Exceptions and Aspects: The Devil is in the Details

Fernando Castor Filho\textsuperscript{1}, Nelio Cacho\textsuperscript{2}, Eduardo Figueiredo\textsuperscript{3}, Raquel Maranhão\textsuperscript{1}, Alessandro Garcia\textsuperscript{2}, Cecília Mary F. Rubira\textsuperscript{1}

\textsuperscript{1} Institute of Computing, State University of Campinas
P.O. Box 6176, 13084-971, Campinas, SP, Brazil
\textsuperscript{2} Computing Department, Lancaster University
South Drive, InfoLab 21, LA1 4WA, Lancaster, UK
\textsuperscript{3} Informatics Department, Pontifical Catholic University of Rio de Janeiro
Rua Marquês de São Vicente, 225 RDC, 22453-900, Rio de Janeiro, RJ, Brazil

fernando@ic.unicamp.br, n.cacho@comp.lancs.ac.uk, emagno@inf.puc-rio.br, raquelmaranhao@gmail.com, garciaa@comp.lancs.ac.uk, cmrubira@ic.unicamp.br

ABSTRACT

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 well-understood. This paper presents an in-depth study 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 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.

Categories and Subject Descriptors: D.1 [Software]: Programming Techniques – Aspect-Oriented Programming; D.2.8 [Software]: Software Engineering – Metrics.

General Terms: Design, Experimentation, Languages.

Keywords: exception handling, AspectJ, empirical studies.

1. INTRODUCTION

Exception handling [13] mechanisms were conceived as a means to improve modularity of programs that have to deal with exceptional situations [6]. 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 [6, 13]. Several researchers [8, 20, 23] 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 [10] that affects almost all the system modules [20], mostly in an application-specific fashion [1]. Also, a large part of the system code is usually devoted to error detection and handling [7, 24], but this part of the code is often the least understood, tested, and documented [7].

Given the broadly-scoped character of exception handling, aspect-oriented programming (AOP) techniques [17] 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 [17, 19, 20]. Some recent research works [18, 20, 22] 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 [20]. 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 is it not?
- How do exception handling aspects affect aspects implementing other concerns?

This paper presents an in-depth study that assesses the adequacy of AspectJ [19], 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, such as a framework. Hence, the exception handling code also implements non-uniform, complex strategies, making it harder to move handlers to aspects. Second, we employ a metrics suite [11] 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 do not attempt to move error detection code to aspects. Error detection involves checking the state of a program against a certain predicate when its control flow graph reaches a certain node, at runtime [7]. Thus, in many cases, error detection code is very strongly coupled with the code that implements a system’s normal behavior. We believe this subject deserves a separate in-depth study.

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 how we moved exception handling code to aspects. 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

AspectJ [19] extends Java with constructs for picking specific points in the program flow, called join points, and executing pieces of code, called advice, when these points are reached. Join points are points of interest in the program execution through which crosscutting concerns are composed with other application concerns. AspectJ adds a few new constructs to Java. A pointcut picks out certain join points and contextual information at those join points. Join points selectable by pointcuts vary in nature and granularity. Examples include method call and class instantiation. Advice may be executed before, after, or around the selected join points. In the latter case, execution of the advice may potentially alter the flow of control of the application, and replace the code that would be otherwise executed in the selected join point. AspectJ also lets programmers suppress the static checks that the Java compiler makes regarding checked exceptions. This feature is called exception softening and is useful when it is necessary to move exception handlers to aspects. Exceptions are softened within a set of join points. When exceptions are thrown in these join points, they are automatically wrapped by a pre-defined unchecked exception called SoftException.

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 parts of a try-catch block. 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.

