Assessing Intra-Application Exception Handling Reuse with Aspects

Júlio César Taveira, Cristiane Queiroz, Rômulo Lima, Juliana Saraiva, Sérgio Soares, Hítalo Oliveira, Nathalia Temudo, Amanda Araújo, Jefferson Amorim
Department of Computing and Systems, University of Pernambuco, Recife, Brazil
{jcft2, ccq, rsl, julianajags, hos, nmt, ara, jsa}@dsc.upe.br

Fernando Castor, Emanoel Barreiros
Informatics Center, Federal University of Pernambuco, Recife, Brazil
{castor, efsb}@cin.ufpe.br

Abstract — **Recent studies have attempted to evaluate the benefits and drawbacks of using aspect-oriented programming to modularize exception handling code. In spite of their many interesting findings, these studies have not reached a consensus when it comes to the impact of aspectization on exception handler reuse. In fact, their results are sometimes in direct contradiction.In this paper we describe a study aiming to answer the question of whether AOP really promotes the implementation of reusable exception handling. 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. Our study targets three industrial-strength, medium-size software systems from different domains and employs a comprehensive set of concern-specific metrics.**

Resumo — **Estudos recentes tentaram avaliar os benefícios e desvantagens de se empregar a programação orientada a aspectos (POA) para modularizar código de tratamento de exceções. Apesar dos muitos achados interessantes desses estudos, eles não atingiram um consenso no tocante ao impacto de POA no reuso de tratadores de exceções. Inclusive, em alguns casos, os resultados desses estudos estão em contradição direta. Tendo isso em vista, este artigo descreve um novo estudo cujo objetivo é responder se aspectos realmente promovem a implementação de tratamento de exceções reusável. Reuso é analisado em um contexto específico: em termos do numero de tratadores de erros duplicados ou muito similares que podem ser removidos de um programa quando código de tratamento de exceções é extraído para aspectos. O estudo tem como alvos três sistemas de software reais, de tamanho médio, oriundos de domínios distintos, e emprega um conjunto abrangente de métricas específicas para o interesse analisado, tratamento de exceções.**

I. INTRODUCTION

Exception handling [14] is a technique used in software development that allows one to structure a program to deal with exceptional situations. Many mainstream programming languages, such as Java, Ada and C++, have an exception handling mechanism. An exception handling mechanism supports the separation between the code responsible for handling errors and the code implementing the main system functionality [22]. Ideally, this separation results in systems that are less complex and, hence, easier to understand and more evolvable and reliable [23].

In spite of all its potential benefits, effectively using exception handling is not a simple task. The main problem is that error handling code usually gets tangled with code associated with other concerns and scattered throughout many program units. This hinders maintainability and understandability and negates some of the expected benefits of exception handing mechanisms. In addition, developers are often only interested in the main system functionality [26]. Exception handling is often addressed at the implementation phase, in an ad hoc way. As a consequence, exception handling code is, in general, not well-understood, tested, and documented [6]. Another problem of exception handling in object-oriented systems written in mainstream programming languages is code duplication. It is often the case that identical exception handlers can be found in many different places within a system [1]. Duplicated code can have a negative impact on several software quality attributes [17], including maintainability and reliability.

The use of new separation of concerns techniques such as aspect-oriented programming (AOP) [18] can help developers in taming the inherently crosscutting nature of error handling. In AOP, crosscutting concerns, such exception handling, authentication, and logging, can be implemented by a new abstraction, called aspect, while the concerns that are not crosscutting are implemented with plain object-oriented programming. AOP can also help developers in reducing the amount of duplicated error handling code in an application, by localizing similar exception handlers within the same program unit. This means that, when an exception handler is refactored to an aspect, all the clones of that handler can be removed and the aspectized handler can be reused instead, without duplication.

The intra-application reuse of exception handling code promoted by AOP has been discussed in at least two previous studies. Lippert and Lopes [21] found out that, when extracting to aspects the error handling code of a reusable infrastructure, a large amount of reuse can be achieved and, as a consequence, system size is reduced. On

the other hand, some of us [3] conducted an independent study targeting four deployable systems and found out that very little reuse could be achieved through the aspectization of exception handling. In fact, system size consistently grew after we extracted error handling to aspects. In our view, these conflicting results stem from the focus of these studies. The first one aimed to assess the feasibility of using AOP to modularize exception handling whereas the second one assessed the quality of the resulting code, when compared to a purely object-oriented version. Even though both studies attempted to measure system attributes typically associated with intra-application exception handling reuse, such as number of lines of code (LOC) and number of catch blocks, none of them was specifically designed with reuse in mind. Therefore, it is difficult to extrapolate their findings to general software development.

