**
	Diff.	**-0.54%**		**0%**		**+13.71%**		**+8.04%**	
Java Pet Store	*Classes*	17482	15593	542	542	2075	2135	339	339
	Aspects	-	2045	-	6	-	180	-	37
	Total	**17482**	**17638**	**542**	**548**	**2075**	**2315**	**339**	**376**
	Diff.	**+0.89%**		**+1.11%**		**+11.57%**		**+10.91%**	
Eclipse CVS Core Plugin	*Classes*	18876	17803	852	854	1832	1848	257	257
	Aspects	-	19423	-	0	-	180	-	4
	Total	**18876**	**19423**	**852**	**854**	**1832**	**2028**	**257**	**261**
	Diff.	**+2.82%%**		**+0.23%%**		**+9.66%**		**+1.43%**	
Health Watcher	*Classes*	5732	4641	152	152	542	553	98	98
	EH Aspects	86	853	3	7	9	73	5	10
	Other Aspects	812	701	12	12	104	104	31	31
	Total	**6630**	**6195**	**167**	**171**	**655**	**730**	**134**	**139**
	Diff.	**-6.56%**		**+2.4%**		**+11.45%**		**+3.73%**	

Table 4. Size Metrics.

Telestrada, the measure of Lack of Cohesion in Operations was more than 20% higher than the corresponding original systems. In the Java Pet Store and the CVS Plugin, the increase was of approximately 8% and 5%, respectively. The main reason for the poor results is the large number of operations that were created to expose join points that AspectJ can capture. These new operations are not part of the implementation of the exception handling concern (and therefore do not affect Concern Diffusion over Operations), but are a direct consequence of using aspects to modularize this concern. Refactoring to expose join points is a common activity in AOP, since current aspect languages do not provide means to precisely capture every join point of interest.

Even though cohesion was worse in the refactored target systems, this was caused mostly by the classes. The value of the cohesion metric for the aspects in the refactored version of Telestrada and the CVS Plugin was 0. In the Java Pet Store, the aspects accounted for less than 1% of the total value of the metric. Only Health Watcher was different. In this system exception handling aspects accounted for 10.8% of the total value. Section 4.1 elaborates more on this.

It is interesting to note that the goal of the Lack of Cohesion in Operations metric is to capture a partial view of cohesion: it considers only the explicit relationships between the attributes and operations (Table 1). It does not consider direct inter-operation relationships and the semantic closeness between elements of a component. The limitation of this metric is somewhat addressed by the separation of concerns metrics. For example, the lower the number of transition points in a component (Table 1), the higher the closeness between the internal members of a class or aspect.

3.3 Size Measures

Contradicting the general intuition that aspects make programs smaller [16, 17, 23] due to reuse, the original and refactored versions of the four target systems had very similar results in two of the four size metrics: LOC and Number of Attributes. The measure of Vocabulary Size grew as expected, due to the introduction of exception handling aspects. Moreover, the Number of Operations of the refactored versions of all the target systems grew significantly. Table 4 summarizes the results for the size metrics.

In Telestrada and Java Pet Store, the number of LOC of the original and refactored versions is similar (less than 1%). In Health Watcher there was a sensible decrease in the amount of exception handling code, even though the influence of this change on the overall number of LOC of the system was only modest (approximately -6.6%). In the CVS Plugin, there was an increase of 2.9% in the number of LOC of the refactored version. Although this is a small percentage of the overall number of LOC of the system, it accounts for almost 550 LOC introduced due to aspectization. The obtained values for LOC were expected. Although some reuse of handler code could be achieved, this was not anywhere near the results obtained by Lippert and Lopes in their study. Moreover, most handlers comprise a few (between 1 and 10) LOC and the use of AspectJ incurs in a slight implementation overhead because it is necessary to specify join points of interest and soften exceptions in order to associate handlers to pieces of code. In the end, the economy in LOC achieved due to handler reuse was more or less compensated by the overhead of using AspectJ.

The general increase in the measures for Vocabulary Size in the refactored systems was entirely due to the aspects. No new classes were introduced or removed. Similarly to Concern Diffusion over Components (Section 3.1), Vocabulary Size depends heavily on how the implementation of the exception handling concern is partitioned among the aspects. Overall, the aspectization of exception handling increased Vocabulary Size by approximately 11% (Java Pet Store) in the worst case and less than 1.55% in the best (CVS Plugin).