We used the Extract Fragment to Advice [21] refactoring to move handlers to aspects. After extracting all the handlers to advice, we looked for reuse opportunities and eliminated identical handlers. The following code snippet shows a trivial example of aspectization of handlers using an around advice. Due to space constraints, we do not show how we extracted handlers to after advice.

```java
// original code // refactored code
class C {
    void m() {
        try {
            throw new SoftException();
        } catch (E e) {
            // former body of try block
        }
    }
}

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

We implemented handlers in the aspects using 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 (handlers that mask the exception). AspectJ requires that an after advice end its execution in the same way as the join point to which it is associated. Therefore, if the code of an after advice is executed following the raising of an exception, the runtime system of AspectJ assumes that the advice ends its execution by raising an exception (either the original or a new one), even if that does not happen explicitly. Thus, it is not possible to implement as an after advice a handler that logs the caught exception and ignores it (the advice would have to raise some exception). In these cases, around advice were employed, since they do not have this restriction. 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 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, creating several simple aspects that encapsulate the possible handling strategies for each type of exception, or creating a separate aspect for each handling strategy. More moderate approaches include creating a handler aspect per class that includes exception handling code or one aspect for each package. For a system where other concerns have been aspectized a priori, a feasible strategy would be to create one exception handling aspect per aspectized concern. Each organization has pros and cons that revolve around the code size vs. modularity trade-off. This is further discussed in Section 3.1.

Whenever possible, we associated exception handling advice to methods through execution pointcut designators. These pointcut designators have a simple semantics and, unlike the call pointcut designator, do not require that the exception handling advice deal directly with SoftException when it is necessary to soften exceptions. For cases where it was not possible to use the execution pointcut designator, we looked for alternate solutions, depending on the circumstances, usually employing the call, new, within code, and cflow pointcut designators.

In several occasions, we modified the implementation of a method in order to expose join points that AspectJ could select more directly or contextual information required by exception handlers, for example, the values of local variables. Usually, this amounted to extracting new methods whose body is entirely contained within a try block, and whose parameters were the contextual information required by the handler. This was often necessary when try-catch blocks were tangled with the normal code, for example, nested within a loop. In these situations, using a pointcut to select the execution of the whole method might not be appropriate. After exception handling, when a tangled try-catch block does not end its execution by raising an exception, it is necessary to resume the execution of the normal code from the statement that textually follows the handler block. However, if no refactoring is performed, this behavior cannot be achieved for cases where it is impossible to specify a pointcut that captures exclusively the statements within the try part of the try-catch block.

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 first three systems was written 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 is a demo for the Java Platform, Enterprise Edition (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 platform. The implementation of the plugin comprises approximately 170 classes and interfaces and approximately 19000 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 versions of the target systems. This suite includes metrics for separation of concerns, coupling, cohesion, and size to evaluate both original and refactored implementations and have already been used in several experimental studies [3, 4, 12, 14]. The coupling, cohesion, and size metrics were defined based on some classic OO metrics [5]. The original OO metrics were extended to be applied in a paradigm-independent way, supporting the generation of comparable results. Also, 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 [11].

---

1http://java.sun.com/developer/releases/petstore/
2http://java.sun.com/j2ee
3http://www.eclipse.org
TABLE 1: METRICS SUITE

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Metrics</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separation of Concerns</td>
<td>Concern Diffusion over Components</td>
<td>Counts the number of components that contribute to the implementation of a concern and other components which access them.</td>
</tr>
<tr>
<td></td>
<td>Concern Diffusion over Operations</td>
<td>Counts the number of fields of each class or aspect.</td>
</tr>
<tr>
<td></td>
<td>Concern Diffusion over LOC</td>
<td>Counts the number of transition points for each concern through the LOC.</td>
</tr>
<tr>
<td></td>
<td>Coupling Between Components</td>
<td>Counts the number of components declaring methods or fields that may be called or accessed by other components.</td>
</tr>
<tr>
<td></td>
<td>Lines of Code (LOC)</td>
<td>Counts the lines of code.</td>
</tr>
<tr>
<td></td>
<td>Number of Attributes</td>
<td>Counts the number of fields of each class or aspect.</td>
</tr>
<tr>
<td></td>
<td>Number of Operations</td>
<td>Counts the number of methods and advice of each class or aspect.</td>
</tr>
<tr>
<td></td>
<td>Vocabulary Size</td>
