SOFTWARE—PRACTICE AND EXPERIENCE
*Softw. Pract. Exper.* 2009; **39**:1377–1417
Published online 2 October 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.939

# On the modularization and reuse of exception handling with aspects

Fernando Castor[1,*,†], Nélio Cacho[2], Eduardo Figueiredo[3], Alessandro Garcia[4], Cecília M. F. Rubira[5], Jefferson Silva de Amorim[6] and Hítalo Oliveira da Silva[6]

[1]*Informatics Center, Federal University of Pernambuco (UFPE), Recife, Brazil*
[2]*School of Science and Technology, Federal University of Rio Grande do Norte (UFRN), Natal, Brazil*
[3]*Computing Department, Lancaster University, Lancaster, U.K.*
[4]*Informatics Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil*
[5]*Institute of Computing, University of Campinas (UNICAMP), Campinas, Brazil*
[6]*Department of Computing and Systems, University of Pernambuco (UPE), Recife, Brazil*

**SUMMARY**

**This paper presents an in-depth study of the adequacy of the AspectJ language for modularizing and reusing exception-handling code. The study consisted of refactoring existing applications so that the code responsible for implementing error-handling strategies was moved to newly created exception handler aspects. We have performed quantitative assessments of five systems—four object-oriented and one aspect-oriented—based on four key 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 extent to which error-handling aspects can be reused, (ii) the beneficial and harmful aspectization scenarios for exception handling, and (iii) the scalability of AOP to support the modularization of exception handling in the presence of other aspects. Copyright © 2009 John Wiley & Sons, Ltd.**

*Received 22 April 2009; Revised 21 July 2009; Accepted 22 July 2009*

*Correspondence to: Fernando Castor, Informatics Center, Federal University of Pernambuco (UFPE), Av. Prof. Lus Freire s/n, 50740-540, Recife, PE, Brazil.
†E-mail: castor@cin.ufpe.br

Contract/grant sponsor: CNPq/Brazil; contract/grant numbers: 481147/2007-1, 550895/2007-8, 308383/2008-7, 301446/2006-7, 484138/2006-5
Contract/grant sponsor: National Institute of Science and Technology for Software Engineering (INES)
Contract/grant sponsor: CNPq and FACEPE; contract/grant numbers: 573964/2008-4, APQ-1037-1.03/08
Contract/grant sponsor: CAPES/Brazil
Contract/grant sponsor: FACEPE/Brazil

KEY WORDS: aspect-oriented programming; exception handling; metrics; reuse; modularity; AspectJ

## 1. INTRODUCTION

In spite of the incorporation of exception-handling mechanisms [1] in modern programming languages, reuse of error-handling code is still difficult for software engineers. Exception-handling mechanisms were conceived in the mid-seventies as a means to improve the organization of programs that have to deal with exceptional situations [1,2]. Ideally, an exception-handling mechanism should provide programming constructs that facilitate the construction of error handling code that is modular and can be more easily reused. The designs of exception-handling mechanisms were aimed at promoting an explicit textual separation, with well-defined interfaces, between normal behavior and exceptional (or abnormal) behaviors. Several researchers [3–5] have explored new programming techniques in order to reap the promised benefits of existing exception-handling mechanisms. In spite of this, implementing modular and reusable error-handling code is far from being straightforward.

The main problem is that in realistic software systems normal processing code and error-handling code can be tightly coupled. Exception-handling is known to be a global design issue [6] that affects almost all the system modules [4], mostly in an application-specific manner [7].

Given the broadly scoped character of exception-handling in software systems, aspect-oriented programming (AOP) techniques [8] are increasingly seen as a natural candidate to promote enhanced modularity and reusability of programs in the presence of exceptions. It is usually assumed that the exceptional behavior of a system is a crosscutting concern that can be better modularized by the use of AOP [4,8,9]. However, the existing research works [4,10,11] usually restrict their analysis to 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 [4]. The authors had the goal of evaluating whether 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 reuse of exception-handling behavior. 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 to separate exceptional code from the normal application code is beneficial, the trade-offs involving modularity and reuse 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. In addition, the reusability of exception-handling aspects in the presence of interaction with aspects that implement other equally important concerns, such as distribution and security, has not been explicitly studied. Hence, some important research questions remain unaddressed:

RQ1. Does AOP improve well-accepted quality attributes other than separation of concerns, such as coupling, cohesion, and conciseness?



DOI: 10.1002/spe

RQ2. To what extent is the modularity of exception handler aspects affected by interactions with other aspects?
RQ3. Is exception handling a reusable aspect in real, deployable, software systems?
RQ4. When is it possible to aspectize exception handling in a modular manner? When is it not?

In order to address these research questions, this paper presents an in-depth study that assesses the adequacy of AspectJ [9] for modularizing exception handling code. AspectJ was chosen as it is a general-purpose AO extension to Java and offers a programming model that was mimicked by many other mainstream AOP frameworks. Initially, the study consisted of refactoring five different applications so that the code responsible for handling exceptions was moved to aspects. Four of these applications were originally written in Java and one was implemented in AspectJ. The targets of this 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 exception handlers to aspects. We employed a conventional metrics suite [12] to quantitatively assess modularity attributes, which have well-known impact on reuse, such as coupling and cohesion, in both the original and the refactored systems. We have also directly quantified the degree of exception-handling reuse in the target applications. Finally, we evaluated how exception handler aspects interact with aspects implementing other concerns.

We have found that, in general, AOP improved the separation of concerns between exception-handling code and normal application code. Moreover, we have noticed that aspects can promote exception handler reuse, but this kind of reuse may require careful design planning and is sometimes difficult to achieve in practice. Furthermore, contradicting the general intuition, we have observed that, for systems with application-specific exception-handling strategies, an AO solution does not result in a reduced number of lines of code (LOC). Another consequence of using aspects was that, in many cases, it was necessary to refactor the base application code to expose join points that AspectJ can capture. This produced code that did not appropriately express the intent of the programmer and had a negative impact on the overall cohesion of the systems.

We share the lessons learned from the study by proposing a classification for error-handling code. This classification revolves around the factors that we have determined to have more influence on the task of refactoring exception handlers to aspects. We use the proposed classification to devise a set of scenarios that comprise combinations of these factors and indicate whether aspectization is beneficial or harmful in each of the scenarios. Our goal is twofold: (i) to assist developers of new systems to modularize the exceptional behavior with reusable aspects, so that they can focus on the beneficial scenarios and avoid the harmful ones and (ii) to assist maintainers of existing systems in the task of refactoring error-handling code to aspects by showing when aspectization is worth the effort and when it is not.

This paper is organized as follows. The next section (Section 2) describes the setting of our study. The results of the study are presented in Section 3. Section 4 makes a qualitative evaluation of the study results, providing insights into research questions RQ1 (Section 4.1) and RQ2 (Section 4.2). Section 5 describes the results of a more recent study we have conducted in an attempt to more precisely answer research question RQ3 (Section 5). Section 6 addresses research question RQ4 and can be used as a guide to support developers in the task of refactoring error-handling code to aspects. It first indicates the factors that have the strongest influence on the aspectization of error handling (Section 6.1). Afterwards, it provides a detailed analysis of these factors and



DOI: 10.1002/spe

their interactions (Section 6.2). Section 7 indicates some constraints on the validity of our study. Section 8 discusses the related work. The last section presents a summary of our contributions and points directions for the future work.

## 2. STUDY SETTING

This section describes the configuration of our study. Section 2.1 briefly explains how we have 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 [9] 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 calls and class instantiations. 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 it is useful when it is necessary to move exception handlers to aspects. Exceptions are softened within a set of join points. Then, 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 have moved `try-catch`, `try-catch-finally` and `try-finally` blocks in the five applications to aspects. Hereafter, we refer to these types of blocks collectively as `try-catch` blocks, unless otherwise noted. We use the terms `try` block, `catch` block (or exception 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. Furthermore, we did not extract `try-catch` blocks that appear within `catch` and `finally` blocks, since they already pertain to the exception-handling concern and it does not make sense to separate them.

We have applied the Extract Fragment to Advice refactoring [13] to move handlers to aspects. After extracting all the handlers to advice, we have identified reuse opportunities and attempted to eliminate identical exception handlers. The code snippet of Figure 1 shows a simple example of exception handler aspectization using an around advice. In this figure, the `try-catch` block of method `m()` (left-hand side) was moved to the `A` aspect (right-hand side). Then, when method `m()` raises exception `E` in the AO implementation, this event is intercepted and execution is transferred to the *around* advice, which handles the exception.

Exception handler aspects were implemented using *after* or *around* advice. Whenever possible, we have used *after* advice, since they are easier to implement than around advice in AspectJ.



DOI: 10.1002/spe

```
class C
 void m(){
  try{ ... // body of the try block
  }catch(E e){log(e);}
 }
}
```

⇒

```
class C
 void m(){
  ... body of
 }    the try block
}
aspect A {
 pointcut execM():
  execution(* C.m()) &&
 around() : execM() {
  try { proceed();}
  catch(E e) {log(e);}
 }
}
```

Figure 1. An example of exception handler aspectization.

*After* advice are not appropriate, though, to implement handlers that do not raise an exception, that is, handlers that mask the exception returning to the normal control flow. AspectJ requires that an *after* advice end its execution in the same way as the join point with 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 this does not happen explicitly. Thus, it is not possible to implement a handler that logs the caught exception and ignores it (the advice would have to raise some exception) as an *after* advice. 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. In situations where multiple `catch` blocks are associated with a single `try` block, in general, we have 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 with the same join point.

For each target system, we have employed a different strategy for structuring exception handler aspects. This approach helped us in understanding how different structuring strategies could influence handler reuse. Various kinds of strategies are possible, including (i) putting all the exception-handling code in a single aspect, (ii) creating several simple aspects that encapsulate the possible handling strategies (e.g. re-throwing, logging, rolling back) for each type of exception, or (iii) creating a separate aspect for each exception-handling strategy. More moderate structuring strategies include (iv) creating a handler aspect per class that includes exception-handling code or (v) creating a single aspect for each package. For a system where other crosscutting concerns have been aspectized *a priori*, a feasible strategy would be (vi) creating an exception handler aspect per aspectized concern. Each strategy 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 have associated exception-handling advice with methods through `execution` pointcut designators. These pointcut designators have a simple semantics (they capture the entire method execution) and are easy to associate with the base code. For cases where it was not possible to use the `execution` pointcut designator, we have identified alternate solutions, depending on the circumstances, usually employing the `call`, `new`, and `withincode` pointcut designators.



DOI: 10.1002/spe

Table I. Metrics suite.

| Name | Brief description | Programming languages |
|---|---|---|
| Java Pet Store | Java EE-based on-line pet store. This is the demo application of the Java EE platform | Java |
| Telestrada | Web-based traveler information system that allows its users to register and visualize information about Brazilian roads | Java |
| CVS Plugin | Eclipse Plugin that implements the basic functionalities of a CVS client, such as checkin and checkout of a system stored in a remote repository | Java |
| EImp | Eclipse Plugin that supports collaborative software development for distributed teams | Java |
| Health Watcher | Web-based heath care information system that allows users to register complaints about the public health care system | Java and AspectJ |

On several occasions, we have 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 are the contextual information required by the handler. We discuss this subject in more detail in Section 6.2.

### 2.2. Our case studies

Five different applications were refactored in our study, four of them OO and one of them AO. Hereafter we call them 'target systems'. Table I presents an overview of the five target systems of our study. 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 different exception-handling strategies using simple and also sophisticated error recovery scenarios. 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 behavior, the exceptional behavior, the clean-up actions, and other crosscutting concerns, such as persistence and distribution. The rest of this subsection describes the five targets systems of the study.