The Number of Operations was sensibly higher in the refactored target systems. It grew 13.7% in the refactored version of Telestrada, 11.6% in the Java Pet Store, 10.7% in the CVS Plugin, and approximately %11.5 in Health Watcher. The main reason for this result was the creation of advice implementing handlers. Since there is a one-to-one correspondence between `try` blocks and advice (except for cases where handlers are reused) and handlers do not count as methods in the original systems, this increase was expected. Another reason for the increase in the Number of Operations was the refactoring of methods to expose join points that AspectJ can capture.

4 Discussion

This section makes a qualitative analysis of the obtained results (Section 3) focusing on the research questions posed in Section 1. We also base the analysis on our experience in modularizing exception handling in the four target systems. Furthermore, we discuss the constraints on the validity of our empirical evaluation.

4.1 Coupling, cohesion, and conciseness

Our empirical study confirms some of the findings of the study conducted by Lippert and Lopes [17], who claim that the use of aspects decreases interference between concerns in the program texts. The results achieved by the refactored versions of the target systems in the separation of concerns metrics (Section 3.1) provide convincing evidence for this.

According to Section 3.2, the aspectization of exception handling code does not seem to influence the coupling between the components of a system. Basically, the overall coupling remains the same and the refactored versions have a larger number of less strongly coupled components. However, a closer examination on the code of the four applications reveals a subtle kind of coupling that is not captured by the employed metrics. When exception handling is aspectized using AspectJ, it is sometimes necessary to soften exceptions to suppress the static checks performed by the Java compiler. The issue with exception softening is that it creates an implicit, compile-time dependency of the base code on the exception handling aspect. The dependency is implicit because it can not be inferred just by looking at the base code. Moreover, if the exception handling aspects are not present, the base code will not compile. When aspectizing design patterns, one of the most important benefits obtained is the fact that dependencies are inverted and code implementing design patterns will depend on the participants of the pattern, but not the other way around [13]. This principle does not apply to the aspectization of exception handling with AspectJ because, in many situations, it is not possible to eliminate the dependency of the base code on the aspects. A direct consequence of this is that, in AspectJ, exception handling is not a pluggable aspect, differently from other concerns like distribution [23], assertion checking [17], and some design patterns [13].

As seen in Section 3.3, handler advice accounted for a significant increase in the Number of Operations of all the target systems (+10.4% in Telestrada +8.7% in the Java Pet Store, and +9.8 in the CVS Plugin and Health Watcher). As with all size metrics, this value cannot be evaluated in isolation. Although a developer getting acquainted to the refactored version will have to understand more operations, these operations are smaller and do not mix a system's normal activity with the code that handles exceptions. Therefore, the increase in the Number of Operations caused by the handler advice should not be seen as a negative factor.

Operations extracted in order to expose join points that AspectJ could capture corresponded to 3.3% of the total Number of Operations in Telestrada, 2.9% in the Java Pet Store, 0.79% in the CVS Plugin, and 1.7% in Health Watcher. Unlike the increase caused by handler advice, the increase caused by refactored operations, albeit small, is negative in most situations. These new operations are not part of the original design of the system and possibly do not clearly state the intent of the developer. In some cases, a refactored operation comprises just a few lines that do not make sense when separated from their original contexts.

The increase in Lack of Cohesion in Operations in the refactored versions of Java Pet Store and the CVS Plugin is much lower than the increase in the same metric in Telestrada and Health Watcher. We tried to identify the reasons for this difference by inspecting the collected data for the target systems. We discovered that, in Telestrada, almost 90% of

the increase in the cohesion metric is due to only three classes. These classes have a large number of complex methods, constrasting with the other classes of the system. Since the classes on the system are, in general, very simple (the Number of Operations/Vocabulary Size ratio of the original Telestrada is less than 2), we believe that the large increase in the cohesion metric exhibited by the refactored system was mainly due to the its small size. In Health Watcher, unlike the other target systems, the large increase in the cohesion metric was caused mainly by the exception handling aspects. Three such aspects can be accounted for almost 50% of the increase in the cohesion metric. It is important to stress that this result does not seem to be related to the existence of aspects in the system that implement other concerns.

