

# A Quantitative Study on the Aspectization of Exception Handling

Fernando Castor Filho\* and Cecília Mary F. Rubira\*\*

Institute of Computing - State University of Campinas  
P.O. Box 6176. CEP 13083-970, Campinas, SP, Brazil.  
{fernando, cmrubira}@ic.unicamp.br

Alessandro Garcia\*\*\*

Computing Department - Lancaster University  
South Drive, InfoLab 21, LA1 4WA, Lancaster, UK.  
garciaa@comp.lancs.ac.uk

**Abstract.** It is usually assumed that the implementation of exception handling can be better modularized by the use of aspect-oriented programming (AOP). However, the trade-offs involved in using AOP with this goal are not yet well-understood. To the best of our knowledge, no work in the literature has attempted to assess whether AOP really promotes an enhancement in well-understood quality attributes other than separation of concerns, when used for modularizing non-trivial exception handling code. This paper presents a quantitative study of the adequacy of aspects for modularizing exception handling code. The study consisted of refactoring part of a real object-oriented system so that the code responsible for handling exceptions was moved to aspects. We employed a suite of metrics to measure quality attributes of the original and refactored systems, including coupling, cohesion, and conciseness. We found that AOP improved separation of concerns between exception handling code and normal application code. However, contradicting the general intuition, the aspect-oriented version of the system did not present significant gains for any of the four size metrics we employed.

## 1 Introduction

Aspect-oriented programming (AOP) [9] has been proposed recently as a means for modularizing systems that present crosscutting concerns. A crosscutting concern can affect several units of a software system and usually cannot be modularized by traditional object-oriented programming techniques. A typical example of crosscutting concern is logging. The implementation of this concern should be scattered across all the modules in a system, tangled with code related to other concerns, because some contextual information must be gathered in order for the recorded information to be useful. Other common examples of crosscutting concerns include profiling and authentication [11].

\* Supported by FAPESP/Brazil under grant 02/13996-2.

\*\* Partially supported by CNPq/Brazil under grant 351592/97-0.

\*\*\* Supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008

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 [9, 11, 12]. The most well-known study on the subject, performed by Lippert and Lopes [12], had the goal of evaluating if AOP could be used to separate the code responsible for detecting and handling exceptions from the normal application code in a large object-oriented (OO) framework. The authors found that the use of AOP brought several benefits, such as less interference in the program texts and a drastic reduction in the number of lines of code (LOC). However, this first study has not investigated the "aspectization" of application-specific error handling, which is often the case in large-scale software systems. Moreover, in spite of the assumption made by many authors that using AOP for separating exception handling code from the normal application code is beneficial, the trade-offs involved in using AOP with this goal are not yet well-understood. To the best of our knowledge, no work in the literature has attempted to assess whether AOP really promotes an enhancement in well-understood quality attributes such as separation of concerns, coupling, cohesion, and conciseness, when used for modularizing non-trivial exception handling code.

This paper presents a study performed to assess the adequacy of AspectJ [11], a general purpose aspect-oriented extension to Java, for modularizing exception handling code. The study consisted of refactoring part of a real OO system so that the code responsible for handling exceptions was moved to aspects. This study differed from the Lippert & Lopes study in the following points:

- The target of the study is part of a complete, deployable system, not a reusable infrastructure, like a framework. Hence, the exception handling code implements non-uniform, complex strategies, making it harder to move handlers to aspects.
- We employ the metrics suite proposed by Sant'Anna et al [15] to assess attributes such as coupling, conciseness, cohesion, and separation of concerns in both the original and the refactored system.
- We assess the overall quality of both the error handling aspects and the application classes affected by them.
- We have not attempted to move error detection code to aspects.

We have found that, in general, AOP improved separation of concerns between exception handling code and normal application code. Moreover, we noticed that aspects promote handler reuse, but reusing handlers requires careful design planning. Otherwise, the behavior of the system may be unintentionally altered when the handlers are extracted to aspects. Furthermore, contradicting the general intuition, we observed that, for systems with application-specific exception handling strategies, an aspect-oriented (AO) solution does not result in a reduced number of LOC. For the system we have refactored, the AO version had almost the same number of LOC as the OO version. Another consequence of using aspects was that, in many cases, it was necessary to refactor the 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 in the overall cohesion of the system.

This paper is organized as follows. Section 2 describes the setting of our study, while providing very brief descriptions to the AspectJ language and to the Lippert &

Lopes study. The results of the study are presented and analyzed in Sections 3 and Section 4, respectively. Section 5 discusses some limitations of our study and the last section points directions for future work.