The original implementation of the first four systems was written in Java and, afterwards, all the exception-handling codes were refactored to create exception handler aspects. Telestrada [14] 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).

[‡] http://java.sun.com/developer/releases/petstore/.
[§] http://java.sun.com/j2ee.



DOI: 10.1002/spe

The system uses various technologies based on the Java EE platform and it is representative of the existing e-commerce applications. Its implementation comprises approximately 17 500 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 near 19 000 LOC. It is the target system with the most complicated exception-handling scenarios. The fourth target system is EImp[‖] , an open-source Eclipse plugin supporting distributed software development. It allows developers in different geographical locations to communicate and exchange information as if they were sharing their development environments. This plugin comprises approximately 120 classes and interfaces and almost 9000 LOC.

Health Watcher [11,15] was the only system originally implemented in AspectJ. It is a web-based information system that was developed for the health care bureau of the city of Recife, Brazil. The original system version involved the aspectization of distribution, persistence, and concurrency control crosscutting concerns. Furthermore, the original system includes some simple exception handler aspects whose handling strategy consists of 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 of moving exception handling code from classes to newly created exception handler aspects. Moreover, we have also extracted exception handling code from existing aspects (not only from classes) related to the other crosscutting concerns, creating a set of exception handler aspects that intercept aspects.

### 2.3. Metrics suite

The quantitative assessment of both the original and refactored versions of the target systems was based on the application of a metrics suite. This suite includes metrics for separation of concerns, coupling, cohesion, and size [12] which have already been used in several experimental studies [15–20]. The coupling, cohesion, and size metrics were defined based on classic OO metrics [21]. The original OO metrics were extended to be applied in a paradigm-independent way, supporting the generation of comparable results. In addition, 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 design components (classes and aspects), operations (methods and advice), and LOC. In the latter case, we use a metric that counts the number of transition points in a program. Transition points are points in the code where there is a 'concern switch', that is, two consecutive LOC where each pertains to a different concern. For example, in the code snippet of Figure 2, there are four transition points, as indicated by the comments. The grey regions represent LOC pertaining to the error-handling concern.

Table II presents brief definitions of the metrics and associates them with the attributes measured by each one. For all the employed metrics, a lower value implies a better result. Detailed descriptions of the metrics appear elsewhere [12]. The gathering of data for these metrics has been automated through the use of the AOPMetrics tool [22]. We have employed AOPMetrics to gather data

[¶] http://www.eclipse.org.
[‖] http://eimp.sourceforge.net.



DOI: 10.1002/spe

```
void m(){
 doSomething(); // some concern
 try{ // transition point #1 -- error handling concern
  ... // transition point #2 -- some concern
 }catch(E e){ // transition point #3 -- error handling concern
  log(e);
 }
 doSomethingElse(); // transition point #4 -- some concern
}
```

Figure 2. A code snippet with four transition points.

Table II. Metrics suite.

| Attributes | Metrics | Definitions |
|---|---|---|
| Separation of concerns | Concern Diffusion over Components | The number of components (classes, aspects, interfaces) contributing to the implementation of a concern and other components that access them |
| | Concern Diffusion over Operations | The number of operations (methods and advice) that contribute to a concern's implementation plus the number of other operations accessing them |
| | Concern Diffusion over LOC | The number of transition points for each concern through the LOC |
| Coupling | Coupling Between Components | 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 component is declared |
| Cohesion | Lack of Cohesion in Operations | Measures the lack of cohesion of a component in terms of the amount of method and advice pairs that do not access the same field |
| Size | Lines of Code (LOC) | Counts the LOC |
| | Number of Fields | Counts the number of fields of each component |
| | Number of Operations | Counts the number of operations of each component |
| | Vocabulary Size | Counts the number of components of the system |

regarding the coupling, cohesion, and size metrics. Only the separation of concerns metrics had to be measured manually.

A common criticism against the employed separation of concerns metrics is that they do not faithfully reflect the degree of separation of concerns in a program, since sometimes it is hard to determine which parts of the code pertain to the implementation of each concern [23]. However, for the exception-handling concern, this issue is not applicable, since this concern is syntactically defined in Java programs by `try-catch-finally` blocks. This factor supports our assumption



DOI: 10.1002/spe

that the employed metrics work as reasonably precise indicators of the quality attributes of object- and AO software systems.

## 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 organized in three parts. Section 3.1 describes the results for the separation of concerns metrics. Section 3.2 presents the results for the coupling and cohesion metrics. Section 3.3 considers the results for the size metrics.

The results are organized in tables that put side-by-side the values of the metrics for the original and refactored versions of each target system. The results for the four OO target systems are presented in two parts, in order to highlight the contribution of classes and aspects to the value of each metric. For the AO application (Health Watcher), the results are discussed in three parts, in order to clarify the contribution of classes, exception handler aspects, and aspects related to the other crosscutting concerns to the value of each metric. Hereafter, we use the term 'class' to refer to both classes and interfaces. Rows labeled 'Diff.' indicate the percentual 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. Concerns measures

Table III shows the obtained results for the concern metrics. In general, the refactored versions of the target systems performed better than the original ones. In the refactored versions, the entire 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 Pet Store includes machine-generated code, produced by the Java EE compiler. In the rows pertaining to Health Watcher, '*EH*' refers to aspects implementing exception handling, whereas '*others*' refers to aspects implementing other crosscutting concerns.

Although the measures of Concern Diffusion over Components diverged strongly amongst the five 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 have 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 handler aspect was created per package. In Health Watcher, an exception handler aspect was created for each other crosscutting concern. For the two Eclipse plugins (CVS Core and EImp), the developer created exception handler aspects based on the subjective criteria. The results for Concern Diffusion over Components in Table III 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 handler 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 handler aspects (10) represented a reduction of 78.7% in



DOI: 10.1002/spe

Table III. Concerns metrics.

| 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%** |
| EImp Eclipse Plugin | Classes | 25 | 0 | 58 | 0 | 338 | 0 |
| | Aspects | — | 2 | — | 33 | — | 0 |
| | Total | **25** | **2** | **58** | **33** | **338** | **0** |
| | Diff. | | **−92.00%** | | **−43.10%** | | **−100%** |
| Health Watcher | Classes | 35 | 0 | 115 | 0 | 488 | 0 |
| | EH | 5 | 10 | 9 | 70 | 0 | 0 |
| | Others | 7 | 0 | 12 | 0 | 48 | 0 |
| | Total | **47** | **10** | **136** | **70** | **536** | **0** |
| | Diff. | | **−78.72%** | | **−48.53%** | | **−100%** |

the value of the metric. For the CVS Plugin and EImp, 4 and 2 exception handler aspects were created, respectively, resulting in reductions above 90%.

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. 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.

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 EImp, this number was more than 40% lower in the refactored version. 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 the best, when compared with the original ones. The refactored versions of four out of five target systems did not have any concern switches and thus had value 0 for this metric. The only exception was



DOI: 10.1002/spe

Table IV. Coupling and cohesion metrics.

| 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%** |
| EImp Eclipse Plugin | Classes | 226 | 205 | 249 | 249 | 188 | 195 |
| | Aspects | — | 30 | — | 2 | — | 17 |
| | Total | **226** | **235** | **249** | **251** | **188** | **212** |
| | Diff. | | **+3.98%** | | **+0.80%** | | **+12.77%** |
| Health Watcher | Classes | 217 | 197 | 69 | 69 | 766 | 867 |
| | EH | 5 | 27 | 3 | 3 | 4 | 130 |
| | Others | 66 | 61 | 17 | 17 | 210 | 210 |
| | Total | **288** | **285** | **89** | **89** | **980** | **1207** |
| | Diff. | | **−1.04%** | | **0%** | | **+23.16%** |

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. In addition, 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 IV 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 have employed, Coupling between Components was the



DOI: 10.1002/spe

least affected by the aspectization of exception handling. None of the target systems had a difference greater than 4.0% between the original and refactored versions and for only one system, EImp, this difference was more than 1.5%. New couplings were introduced only when exception-handler 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 for 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, the CVS Plugin, and Eimp, the increase was of approximately 8, 5, and 13%, 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 the means to precisely capture every join point of interest.

Although 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 and EImp were different. In the former, exception handler aspects accounted for 10.8% of the total value of the metric, whereas aspects pertaining to other concerns were responsible for 17.4%. In the latter, exception-handler aspects accounted for 8.02% of the total value of the metric.

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 fields and methods (Table II). It does not consider direct inter-operation relationships and the semantic closeness between the 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 II), 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 [4,9] due to reuse, the original and refactored versions of the five target systems had very similar results in two of the four size metrics: LOC and Number of Fields. The measure of Vocabulary Size grew as expected, due to the introduction of exception-handler aspects. Moreover, the Number of Operations of the refactored versions of almost all the target systems grew significantly. Table V summarizes the results for the size metrics.

For Telestrada and Java Pet Store, the number of LOC of the original and refactored versions is similar (less than 1% difference). For 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 case, 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. For EImp, the 3.82% increase accounts for 333 extra LOC in the refactored version. The obtained values for LOC were expected. Although some reuse of handler



DOI: 10.1002/spe

Table V. Size metrics.

| Application | | Lines of code | | Number of fields | | Number of operations | | Vocabulary size | |
|---|---|---|---|---|---|---|---|---|---|
| | | Orig. | Refac. | Orig. | Refac. | Orig. | Refac. | Orig. | 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 | 17 482 | 15 593 | 542 | 542 | 2075 | 2135 | 339 | 339 |
| | Aspects | — | 2045 | — | 6 | — | 180 | — | 37 |
| | Total | **17 482** | **17 638** | **542** | **548** | **2075** | **2315** | **339** | **376** |
| | Diff. | **+0.89%** | | **+1.11%** | | **+11.57%** | | **+10.91%** | |
| Eclipse CVS Core Plugin | Classes | 18 876 | 17 803 | 852 | 854 | 1832 | 1848 | 257 | 257 |
| | Aspects | — | 1620 | — | 0 | — | 180 | — | 4 |
| | Total | **18 876** | **19 423** | **852** | **854** | **1832** | **2028** | **257** | **261** |
| | Diff. | **+2.82%** | | **+0.23%** | | **+11.07%** | | **+1.43%** | |
| EImp Eclipse Plugin | Classes | 8708 | 8481 | 286 | 285 | 820 | 821 | 123 | 123 |
| | Aspects | — | 560 | — | 0 | — | 41 | — | 3 |
| | Total | **8708** | **9041** | **286** | **285** | **820** | **862** | **123** | **126** |
| | Diff. | **+3.82%** | | **−0.35%** | | **+5.12%** | | **+2.44%** | |
| Health Watcher | Classes | 5732 | 4641 | 152 | 152 | 542 | 553 | 98 | 98 |
| | EH | 86 | 853 | 3 | 7 | 9 | 73 | 5 | 10 |
| | Others | 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%** | |

code could be achieved, this was nowhere 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 an implementation overhead because it is necessary to specify join points of interest and soften exceptions in order to associate handlers with 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.

As in Number of LOC, the measures for Number of Fields were also similar in the original and refactored target systems. They did not change for Telestrada and varied (positively or negatively) by less than 2.5% in the refactored versions of the remaining systems. New fields were only introduced for situations where it was necessary to temporarily store some contextual information beforehand so that it could be used by the exception handlers. In the case of EImp, there was a decrease of one in the Number of Fields of the refactored version.

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. Similar 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 1.5% in the best (CVS Plugin).



DOI: 10.1002/spe

The Number of Operations was sensibly higher in the refactored target systems. It grew by 13.7% in the refactored version of Telestrada, 11.6% in the Java Pet Store, 10.7% in the CVS Plugin, 5.1% in EImp, 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. Amongst the five target systems, the only one that did not follow this trend was EImp. This system has a small

$$\frac{\#\ of\ operations\ including\ handlers}{\#\ of\ operations}$$

ratio of 0.07, whereas the ratios for the CVS Plugin and Health Watcher were 0.21 and 0.13, respectively. Moreover, only one new method creation was necessary to expose the join points of interest.