This paper addresses these limitations of previous studies and presents a new study with the goal of assessing the extent to which AOP promotes intra-application reuse of exception handling code. We compare three alternatives for modularizing exception handling: a "regular" object-oriented approach, a refactored object-oriented approach where exception handlers are implemented in special-purpose classes, as an attempt to maximize the reuse of object-oriented exception handlers, and an aspect-oriented approach where exception handlers are implemented in AspectJ [20], an aspect-oriented extension to Java. For the latter two alternatives, the code was thoroughly and methodically reviewed to spot reuse opportunities and combine identical handlers. Our study targets three medium-size industrial-strength Java applications where exception handling code is non-trivial. We try to elicit situations where AOP creates opportunities for reuse whereas object-oriented programming does not and vice-versa. We also analyze the language constructs that influence these opportunities, i.e., the root causes for one approach to be more powerful than the other. We employ a set of software metrics to assess the quality of each version of the three systems. These metrics have the peculiarity of being capable of directly measuring system features that are related to reuse of exception handling code. To collect these metrics, we have built a measurement tool that works with both Java and AspectJ programs.

It is important to stress the main differences between this study and previous studies addressing the combination of AOP and exception handling. First, we use medium-size, industrial-strength applications with non-trivial exception handlers, whereas Lippert and Lopes [21] have employed a reusable infrastructure with very simple exception handling policies. Our study is also different from the one conducted by Castor Filho et al. [3], since we have employed pairs of developers attempting to systematically reuse exception handling as much as possible. In addition, these pairs have also reviewed each other's work, to spot missed reuse opportunities Finally, this study employs a set of 10 different concern-specific metrics, all of them focusing on exception handling. This set of metrics provides a more precise picture of the impact of each modularization approach on the amount of exception handling reuse that could be achieved.

This paper is organized as follows. Section II presents the target applications of our study (Section II.A), explains how we extracted exception handling code to both aspects and new classes (Section II.B), and describes our approach for finding and eliminating duplicated handlers (Section II.C). It also presents the employed metrics suite (Section II.D). Section III presents the results of the study. Sections IV and V discuss related work and round the paper, respectively.

II. STUDY STETTING

In this section we describe the configuration of our study. Section II.A describes its target systems. Sectin II.B explains how we extracted error handling code to aspects and to new classes. We assume that the reader has a basic knowledge of AOP and AspectJ. The adopted approach for detecting and combining duplicated exception handlers is explained in Section II.C. Section II.D presents the metrics suite that we employed in order to quantify reuse

A. Our Case Studies

In this study we refactored three Java applications. These applications are significant for our study because they are real industrial-strength applications and have some relevant characteristics. The first one is that all of them have at least 15K LOC and at least 1% of LOC pertaining to exception handling. A second characteristic is that the applications are from different domains and written in Java. The third characteristic is that the applications have many non-trivial exception handlers. Finally, the three target systems come from different domains and application development platforms. The original implementations of the three applications are in plain Java. For each application, two new versions were developed: one where exception handling is modularized using AspectJ aspects and another one where exception handling is modularized using Java classes (Section II.A). On the remaining of this section, we describe the three target systems.

The first application is Checkstyle[1], a plugin that checks coding style for Java programs. This application comprises 19,197 LOC (compilable code, excluding comments and blank lines) and more than 290 classes and interfaces. The second application, Java PetStore[2], is a demo for the Java platform, Enterprise Edition [3](Java EE). This application employs various technologies based one the Java Enterprise Edition (EE) platform and is representative of e-commerce applications. Its implementation has approximately 17,500 LOC and 330 classes and interfaces. The third application is JHotDraw[4], a Java GUI framework for technical and structured graphics. JHotDraw comprises 23k LOC and more than 400 classes and interfaces.

B. Extracting Exception Handling

This study compared three different versions of each of the target systems: (i) an original version, implemented in

[1] http://eclipse-cs.sourceforge.net/
[2] http://java.sun.com/developer/releases/petstore/
[3] http://java.sun.com/j2ee
[4] http://www.jhotdraw.org/

Java; (ii) a refactored OO version, also implemented in Java; and (iii) and AO version, implemented in AspectJ. The latter are functionally equivalent versions of the original system where the exception handling code was moved to new classes and new aspects, respectively. To obtain versions (ii) and (iii) of each target systems, we extracted, from original version, exception handling code to new classes and aspects whose sole responsibility is to implement the exception handling concern.