4.2 Is exception handling a reusable aspect?

In general, we found out that reusing handlers is much more difficult than is usually advertised [16, 17]. This can be noticed by observing the measures for Concern Diffusion over Operations in Section 3.1. In the target system where the highest amount of reuse of exception handling code was achieved, a reduction of 48.5% was observed for this metric. Albeit very positive, this result still contrasts strongly with the findings of Lippert and Lopes, who claim to have achieved a reduction of more than 85% in the number of exception handlers in the target of their study. Moreover, when exception handlers are non-trivial, it may be difficult to fully understand the implications of moving a handler to an aspect. Hence, reusing handlers requires careful design, in order to avoid changing the behavior of the system.

Handler reuse directly depends several factors. In our study, the factors that had the strongest influence on reuse were: (i) the type of exception being handled; (ii) what the handler does and whether it ends its execution by returning, raising an exception, etc.; (iii) the kind of contextual information required, if any; and (iv) what the method that handles the exception returns and what exceptions appear in its `throws` clause. Some of these factors can assist in determining if aspectizing exception handling in a given context is beneficial or harmful (Section 4.3), independently of reuse.

The difficulty of reusing handler code is illustrated by Figure 2. The figure shows three advice that look similar, but can not be merged into a single one because of small differences. Advice #1 and #2 can not be combined because they log different error messages and handle different exceptions. A possible solution to the second problem is to implement a single advice that catches a supertype of both `CVSException` and `IOException`. It is also not possible to combine advice #1 and #3. The former returns a value that depends on the call to `proceed()` (Line 3) while the latter always returns `false` (Lines 17 and 19). For the same reasons, advice #2 and #3 can not be combined.

In our study, the different organizations for exception handling aspects (Section 3.1) produced little or no effect on the overall reuse of exception handlers. From our assessment, in none of the target systems a different arrangement of exception handling aspects could result in substantial improvement in reuse (for example, by moving common advice to abstract aspects).

The value of Concern Diffusion over Operations in the refactored version of Telestrada was almost 5% higher than in the original one (Section 3.1). This happened because, in some

```
// ADVICE #1
boolean around() : ...  {
 try { return proceed();
 } catch (CVSException e) {
   CVSProviderPlugin.log(e); }
 return false;
}
// ADVICE #2
boolean around() : ... {
 try { return proceed();
 }catch (IOException e) {
   CVSProviderPlugin.log(
     IStatus.ERROR, e.getMessage(), e); }
 return false;
}
// ADVICE #3
boolean around() : ... {
 try { proceed();
 } catch (CVSException e){
   CVSProviderPlugin.log(e); }
 return false;
}
```

Fig. 2. Three similar advice that can not be combined.

packages, reuse of handler code was virtually inexistent and some classes had operations with more than one `try-catch` block. Hence, when exception handling code in these classes was moved to aspects, each handler had to be put in a separate advice, contributing to the increase.

4.3 When to aspectize exception handling

From our experience in refactoring the four target systems, we derived a simple classification for exception handling code. This classification emphasizes factors that influence the aspectization of exception handling. It aims at helping developers to identify the situations where moving exception handlers to aspects is beneficial and when it is not worth the effort. We use three categories to classify exceptional code in Java-like languages: (i) placement of `try-catch` blocks; (ii) dependency on local variables; and (iii) flow of control after handler execution. In the rest of this section, we describe these categories and show how they capture many of the situations that developers are likely to find when attempting to aspectize exception handling.

Placement of `try-catch` Blocks. The first category is related to where in the text of a method a handler block appears. This impacts the pointcut designators employed to capture the body of the `try` block and whether refactoring is required in order to expose join points of interest. If all the statements implementing the normal behavior of a method, including variable declarations, appear within a `try` block, we say that the placement of the containing `try-catch` block is *basic*. Aspectizing exception handling for a basic `try-catch` block is a simple matter of refactoring handlers and clean-up actions to advice, softening exceptions as necessary, and defining a pointcut to capture the whole method execution. No additional refactoring is required and the pointcut definition is usually quite simple.