## 4. QUALITATIVE ANALYSIS

This section makes a qualitative analysis of the obtained quantitative results (Section 3). We begin by assessing the results, in terms of three quality attributes (Section 4.1): coupling, cohesion, and size. We then proceed to discuss how exception-handling aspects interfere with aspects implementing other crosscutting concerns (Section 4.2). In both cases, we strongly base the analysis on our experience in modularizing exception handling in the five target systems.

### 4.1. Analysing coupling, size, and cohesion

In this section we thoroughly discuss the impact of aspectization on the coupling, size, and cohesion of the target systems.

#### *4.1.1. Coupling*

Our empirical study confirms some of the findings of the study conducted by Lippert and Lopes [4], claiming that the use of aspects decreases the 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. Aspects keep all the exception-handling code encapsulated within program units whose sole purpose is to implement the exception-handling concern. This encapsulation can simplify system maintenance, as developers do not have to search through a whole program to change a certain exception handler. They also have the potential to improve the understandability of the exception-handling code, since it is possible to get an intuitive understanding of how error-handling works in a given system just by looking at the exception-handler aspects. In addition, an exception-handler aspect can be easily replaced by another exception-handler aspect implementing different error-handling strategies. This capability is desirable in cases where different contexts of use impose different requirements in terms of error-handling strategies.



DOI: 10.1002/spe

On the other hand, 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 (Section 3.2). However, a closer examination on the code of the five 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 the 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-handler aspects. The dependency is implicit because it cannot be inferred just by looking at the base code. Moreover, if the exception-handler aspects are not present, the base code will not compile. Hence, Java-based AO languages that do not allow the softening of exceptions, such as CaesarJ [24] and HyperJ [25], cannot be employed to modularize exception handling. They can only be used in situations where the aspectization of error-handling results in programs whose base (non-aspect) code does not violate the language rules for checked exceptions. This is the case, for example, when a handler throws an exception of the same type as (or a subtype of) the exceptions it catches.

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 [26]. 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 aspects. A direct consequence is that, in AspectJ, exception handling is not a pluggable aspect, differently from other concerns, such as distribution [11], assertion checking [4], and some design patterns [26]. We consider a concern to be pluggable if, without affecting any part of the system other than the implementation of the concern, it is possible to switch between using it or not [26]. It is important to emphasize that this does not influence the ease of switching between different exception-handler aspects. In addition, this restriction only applies to base languages that implement checked exceptions, such as Java.

A problem related to exception softening happens when one tries to soften an exception that is a supertype of another exception raised within the same context, for example, thrown by an exception handler aspect. This causes the subtype to be softened as well, possibly with unanticipated effects. We believe that developers should avoid softening exceptions that are supertypes of many other exceptions, such as `Exception` and `Throwable` in Java. Even for less general types, care should be taken in order to avoid unintentional softening (e.g. softening `FileNotFoundException` because `IOException` was declared soft in a given context).

#### *4.1.2. Size*

As expected, we have found out that aspects reduce the amount of duplicated code. It is easy to encapsulate an exception handler that would otherwise appear in several parts of a system in a single handler advice (an advice implementing error-handling code). These parts of the system then become the join points of interest with which the handler advice will be associated. It is important to notice, though, that this does not necessarily mean that the amount of error-handling code reuse is high, neither that it is easy to combine similar (but not identical) advice. In the early days of AOP, it was often claimed that this reduction would yield a reduction in the overall application size [4]. However, more recent studies [27–29] have shown that this is only true if error-handling code is uniform and context-independent. If exception-handling code in an application is non-uniform or



DOI: 10.1002/spe

strongly context-dependent, the reuse of handler code becomes low and the number of LOC in an application can grow due to the implementation overhead of AOP. Moreover, the number of operations (methods and advice) and components (aspects and classes) will almost always grow due to the use of aspects. This subject is further discussed in Section 5.

As discussed in Section 3.3, the new 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, +5% in EImp, 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 with 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 always be seen as a negative factor. Notwithstanding, it is a significant increase if we consider that all of these operations are a consequence of the exception-handling concern.

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, 0.12% in EImp, 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. Moreover, it was consistent throughout all the target systems. 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. This suggests that there is still room for improving AspectJ in order to more precisely select join points of interest. In fact, this has been the subject of many recent papers on AO programming [20,28,30].

#### *4.1.3. Cohesion*

The fundamental precept of the approach adopted in this study is that advice implement exception handlers and are associated with exception throwing code (ETC) through pointcuts. Because of this assumption, the only limitation to the types of exception-handling contexts that can be defined is the join point model of the employed AO language. However, current production-quality AO languages cannot, in some fairly common situations, simulate the exception-handling mechanisms of existing programming languages. The design of the base code must take this into account and avoid these situations. If the base code is developed obliviously to aspects, it is highly probable that it will have to be refactored to make it easier to associate exception-handler aspects with it. Non-coincidentally, in many situations where we have detected increases in Lack of Cohesion in Operations, it was necessary to refactor the base code to expose join points that the exception-handler aspects could capture.

The increase in Lack of Cohesion in Operations for the refactored versions of Java Pet Store, EImp, and the CVS Plugin is much lower than the increase in the same metric for Telestrada and Health Watcher. For 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, contrasting 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 its small size. For Health Watcher, unlike the other target systems, the large increase in the cohesion metric was



DOI: 10.1002/spe

caused mainly by the exception-handler aspects. In particular, 3 exception-handler aspects were 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 original system implementing other crosscutting concerns. In EImp, most of the increase was also a consequence of exception handling aspects. However, both versions of the system exhibited a low overall Lack of Cohesion in Operations. Moreover, although the value of this metric grew by 12% in the refactored version, the absolute growth in its value was very small. Hence, the results were not considered significant.

### 4.2. Exception-handling crosscutting other aspects

Sections 4.1 and 5 have analyzed how AspectJ scaled to support modularization and reuse of diverse forms of exception handling. They consisted of several 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-handler aspects.

Our investigation was carried out mainly in the context of the Health Watcher system since it is the only one originally developed using AOP in our set of target systems. However, as discussed in the previous sections and evidenced by the measurements (Section 3), this system has a simple exceptional behavior, when compared with the other four 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. The security concern supports access control to both executed methods and sent messages in the system. The distribution concern uses different communication protocols, such as http and SOCKS5 proxies, to get SSH2 access beyond a firewall.

We have selected these concerns because they are well known as traditional crosscutting concerns in the literature and tend to have a broadly scoped influence at the system's architecture. 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-handler aspects and aspects implementing other concerns. AspectJ was effective to deal with those inter-aspect interactions, as indicated by the measures in Tables III–V.

We have observed different categories of interactions involving the exception-handling concern and other crosscutting concerns. 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 five 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 [16]. Here we refine the previous categorization by taking exception-handling structures into account, namely protected



DOI: 10.1002/spe

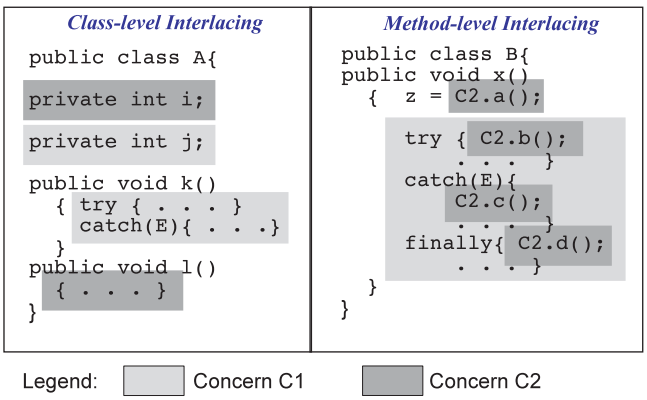


Figure 3. Aspect interaction categories.

regions (`try {}`), handlers (`catch (E) {}`), and clean-up actions (`finally {}`). Figure 3 illustrates these categories, where Concern 1 (C1) corresponds to the exception-handling concern and Concern 2 (C2) represents a different concern. In Health Watcher, C2 can be part of the concurrency control, distribution, or persistence concerns, whereas in the Eclipse CVS plugin it can be 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 concerns. In this case, the implementation concerns C1 and C2 have one class in common. However, each concern encompasses disjoint sets of methods and fields in the same class. As illustrated on the left-hand side of Figure 3, C1 and C2 have a coinciding participant class, but there is no method or field 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*. The distinguishing feature of the four categories of method-level interlacing is where a method call is placed in terms of the exception-handling elements. Different 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, the exception-handling concern is interlaced at the method level with elements of C2. On the right-hand side of Figure 3, method `x()` has code pertaining to both C1 and C2. In most of the situations in the Health Watcher and the CVS plugin, interaction between the concerns C1 and C2 consisted of calls to methods from `C2` by code pertaining C1.

The right-hand side of Figure 3 depicts all the four 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 concerns relative to C2 was straightforward when the situation exhibited interlacing at the unprotected- or protected-region level. The reason is that there is no explicit link between the exception-handler aspects and the C2 code being aspectized. In Health Watcher, all 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



DOI: 10.1002/spe

distribution and security in the CVS Plugin: the advice in the exception-handler 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 handler advice implementation in order to (i) use reflective features of AspectJ to access the elements of C2 or (ii) use the handler advice execution as join points of interest in pointcuts of the C2-specific aspects.

The situation becomes more complicated when a handler advice depends on local variables (Section 6.1) that are initialized through calls to C2-specific methods, such as the z variable in Figure 3. After refactoring, exception-handler aspects would be advising a method call such as C2.a() 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 an aspect. This would require the exception-handler aspect to take this C2-specific aspect into account, creating a dependency between the two aspects.

## 5. REUSE OF EXCEPTION HANDLER ASPECTS

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 encapsulated is a nice reason for using aspects. However, some researchers [4,9,26] claim that reuse is often a natural consequence of aspectization, specially when it comes to exception-handling code [4,9]. In our study, we have found that reusing handlers is much more difficult than is usually advertised [4,9]. This can be noticed by observing the Concern Diffusion over Operations measures 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.

The difficulty of reusing handler code is illustrated by Figure 4, which shows three advice that look similar, but cannot be merged into a single one because of small differences. advice #1 and #2 should 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()` while the latter always returns `false`. For the same reasons, advice #2 and #3 cannot be combined.

In spite of the aforementioned difficulties, determining whether exception handling is a reusable aspect or not has proved to be harder than we have initially assumed. This is due to two reasons. First, the overall number of LOC in the refactored versions of most of the target systems did not change much when compared with the original versions. Since the use of AOP introduces an implementation overhead, the consequence of implementing pointcuts, advice, and intertype declarations, it becomes difficult to tell whether low reuse was responsible for the increase in LOC in the AO versions of the Java Pet Store and the two Eclipse plugins. Moreover, for the target systems where the number of LOC has decreased, the amount of reuse that was achieved is not clear. The second reason is the value of the Concern Diffusion over Operations metric. This metric measures the number of operations in the system that contribute to the implementation of a concern (Section 2.3). In four out of five target systems, there was a reduction in this metric in the refactored versions. This means that there are less operations contributing to the implementation of exception



DOI: 10.1002/spe

```
// 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;
}
```

Figure 4. Example of three similar advice that cannot be combined.

handling, a clear indication that some degree of reuse was achieved. Notwithstanding, this does not seem to be correlated with a reduction in the number of LOC, as we would expect. In the refactored versions of Java Pet Store and EImp, the number of LOC grew by 0.89 and 3.82%, respectively, while the values for Concern Diffusion over Operations were reduced by 21.88 and 43.10%. At the same time, the number of LOC of the AO version of Telestrada was 0.54% smaller, while Concern Diffusion over Operations grew by 4.76%. It is important to stress that this growth (or reduction), in terms of LOC, pertains to the overall system size but is caused solely by the aspectization of exception handling. Considering that, for many systems, all the exception-handling code accounts for between 1 and 5% of the overall number of LOC of the system [31], this is a large increase (or decrease) and not just a random fluctuation.