It is important to emphasize that, in the refactored OO versions, the code that captures exceptions (try-catch-finally blocks) cannot be extracted, diferently from the refactored AO versions. Only the internal statements of the catch and finally blocks. We perform this extraction, nonetheless, to evaluate the capability of object-oriented languages to promote error handling reuse, so as to allow a fair comparison with AspectJ. We consider that exception handling code is all the code that exists solely due to the need to handle exceptions (i.e., we could remove it if exceptions never had to be handled). This is in accordance to the ideas of Eaddy et al. [8]. This definition includes catch blocks, finally blocks and the definition of try blocks (but not the code within a try block). It excludes throw statements and throws clauses, as well as code that checks for erroneous conditions. In this study, eight different programmers conducted the extraction of the exception handlers. These programmers worked in pairs. Each pair systematically reviewed the work of the others. The first step of exception handler extraction was to move exception handlers out of the regular Java classes to Java classes concerned only with exception handling (handler classes). One handler class was created for each package in the system. Only the contents of catch and finally blocks were removed to these classes and the catch and finally blocks therefore contained only calls to the new handler methods. Figure 1 present an example of this refactoring. Figure 1(a) presents the original code whereas Figure 1(b) shows the same method after its exception handler is extracted to a handler class. Figure 1(c) shows the affected parts of a handler class. If a catch or a finally block writes (in primitives types) or performs assingments to a private field of the enclosing class, or a local variable or parameter of the enclosing method, this refactoring cannot be performed, because these elements are not accessible to a handler class.

The second step of the extraction consisted of moving to aspects (handler aspects) the error handlers of the original version of each target system. One handler aspect was created for each package in the system. For uniformity's sake, we employed only around advice to implement handlers (handler advice), since they are powerful enough to implement a wide variety of error handling strategies [3]. In this phase of the study, we created new advice on a per-try-block basis, i.e., all the catch blocks associated to a try block in the original version of each system were moved to the same advice. The finally blocks were moved to separate advice to promote the conceptual separation between exception handlers and clean-up actions. Figure 1(d) shows a class affected by AO refactoring, where the try-catch blocks were removed. Figure 1(e) presents the resulting handler advice. In cases where a handler depends on fields of the enclosing class, we declare the aspects to which they are extracted to be privileged. On the other hand, when a handler depends on local variables of the enclosing method, we first employ the Extract Method [10] refactoring to expose these variables as parameters of the new method and then extract the handler to an aspect. In a small number of situations, e.g., a handler that writes to two or more local variables of the enclosing method, it was not possible extract exception handlers to aspects without redesigning of the original system. The handlers in these cases were not modified.

C. Reusing Exception Handlers

To reuse exception handlers, a similar approach was conducted for both the refactored OO versions and the AO versions, taking into account the differences between the two paradigms. For the refactored OO versions, we first put all the exception handling code that could be extracted to classes in a single class per package. Next, we look for similar exception handlers within the handler class of each package. Methods implementing duplicated exception handlers are eliminated and references to these methods are modified to point out to the appropriate handlers. Each handler class is systematically reviewed by at least two pairs of developers. The next step is to combine duplicated handlers appearing in different packages. In this case, we compare handler classes across unrelated packages. In this case, duplicated handlers are moved to a global parent handler class and the handler classes are modified to inherit from it. Whenever, due to reuse, a handler class becomes empty, we remove it from the system.

For the AO versions, we start by putting all the source code relative to the capture and handling of exceptions in a single aspect per package. Then, we check the possibility of reuse inside each package. The complicating factor in this case is the flexibility of AspectJ. For example, different (unrelated) return types do not necessarily mean that two handler advice cannot be combined. This greater flexibility also means that developers have more issues to consider, when combining handler advice, e.g., the return type of the method to which the handler advice will be associated, the number of arguments of the method, if other exceptions can be thrown, etc. In addition, sometimes the sets of catch blocks associated to different try blocks cannot be reused, but the individual catch blocks can. In these fairly frequent situations, it is necessary to create handler advice on a per-catch block basis, instead of a per-try block basis.