If a `try-catch` block is not basic, it is *tangled*. It is usually much harder to modularize exception handling with aspects for tangled `try-catch` blocks. It is often necessary to perform some a priori refactoring, usually Extract Method [7], that makes the `try-catch` block basic. Furthermore, depending on the context of the tangling, it may be necessary to define complex pointcuts to correctly capture the implementation of the `try` block. If all the statements that appear outside of a top-level (non-nested) tangled `try-catch` block can be moved to its corresponding `try` block without altering the behavior of the parent method, this `try-catch` block is considered basic.

A `try-catch` block can be further classified as *nested* or *top-level*. A nested `try-catch` block is contained within a `try` block, whereas a top-level `try-catch` block is not. A nested `try-catch` block is considered basic if it is the only statement in the `try` block of a basic `try-catch` block.

Dependency on Local Variables. The second category is used to separate exception handlers in two groups: those that do and those that do not depend on local variables. By "local variables" we mean variables defined within the containing context. Dependency on local variables hinders aspectization, as the joint point models of AO languages like AspectJ can not capture information stored by these variables. If a handler reads the value of one or more local variables, moving it to an aspect usually requires the use of the Extract Method refactoring, in order to expose the variables as parameters of a method. If the handler performs assignments to local variables, other refactorings might be necessary, depending on whether the variables are primitive or object types. Although ths discussion focuses on exception handlers, it also applies to clean-up actions.

Flow of Control after Handler Execution. The third category, flow of control after handler execution, is related to how an exception handler ends its execution. After a *termination* exception handler executes its last statement, the statement that textually follows the corresponding `try-catch` block is executed. A *propagation* exception handler finishes its execution by signaling an exception. In this case, system execution resumes when some other handler catches the signaled exception. A *return* exception handler is, in some ways, similar to a *propagation* handler. The difference is that, in the former, system execution resumes from site where the handler's method was called. Finally, a *loop iteration* exception handler occurs when a `try-catch` block is nested within a loop and at least one of its `catch` blocks executes a statement such as `break` or `continue`. Flow of control after handler execution affects several design choices related to the aspectization of exception handling. For example, it is generally straightforward to define a pointcut that selects a tangled, top-level `try-catch` block if its handlers are all propagation or return. However, the same does not apply to termination handlers.

Table 5 shows the number of instances of each category in the four target systems. Using the proposed classification, it is possible to describe several interesting scenarios. These scenarios represent recurring situations with which a developer would have to deal if faced with the task of modularizing exception handling code using aspects. Table 6 enumerates the ones that we encountered the most frequently while conducting our study. Each row represents one or more scenarios. To avoid repetition, if a category is marked more than once in the same row (e.g. "basic" and "tangled" marked), the row represents more than one

Target system	Placement of `try-catch` blocks				Dependency on local vars.		Flow of control after handler execution				Total
	basic	**tangled**	**nested**	**top-level**	**yes**	**no**	**term.**	**prop.**	**ret.**	**loop**	
Telestrada	22	19	4	37	10	31	10	31	0	0	41
Java Pet Store	185	70	16	239	24	231	72	161	22	0	255
CVS Plugin	74	88	23	139	15	147	72	62	27	1	162
Health Watcher	101	34	5	130	30	105	71	64	0	0	135

Table 5. Instances of the proposed categories in the four target systems.

#	Placement of `try-catch` blocks				Dependency on local vars.		Flow of control after handler execution				Should be aspectized?
	basic	**tangled**	**nested**	**top-level**	**yes**	**no**	**term.**	**prop.**	**ret.**	**loop**	
1	□		□	□		□	□	□	□		**Yes**.
2		□		□		□		□	□		**Yes**.
3		□		□		□	□				**Depends**.
4		□	□			□	□		□		**No**.
5		□	□			□		□			**Depends**.
6		□		□	□		□	□	□		**Depends**.
7		□	□		□		□	□	□		**No**.
8		□	□	□	□	□				□	**No**.

Table 6. Some exception handling scenarios according to the proposed classification

scenario and an OR semantics is adopted. For category Placement of `try-catch` Block, we assume that the two subcategories (basic/tangled and nested/top-level) count as different categories. For example, row 1 refers to scenarios where the placement of `try-catch` blocks is basic and either nested or top-level. Also, the corresponding `catch` block does not depend on local variables and can end its execution by terminating, propagating an exception, or returning. For simplicity, the table does not show scenarios that could not happen in practice, for example, a basic `try-catch` block that has a loop iteration handler or that accesses local variables.