To better understand the impact of aspects on exception-handling code reuse, we have conducted a complementary study aiming to quantify, at the same time, the amount of intra-application reuse that AOP can achieve and the implementation overhead imposed by AspectJ. We analyze reuse in a specific context: in terms of the number of duplicated or very similar error handlers that can be removed from a program when extracting error-handling code to aspects. We have chosen the Java Pet Store as the target of this new study for three reasons: (i) it is the second largest target system of the original study; (ii) as mentioned before, the number of LOC grew in the refactored version of the system whereas its Concern Diffusion over Operations was reduced; and (iii) it has the greatest number of exception handlers amongst the five target systems.

### 5.1. Handler reuse study: Java Pet Store

The complementary reuse study comprised three steps. First, we attempted to further combine similar exception handlers in the refactored version of the Java Pet Store to obtain a more precise estimate of the amount of reuse that AOP can achieve. In the remainder of this section, we call this new version of the Java Pet Store 'optimized version'.



DOI: 10.1002/spe

```
// Original version ...
try{ ... }
// catch clauses
catch(E1 a){ /* do something */ }
catch(E2 b){ /* do something else */ }
// more catch clauses
...
try{ ... }
// catch clauses
catch(E1 c){ /* do something */ }
catch(E2 d){ /* do something else */ }
// other catch clauses
...
```

```
// optimized version ...
around() : ... {
 try { proceed();}
 catch(E1 e1){ /* do something */ }
 catch(E2 e2){ /* do something else */ }
}
...
```

Figure 5. An example of a reusable handler advice.

In the original study, one advice responsible for-handling exceptions was created for each `try` block in the original version. Identical or very similar advice were combined to reduce duplicated code. However, identical `catch` blocks associated with different `try` blocks were **not** extracted to separate advice to maximize reuse. We avoided doing so because creating one advice for each `catch` block requires the advice to be ordered to mimic the original exception-handling behavior of the system. For this new study, we waived this constraint and, if beneficial for reuse, created new advice on a per-`catch` block basis. At the same time, we devoted additional effort to the task of correctly ordering the advice, so as to refrain from changing the application behavior. Moreover, to make reuse opportunities easier to identify, due to uniformity, we have implemented all the exception handlers of the optimized version using the same kind of advice: *around*. *Around* advice are more powerful than *after* advice, but have the disadvantage of requiring more LOC to implement the same error-handling policy.

The second step consisted of selecting a set of metrics that can be applied to both the original (OO) and optimized versions of the system. In order to quantify reuse of error-handling code, it is necessary to employ metrics that specifically address this concern. At the same time, we need to clearly state what we mean by reuse to select an appropriate set of metrics. According to Frakes and Kang [32], *'**Software reuse** is the use of existing software or software knowledge to construct new software'* and ***Reusability** is a property of a software asset that indicates its probability of reuse*. We adapt these broad definitions to our study taking three important issues into account: (i) we are interested in reuse within a system, instead of across different systems, (ii) we are considering reuse of code pertaining to a specific concern, that is, exception handling, and (iii) we need to be specific, in order to select metrics that appropriately quantify error-handling reuse.

In the context of this study, reuse is the activity of using an existing piece of code more than once with the same goal within the same system. In other words, a piece of code is reusable if its use reduces the need to produce duplicated code and/or duplicated work. In this work, the 'piece of code' is a handler advice and the basic unit of reuse is the `catch` block (or a sequence of `catch` blocks) the code responsible for capturing and handling exceptions. Figure 5 presents an example of reusable handler advice. The left-hand side of the figure presents two sequences of `catch` blocks that are functionally equivalent. The right-hand side of the figure exhibits an advice that can replace both sequences and is, therefore, considered to be reusable.



DOI: 10.1002/spe

Table VI. Error-handling metrics.

| Metrics | Definitions |
|---|---|
| Number of Lines of Error Handling Code | The number of LOC in the system contributing to the implementation of error handling. In a Java program, it comprises all the `catch` and `finally` blocks, including the line that declares a `try` block In AspectJ, it counts the overall number of LOC of the exception handler aspects |
| Number of Protected Regions | A protected region is a region of a program where an exception is always treated in the same way. Protected regions are declared by means of `try blocks`. Hence, this metric counts the number of `try` blocks in the program, excluding the ones appearing within `catch` and `finally` blocks (Section 2.1). In AspectJ, it also counts the number of *after throwing* advice, assuming that each one corresponds to a single `try` block |
| Number of Handlers | The number of `catch` and `finally` blocks in a program. For AspectJ programs, each *after throwing* advice counts as an additional handler |

Table VI shows the metrics used in this complementary study. Since these metrics refer to quantities of program elements that are directly related to error handling, they are likely to be more effective indicators of exception-handling reuse than conventional metrics, such as Number of LOC. In addition, these metrics support the quantification of error handler intra-application reuse, because one can consider a handler advice to be reusable if it can replace duplicated sequences of `catch` blocks that are scattered throughout the system. By comparing the measures of the original and optimized versions of the system, we can estimate the amount of reuse yielded by AOP techniques.

The third step consisted of measuring both the original and optimized versions of Java Pet Store and analyzing the results, similar to the procedure described in the previous sections.

### 5.2. Non-zero overhead reuse of exception-handling aspects

Table VII presents the results of our complementary study. The Number of Lines of Error Handling Code in the optimized version of the Java Pet Store grew by almost 50% when compared with the original version of the system. It is important to stress that these results cannot, in all fairness, be compared with the results of Table V.

In our complementary study, the measures for the three metrics were collected manually. We are not aware of any tool capable of collecting the metrics described in Table VI, that is, metrics related to a specific crosscutting concern, in this case, exception handling. Moreover, as mentioned in the previous section, to make the code more uniform, all error handlers of the optimized version of Java Pet Store were implemented using *around* advice, whereas the refactored version (Section 3.3) includes both *around* and *after* advice.

The Number of Protected Regions of the optimized version of the Java Pet Store presented a reduction of more than 50% when compared with the original version. In the original, OO version, the number of protected regions is strongly dependent on the number of sites in the code where



DOI: 10.1002/spe

Table VII. Error-handling metrics.

| | Number of lines of error handling code | | Number of protected regions | | Number of handlers | |
|---|---|---|---|---|---|---|
| Application | Orig. | Optim. | Orig. | Optim. | Orig. | Optim. |
| Java Pet Store | 1543 | 2313 **+49.9%** | 281 | 135 **−51.96%** | 413 | 166 **−59.8%** |

```
// Original version ...
try{...
}catch (ServiceLocatorException sle){...
}catch (CreateException ce){...
}catch (RemoteException re){...}
...
try{...
}catch (RemoteException re){...
}catch (OPCAdminFacadeException oafee){...}
...
try{ ...
}catch (ServiceLocatorException sle){...
}catch (XMLDocumentException xde) {...
}catch (CreateException ce) {...}
...
try{...
}catch (RemoteException re){...
}catch (OPCAdminFacadeException oafee){...}
...
```

```
// optimized version ...
void around() throws ... : ... {
  try{ proceed();
  }catch(ServiceLocatorException
         sle){...}
} ...
void around() throws ... : ... {
  try{ proceed();
  }catch(CreateException ce){...}
} ...
Object around() throws ... : ... {
  try{ proceed();
  }catch(RemoteException re){...}
} ...
Object around() throws ... : ... {
  try{ proceed();
  }catch(OPCAdminFacadeException
         oafee){...}
} ...
void around() throws ... : ... {
  try{ proceed();
  }catch(XMLDocumentException
         xde){...}
} ...
```

Figure 6. Extracting `catch` blocks to create textually separate exception-handler advice.

exceptions can be raised. On the other hand, in the optimized (AO) version, the number of protected regions is a direct consequence of the number of error-handling strategies for the different exception types that can be raised in the system. This stems from the handlers not needing to be textually close to the raising sites of the exceptions (Figure 6). Therefore, we can conclude that, in the context of this specific study, AOP improved the reuse of error-handling code, since a large number of handlers that were duplicated and scattered throughout the code in the original version are now encapsulated within concern-specific implementation units. AOP makes it possible to implement each specific error handler only once and associating it with the parts of the code to which it is relevant.

The Number of Handlers of the optimized version of Java Pet Store was 60% lower than that of the original version. This is an indication that it is sometimes advantageous to extract `catch` blocks in a `try-catch` block to separate handler advice in order to improve reuse, as illustrated by Figure 6. The snippet on the left-hand side of the figure shows 4 different `try-catch` blocks that are part of 4 different methods appearing in the Java Pet Store implementation. Amongst the



DOI: 10.1002/spe

associated `catch` blocks, handlers for the same exception type implement the same error-handling strategies (which we omit due to space constraints). Simply extracting these `try-catch` blocks to advice on a 'one handler advice per `try` block' basis yields some duplicated `catch` blocks. To avoid this, we have extracted each individual `catch` block to a handler advice, as shown on the right-hand side of Figure 6, and associated the latter with the base code in order to mimic the behavior of the original system. As a consequence, the number of `catch` blocks, 10 in the original version, was reduced to only 5 in the optimized one.

In sharp contrast to the Number of Protected Regions and the Number of Handlers, the obtained measures for Number of Lines of Error-handling Code indicate that the overhead of using AspectJ to modularize exception handling, in terms of LOC, is significant. In other words, the optimized version of the system includes a large portion of code that is a direct consequence of using AOP, but does not directly contribute to the implementation of the exception-handling concern. Therefore, one can argue that this study, to some extent, confirms the results presented in Section 3.3, in the sense that, independently of reuse, exception handler aspects increase the number of LOC in a system. Notwithstanding, it is important to stress that the kind of overhead created by the use of AOP also occurs in programs written in different paradigms, such as OO programming. As mentioned previously, Number of LOC is not, by itself, a measure of the quality of a software system.

Overall, we found that there is no clear correlation between reduction in the number of LOC and reuse of error handling code. Although the number of LOC of the optimized version of Java Pet Store is considerably larger than that of the original version, the optimized version includes a large amount of reused exception-handling code. This contradicts both our previous results [29] and the findings of Lippert and Lopes [4]. The former suggested that AOP promotes only a small amount of reuse of error-handling code, while the system size tends to grow. The latter argued that the use of AOP to modularize exception handling can reduce the system size due to reuse. This may be the case for some specific systems, but is not the general trend. Further studies are still necessary to extrapolate the findings discussed in this section.

## 6. A GUIDE FOR EXTRACTING ERROR HANDLING TO ASPECTS

One of the most important lessons learned from our studies was that a combination of several factors determine if the use of AOP to separate exception handling from base code can be beneficial. In many recurrent situations, to evolve a system so that it uses aspects to implement exception handling, some *a priori* redesign is necessary. If the exception-handling code in an application is non-uniform, strongly context-dependent, or too complex, *ad hoc* aspectization might not be possible or lead to code which: (i) worsens the overall quality of the system; (ii) does not represent the original design of the system; and/or (iii) exhibits 'bad smells' [33], such as Temporary Field and Middle Man.

The existence of many situations where aspectization is not beneficial motivated us to try to understand precisely what factors make it easier or harder to modularize error handling with aspects. Section 6.1 describes a classification for exception-handling code emphasizing factors that have a strong impact on the aspectization of exception handling. Section 6.2 presents a set of scenarios corresponding to the combinations of these factors. Together, the classification and the set of scenarios work as a cookbook that aims to share the acquired experience with developers of AO software.