## 2 Study Setting

This section describes the configuration of our study. Section 2.1 briefly describes the AO programming language we have used, AspectJ. Section 2.2 provides an overview of the Lippert & Lopes study. Section 2.3 describes the Telestrada system, the target of our study. Section 2.4 presents an example of how exception handling code was moved to aspects in our study. Section 2.5 presents the metrics we have used to evaluate the OO and AO versions of Telestrada.

### 2.1 AspectJ Overview

AspectJ [11] is a general purpose aspect-oriented extension to Java. It 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, in order to support the selection of join points and the execution of advice in these points. A *pointcut* picks out certain join points and contextual information at those join points. Join points selectable by pointcuts vary in nature and granularity. Examples include method call and class instantiation. Advice may be executed *before*, *after*, or *around* the selected join points. In the latter case, execution of the advice may potentially alter the flow of control of the application, and replace the code that would be otherwise executed in the selected join point. AspectJ also allows programmers to modify the static structure of a program by means of static crosscutting. With static crosscutting, one can introduce new members in a class or interface, or make a checked exception unchecked.

*Aspects* are units of modularity for crosscutting concerns. They are similar to classes, but may also include pointcuts, advice, and static crosscutting. Aspects are combined with Java code by means of a process called weaving. Therefore, the tool responsible for performing weaving is called *weaver*.

### 2.2 Lippert and Lopes' Study

The study of Lippert and Lopes used 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 lines of code. The

Lippert & Lopes study was a important initial evaluation of the applicability of AspectJ and aspects in general for solving a real software development problem. However, it has some shortcomings that hinder its results to be extrapolated to the development of real-life software systems.

First, the target of the study was a system where exception handling is generic (not application-specific). However, it is well-known that exception handling is an inherently application-specific error recovery technique [1]. In other words, the real exception handling would be implemented by systems built using JWAM as an infrastructure and not by the framework itself. The authors report that most of the handlers in JWAM implemented policies such as “log and ignore the exception”. This helps explaining the vast economy in LOC the authors achieved by using AOP.

Second, the qualitative assessment was performed in terms of quality attributes that are not well-understood, such as (un)pluggability and support for incremental development. The authors did not evaluate some attributes that are more fundamental and well-understood in the Software Engineering literature, such as coupling and cohesion.

Third, quantitative evaluation was performed only in terms of number of LOC. Although the number of LOC may be relevant if analyzed together with other metrics, its use in isolation is usually the target of severe criticisms [17]. In the context of the Lippert & Lopes study, the use of LOC as the sole metric provided a narrow view of the effects of the aspectization of exception handling on the program quality. It portrayed the AO solution as very superior to the OO solution even though, as described previously, this owed more to the nature of the target of the study than to the quality of the AO solution.

### 2.3 Telestrada: Our Case Study

Telestrada [4] is a large traveler information system being developed for a Brazilian national highway administrator. It comprises five subsystems: Central Database Subsystem, GIS (Geographic Information System) Subsystem, Call-Center Operations Subsystem, Roadside Operations Subsystem, and Complaint Management Subsystem.

For our study, we have selected some self-contained packages of the Complaint Management Subsystem (CMS). The implementation of the CMS comprises more than 12000 LOC and more than 300 classes. The packages we selected for the study comprise approximately 1600 LOC (excluding comments and blank lines) and more than 120 classes and interfaces.

The classes and interfaces of the selected portion of the CMS include more than 45 `try-catch` blocks of varied complexity. They implement diverse exception handling strategies that range from trivial to sophisticated, for example: (i) do nothing (empty catch block); (ii) log and close database connection; (iii) log the exception, perform a rollback, close the database connection, and raise a different exception; (iv) use Java’s reflection API to create a new `Method` object and use it for logging.

### 2.4 Aspectizing Exception Handling

Our study focuses specifically on the handling of exceptions. We moved all the `try-catch`, `try-catch-finally`, and `try-finally` blocks in the selected

portions of Telestrada to aspects. Method signatures (throws clauses) and the raising of exceptions (throw statements) were not affected because these elements are more related to exception detection than to exception handling.

Handlers moved to aspects were implemented by means of after and around advice, depending on whether or not the handler ended its execution by raising an exception, respectively. Whenever possible, we used after advice, since they are simpler. After advice are not appropriate, though, for implementing handlers that do not raise (or re-raise) an exception because these advice cannot alter the flow of control of a program. In cases where this was necessary, around advice were used.