After checking if there are reuse possibilities inside each package, we proceed as described for the refactored OO versions. A global general aspect was created to contain the similar handlers that are duplicated across different packages. However, differently from the general handler class, there is no inheritance relationship between the handler aspects. Due to the limited form of inheritance supported by AspectJ and the richer reuse scenarios promoted by AOP, we found that aspect inheritance would not suit our needs.

```
public class CheckSelectedFilesAction {
... public void run(IAction action) {
        ... try{
            addFileResources(mSelection.toList(), filesToCheck);
        } catch () {
            CheckstyleLog.errorDialog(mPart.getSite().getShell(), e, true);
            //one or more lines
        }...
    }...
}
```

(a) Original CheckSelectedFilesAction class.

```
public class CheckSelectedFilesAction {
    ActionHandler actionH = new ActionHandler();
... public void run(IAction action) {
        ... try{
            addFileResources(mSelection.toList(), filesToCheck);
        } catch () {
            actionH. errorDialogCheckstyleLog(mPart.getSite().getShell(),e, true);
        } ...
    } ...
}
```

(b) Refactored CheckSelectedFilesAction class. Refactored OO version.

```
public class ActionExceptionHandler {
... public void errorDialogCheckstyleLog(Shell shell , Exception e, boolean b){
        CheckstyleLog.errorDialog(shell, e, b);
    }...
}
```

(c) ActionExceptionHandler exception handler class.

```
public class CheckSelectedFilesAction {
... public void run(IAction action) {
        ...
        addFileResources(mSelection.toList(), filesToCheck);
        ...

    } ...
}
```

(d) Refactored CheckSelectedFilesAction class. AO version.

```
public privileged aspect ActionAspectExceptionHandler
 {
... declare soft: CoreException : CheckSelectedFilesActionHandle_runHandle();
... pointcut CheckSelectedFilesActionHandle_runHandle():
        execution (public void CheckSelectedFilesAction.run(..)) ;
... void around(): CheckSelectedFilesActionHandle_runHandle(){
        try {  proceed();  } catch (CoreException e) {
            CheckSelectedFilesAction c =
                 (CheckSelectedFilesAction) thisJoinPoint.getThis();
            CheckstyleLog.errorDialog(c.mPart.getSite().getShell(), e, true);
        }
    }...
}
```

(e) ActionAspectExceptionHandler exception handler aspect.

Figure 1. **Examples of the extraction of exception handling code.**

Figure 2 illustrates the amount of exception handling code reuse that could be achieved in some situations. Figure 2(a) shows a simple method that is called by twelve different catch blocks. Figure 2(b), shows the same exception handler in the AspectJ version of the system. The handler in this case is not called by any catch blocks. Instead, it is associated to twelve different join points in the system. It is important to stress that scenarios such as this one are only possible when exception handlers are very simple. Even though some amount of reuse is possible with more complicated handlers, this is not always easy to spot and is limited to a few places in the code.

```
...
//Called by some code location
public void rethrowCheckstylePluginException(Exception e) throws
CheckstylePluginException {
    CheckstyleLog.log(e);
    CheckstylePluginException.rethrow(e);
}...
```

(a). Exemple of code reuse using a class.

```
...
Object around() throws CheckstylePluginException:
    RetrowException_runHandle() ||
    ProjectConfigurationFactory_internalLoadFromPersistenceHandler() ||
    checkConfigurationMigrator_migrateHandler() ||
    ProjectConfigurationEditor_internalEnsureFileExistsHandler() ||
    ExternalFileConfiguration_internalEnsureFileExistsHandler() ||
    checkConfigurationMigrator_ensureFileExistsHandler() ||
    auditor_runAuditHandle() ||
    PackageNamesLoader_getPackageNames() ||
    CustomLibrariesClassLoader_get() ||
    RetrowException_setModulesHandle() ||
    RemoteConfigurationType_internalGetCheckstyleConfigurationHandler() {
        Object result = null;
        try {
            result = proceed();
        } catch (IOException ioe) {
            CheckstyleLog.log(ioe);
            CheckstylePluginException.rethrow(ioe);
        }
        return result;
}
...
```

(b). Exemple of code reuse using an aspect.

Figure 2. **Examples of the reused exception handling code.**

D. Metrics Suite

We evaluated the three versions of each target system based on a suite of code metrics. To collect them, we used a tool that we have developed as an extension to AopMetrics[5], EH-Meter[6]. The latter is a metrics collection tool that works with both Java and AspectJ programs. Since this study focuses on evaluating reuse, we choose a set of metrics capable of measuring quantities that, we believe, indicate exception handling reuse or the lack thereof. More specifically, we have selected 11 metrics, all but one of them directly applicable to the error handling concern. For all the employed metrics, a lower value is better.