<td>Counts the number of components (classes, interfaces, and aspects) of the system.</td>
</tr>
</tbody>
</table>

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 percentage difference between the original and refactored versions of each system, relative 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 separation of concerns metrics. In general, the refactored versions of the target systems performed better than the original ones. In the refactored versions, all the code related to exception handling that was not machine-generated was moved to aspects. We did not consider machine-generated code because it typically does not need to be maintained by developers. Among the target systems, only the Java PetStore includes machine-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, an exception handling aspect was created for each other crosscutting concern. For the CVS Plugin, the programmer was left free to create aspects as he deemed necessary. 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 a middle ground between Telestrada and Health Watcher, with a reduction of 48.18%. In the refactored Health Watcher, the small number of exception handling aspects (10) represented a reduction of 78.7% in the value of the metric. For the CVS Plugin, only 4 exception handling aspects were created, resulting in a reduction of 93.22%.

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 machine-generated code was not moved to aspects. In spite of this, 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.

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 increase in the value of Depth of Inheritance Tree for some of the target systems was due to the creation of abstract aspects from which other handler aspects inherit. The greater the number of concrete aspects in a system that inherit from a newly created abstract aspect, the greater the value of the metric. Amongst all the metrics we employed, Coupling between Components was the least affected by the aspectization of exception handling. None of the target 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.

Lack of Cohesion in Operations was the metric for which the refactored target systems presented the worst results. The refactored versions of all target systems performed worse in this metric. For the refactored versions of Health Watcher and 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.

### 3.3 Size Measures

Contradicting the general intuition that aspects make programs smaller [19, 20] 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 aspectiza-
<table>
<thead>
<tr>
<th>Application</th>
<th>Lines of Code</th>
<th>Number of Attributes</th>
<th>Number of Operations</th>
<th>Vocabulary Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telestrada</td>
<td>3352</td>
<td>2885</td>
<td>121</td>
<td>127</td>
</tr>
<tr>
<td>Aspects</td>
<td>0</td>
<td>159</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>3352</td>
<td>3334</td>
<td>121</td>
<td>127</td>
</tr>
<tr>
<td>Diff.</td>
<td>-0.54%</td>
<td>0%</td>
<td>+13.14%</td>
<td>+8.04%</td>
</tr>
<tr>
<td>Java Pet Store</td>
<td>17482</td>
<td>15992</td>
<td>542</td>
<td>542</td>
</tr>
<tr>
<td>Classes</td>
<td>-</td>
<td>2045</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>Aspects</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>17482</td>
<td>17638</td>
<td>542</td>
<td>548</td>
</tr>
<tr>
<td>Diff.</td>
<td>+0.89%</td>
<td>+1.11%</td>
<td>+11.57%</td>
<td>+10.91%</td>
</tr>
<tr>
<td>Eclipse CVS Core Plugin</td>
<td>18876</td>
<td>17863</td>
<td>892</td>
<td>854</td>
</tr>
<tr>
<td>Classes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Aspects</td>
<td>0</td>
<td>1620</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>18876</td>
<td>19423</td>
<td>892</td>
<td>864</td>
</tr>
<tr>
<td>Diff.</td>
<td>+2.82%</td>
<td>-0.23%</td>
<td>+11.66%</td>
<td>+1.43%</td>
</tr>
<tr>
<td>Health Watcher</td>
<td>5732</td>
<td>4641</td>
<td>152</td>
<td>152</td>
</tr>
<tr>
<td>EH Aspects</td>
<td>86</td>
<td>855</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Other Aspects</td>
<td>812</td>
<td>701</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Total</td>
<td>6630</td>
<td>6195</td>
<td>167</td>
<td>171</td>
</tr>
<tr>
<td>Diff.</td>
<td>-6.50%</td>
<td>+2.4%</td>
<td>+11.46%</td>
<td>+3.73%</td>
</tr>
</tbody>
</table>

Table 4: Size Metrics.

Aspects

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 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 [20], 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.

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. An important benefit of aspectizing design patterns 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 [15]. 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, such as distribution [22], assertion checking [20], and some design patterns [15].

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. 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, constraining 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?

Reuse of handler code is not the main expected benefit of using aspects for modularizing crosscutting concerns in software systems. Rather, an implementation of concern code which is localized and consistent is a stronger reason for using aspects. However, some researchers [15, 19, 20, 22] claim that reuse is often a natural consequence of aspectization, specially when it comes to exception handling code [19, 20]. In our study, we found that reusing handlers is much more difficult than is usually advertised [19, 20]. 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.

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 1. 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. Since the nearest common supertype is Exception, the advice should re-throw unchecked exceptions to avoid changing the system’s behavior. 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.

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 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.