New advice were created on a per-try-block basis, excluding cases where handlers could be reused. For each class in the original system, we defined an aspect to handle exceptions raised by the members of the class. In many cases, moving handlers to aspects required some refactoring of the original code. The following code snippet presents an example:

```
public class GenericOperations {
    public static boolean closeResultSet(ResultSet aResultSet) {
        boolean r = true;
        try { ... // body of the "try" block.
        } catch (SQLException e) { System.out.println(e.toString());
            r = false;
        }
        return r;
    } ... // implementation of the class
}
```

The procedure we used to move handlers to aspects is very similar to the *Extract Method* refactoring [5] and the same restrictions apply. After extracting all the handlers to aspects, we searched for reuse opportunities and eliminated identical handlers. For the example above, it was necessary to remove references to the local variable `r` from the `try-catch` block before moving it to an advice. Moving the `try-catch` block to an aspect named `GOHandler` produces the following code:

```
public class GenericOperations {
    public static boolean closeResultSet(ResultSet aResultSet) {
        ... // body of the original "try" block.
        return true;
    } ... // implementation of the class
}
public aspect GOHandler { // another source file
    pointcut crsHandler() :
        execution(public static boolean closeResultSet(..));
    boolean around(ResultSet rs) : crsHandler() && args(rs){
        try { return proceed(rs);
        } catch (SQLException e) { System.out.println(e.toString());
            return false;
        }
    }
}
declare soft : SQLException : crsHandler();
}
```

Method `closeResultSet()` now consists of the body of the original `try` block, plus a return statement. In the `GOHandler` aspect, we defined a `pointcut` named `crsHandler` to select the execution of `closeResultSet()`. The around advice to which we extracted the `try-catch` block is executed at this join point. This

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

**Table 1.** The Metrics Suite

advice calls `closeResultSet()` by means of the `proceed()` statement of `AspectJ` and, if no exceptions are raised, returns the result of the method execution. If an `SQLException` is raised, the exception handler is executed. In this example, `SQLException` was *softened*, meaning that Java’s static checks are suppressed at `crsHandler`. This is necessary because the body of `closeResultSet()` can still raise `SQLException` but, from the viewpoint of the Java compiler, the exception is not being handled since the handler is in an aspect.

## 2.5 Metrics Suite

In our study, we have selected a suite of metrics for separation of concerns, coupling, cohesion, and size [15] to evaluate both OO and AO implementations. These metrics have already been used in three different experimental studies [8, 7, 10] and have been effective to assess several internal quality attributes of Java and AspectJ programs. Some of them have been automated in the context of a measurement tool [16]. This metrics suite was defined based on the reuse and refinement of some classical OO metrics [2, 3]. The original definitions of the OO metrics [2] were extended to be applied in a paradigm-independent way, supporting the generation of comparable results.

The metrics suite also encompasses new metrics for measuring separation of concerns. They were used in our study to measure the degree to which the exception handling concern in Telestrada maps to the design components (classes and aspects), op-

erations (methods and advice), and lines of code. Table 1 presents a brief definition of each metric, and associates them with the attributes measured by each one. In general, the higher the value of a measure, the worse the performance of the assessed system with respect to that metric. Detailed descriptions of the metrics appear elsewhere [15].

Regarding the Concern Diffusion over LOC metric, concern switches are pairs of lines of code where the first line is related to the implementation of a concern and the following line is part of the implementation of a different concern. Each such pair of lines is a transition point.

### 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 2.5). The presentation is broken in three parts. Section 3.1 presents the results for the separation of concerns metrics. Section 3.2 presents the results for the coupling and cohesion metrics. Section 3.3 presents the results for the size metrics.

We present the results by means of tables that put side-by-side the values of the metrics for the OO and AO version of Telestrada. Where relevant, results are broken in two parts, in order to make it clear the contribution of classes and aspects to the value of each metric. Hereafter, we use the term “class” to refer to both classes and interfaces.

#### 3.1 Separation of Concerns Measures

Table 2 shows the obtained results for the three separation of concerns metrics. The AO version of Telestrada performed better for two of the three separation of concerns metrics, Concern Diffusion over Operations and Concern Diffusion over LOC. The two versions had the same value for Concern Diffusion over Components.