The rightmost column of Table 6 indicates whether it is beneficial ("yes") or harmful ("no") to modularize exception handling with aspects in the presented scenarios. In general, we considered aspectization to be beneficial in a given scenario if it has a positive effect on the values of the metrics of Section 2.3, when comparing the original and aspectized code for instances of the scenario. Moreover, in some rows, the rightmost column column indicates that the choice of aspectizing exception handling in a given scenario depends on factors that are not taken into account by the proposed classification. These cases are marked as "depends". We have chosen not to include these factors in the classification because they are very specific and subjective, and to keep it simple. Table 7 justifies the "no" and "depends" scenarios.

#	Reason
3	Aspectization is beneficial in this scenario if: (i) the code within the `try` block can be selected by a pointcut without the need for additional refactoring; or (ii) it is necessary to use Extract Method to expose a joint point that AspectJ can capture but the new method makes sense by itself, i.e., it could have been created by the developers of the system.
4 & 7	In our experience, combinations of tangling and nesting, and nesting and access to local variables usually result in complex code that needs to be refactored before it can be aspectized. In many cases more than one new operation needs to be created, negatively affecting the cohesion and conciseness of the code.
5	If the outer `try-catch` blocks do not have handlers for the exception caught by the innermost handler nor to the exceptions signaled by it, aspectization is beneficial because the advice implementing the handler can be associated to the execution of the whole method.
6	Aspectization is only beneficial if: (i) the handler accesses just a few variables (< 4) and only for reading; and (ii) the refactoring employed to expose these variables creates a method that makes sense by itself.
8	Loop iteration handlers are usually too strongly coupled with the context where they appear.

Table 7. Justification for the "no" and "depends" scenarios of Table 6.

4.4 Exception handling and other aspects

The previous sections have analyzed how AspectJ scaled to support modularization and reuse of diverse forms of exception handling. They consisted of a plethora of combinations involving the normal code and exception handling code. This section analyzes the scalability of AOP when there are interactions between the implementation of exception handling and other crosscutting behaviors. The idea is to examine how easy it is to aspectize such crosscutting concerns in the presence of exception handling aspects.

Our investigation was carried out mainly in the context of the Health Watcher system. However, as discussed in the previous sections and evidenced by the measurements (Section 3), this system has a simple exceptional behavior, when compared to the other three target systems. To obtain a more comprehensive perspective of the possible difficulties caused by interactions between exception handling and other aspects, we have also refactored part of the AspectJ version of the CVS Plugin, where the exception handling concern was already modularized with aspects. We have chosen this system, instead of the other two, because it was the case study that exhibited the most complex exception handling strategies. In the CVS Plugin, we have opted for aspectizing security (access control) and distribution (remote access through HTTP and SOCKS5 proxies) concerns, which would otherwise be naturally tangled and scattered through several classes. We selected these concerns because they are well-known as traditional crosscutting concerns in the literature, and tend to have a broadly-scoped influence in the system. More importantly, we have detected a number of different relationships between exception handling and these crosscutting concerns. Such relationships were not captured in the Health Watcher system, which generally exhibited a loose coupling between the exception handling aspects and aspects implementing other concerns.

We have observed different categories of interactions involving the exception handling code and other crosscutting behaviors. They range from (i) simple invocations linking exceptional behaviors and methods relative to the other concern to (ii) the sharing of one or more module members by two different concerns. The set of interactions analyzed in this

study was classified into 5 categories, which are described in the following. Such categories involve either *class-level interlacing* or *method-level interlacing*. Our categorization is a specialization of interaction categories defined in a previous study where we have analyzed design pattern compositions [2]. Here we refine the previous categorization by also taking exception handling structures into account, namely protected regions (`try {}`), handlers (`catch (E) {}`), and clean-up actions (`finally {}`). To illustrate these categories, we use Figure 3. In the figure, Concern 1 (C1) corresponds to exception handling and Concern 2 (C2) is a second concern. In Health Watcher, C2 can be part of the concurrency control, distribution, or persistence concerns, whereas in the Eclipse CVS plugin it is part of the security or distribution concerns.