The selected metrics are divided in two groups: size metrics and concern metrics. Table 1 lists these metrics and briefly describes them. The first group, size metrics, includes metrics for both general system attributes (e.g. Number of Lines of code) and quatities that are specific to exception handling (e.g. Number of try Blocks). The selected size metrics are: number of Lines Of Code (LOC), Number of try blocks (NOT), Number Of Catch blocks (NOC), Number Of Finally blocks (NOF), number of Lines Of Code implementing Exception Handling (LOCEH), and Number of Exception Handlers (NEH). The latter counts the number of blocks of code that implement exception handling strategies. It is useful to quantify reuse of exception handling in the refactored OO versions.

Concerns metrics, measure how the source code is directed to analyze concerns [27]. We employ these metrics to assess the extent to which the modularization of exception handling influences reuse and vice-versa. Moreover, some of these metrics simply count the number of occurrences of a certain kind of element pertaining to a specific concern. This is not different from what the aforementioned size metrics do. In this study, we consider only two concerns: exception handling and the main system functionality, which comprises all the remaining system concerns. The first three metrics [25] are well-known in the AO software development community: Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO), and Concern Diffusion over LOC (CDLOC). The justification for using those metrics is to verify the diffusion of the exception handling concern, analising: (i) how many components (classes and aspects) and operations (methods) implement the exception handling concern; and (iii) how many transitions exists between exception handling and normal behaviour.

[5] http://aopmetrics.tigris.org/

[6] http://www.dsc.upe.br/~jcft2/eh-meter

Table 1. The employed metrics suite

Group	Metric	Description
Size Metrics	LOC	Number of LOC in a program, excluding comments and blank lines.
	NOT	Number of try blocks.
	NOC	Number of catch blocks.
	NOF	Number of finally blocks.
	LOCEH	Lines of code that are relative to the handling and capture of exceptions.
	NEH	Number of blocks of code that to contribute to handle exceptions.
Concerns Metrics	CDC	The number of components (classes and aspects) that include some exception handling code.
	CDO	The number of operations (methods) that include exception handling code.
	CDLOC	The number of transition points for each concern through the lines of code. The use of this metric requires a shadowing process that partitions the code into shadowed areas and non-shadowed areas.
	LOF	Measures the variance of the dedication of a component to every concern. The result is normalized between 0 (*completely focused*) and 1 (*completely unfocused*).
	DOS	A measure of the variance of the concentration of a concern over all components. The result is normalized to be between 0 *(completely localized)* and 1 *(completely delocalized, uniformly distributed)*.

In addition, we use two more recent metrics that were proposed in order to circumvent some of the limitations of the aforementioned concern metrics. The first one, Lack Of Focus (LOF), is an adaptation of the Degree Of Focus (DOF) metric, proposed by Eaddy et al. [8]. DOF measures the extent to which a component's lines of code are dedicated to the implementation of a given concern. Its value ranges between 0 (completely unfocused) and 1 (completely focused). A component's LOF is equal to 1 – DOF. The last concern metric, Degree of scattering (DOS) [8] measures the extent to which the LOC related to a concern are contained within a specific component. More information about DOF and DOS is available elsewhere [8].

III. Study Results

The main goal of this study is to check if AOP promotes greater reuse of exception handling code than a traditional, object-oriented approach. It also attempts, to a certain extent, to understand the relation between reuse and modularization of exception handling. The employed metrics support the evaluation of the changes applied to code, highlighting the benefits and drawbacks stemming from these changes. We investigate the impacts in reuse and separation of concerns of using aspect-oriented programming to modularize exception capture and handling. Besides this, refactored object-oriented versions of each application were developed to guarantee a better reuse. The refactored OO version was employed in order to make the comparison between OOP and AOP fair, since the original version of each target system was not designed or implemented with exception handling reuse in mind.

Table 2 presents size metrics collected from the three versions of each target system. Original OO indicates the original OO versions. Ref. OO refers to the object-oriented refactored versions, and Ref. AO indicates the aspect-oriented refactored versions. A percentage increase (% Incr.) indicates that the values of the metrics have increase after refactoring an application, when compared with the original version. As previously mentioned, for almost all the employed metrics, a lower value is better. The only exception is metric LOC=0 (Table 3), where it is not really possible to infer if a value is good or not without comparing it with the value of the same metric for a different version of the same system. In this case, higher is better.