Metrics	# components	OO version	AO version
Concern Diffusion over Components	classes	8	0
	aspects	-	8
	<b>total</b>	<b>8</b>	<b>8</b>
Concern Diffusion over Operations	classes	25	0
	aspects	-	21
	<b>total</b>	<b>25</b>	<b>21</b>
Concern Diffusion over LOC	classes	131	0
	aspects	-	24
	<b>total</b>	<b>131</b>	<b>24</b>

**Table 2.** Separation of Concerns Metrics

In the AO version of Telestrada, code related to the implementation of the exception handling concern was moved to aspects. Therefore, for all the metrics, the number of classes implementing exception handling was zero. The identical values for Concern Diffusion over Components in the OO and AO versions of Telestrada are due to

the design choice of creating one “handler aspect” for each public class that implemented exception handling in the OO version. Other possible design choices would be to put the exception handling code in a single aspect or, for each exception, create an aspect that encapsulates the possible handling strategies for a given exception. The three approaches have pros and cons that revolve around the code size vs. modularity trade-off. This trade-off is also faced by developers applying design patterns [6] to unstructured OO systems. Our design choice was a middle-ground between a single, possibly bloated, aspect and more than twenty, possibly too fine-grained, aspects.

The AO version exhibited a better Concern Diffusion over Operations (16% lower than the OO version). For most components, the AO solution was either equivalent or superior to the OO one. Two exceptions were the AO versions of `db.ConnectionPool` and `system.modifyComplaint.Façade`. The AO versions of these components had higher values in Concern Diffusion over Operations because the OO version had operations with more than one `try-catch` block. When these handlers were moved to aspects, each one had to be put in a separate advice. Moreover, handler reuse was low for these components, since they implement very context-specific exception handlers.

When moving handlers to aspects, we reused handler advice as much as possible. For example, even though `GenericOperations` had some methods that had more than one `try-catch` block, the AO version exhibited a lower value in Concern Diffusion over Operations, since some of the handler advice could be reused. However, we avoided situations where handler reuse could cause exceptions to be swallowed, since this could change the behavior of the system. For instance, we did not merge the following two advice, from the `ConnectionPoolHandler`<sup>1</sup> aspect, in a single one:

```
void around() : setPropertiesHandler() {
    try { proceed();
    } catch (MissingResourceException mre) { // do nothing
    } catch (NumberFormatException nfe) { // do nothing }
}
void around() : logHandler() {
    try { proceed();
    } catch (Exception e) {} // ignore exceptions when logging
}
```

Associating the `setPropertiesHandler` pointcut to the second advice would cause unchecked exceptions to be caught and ignored. However, the advice to which this pointcut is associated does not interfere with the propagation of these exceptions.

Concern Diffusion over LOC was the metric where aspects performed best. The AO version of `Telestrada` had less than 20% of the number of concern switches of the OO version. This finding confirms the results in the Lippert & Lopes study. The authors claim that the use of aspects decreases interference between concerns in the program texts.

### 3.2 Coupling and Cohesion Measures

Table 3 shows the obtained results for the two coupling metrics, Coupling between Components and Depth of Inheritance Tree, and the cohesion metric, Lack of Cohesion in Operations.

<sup>1</sup> Handler aspects have the same name as their corresponding classes, plus the suffix “Handler”.

Metrics	# components	OO version	AO version
Coupling between Components	classes	73	58
	aspects	-	16
	<b>total</b>	<b>73</b>	<b>74</b>
Depth of Inheritance Tree	classes	73	73
	aspects	-	2
	<b>total</b>	<b>73</b>	<b>75</b>
Lack of Cohesion in Operations	classes	171	298
	aspects	-	-
	<b>total</b>	<b>171</b>	<b>298</b>

**Table 3.** Coupling and Cohesion Metrics

The OO and AO versions of Telestrada exhibited very similar measures for the coupling metrics. The Depth of the Inheritance Tree increased by less than 3% in the AO version. This was expected, since the use of aspects alone does not interfere with this metric. The increase of 2 in the value of the measure was due to the creation of a new aspect from which two handler aspects, `system.modifyComplaint.FaçadeHandler` and `system.registerComplaint.FaçadeHandler`, inherited. The super-aspect was created in order to avoid duplicated code.

Coupling between Components in the two versions was almost identical. New couplings were introduced only when aspects had to capture contextual information from classes. In these cases, at most one new coupling is created per aspect, due to a reference from the aspect to its corresponding class.