Class-level Interlacing. The first category is concerned with class-level interlacing of exceptional behavior and other crosscutting behaviors. In this case, the implementations concerns C1 and C2 have one class in common. However, each concern encompasses disjoint sets of methods and attributes in the same class. As illustrated in the left-hand side of Figure 3, C1 and C2 have a coinciding participant class, but there is no method or attribute pertaining to the two concerns. This interaction category did not bring any kind of problem while aspectizing elements of C2. Hence we can say that AspectJ has scaled up well in scenarios involving class-level interlacing.

Method-level Interlacing. Four categories involve some form of *method-level interlacing*: *unprotected-region level*, *protected-region level*, *handler level*, and *cleanup-action level*. Differently from class-level interlacing, all these categories have a similar characteristic: the implementations of concerns C1 and C2 have one or more methods in common. Hence exception handling code is interlaced at the method level with elements of C2. In the right-hand side of Figure 3, method `x()` has code pertaining both C1 and C2. In most of the situations in Health Watcher and the CVS plugin involved, interaction between concerns C1 and C2 consisted of calls to methods from C2 by code pertaining C1. The distinguishing feature of the four categories of method-level interlacing is where such a call is placed in terms of the exception handling elements.

The right-hand side of Figure 3 depicts all the 4 types of interactions encountered in the two systems. The interaction types influenced the way in which the AspectJ code of the two systems was refactored to expose the appropriate join points to the aspects of C2. The aspectization of crosscutting behaviors relative to C2 was straightforward when the situation exhibited interlacing at the unprotected- or protected-region level. The reason was that there was no explicit link between the exception handling aspects and the C2 code being aspectized. In Health Watcher, all the instances of method-level interlacing fit into one of these two categories. More explicit aspect interactions appear in methods with catch- and finally-level interlacings. These cases complicated the aspectization of distribution and security in the CVS Plugin: the advice in the exception handling aspects, which implemented handlers and clean-up actions, also contained calls to C2 methods that were being moved to aspects. In this case, we needed to change the implementation of the handler advice in order to (i) use reflective features of AspectJ to access the elements of C2, or (ii) use the execution of handler advice as a join point of interest in poincut(s) of the C2-specific aspects.

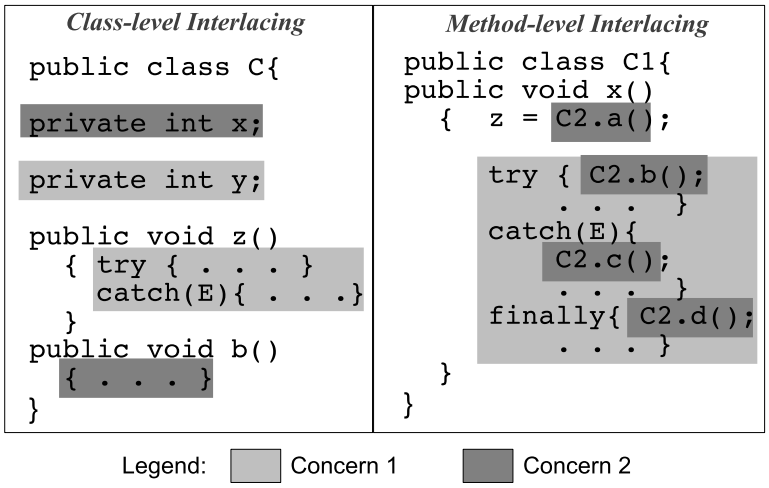

Fig. 3. Aspect interaction categories.

The situation becomes more complicated when a handler advice depends on local variables (Section 4.3) that are initialized through calls to C2-specific methods, such as the z variable in Figure 3. After refactoring, exception handling aspects would be advising such a method call (C2.a()) in order to save the value being assigned to z in an aspect variable. However, with the aspectization of C2, the call C2.a() would either be moved to an aspect related to C2 or be advised by such aspect. This would require the exception handling aspect to take this C2-specific aspect into account, creating a dependency between the two aspects.

4.5 Limitations of this Study

Our study focuses on a single AO language, namely, AspectJ. Although many ideas presented here also apply to other AO languages, some surely do not. For example, it not necessary to soften exceptions in Eos [19], an AO extension to C#, because C# does not have checked exceptions.