DOI: 10.1002/spe

### 6.1. A classification for exception-handling code

We classify exception-handling code in Java-like languages according to the following categories: (i) tangling of `try-catch` blocks; (ii) terminal ETC; (iii) nesting of `try-catch` blocks; (iv) dependency of exception handlers on local variables; and (v) flow of control after handler execution. In the rest of this section, we describe these categories.

#### *6.1.1. Tangling of* `try-catch` *blocks*

The first category describes *where* in the body of a method a `try-catch` block appears. We consider a `try-catch` block *tangled* if it is textually preceded or followed by a statement in the body of the method where it appears. Declarations and statements that appear textually before a `try-catch` block make it *tangled by prefix*. Accordingly, statements that appear after a `try-catch` block make it *tangled by suffix*. A `try-catch` block appearing within a loop is considered tangled by prefix and suffix, independently of the placement of other statements. A `try-catch` block whose `try` block surrounds the whole method body is considered *untangled*. Figure 7(a) presents an untangled `try-catch` block. The shaded `try-catch` block in Figure 7(b) is tangled by suffix because it is followed by a call to `n()`. The `try-catch` block in Figure 7(c) is tangled by prefix, because the call to method `n()` precedes it. Tangling of `try-catch` blocks impacts the selection of parts of the code where exceptions are thrown. It might also make it difficult for an aspect to simulate the flow of control after handler execution of the original implementation.

#### *6.1.2. Terminal ETC*

An exception-throwing statement is a statement within a `try` block, which can throw exceptions that some handler within the same method catches. ETC is the set of exception-throwing statements within the same `try` block. An exception-throwing statement is *terminal* if it is not followed by any statements in the `try` block. When referring to a `try-catch` block, we consider its ETC terminal if all of its exception-throwing statements are terminal. In the general case, terminal ETC includes a single terminal exception-throwing statement. However, there are special cases such as when a `try-catch` has multiple `if` statements. In Figure 7(d), the call to method `n()`, which throws exception `E`, is *non-terminal* because it is followed by a call to `p()`. On the other hand, the call to `p()` is terminal because it is the last statement of the `try` block. This factor impacts the choice of pointcut designator to be employed and influences aspectization, because it can make it difficult for aspects to simulate the control flow of the original program.

#### *6.1.3. Nesting of* `try-catch` *blocks*

A `try-catch` block can be also classified as *nested* or *non-nested*. A nested `try-catch` block is one that is contained within a `try` block. The shaded code snippet of Figure 7(e) is an example of nested `try-catch` block. Nesting of `try-catch` blocks can make it difficult to understand the flow of exceptions in a given context. Moreover, implementing nested `try-catch` blocks as advice may require these advice to be ordered, so that they mimic the behavior of the original code. This is necessary if some of these advice are associated with the same join points. We do



DOI: 10.1002/spe

```
class C0 {
 void m(){
  try{...}
  catch(E e){...}
 } //m()
} //C0
```
(a)

```
class C1 {
 void m(){
  try{...}
  catch(E e){...}
  n();
 } //m()
} //C1
```
(b)

```
class C2 {
 void m(){
  n(); // throws E
  try{...}
  catch(E e){...}
 } //m()
} //C2
```
(c)

```
class C3 {
 void m(){
  try{
   n(); //throws E
   p(); // throws E
  }catch(E e){...}
  q();  } //m()
} //C3
```
(d)

```
class C4 {
 void m(){
  try{
   try{...}
   catch(E1){...}
  }catch(E2 e){...}
 } //m()
} //C4
```
(e)

```
class C5 {
 void m(){
  int x = 0;
  try{ x = n();//throws E
  }catch(E e){ p(x); }
 } // m()
} //C5
```
(f)

```
class C6 {
 void m(){
  int x = 0;
  try{ x = n(); //throws E
  }catch(E e){x = p();}
  q(x);
 } // m()
} //C6
```
(g)

```
class C7 {
 void m(){
  try{ n();//throws E
  }catch(E e){
   log(e);
  }
  p(); } //m()
} //C7
```
(h)

```
class C8 {
 void m() throws E2{
  try{ n();//throws E1
  }catch(E1 e){
   throw new E2(e);
  }
 } // m()
} //C8
```
(i)

```
class C9 {
 void m(){
  try{ n();//throws E
  }catch(E e){
   return;
  }
 } // m()
} //C9
```
(j)

```
class C10 {
 void m(){
  for(int i=0;i++;i<10){
   try{ n();//throws E
   }catch(E e){break;}
  }
 } //m()
} //C10
```
(k)

```
class C11 {
 void m(){
  for(int i=0;i++;i<9){
   try{n();//throws E
   }catch(E e){
    continue;
   }
   o(); } } //m()
 } //C11
```
(l)

Figure 7. Examples of the categories of the proposed classification for exception-handling code.



DOI: 10.1002/spe

not consider nested a `try-catch` block that appears within a `catch` or `finally` block. Such a `try-catch` block is naturally part of the system's exceptional behavior. In other words, it does not need to be separately extracted to an error-handling aspect. Instead, its enclosing `catch` or `finally block`, as a whole, is extracted. The shaded `try-catch` block in the following code snippet is **not** nested for the purposes of aspectizing error handling. Instead, it is considered part of the outer `catch` block.

```
class C {
 void m(){
  try{...}
  catch(E1 e){
   try{...}
   catch(E2){...}
  } //catch
 } //m()
} //C
```

### *6.1.4. Dependency of exception handlers on local variables*

This category classifies exception handlers into two groups: those that do and those that do not depend on local variables. By 'local variables' we mean variables defined within the containing block of a given `try-catch` block. Dependency on local variables hinders aspectization, as the join point models of the most popular AO languages (AspectJ included) cannot capture information stored by these variables. If a handler reads the value of one or more local variables, we say that it is `context-dependent`. Moving such a handler to an aspect often requires the use of the Extract Method refactoring [33] to expose the variables as parameters of a method. Often, it is possible to avoid such refactoring by selecting the join point where the value is generated (e.g. the return value of a method) and saving it for later retrieval during error handling (e.g. in a table in the error-handling aspect). In Figure 7(f), the `catch` block reads the value of local variable `x`. If a handler performs assignments to local variables, we call it a **context-affecting** handler. In this case, more radical refactoring may be necessary. The resulting code often exhibits bad smells such as 'Temporary Field' [33]. The handler in Figure 7(g) performs an assignment to local variable `x`.

### *6.1.5. Flow of control after handler execution*

The fifth category describes how a `catch` block ends its execution. After the execution of a *masking* handler, control passes to the statement that textually follows the corresponding `try-catch` block. Figure 7(h) presents a masking handler. After its execution, method `p()` is invoked. A *propagation* handler finishes its execution by throwing an exception, as shown in Figure 7(i). In this case, control is transferred to the nearest handler that catches the exception. A *return* exception handler ends its execution with a `return` statement, as shown in Figure 7(j). After the handler returns, system execution resumes at the site where the handler's method was called. Finally, a handler declared inside a loop might affect its flow of control if it includes commands such as `break` and `continue`. In the former case, we say that the handler is a *loop escape* handler. In the latter, we call it a *loop continuation* handler. Figures 7(k) and (l) present examples of loop escape and loop continuation handlers. If a `catch` block includes alternation commands (e.g. `if` statements) and,



DOI: 10.1002/spe

as a consequence, can end its execution in different ways, we assume the worst case in terms of ease of aspectization. We consider loop continuation handlers the hardest to aspectize, followed by loop escape, masking, return, and propagation handlers.

Flow of control after handler execution affects several design choices related to the aspectization of exception handling. For example, propagation handlers can be easily implemented using *after* advice, whereas masking and return handlers have to be implemented using *around* advice, as they stop the propagation of the exceptions they catch. Moreover, it is generally straightforward to simulate the flow of control of the original program (i.e. the program before error handling is aspectized) for a tangled `try-catch` block if its handlers are all propagation or return. These types of handlers ignore the code that follows the `try-catch` block, and thus the fact that it is tangled. However, the same does not apply to masking handlers.

### 6.2. A catalog of exception-handling scenarios

In this section, we describe several exception-handling scenarios representing recurring situations in software development, in the specific context of Java-like languages. We have derived these scenarios by analyzing combinations of the factors of the proposed classification (Section 6.1) and the ways in which these combinations affect the task of extracting error handling to aspects. It is important to stress that this is not a refactoring catalog, as we do not explain the mechanics of the actual extraction process. There are already some refatoring catalogs [13,34] in the literature which provide guidance on the extraction of exception-handling code to an aspect. Instead, we use these scenarios to illustrate the cases where aspectizing error handling is advantageous and where it is not.

Table VIII consolidates the most important scenarios we have. Each scenario is named after its distinctive features. For example, the first scenario of Table VIII is simply named 'Untangled handlers', because it deals with untangled `try-catch` blocks. The second one is named 'Tangled, Non-masking handlers', because it refers specifically to `try-catch` blocks that include propagation or return handlers, but no other complicating factors. To avoid repetition, if a category is marked more than once in the same row, for example, return ('r') and propagation ('p') handlers marked, the row represents more than one scenario and an OR semantics is adopted. For simplicity, hereafter we use the term 'scenario' to refer to all the scenarios that each row represents. For example, the Untangled Handler scenario describes situations where a `try-catch` block is untangled, independently of nesting and terminal ETC. In addition, the corresponding `catch` block does not depend on local variables and can end its execution by masking, propagating an exception, or returning.

The rightmost column of Table VIII indicates whether it is beneficial ('Yes') or harmful ('No') to modularize exception handling with aspects in a given scenario. In some rows, it 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 considered aspectization to be beneficial in a scenario if: (i) the code, after aspectization, does not generally exhibit bad smells; (ii) little or no *a priori* redesign is necessary; (iii) the solution we have used to extract the handlers to aspects applies to most or all of the instances of the scenario; and (iv) the implementation overhead introduced by aspectization is not very large, that is ideally, for each `try-catch` block extracted to an aspect, at most one new advice should be created.



DOI: 10.1002/spe

Table VIII. Exception-handling scenarios according to the proposed classification.

| Scenario | Tangled `try-catch` block | | Nested `try-catch` block | | Terminal exception-throwing code | | Handler depends on local vars. | | | Flow of control after handler execution | | | | | Scr. | Should extract? |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | yes | no | yes | no | yes | no | read | write | no | m | p | r | le | lc | | |
| Untangled Handler | | X | X | X | X | X | | | X | X | X | X | | | 0–1 | Yes |
| Tangled, Non-Mask. Handler | X | | | X | X | X | | | X | | X | X | | | 0 | Yes |
| Nested, Non-Mask. Handler | X | | X | | X | X | | | X | | X | X | | | 1 | Yes |
| Tangled Handler, Term. ETC | X | | | X | X | | | | X | X | | | | | 0 | Yes |
| Nested Handler, Term. ETC | X | | X | | X | | | | X | X | | | | | 1 | Yes |
| Block Handler | X | | X | X | | X | | | X | X | | | | | 2–3 | Depends |
| Loop Esc. Handler | X | | X | X | X | X | | | X | | | | X | | 2–3 | Depends |
| Loop Cont. Handler | X | | X | X | X | X | | | X | | | | | X | 2–3 | Depends |
| Context-Dependent Handler | X | X | | X | X | X | X | | | X | X | X | X | X | 2–4 | Depends |
| Nested, Context-Dependent Handler | X | X | X | | X | X | X | | | X | X | X | X | | 3–5 | No |
| Context-Affecting Handler | X | X | X | X | X | X | X | X | | X | X | X | X | X | 3–8 | No |

Additionally, considering the various factors that influence aspectization, we have devised a simple scoring system to more objectively judge whether a given scenario is beneficial or harmful. The scores for each scenario appear in the column labeled 'Scr.'. They can be obtained by summing the score associated with each factor that is (potentially) present in each scenario. Table IX provides a key for calculating the scores. Basically, a score between 0 and 1 indicates a 'Yes' scenario, a score between 2 and 4 is a 'Depends' scenario, and a score of 5+ is a 'No' scenario. For example, scenario Untangled Handler has a score that ranges between 0 and 1. Its only complicating factor is nesting, whose score is 1, and even then it does not appear in all instances of the scenario (hence the range, instead of a fixed value).