Some results of the metrics presented expected values. For number of try blocks (NOT), number of catch blocks (NOC) and number of finally blocks (NOF) there was no variation between Original OO and Ref. OO versions. However, their values decreased sensibly for all the target systems, in many cases by more than 40%. This indicates that try, catch, and finally blocks are being reused in the Ref. AO versions, although not as much as reported by Lippert and Lopes [21].

The number of LOC grew in the refactored versions of all applications. Even though the increment might seem to be small (between 0.51% and 3.38%), it is considerable if we consider that it is solely a consequence of attempting to better modularize exception handling. In a similar vein, the Lines of Exception Handling Code (LOCEH) metric increased for all the refactored versions of the target systems. This overhead, affecting LOC and LOCEH, occurred because: (a) the Ref. OO version has more source code from calling new methods and new classes, and from method declarations; (b) in the Ref. AO version we need to create new aspects, advice, pointcuts and sometimes use the declare soft intertype declaration to suppress the checks that the Java compiler performs for checked exceptions.

The Number of Exception Handlers (NEH) metric decreased in both Ref. OO and Ref. AO versions. The reduction in the value of the metric for these versions of the applications stems from the separation between exception capture and exception handling. Since the handlers in the Ref. OO versions are methods whose parameters are the exceptions to be handled, in many cases these methods can take as arguments exceptions of types that are more general than the actual types of the exceptions being handled, thus creating new opportunities for reuse. This is possible because the exceptions are still captured in terms of their specific types, avoiding exception subsumption [24]. In the Ref. AO versions, since capture and handling of exceptions are performed by the advice, the latter must catch the exceptions using their specific types, to avoid catching exceptions unintentionally. If, in the Ref. AO version, the handler advice were responsible solely for handling exceptions and regular try-catch blocks were responsible for capturing them, it would be possible to achieve an even greater amount of reuse than was achieved in the Ref. OO versions.

Table 2. Size Metrics Results. Shows the absolute values and the increments with respect to measurements for the original version.

System	Version	LOC	NOT	NOC	NOF	LOCEH	NEH
JHD	**Original OO**	**22820**	64	71	4	**293**	65
	Ref. OO (% Incr.)	22937 (0.51%)	64 (0%)	71 (0%)	4 (0%)	375 (27.99%)	**34 (-47.69%)**
	Ref. AO (% Incr.)	23089 (1.18%)	**47 (-26.56%)**	**56 (-21.13%)**	4 (0%)	595 (103.07%)	47 (-27.69%)
Checkstyle	**Original OO**	**19197**	230	271	27	**1211**	282
	Ref. OO (% Incr.)	19475 (1.45%)	229 (-0,43%)	270 (-0.37%)	27 (0%)	1379 (13.87%)	**104 (-63.12%)**
	Ref. AO (% Incr.)	19845 (3.38%)	**131 (-43.04%)**	**142 (-47.60%)**	**22 (-18.52%)**	1700 (40.38%)	154 (-45.39%)
PetStore	**Original OO**	**17177**	294	418	13	**1709**	397
	Ref. OO (% Incr.)	17798 (3,62%)	294 (0%)	418 (0%)	13 (0%)	2214 (29.55%)	**205 (-48.36%)**
	Ref. AO (% Incr.)	17401 (1.30%)	**165 (-43.88%)**	**218 (-47.85%)**	**7 (-46.15%)**	2281 (33.47%)	210 (-47.10%)

Table 3 exposes results relative to the concern metrics. Metrics CDO (Concern Diffusion of Operation), CDLOC (Concern Diffusion of Lines of Code) and CDC (Concern Diffusion of Components) indicate how the exception handling concern is scattered and tangled throughout the application. Version Ref. AO obtained the best results for all the target systems. This result is consistent with results obtained in previous studies [3], pertaining to the ability of AOP to textually separate error handling from other system concerns. The values of CDO and CDLOC for the Ref. AO versions of the target applications decreased by more than 50%. The values for CDO indicate that fewer operations in the systems implement the exception handling concern (less scattering). The obtained values for CDLOC point out that there is almost no intermingling between error handling code and code implementing other concerns (less tangling). In the Ref. OO versions, try, catch and finally were not removed from the Java classes, unlike the Ref. AO versions. In additions, new methods solely responsible for handling exceptions were created in the handler classes. As a consequence, the values of CDO and CDC were strongly affected. On the other hand, the values for CDLOC remained largely unaffected, since no new concern transition points (Section II.D) were created.