```
// ADVICE #1:
2 boolean around() : ... {
3 try { return proceed(); }
4 catch (CVSException e) { CVSProviderPlugin.log(e); } return false;
5 }
6 }
7 // ADVICE #2
8 boolean around() : ... {
9 try { return proceed(); }
10 catch (IOException e) { CVSProviderPlugin.log(
11 IStatus.ERROR, e.getMessage(), e); } return false;
12 }
13 }
14 // ADVICE #3
15 boolean around() : ... {
16 try { proceed(); }
17 catch (CVSException e) { CVSProviderPlugin.log(e); }
18 return false;
19 }
```

Figure 1: Three similar advice that can not be combined.

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 aims to help 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 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
### Table 5: Some exception handling scenarios according to the proposed classification

<table>
<thead>
<tr>
<th>#</th>
<th>Placement of try-catch blocks</th>
<th>Dependency on local vars.</th>
<th>Flow of control after handler execution</th>
<th>Should be aspectized?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>basic</td>
<td>tangled</td>
<td>nested</td>
<td>top-level</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>6</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>7</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>8</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

The rightmost column of Table 5 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 6 justifies the “no” and “depends” scenarios.

### 4.4 Exception handling and other aspects

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. These 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 in-
volving 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. These 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 [3]. 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 2. 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 of 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 2, 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 dependency between aspects. This would require the exception handling aspect to take this fact into account, 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 2, 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 2 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 join points of interest in pointscuts of the C2-specific aspects.

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 f variable in Figure 2. After refactoring, exception handling aspects would be advising such a method call (C2.a()) in order to save the value being assigned to f 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.

<table>
<thead>
<tr>
<th>#</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>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.</td>
</tr>
<tr>
<td>4 &amp; 7</td>
<td>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.</td>
</tr>
<tr>
<td>5</td>
<td>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.</td>
</tr>
<tr>
<td>6</td>
<td>Aspectization is only beneficial if: (i) the handler accesses just a few variables (&lt; 4) and only for reading; and (ii) the refactoring employed to expose these variables creates a method that makes sense by itself.</td>
</tr>
<tr>
<td>8</td>
<td>Loop iteration handlers are usually too strongly coupled with the context where they appear.</td>
</tr>
</tbody>
</table>

Table 6: Justification for the “no” and “depends” scenarios of Table 5.

---

**Figure 2:** Aspect interaction categories.
4.5 Limitations of this Study

This study does not attempt to assess how different refactoring strategies for moving exception handlers to aspects affect the results. Furthermore, only one team of developers was responsible for conducting the study. More general results could be obtained by employing different teams of developers and performing measurements on the refactored systems produced by each team. Another limitation is that we have not evaluated how aspects affect execution time in the target systems.

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. More powerful join point models would make it possible to deal more appropriately with some complicated cases, such as those where handler blocks are tangled or nested, thus affecting the study results. For example, using the loop pointcut designator of the LoopsAJ language [16] it is possible to associate handlers to exceptions raised and not handled within loops. Moreover, one of the extensions to AspectJ proposed by the developers of the abc AspectJ compiler [2] allows the selection of throw statements as join points. In some situations, this feature makes it possible to easily aspectize termination handlers (Section 4.3) that are nested or tangled in the original code (scenarios 3, 6, and 7 of Table 5).

Arguably, the employed metrics suite is a limitation of this work. There are a number of other existing metrics and other modularity dimensions that 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 [3, 4, 12, 14]. 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 to explore all the possible measures in a single study.

5. RELATED WORK

Even though introductory texts [17, 19] 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 [20] 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 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 Lipert 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.

An initial assessment of the use of AspectJ for modularizing exception handling in software systems with non-trivial exception handling code has appeared elsewhere [4]. 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 [18]. The study consisted of using AOP to separate concurrency control and failure management concerns from other parts of distributed 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 colleagues [22] employed AspectJ to separate persistence and distribution concerns from the functional code of a health care application written in Java. The authors found 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 Siidlott [9] 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 that, although the use of AOP to separate exception handling with aspects is beneficial or harmful. Also, it did not investigate how exception handling aspects interact with aspects implementing other concerns.
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 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 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.

As mentioned in Section 1, this work does not attempt to measure the effects of aspectizing exception detection code. We believe that investigating if AOP can be employed to modularize error detection is an exciting direction for future work. We also intend to study the feasibility of devising a tool that (semi-)automatically detects and refactors exception handling code that is beneficial to aspectize.

Empirical studies about interactions between aspects have only now started to surface [3]. 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.

Acknowledgements. We would like the anonymous referees for the insightful comments. Fernando is supported by FAPESP/Brazil (02/13996-2). Cecília is supported in part by FAPESP/Brazil (2004/10663-8) and CNPq/Brazil (351592/97-0). Fernando and Cecília are partially supported by FINEP/Brazil (1843/04), project CompGov, which aims to develop a component library for e-Government. Nelio and Alessandro are supported by European Commission, grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008. Alessandro is also supported by the TAO project, funded by Lancaster University Research Committee.

7. REFERENCES