*Untangled Handler*: Since the `try-catch` block is not tangled, it is possible to select the entire execution of the method as the join point of interest. Moreover, it is straightforward to extract handlers to an aspect. It does not matter whether the ETC is terminal or not. Flow of control after handler execution is not a problem, as long as it is not loop escape or loop continuation. If there are nested `try-catch` blocks, it may be necessary to rank the handler advice if some of them are associated with the same join points.

*Tangled, Non-Masking Handler*: In this scenario, the `try-catch` block is tangled but it is possible to specify a pointcut that captures the execution of the whole method and associates the handler advice with it. In the original code, the handler ends its execution by either returning or throwing an exception; thus, ignoring whatever comes after the `try-catch` block. Hence, the tangling does not matter because the code that textually follows the `try-catch` block is never executed when an exception is thrown from the `try` block. The ETC does not matter in this case because the execution of the whole method is the join point of interest.



DOI: 10.1002/spe

Table IX. Scores for the factors that hinder the aspectization of exception handling.

| Score | Description | Factors |
|---|---|---|
| 1 | Hinders aspectization in a general and non-limiting way | Nesting of `try` blocks |
| 2 | Some priori refactoring or a considerable implementation overhead are usually necessary | Combinations of non-term. ETC, masking handler, and tangled `try-catch` block |
| | However, the process involves well-known refactorings and results in a generic solution that may or may not exhibit bad smells | Handler that reads from local variables<br>Loop escape handlers<br>Loop continuation handlers |
| 3 | Refactoring is almost always necessary and the solutions that apply to each instance of the scenario are often context-dependent | Handlers that write to local variables |

```
// within class C
 void m(){
  try{
   n();
   p(); // throws E
  }catch(E e){log(e);}
  q();
 } //m()
```

```
// within aspect A
 pointcut callP():
  call(* C.p()) &&
  withincode(* C.m());
 around() : callP() {
  try { proceed();}
  catch(E e) {log(e);}
 } //advice
```

Figure 8. Tangled Handler, Terminal ETC scenario and an AspectJ implementation.

*Nested*, *Non-Masking Handler*: This scenario bears strong resemblance to the previous one, but the nesting of `try-catch` blocks introduces some complications that hinder aspectization. Since the `try-catch` block is nested, it may be necessary to order handler advice associated with the same join points. Moreover, many (3+) levels of nesting combined with tangled `try-catch` blocks often result in complex code and special care should be taken in order to avoid the introduction of subtle bugs and changing the system behavior. In spite of these complications, aspectization is still beneficial in this scenario. In the general case, no *a priori* refactoring has to be applied to the original code and the aspectized code often exhibits good structuring that reflects the original design of the exceptional behavior.

*Tangled Handler*, *Terminal ETC*: It is easy to aspectize exception handling in this case because it is possible to directly associate a handler advice with the exception-throwing statement. Since the latter is terminal and the handler has a termination behavior, after execution of the aspectized handler, control passes to the statement that follows the `try-catch` block in the original implementation. The code snippet in Figure 8 presents a simple instance of this scenario and a possible implementation of the handler as part of an aspect.

On the left-hand side, the call to `p()` is a terminal exception-throwing statement and the handler has a termination behavior. The right-hand side defines a pointcut that selects calls to method `p()` made within the code of method `m()`. An *around* advice implements the handler. The `proceed()` statement is defined by AspectJ and executes the selected join point from an *around* advice, in this



DOI: 10.1002/spe

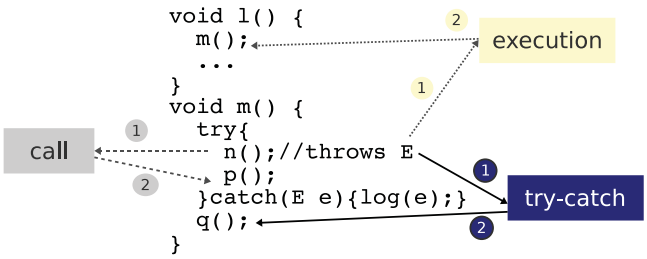


Figure 9. Control flow in the presence of block handlers.

example, the call to `p()`. After exception handling, control will return to the following statement, that is the call to `q()`. This mimics the behavior of the original program.

*Nested Handler*, *Terminal ETC*: Nesting makes it slightly harder to modularize error handling using aspects in this scenario, when compared with the previous one. In general, though, it does not negatively affect the quality of the final code. Therefore, we believe that aspectization is beneficial in this scenario.

*Block Handler*: Moving the error-handling code to an aspect in this scenario is hard because of the combination of non-terminal ETC, tangled `try-catch` block, and a handler that has a masking behavior. These factors make it difficult to mimic the behavior of the original code using an aspectized handler. Figure 9 presents an example of this scenario. It illustrates the control flow of a simple program in a scenario where an exception is raised, control is transferred to a block handler, and, after handling, normal execution continues. It also shows what control flow would be like if a handler advice were employed, instead of the regular `try-catch`, and how two different pointcut designators, `execution` and `call`, would impact the control flow.

Figure 9 depicts a situation where a call to a method `n()` within a method `m()` raises an exception and control is transfered to the handler. If we refactor the handler to an advice and associate the latter with the call to method `n()`, after exception handling the call to method `p()` will be executed. However, in the original implementation, control should be passed to the statement following the `try-catch` block, the call to `q()`. On the other hand, if we associate the handler advice with the execution of method `m()`, control will return to `m()`'s caller after exception-handling (method `l()`). This implies that the call to `q()` will not be executed. The bottom line is: we would like to associate a handler advice to a block containing more than one statement, just like a `try` block, instead of a single statement or a whole method. Hence, we call this scenario *block handler*.

Recently, some solutions to this problem have appeared in the literature [20,30]. However, to the best of our knowledge, no AO language in widespread use provides constructs for implementing block handlers. A possible workaround to this limitation is to extract the code within the `try` block to a new method and associate a handler advice with the execution of the extracted method. This solution has drawbacks; however: (i) it modifies the original design of the system's normal behavior; (ii) it might result in methods that do not make sense by themselves; and (iii) it is not effective in all cases because it is not always possible to extract a piece of code to a new method [33]. In general, in this scenario, aspectization is only beneficial if it is possible to extract the code within the `try` block to a new method that could as well have been created by the developers of the system.



DOI: 10.1002/spe

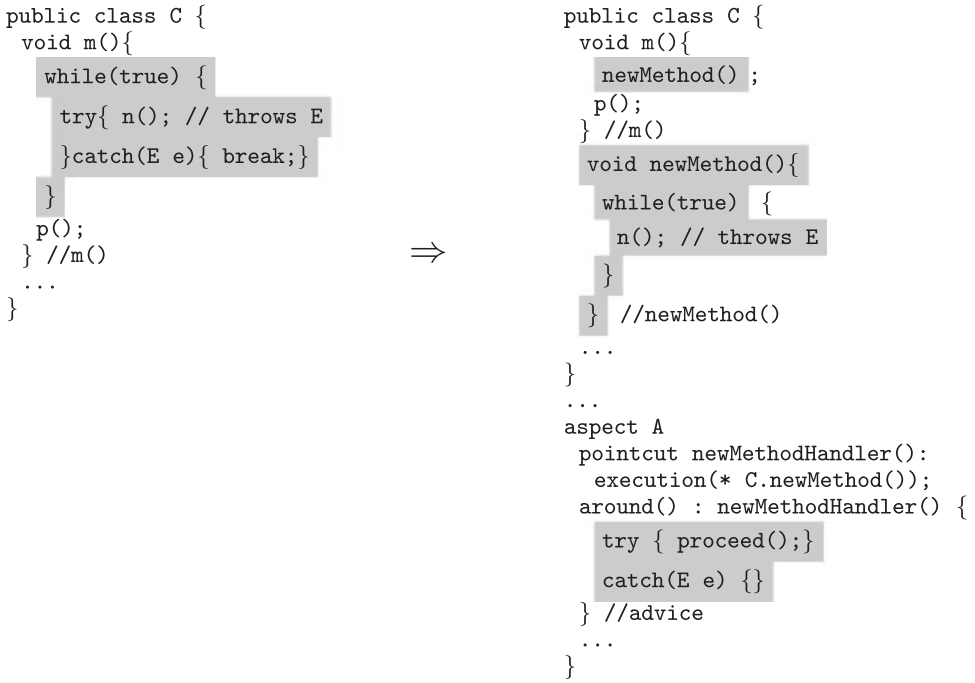


Figure 10. Extraction of a loop escape handler to an aspect.

*Loop Escape Handlers*: In this scenario, the problem is that a `break` statement appears inside a `catch` block and is part of the error-handling concern. However, the loop where the corresponding `try-catch` block appears is part of the normal behavior of the system. Loop escape handlers are similar to block handlers in the sense that it is not possible to extract the exception handler to an aspect and mimic the original application control flow without some *a priori* refactoring. Therefore, the same pros and cons apply.

Figure 10 presents an example of the extraction of a loop escape handler. First, it is necessary to extract the entire loop to a new method (`newMethod()` on the right-hand side of the figure). Afterwards, handler extraction proceeds as in the previous cases, with the difference that it is necessary to remove the `break` statement from the handler when the latter is moved to an advice (since the handler is not within a loop anymore). The handler advice can then be associated with either the call to the new method or with its execution. In this manner, when an exception is raised within the new method, it is handled by the advice and, afterwards, control is returned to the statement following the call to the new method (the call to `p()` on the right-hand side of the figure).

*Loop Continuation Handlers*: A loop continuation handler is very similar to a loop escape handler, with the difference that there is a `continue` statement in the `catch` block. This small difference impacts the way these handlers are extracted to aspects. Unlike a `break` statement, which jumps out of the enclosing loop, `continue` skips the rest of the loop body and goes back to check the loop's condition. To emulate this control flow with an aspectized handler, *a priori* refactoring



DOI: 10.1002/spe

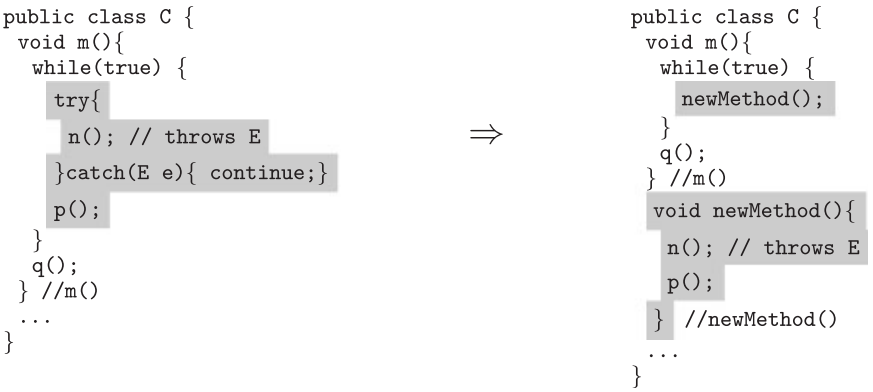


Figure 11. Exposing the join point of interest in a Loop Continuation Handler scenario.

must also be performed. More specifically, at runtime, after the execution of the aspectized handler, control should return to the loop condition. To achieve this behavior, instead of extracting the entire loop to a new method, only the code within the loop must be extracted. Figure 11 presents an example. We omit the resulting handler advice because it would be identical to the one presented in Figure 10. On the right-hand side of Figure 11, associating the handler advice with the execution of method `newMethod()` means that, after it handles an exception raised in `newMethod()`, control returns to the loop condition within method `m()`. This is precisely the behavior of the original `m()` method.