The LOF (Lack of Focus) metric is normalized and Table 3 presents two different pieces of information pertaining to it. First, it presents the average LOF for all the components of each target application. Table 3 also indicates the number of components where LOF is 0 (LOF=0). By comparing the values of LOF=0 for the different versions of a target system, it is possible to assess the extent to which each approach for modularizing exception handling results in a system where each system component is focused on a single concern, a desirable property.

In the Ref. AO version both the average LOF and the number of components where LOF is 0 were better than the others versions. These results mean that, in the Ref. AO versions, error handling components are only responsible for handling exceptins and non-error handling components almost never handle exceptions (as highlighter by the CDLOC results). The LOF and LOF=0 results for the Ref. OO versions were slightly better than the ones for the original versions of the applications. This result must be analyzed with care, though. Since CDLOC did not change in the Ref. OO versions, it is possible to deduce that the system components that are present in both original and Ref. OO versions did not get more focused because they still mix error handling and normal behavior. Hence, the improvement in the value of these metrics for the Ref. OO versions is a consequence of the error handling components. They are highly focused and increase the overall number of system components.

The last metric presented in Table 3 is DOS (Degree of Scattering). Similarly to DOF, the value of this metric is normalized between 0 and 1. A value of 0 for this metric means that the implementation of a concern is entirely contained within a single program unit. A value of 1 means that the implementation of a concern is scattered throughout all the components of the system. For this metric, the Ref. AO versions exhibited the best results, although not by a large difference. This is understandable, since the Ref. AO versions concentrate all the error handling code within the aspects, thus reducing the number of program units responsible for this concern. The Ref. OO versions had slightly worse results than the original ones because the implementation of error handling is scattered throughout a larger number of components. Finally, for all the versions of all the systems, the values of DOS were very high. Since this study only considers two concerns (exception handling and the normal behavior), it is natural that at least one of them (the normal behavior) is scattered throughout the system components.

The percentage of increment for each metric is presented in Figures 3 and 4. The size metrics are depicted in Figure 3 whereas the increments related to concern metrics are presented in Figure 4. Looking up these figures we can conclude that, for each approach to modularizing exception handling, the three target applications were consistent when each metric is considered. For example, all the Ref. AO versions exhibited a lower number of try blocks and a greater amount of lines of exception handling code. This suggests that the obtained results might be applicable to a wider range of applications.

Table 3. Concern Metrics Results. Shows the absolute values and the increments with respect to measurements for the original version.

System	Version	CDO	CDLOC	CDC	Average LOF	LOF=0	DOS
JHD	Original OO	62	262	40	0.0333	90.10%	0.9601
	Ref. OO (% Incr.)	81 (30.65%)	262 (0%)	49 (22.50%)	0.0323 (-2.92%)	90.31% (0.24%)	0.9712 (1.16%)
	Ref. AO (% Incr.)	**47 (-24.19%)**	**20 (-92.37%)**	**12 (-70.00%)**	**0.0025 (-92.43%)**	**99.52% (10.46%)**	**0.8765 (-8.71%)**
Checkstyle	Original	200	870	109	0,1486	62.28%	0.9855
	Ref. OO (% Incr.)	253 (26.50%)	870 (0%)	120 (10.09%)	0,1353 (-8.91%)	64.36% (3.33%)	0.9863 (0.08%)
	Ref. AO (% Incr.)	**127 (-36.50%)**	**92 (-89.43%)**	**27 (-75.23%)**	**0,0138 (-90.69%)**	**97.44% (56.44%)**	**0.8983 (-8.85%)**
PetStore	Original	237	1012	109	0,1721	67.27%	0.9836
	Ref. OO (% Incr.)	365 (54.01%)	1008 (-0,40%)	148 (35.78%)	0,1461 (-15.13%)	70.97% (5.50%)	0.9886 (0.50%)
	Ref. AO (% Incr.)	**157 (-33.76%)**	**68 (-93.28%)**	**51 (-53.21%)**	**0,0162 (-90.61%)**	**95.64% (42.18%)**	**0.9578 (-2.62%)**