Among all the metrics, Lack of Cohesion in Operations was the one for which the AO version of Telestrada presented the worst results. Lack of cohesion in the operations of the AO version was more than 75% higher than in the OO version. This is due to 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 aspect-oriented software development [13], since current aspect languages do not provide means to precisely capture every join point of interest.

It is interesting to note that the goal of the Lack of Cohesion in Operations metric is to capture a partial view of cohesion: it considers only the explicit relationships between the attributes and operations. It does not consider direct inter-operation relationships and the semantic closeness between elements of a component. Moreover, even though cohesion was worse in the AO version, the aspects had very good measures for Lack of Cohesion in Operations. This happened because none of the handler advice accesses fields of the classes they refer to and the aspects do not define new fields. Hence, there are no values of Lack of Cohesion in Operations for the aspects in Table 3.

### 3.3 Size Measures

Contradicting the general intuition that aspects make programs smaller [8, 11, 12], the OO and AO versions of Telestrada had very similar results in three of the four size

metrics: LOC, Number of Attributes, and Vocabulary Size. Moreover, the number of operations of the AO version was 21% higher than the OO version. Table 4 summarizes the results for the size metrics.

<b>Metrics</b>	<b># components</b>	<b>OO version</b>	<b>AO version</b>
Lines of Code (LOC)	classes	1594	1290
	aspects	-	285
	<b>total</b>	<b>1594</b>	<b>1575</b>
Number of Attributes	classes	50	50
	aspects	-	0
	<b>total</b>	<b>50</b>	<b>50</b>
Number of Operations	classes	166	180
	aspects	-	21
	<b>total</b>	<b>166</b>	<b>201</b>
Vocabulary Size	classes	113	113
	aspects	-	8
	<b>total</b>	<b>113</b>	<b>121</b>

**Table 4.** Coupling and Cohesion Metrics

The similar values for LOC were expected. As mentioned in Section 3.1, reusing handler aspects in Telestrada was much harder than we had originally predicted. Hence, although some reuse could be achieved, this was not anywhere near the results obtained by Lippert & Lopes in their study. Moreover, most handlers comprise just a few LOC and the use of AspectJ incurred in a slight implementation overhead because it was necessary to specify join points of interest and soften exceptions in order to associate handlers to pieces of code. In the end, the economy in LOC achieved due to handler reuse was compensated by the overhead of using AspectJ.

The 7% increase in the vocabulary size of the AO version was entirely due to the aspects. No new classes were introduced or removed. Similarly to Concern Diffusion over Components (Section 3.1), Vocabulary Size depends heavily on how the implementation of the exception handling concern is partitioned among the aspects.

The number of operations in the AO version of Telestrada was 17% bigger than in the OO version. The main reason for this increase 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 OO version, this increase was expected. Another reason for the increase in the Number of Operations was the refactoring of methods to expose join points that AspectJ can capture.

## 4 Analysis of the Results

In general, we found that reusing handlers is much more difficult than is usually advertised [12]. Handler reuse depends directly on: (i) the type of exception being handled; (ii) what the handler does and whether it ends its execution by returning or raising an exception; (iii) the amount of contextual information required; and (iv) what the method

that raises the exception returns and what exceptions appear in its `throws` clause. Factor (ii) above, whether a handler ends its execution by returning or raising an exception, is important because it restricts the types of advice that can be used.

When exception handlers are non-trivial, it may be difficult to fully understand the implications of moving a handler to an aspect. Hence, reusing handlers requires careful design, in order to avoid changing the exceptional behavior semantics of the system. The same issue applies for exception softening. Softening an exception that is a supertype of another exception raised within the same context causes the subtype to be softened as well, possibly with unexpected effects. We believe that developers should never soften exceptions that are supertypes of many other exceptions, such as `Exception` and `Throwable` in Java.

In spite of the better Concern Diffusion over LOC of the AO version of Telestrada, we expected the difference to be even bigger. This did not happen because, as discussed in Section 2.4, we used around advice whenever a handler did not end its execution by raising an exception. An around advice executes the code of its selected join points explicitly, by means of the `proceed()` statement. Since, in our case study, the code of the selected join points corresponds to the system's normal activity, occurrences of `proceed()` can be seen as concern switches.

Although the Coupling between Components in both versions of Telestrada was almost identical, this does not mean that components are as coupled to each other in the AO version as in the OO version. The measures of Coupling between Components for the classes in the AO version were lower than in the OO version. Moreover, the sums of the measures of Coupling between Components for the classes and their corresponding aspects in the AO version were similar to the measures for the respective classes in the OO version. Therefore, we can say that the AO version has more components but they are, in general, less strongly coupled to one another.