*Context-Dependent Handler*: Aspectizing handlers that use local variables of the enclosing method in AspectJ is difficult because the language does not make it possible for advice to access the local variables visible at a join point of interest. To try to address this problem, we have analyzed four general ways of exposing a local variable to an advice: (#1) to transform the variable into a field of the enclosing class; (#2) to capture some use of the variable and store this information for later retrieval; (#3) to extract the piece of code where the variable is used to a new method and expose this variable as a parameter of the extracted method; and (#4) to extend the class that defines the thrown exception so that its instances can store the values of the variables. We have concluded that none of them is ideal. Table X lists the drawbacks of each one of these solutions.

The decision of refactoring the handler to an advice in this scenario depends on: (i) whether it is possible to expose the values of the local variables of interest in a way that is conceptually reasonable and does not impose a large implementation overhead; and (ii) whether it would be beneficial to aspectize the handler if we ignored the dependency on the local variables.

*Nested Context-Dependent Handler*: Combinations of nested `try-catch` blocks and handlers that read values of local variables are, in general, difficult to aspectize and not worth the effort. Code adhering to this scenario is often very complex and includes handlers accessing local variables defined various nesting levels above. In order to modularize exception handling, it is almost always necessary to intensively refactor the OO implementation. The resulting AO code often includes a sensible implementation overhead. Moreover, the design of the normal code is usually altered beyond recognition.

*Context-Affecting Handler*: In this scenario the `catch` block performs assignments to local variables of the containing method. Amongst the four solutions we have presented for exposing



DOI: 10.1002/spe

Table X. Limitations of various approaches to expose local variables.

| Approach | Problems |
|---|---|
| Transform variable into field | Conceptually bad because local variables only make sense in the method where they appear<br>Must deal explicitly with multi-threaded programs<br>Suffers from the 'Temporary Field' bad smell |
| Capture assignment to variable | Not always possible to capture the join point of interest<br>Imposes an additional implementation overhead<br>Must also deal explicitly with multi-threaded programs |
| Extract a new method | Modifies the design of the system's normal behavior<br>Might result in methods that do not make sense<br>Not effective in all cases |
| Extend exception class | The resulting exception classes might include information specific to certain-handling strategies<br>Requires non-local changes to the program |

local variables to advice, only the first one makes it possible for the aspect to change the value of the variable without duplicating code. As pointed out previously, this solution has some severe problems. Another possibility is to transform the local variable into an array of its type comprising a single element and then use the fourth approach described above to store this array. In this manner, the handler becomes capable of modifying the variable directly. Besides the limitations highlighted above, this approach is obviously a hack that uses arrays as a means to simulate a pass-by-result semantics.

The task of moving a `catch` block to an advice is described by the Extract Fragment to Advice [13] refactoring, an AO adaptation of the Extract Method refactoring. Hence, similar constraints apply. In general, it is not possible to extract a code snippet to a new method if this snippet performs assignments to more than one local variable of the containing method [33]. The same holds when one tries to extract a code snippet to an advice. Thus, as a rule, it is harmful to aspectize exception handling in this scenario.

Table XI summarizes the general strategies that should be applied in each scenario. The second and third columns of the table specify the type of advice and the type of pointcut to use, respectively. In the latter case, `call` is a shorthand for `call` and `withincode`. The fourth column indicates whether it is necessary to order handler advice associated with the same join points in each scenario. In some rows, this column is marked "maybe", meaning that each instance of the scenario must be independently analyzed. The fifth column indicates the scenarios where the join point of interest is a block (sequence of statements). The sixth points out the scenarios where it is necessary to expose local variables read or written by the exception handlers.

## 7. LIMITATIONS OF THIS STUDY

This study, like any other empirical study, has several limitations. First, it does not attempt to assess how different structuring strategies for moving exception handlers to aspects affect the results.



DOI: 10.1002/spe

Table XI. General strategies for structuring error handling with aspects in each scenario.

| Scenario | Advice | Type of Pointcut | Order advice | Block join point | Expose variables |
|---|---|---|---|---|---|
| Untangled Handler | *after* or *around* | *execution* | no | no | no |
| Tangled, Non-Masking Handler | *after* or *around* | *execution* | no | no | no |
| Nested, Non-Masking Handler | *after* or *around* | *execution* | yes | no | no |
| Tangled Handlers, Terminal ETC | *around* | *call* | no | no | no |
| Nested Handler, Terminal ETC | *around* | *call* | yes | no | no |
| Block Handler | *around* | *call or execution* | no | yes | no |
| Loop Escape Handler | *around* | *call* or *execution* | no | yes | no |
| Loop Continuation Handler | *around* | *call* or *execution* | no | yes | no |
| Context-Dependent Handler | *after* or *around* | *call* or *execution* | no | maybe | yes |
| Nested, Context-Dependent Handler | *after* or *around* | *call* or *execution* | yes | maybe | yes |
| Context-Affecting Handler | *around* | *call* or *execution* | maybe | maybe | yes |

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 the execution time of 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. For example, it is not necessary to soften exceptions in Eos [35], an AO extension to C#, because C# does not have checked exceptions. 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 [36] it is possible to associate handlers with exceptions raised and not handled within loops. Moreover, one of the extensions to AspectJ proposed by the developers of the abc AspectJ compiler [37] allows the selection of `throw` statements as join points. In some situations, this feature makes it possible to easily aspectize termination handlers (Section 6.1) that are nested or tangled in the original code.

Not all possible strategies for implementing the exceptional behavior of the five target systems are covered. In all of them the handlers are implemented exclusively by means of `catch` blocks. However, more complex applications may include methods and fields which are specific to the implementation of the exceptional behavior. Moving these additional elements to aspects would probably affect the quality attributes of the refactored systems.



DOI: 10.1002/spe

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 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 [15,16,19,20]. The metrics of Section 5.1, on the other hand, are not widely known. Nevertheless, they directly quantify elements of interest in the program, from a reuse standpoint. In fact, despite the well-known limitations of these metrics [23], they complement each other and are very useful when analyzed together. In addition, for every possible metrics suite there will be some dimensions that will remain uncovered. Future case studies can use additional metrics and assess the aspectization of exception handling using different modularity dimensions.

## 8. RELATED WORK

The related work can be categorized into four groups: (1) assessment and application of AO techniques to modularize error-handling code; (2) evaluation of AO techniques to structure other fault tolerance mechanisms, most notably transactional execution; (3) best practices and anti-pattern catalogs for the design and implementation of error handling; and (4) approaches to refactor exception-handling code to aspects.

### 8.1. AOP and exception handling

Although introductory texts [8,9] 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 [4] 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 Lippert and Lopes study was an 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 the extrapolation of its results 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 [7]. 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 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.



DOI: 10.1002/spe

Recently, Cacho and colleagues [28] devised a domain-specific extension of AspectJ, EJFlow, whose goal is to modularize error-handling code. EJFlow includes mechanisms to associate exception handlers with exception flows in a Java program, providing developers with local control over global exceptions. In addition, it leverages anchored exception declarations [5] as a workaround to some of the maintenance problems inherent to Java-based exception handling. The authors have shown that the use of EJFlow yields programs with better quality attributes, when compared with purely OO and AO, AspectJ-based, approaches. Hoffman and Eugster [20] propose the concept of explicit join point (EJP) as a means to reduce the amount of obliviousness [38] of AOP. The authors believe that this approach results in improved maintenance and understandability, at the cost of losing some of the textual separation yielded by AOP. In fact, the authors have shown that EJPs can circumvent some of the limitations of AspectJ, in particular for the modularization of Block Handlers (Section 6.2). The evaluation of their approach consisted of comparing three versions of three different industrial strength applications: (i) an OO version; (ii) an AO version where exception handling was extracted to aspects in AspectJ; and (iii) an AO version where exception handling was extracted to aspects using an extension of AspectJ that includes EJPs. These two works represent important improvements over AspectJ and are orthogonal to ours. We believe they could be greatly enhanced by in-depth empirical studies such as the one described in this paper.

Coelho *et al.* [17] conducted an empirical study with the goal of understanding the impact of AspectJ in program reliability. In particular, the authors were interested in bugs that are the consequences of convoluted exception flow caused by AO constructs. This study targeted OO and AO versions of three different applications, analyzing the impact of several maintenance scenarios over their reliability. The authors concluded that the AO versions of the systems became less reliable as a consequence of the maintenance scenarios, mainly because AOP obscures exception flow and complicates it in non-obvious ways. The authors then documented the most important bug patterns they discovered as a pattern language [39].

### 8.2. AOP and fault tolerance

One of the first studies of the applicability of AOP for developing dependable systems has been conducted by Kienzle and Guerraoui [10]. 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 instead of transactions to deal with errors; (ii) we substantiate our conclusions with measurements based on a metrics suite for AO software, instead of examples; (iii) we do not address concurrency; and (iv) our study is more general and based on a varied set of applications with diverse error-handling strategies.

Soares and his colleagues [11] 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.



DOI: 10.1002/spe

Kienzle and Gélineau [40] use aspects to ensure the ACID (atomicity, consistency, isolation, and durability) for transactional objects. The authors built 10 base aspects that implement different concurrency control and recovery strategies. They report that, in order to actually use these aspects, careful configuration is necessary because the aspects are not independent and their composition results in conflicts. This work is different from ours because it does not consider exception handling as an alternative to implement error recovery.

### 8.3. Best and worst practices for error handling

Some authors describe their experience in the application of exception handling to real OO software systems in the form of best practices [41] and anti-pattern [42,43] catalogs. An early paper by Cargill [42] describes some of the problems that often happen in C++ programs due to the complex control flow that exception handling creates. Reimer and Srinivasan [44] have analyzed how developers have applied the exception-handling mechanism of Java in some large-scale Java Enterprise Edition applications. They comment on some of the most common bad programming practices and hint at some approaches that can help developers in alleviating these practices (e.g. static analysis tools). Weimer and Necula [31] present several examples of situations where error handling complicates code that frees resources, creating complex control flow that programmers do not understand completely.

Doshi [41] presents two small catalogs of best practices: one for specifying APIs whose services can throw exceptions and another one for developing the client code of these APIs (the actual exception handlers). A recent paper by McCune [43] presents a list of error-handling anti-patterns. Some of them are well accepted (e.g. 'catch and ignore' and 'throwing `Exception`'), whereas others seem to be based on the personal opinions of the author (e.g. 'log and throw'). Best practices and anti-pattern catalogs provide guidelines on what exception handlers should and should not do. Our work complements these approaches by providing design guidance to developers using AOP techniques to enhance the modularity of error-handling code.

### 8.4. Refactoring exception handlers to aspects

In the recent years, several works have appeared which try to amass design/implementation time-tested knowledge regarding AO software development activities [13,34,45,46]. Many authors have devoted attention to developing refactoring catalogs [13,34,47–49] for AO software. A few of them [34,48] include specific procedures for moving exception-handling code to aspects.

Laddad presents the 'Extract Exception-Handling' refactoring. The refactoring centers around the effects of using it to extract trivial error-handling code, but does not explain when it is useful (or possible) to apply it in practice. This refactoring was later implemented in a tool developed by Binkley *et al.* [49]. Cole and Borba describe a set of behavior-preserving programming laws that can be combined in order to specify a refactoring that works along the same lines as 'Extract Exception Handling'. These laws include preconditions that indicate when it is possible to apply them. While refactorings targeting error-handling code concentrate on the mechanics of moving a `try-catch` block to an aspect, the work we present identifies situations where this is beneficial and where it is not.