Arguably, the employed metrics suite is a limitation of this work. There are a number of other existing metrics and other modularity dimensions that we could be exploited in our study. We have to decided to focus on the metrics described in Section 2.3 because they have already been proved to be effective quality indicators in several case studies [2, 10, 11, 3]. In fact, despite the well-known limitations of these metrics, they complement each other and are very useful when analyzed together. In addition, there is no way in a single study to explore all the possible measures. For every possible metrics suite there will be some dimensions that will remain uncovered. In addition, future case studies can use additional metrics and assess the aspectization of exception handling using different modularity dimensions.

5 Related Work

Even though introductory texts [14, 16] often cite exception handling as an example of the (potential) usefulness of AOP, only a few works attempt to evaluate the suitability of this new paradigm to modularize exception handling code. The study of Lipert and Lopes [17] employed an old version of AspectJ to refactor exception handling code in a large OO framework, called JWAM, to aspects. The goal of this study was to assess the usefulness of aspects for separating exception handling code from the normal application code. The authors presented their findings in terms of a qualitative evaluation. Quantitative evaluation consisted solely of counting LOC. They found out that the use of aspects for modularizing exception detection and handling in the aforementioned framework brought several benefits, for example, better reuse, less interference in the program texts, and a decrease in the number of LOC.

The Lippert and Lopes study was a important initial evaluation of the applicability of AspectJ in particular and aspects in general for solving a real software development problem. However, it has some shortcomings that hinder its results to be extrapolated to the development of real-life software systems. First, the target of the study was a system where exception handling is generic (not application-specific). However, exception handling is an application-specific error recovery technique [1]. In other words, the "real" exception handling would be implemented by systems using JWAM as an infrastructure and not by the framework itself. Most of the handlers in JWAM implemented policies such as "log and ignore the exception". This helps explaining the vast economy in LOC that was achieved by using AOP. Second, the qualitative assessment was performed in terms of quality attributes that are not well-understood, such as (un)pluggability and support for incremental development. The authors did not evaluate some attributes that are more fundamental and well-understood in the Software Engineering literature, such as coupling and cohesion. Third, quantitative evaluation was performed only in terms of number of LOC. Although the number of LOC may be relevant if analyzed together with other metrics, its use in isolation is usually the target of severe criticisms.

Tilevich et al [24] present the GOTECH framework which can make local Java applications distributed in a seamless and automated way. The framework is based on three components: (i) a middleware infrastructure, (ii) a code generation engine, and (iii) AspectJ. One of the issues that the framework addresses is the handling of exceptions derived from network errors. This work does not, however, evaluate the adequacy of AspectJ to modularize exception handling code in general.

An initial assessment of the use of AspectJ for modularizing exception handling in software systems with non-trivial exception handling code has appeared elsewhere [3]. This previous assessment was based solely on a small part of Telestrada (+- 2000 LOC). Furthermore, it did not attempt identify the situations where modularizing exception handling with aspects is beneficial or harmful. Also, it did not investigate how exception handling aspects interact with aspects implementing other concerns.

One of the first studies of the applicability of AOP for developing dependable systems has been conducted by Kienzle and Guerraoui [15]. The study consisted of using AOP to separate concurrency control and failure management concerns from other parts of dis-

tributed applications. It employed AspectJ and transactions as a representative of AOP languages and a fundamental paradigm to handle concurrency and failures, respectively. This work is similar to ours in its overall goal, namely, to assess the benefits of using aspects to modularize error recovery code. However, there are some fundamental differences: (i) we use exception handling to deal with errors, instead of transactions; (ii) we substantiate our conclusions with measurements based on a metrics suite for AO software, instead of examples; (iii) we do not address concurrency; (iv) our study is more general and based on a varied set of applications with diverse error handling strategies.

Soares and his coleagues [23] employed AspectJ to separate persistence and distribution concerns from the functional code of a health care application written in Java. The authors found out that, although AspectJ presents some limitations, it helps in modularizing the transactional execution of methods in many situations that occur in real systems. Furthermore, they employed aspects to modularize part of the exception handling code of an application, but did not attempt to assess the suitability of AspectJ for this task.

An early position paper by Fradet and Südolt [8] discusses the features that an AO language for detecting errors in numeric computations should provide. It proposes pointcut designators that work as global invariants whose violations trigger the execution of recovery code (advice). This work is complementary to ours because it focuses on error detection while ours emphasizes error recovery.