As seen in Section 3.3, handler advice accounted for a 10% increase in the number of operations. As with all size metrics, this value cannot be evaluated in isolation. Although a developer getting acquainted to the AO version will have to understand more operations, these operations are simpler and do not mix the system's normal activity with the code that handles exceptions. Therefore, the increase in the Number of Operations caused by the handler advice can be seen as a positive factor.

The number of operations refactored to expose join points that AspectJ could capture corresponded to 7% of the total Number of Operations measure. Unlike the increase caused by handler advice, the increase caused by refactored operations is definitely negative. 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, the refactored operations comprised just a couple of lines that did not make much sense when separated from their original contexts. This problem may suggest that there is still room for improving AspectJ so that more join points of interest can be captured.

## 5 Limitations of this Study

Our study focuses on a single aspect-oriented language, namely, AspectJ. Although many ideas presented here also apply to other AO languages, some surely do not. For

example, it is not necessary (or possible) to soften exceptions in Eos [14], an aspect-oriented extension to C#, because C# does not have checked exceptions.

Not all possible strategies for implementing exceptional behavior of systems are covered. In Telestrada, 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 system.

We do not attempt to evaluate the scalability of aspects for modularizing exception handling. Although the target of our study implements non-trivial exception handling policies, it is still just part of a system and comprises less than 2000 LOC. Moreover, we only modularize exception handling using aspects. We do not evaluate interactions between exception handling aspects and aspects implementing other concerns.

## 6 Future Work

Our most immediate future work is to derive a predictive model for using aspects to implement exception handling, based on the lessons learned from this study. With this model, developers will be able to recognize the situations in which it is advantageous to use aspects to modularize exception handling code. Moreover, we intend to document as patterns some strategies for structuring exception handling aspects.

As mentioned in Section 5, we have not evaluated the scalability of AspectJ for implementing exception handling. In the near future, we intend to analyze two scenarios: (i) whether aspects scale up well when the number of handlers grows; and (ii) whether it is difficult to integrate exception handling aspects with aspects implementing other concerns, such as distribution and persistence.

### Acknowledgements

We would like to thank Alexandra Barros for the many insightful comments and suggestions.

## References

1. T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 2nd edition, 1990.
2. S. Chidamber and C. Kemerer. A metrics suite for oo design. *IEEE Trans. on Soft. Eng.*, 20(6):476–493, June 1994.
3. N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous Practical Approach*. PWS, 1997.
4. F. Castor Filho et al. A systematic approach for structuring exception handling in robust component-based software. *Journal of the Brazilian Computer Society*, 2005. To appear.
5. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
6. E. Gamma et al. *Design Patterns: Elements of Reusable Software Systems*. Addison-Wesley, 1995.
7. A. Garcia et al. Separation of concerns in multi-agent systems: An empirical study. In C. Lucena et al., editors, *Software Engineering for Multi-Agent Systems II*, LNCS 2940. Springer-Verlag, February 2004.

8. A. Garcia et al. Modularizing design patterns with aspects: A quantitative study. In *Proceedings of the 4th AOSD*, pages 3–14, Chicago, IL, USA, March 2005.
9. G. Kiczales et al. Aspect-oriented programming. In *Proceedings of the 11th ECOOP'97*, pages 220–242. Springer Verlag LNCS 1241, 1997.
10. U. Kulesza et al. Aspectization of distribution and persistence: Quantifying the effects of aop. *Submitted to IEEE Software, Special Issue on AOP*, May 2005.
11. R. Laddad. *AspectJ in Action*. Manning, 2003.
12. M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd ICSE'2000*, pages 418–427, Limerick, Ireland, June 2000.
13. G. Murphy et al. Does aspect-oriented programming work? *Communications of the ACM*, 44(10):75–77, October 2001.
14. Hridesh Rajan and Kevin Sullivan. Eos: Instance-level aspects for integrated system design. In *Proceedings of ESEC/FSE'2003*, Helsinki, Finland, September 2003.
15. C. Sant'Anna et al. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings of the 17th SBES*, pages 19–34, October 2003.
16. Tigris. aopmetrics home page, 2005. Address: <http://aopmetrics.tigris.org>.
17. H. Zuse. History of software measurement, 2005. Address: [http://irb.cs.tu-berlin.de/zuse/metrics/History\\_00.html](http://irb.cs.tu-berlin.de/zuse/metrics/History_00.html).