DOI: 10.1002/spe

## 9. CONCLUDING REMARKS

In this paper, we have 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 have found that a combination of several factors determines if the use of AOP to separate exception-handling code and normal application code can be beneficial. 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. For exception handling, *ad hoc* aspectization using AspectJ is beneficial only in simple scenarios. The main contributions of this work are: (i) a substantial improvement, based on experience acquired from refactoring five 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; (iii) an initial assessment of the effects of aspect interaction when exception handling gets in the mix; and (iv) an analysis of the impact of aspects on the reuse of exception handling code.

The general conclusion that can be drawn from this work is that AOP does not fix poor designs. In other words, in OO systems where exception-handling code is already well structured, AOP further improves the structure by completely separating the system's normal and exceptional activities. However, for systems with very complex and/or poorly structured exception-handling code, AOP merely adds insult to injury and worsens the overall quality of the system. We believe that a better approach than extracting error handling to aspects is to take error-handling aspects into account from the start, throughout all software development phases. In this manner, the more difficult scenarios can be avoided 'by construction', thus improving the overall quality of the code. When this is not possible, though, this work provides design guidance to make the process of aspectizing exception-handling code straightforward. In particular, the catalog of Section 6.2 covers a wide range of scenarios and can certainly be of used for both AOP novices and experienced developers of AO systems.

Since exception handlers can be identified by syntactically analyzing the code of an application, we believe that a significant part of the work necessary to refactor it to aspects can be automated by a tool. Therefore, in the future, we intend to devise a static analysis, based on the presented catalog scenario (Section 6.2), to detect situations where it is beneficial to modularize error-handling code using aspects. We hope that this static analysis will serve as a starting point to the construction of a tool to (semi-)automatically refactor exception handler code to aspects. We also intend to study workarounds for the harmful scenarios, possibly by proposing alternate designs that can be automatically introduced in a system through tool-assisted refactoring.

#### ACKNOWLEDGEMENTS

We would like to thank the anonymous referees for the insightful comments which helped to improve this paper. Fernando is partially supported by CNPq/Brazil, grants 308383/2008-7, 481147/2007-1 and 550895/2007-8, and by the National Institute of Science and Technology for Software Engineering (INES** ), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08. Eduardo is supported by CAPES/Brazil.

**http://www.ines.org.



DOI: 10.1002/spe

Cecília is partially supported by CNPq/Brazil, grants 301446/2006-7 and 484138/2006-5. Hítalo is supported by FACEPE/Brazil.

**REFERENCES**


1. Goodenough JB. Exception handling: Issues and a proposed notation. *Communications of the ACM* 1975; **18**(12):683–696.
2. Cristian F. A recovery mechanism for modular software. *Proceedings of the 4th ICSE*, Munich, Germany, 1979; 42–51.
3. Fetzer C, Högstedt K, Felber P. Automatic detection and masking of nonatomic exception handling. *IEEE Transactions on Software Engineering* 2004; **30**(8):547–560.
4. Lippert M, Lopes CV. A study on exception detection and handling using aspect-oriented programming. *Proceedings of the 22nd ICSE*, Limerick, Ireland, June 2000; 418–427.
5. van Dooren M, Steegmans E. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. *Proceedings of 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, U.S.A., October 2005; 455–471.
6. Garcia A, Rubira C, Romanovsky A, Xu J. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software* 2001; **59**(2):197–222.
7. Anderson T, Lee PA. *Fault Tolerance: Principles and Practice* (2nd edn). Springer: Berlin, 1990; 302.
8. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier J-M, Irwin J. Aspect-oriented programming. *Proceedings of the 11th ECOOP* (*Lecture Notes in Computer Science*, vol. 1271). Springer: Berlin, 1997; 220–242.
9. Laddad R. *AspectJ in Action*. Manning, 2003; 512.
10. Kienzle J, Guerraoui R. AOP: Does it make sense? The case of concurrency and failures. *Proceedings of 16th European Conference on Object-Oriented Programming*, Bonn, Germany, June 2002; 37–61.
11. Soares S, Laureano E, Borba P. Implementing distribution and persistence aspects with AspectJ. *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, U.S.A., 2002; 174–190.
12. Garcia A, SantAnna C, Figueiredo E, Kulesza U, de Lucena CJP, von Staa A. Modularizing design patterns with aspects: A quantitative study. *Transactions on Aspect-Oriented Software Development I* 2006; **1**:36–74.
13. Monteiro MP, Fernandes JM. Towards a catalog of aspect-oriented refactorings. *Proceedings of the 4th AOSD*, Chicago, U.S.A., March 2005; 111–122.
14. Castor Filho F, Guerra PA de C, Pagano VA, Rubira CMF. A systematic approach for structuring exception handling in robust component-based software. *Journal of the Brazilian Computer Society* 2005; **10**(3):5–19.
15. Greenwood P, Bartolomei T, Figueiredo E, Dosea M, Garcia A, Cacho N, SantAnna C, Soares S, Borba P, Kulesza U, Rashid A. On the impact of aspectual decompositions on design stability: An empirical study. *Proceedings of the 21st European Conference on Object-Oriented Programming*, Berlin, Germany, July 2007; 176–200.
16. Cacho N, Sant'Anna C, Figueiredo E, Garcia A, Lucena TBC. Composing design patterns: A scalability study of aspect-oriented programming. *Proceedings of 5th ACM Conference on Aspect-Oriented Software Development*, March 2006; 109–121.
17. Coelho R, Rashid A, Garcia A, Ferrari FC, Cacho N, Kulesza U, von Staa A, de Lucena CJP. Assessing the impact of aspects on exception flows: An exploratory study. *Proceedings of the 22nd European Conference Object-Oriented Programming*, Paphos, Cyprus, Julho, 2008; 207–234.
18. Figueiredo E, Cacho N, Monteiro CSM, Kulesza U, Garcia A, Soares S, Ferrari F, Khan S, Castor Filho F, Dantas F. Evolving software product lines with aspects: An empirical study on design stability. *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering*, Leipzig, Germany, May 2008; 261–270.
19. Godil I, Jacobsen H. Horizontal decomposition of prevlayer. *Proceedings of CASCON 2005*, Toronto, Canada, 2005; 83–100.
20. Hoffman KJ, Eugster P. Towards reusable components with aspects: An empirical study on modularity and obliviousness. *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008; 91–100.
21. Chidamber S, Kemerer C. A metrics suite for oo design. *IEEE Transactions on Software Engineering* 1994; **20**(6):476–493.
22. Tigris. Aopmetrics home page, 2005. Avaialble at: http://aopmetrics.tigris.org [10 September 2009].
23. Eaddy M, Aho A, Murphy GC. Identifying, assigning, and quantifying crosscutting concerns. *Proceedings of the ICSE'2007 Workshop on Assessment of Contemporary Modularization Techniques*, Minneapolis, MN, May 2007; 1–6.
24. Mezini M, Ostermann K. Conquering aspects with caesar. *Proceedings of the 2nd AOSD*, Boston, U.S.A., 2003; 90–99.
25. Tarr P *et al.* N degrees of separation: Multi-dimensional separation of concerns. *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, U.S.A., May 1999; 107–119.
26. Hannemann J, Kiczales G. Design pattern implementation in Java and AspectJ. *Proceedings of 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, U.S.A., November 2002; 161–173.
27. Bartolomei TT. On modularity assessment of aspect-oriented software. *Master's Thesis*, Kiel University of Applied Sciences, Kiel, Germany, October 2006.




DOI: 10.1002/spe

28. Cacho N, Castor Filho F, Garcia A, Figueiredo E. Ejflow: Taming exceptional control flows in aspect-oriented programming. *Proceedings of the 7th ACM Conference on Aspect-Oriented Software Development*, Brussels, Belgium, March 2008; 72–83.
29. Castor Filho F, Cacho N, Figueiredo E, Maranhao R, Garcia A, Rubira C. Exceptions and aspects: The devil is in the details. *Proceedings of the 14th SIGSOFT FSE*, Portland, U.S.A., November 2006; 152–162.
30. Akai S, Chiba S, Nishizawa M. Region pointcut for Aspectj. *Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Charlottesville, U.S.A., 2009; 43–48.
31. Weimer W, Necula GC. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems* 2008; **30**(2):1–51.
32. Frakes WB, Kang K. Software reuse research: Status and future. *IEEE Transactions on Software Engineering* 2005; **31**(7):529–536.
33. Fowler M. *Refactoring*: *Improving the Design of Existing Code*. Addison-Wesley: Reading, MA, 1999; 464.
34. Cole L, Borba P. Deriving refactorings for Aspectj. *Proceedings of the 4th ACM Conference on Aspect-Oriented Software Development*, Chicago, U.S.A., March 2005; 123–134.
35. Rajan H, Sullivan K. Eos: Instance-level aspects for integrated system design. *Proceedings of Joint 9th European Conference on Software Engineering/11th SIGSOFT Symposium on Foundations of Software Engineering*, Helsinki, Finland, September 2003; 291–306.
36. Harbulot B, Gurd JR. A join point for loops in Aspectj. *Proceedings of the 5th ACM Conference on Aspect-Oriented Software Development*, Bonn, Germany, March 2006; 63–74.
37. Avgustinov P, Christensen AS, Hendren LJ, Kuzins S, Lhoták J, Lhoták O, de Moor O, Sereni D, Sittampalam G, Tibble J. abc: An extensible Aspectj compiler. *Proceedings of 4th ACM Conference on Aspect-Oriented Software Development*, Chicago, U.S.A., March 2005; 87–98.
38. Filman R, Friedman D. Aspect-Oriented programming is quantification and obliviousness. *Proceedings of the OOPSLA'2000 Workshop on Advanced Separation of Concerns*, Minneapolis, U.S.A., October 2000; 1–7.
39. Coelho R, Rashid A, Kulesza U, von Staa A, Lucena C, Noble J. Exception handling bug patterns in aspect oriented programs. *Proceedings of the 15th Conference on Pattern Languages of Programs*, Chicago, U.S.A., October 2008; 1–19.
40. Kienzle J, Gélineau S. Ao challenge—Implementing the acid properties for transactional objects. *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, Bonn, Germany, 2006; 202–213.
41. Doshi G. Best practices for exception handling, 2003. Available at: http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html [10 September 2009].
42. Cargill T. Exception handling: A false sense of security. *C++ Report*, 6(9), 1994.
43. McCune T. Exception-handling antipatterns, 2006. Available at: http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html [10 September 2009].
44. Reimer D, Srinivasan H. Analyzing exception usage in large Java applications. *Proceedings of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems*, Darmstadt, Germany, July 2003; 10–19.
45. Castor Filho F, Garcia A, Rubira CMF. Extracting error handling to aspects: A cookbook. *Proceedings of 23rd International Conference on Software Maintenance*, Paris, France, October 2007; 134–143.
46. Garcia A *et al.* Modularizing design patterns with aspects: A quantitative study. *Proceedings of the 4th ACM Conference on Aspect-Oriented Software Development*, Chicago, U.S.A., March 2005; 3–14.
47. Hanenberg S, Oberschulte C, Unland R. Refactoring of aspect-oriented software. *Proceedings of 4th Net.ObjectDays Conference*, Erfurt, Germany, September 2003; 19–35.
48. Laddad R. Aspect-Oriented refactoring, parts 1 and 2, 2003. The Server Side. Available at: www.theserverside.com [10 September 2009].
49. Binkley D, Ceccato M, Harman M, Ricca F, Tonella P. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering* 2006; **32**(9); 698–717.



DOI: 10.1002/spe