6 Concluding Remarks

In this paper, we presented an in-depth study to assess if AOP improves the quality of the application code when employed to modularize non-trivial exception handling. We found out that, although the use of AOP to separate exception handling code and normal application code can be beneficial, that depends on a combination of several factors. As discussed in the previous sections, if exception handling code in an application is non-uniform, strongly context-dependent, or too complex, aspectization can bring more harm than good. We believe that effective use of AOP requires some *a priori* planning and must be incorporated in the software development process. For exception handling, ad-hoc aspectization is beneficial only in simple scenarios.

The main contributions of this work are: (i) a substantial improvement, based on experience acquired from refactoring four different applications, to the existing body of knowledge about the effects of AOP on exception handling code; (ii) a useful set of scenarios that can be used by developers to better understand when it is beneficial to aspectize exception handling and when it is not; and (iii) an initial assessment of the effects of aspect interaction when exception handling gets in the mix.

Empirical studies about interactions between aspects have only now started to surface [2]. In the specific case of interactions between exception handling and other aspects, the presented empirical study provides an important starting point, but much remains to be done. For example, our study does not assess how handler and clean-up method-level interlacing (Section 4.4) affect the employed metrics. Also, the classification of Section 4.3 does not apply to systems where other concerns are modularized as aspects *a priori*.

References

1. T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Springer, 2nd edition, 1990.
2. N. Cacho et al. Composing design patterns: A scalability study of aspect-oriented programming. In *Proceedings of AOSD'06*, March 2006.
3. F. Castor Filho, A. Garcia, and C. M. F. Rubira. A quantitative study on the aspectization of exception handling. In *Proceedings of the ECOOP'2005 Workshop on Exception Handling in Object-Oriented Systems*, 2005.
4. F. Cristian. A recovery mechanism for modular software. In *Proceedings of the 4th ICSE*, pages 42–51, 1979.
5. F. Cristian. Exception handling. In *Dependability of Resilient Computers*. BSP Professional Books, 1989.
6. C. Fetzer et al. Automatic detection and masking of nonatomic exception handling. *IEEE Trans. Soft. Eng.*, 30(8):547–560, 2004.
7. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
8. P. Fradet and M. Südolt. An aspect language for robust programming. In *Proceedings of the ECOOP'99 Workshop on AOP*, June 1999.
9. A. Garcia et al. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software, Elsevier*, 59(2):197–222, 2001.
10. A. Garcia et al. Modularizing design patterns with aspects: A quantitative study. In *Proceedings of AOSD'05*, pages 3–14, March 2005.
11. I. Godil and H. Jacobsen. Horizontal decomposition of prevlayer. In *Proceedings of CASCON 2005*, 2005.
12. J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.
13. J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of OOPSLA'02*, pages 161–173, November 2002.
14. G. Kiczales et al. Aspect-oriented programming. In *Proceedings of ECOOP'97*, LNCS 1241, pages 220–242, 1997.
15. J. Kienzle and R. Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In *Proceedings of ECOOP'02*, LNCS 2374, pages 37–61, June 2002.
16. R. Laddad. *AspectJ in Action*. Manning, 2003.
17. M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of ICSE'2000*, pages 418–427, June 2000.
18. M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of AOSD'05*, pages 111–122, March 2005.
19. H. Rajan and K. Sullivan. Eos: Instance-leve aspects for integrated system design. In *Proceedings of ESEC/FSE'2003*, September 2003.
20. M. Robillard and G. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM TOSEM*, 12(2):191–221, April 2003.
21. C. Sant'Anna et al. On the reuse and maintanance of aspect-oriented software: An assessment framework. In *Proceedings of the 17th Brazilian Symposium on Software Engineering*, pages 19–34, October 2003.
22. S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Soft. Eng.*, 26(9):849–871, 2000.
23. S. Soares et al. Implementing distribution and persistence aspects with aspectj. In *Proceedings of OOPSLA'02*, pages 174–190, November 2002.
24. E. Tilevich et al. Aspectizing server-side distribution. In *Proceedings of ASE'2003*, pages 130–141, October 2003.
25. M. van Dooren and E. Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *Proceedings of OOPSLA'05*, pages 455–471, October 2005.
26. W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of OOPSLA'04*, pages 419–433, October 2004.