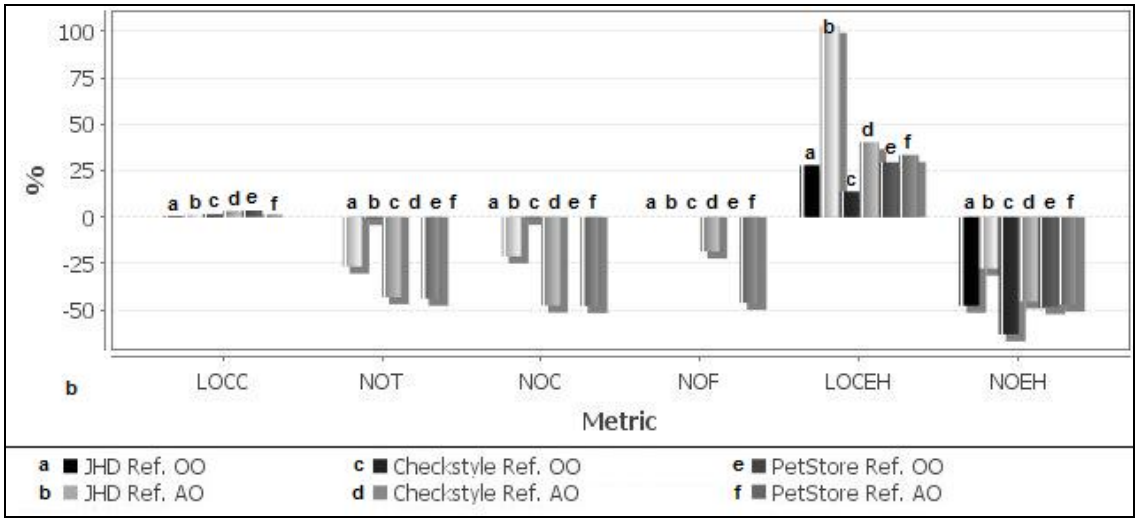

Figure 3. How the size metrics changed for the Ref. OO and Ref. AO versions.

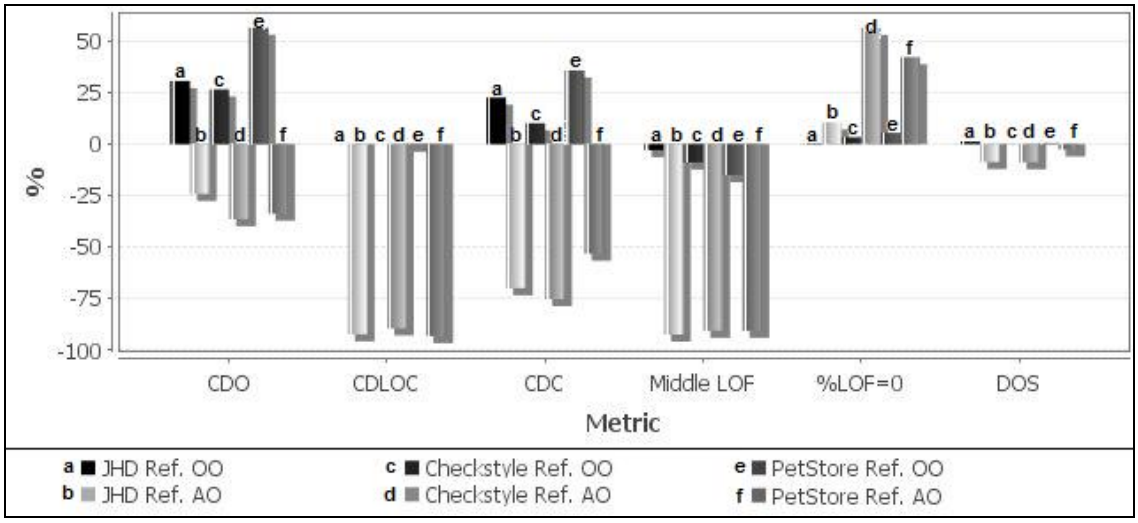

Figure 4. How the concern metrics changed for the Ref. OO and Ref. AO versions.

A careful analysis of the figures reveals some interesting points. First of all, there is a relationship between reduction in LOC and reuse when using AOP. JHD, the system where the lowest amount of reuse could be achieved, was also the one where LOCEH grew the most. This indicates that reuse of exception handlers actually influences the final number of LOC. Nonetheless, the overhead of using AOP, in terms of LOC, is considerable. For all the systems, LOCEH grew by at least 25%. Most importantly, this overhead is, for all the situations we have analyzed so far, much greater than the economy in LOC afforded by AOP. This observation raises another important point. This study has evidenced that, when applied in the modularization of error handling, AOP promotes reuse. However, it is not clear whether this reuse results in code that is more maintainable or understandable. Arguably, developers attempting to understand a system where exception handling is modularized with aspects would need to understand a greater number of modules and a larger number of LOC. Moreover, exception handling aspects influence program control flow in subtle ways that are not evidenced by the code [5][2].

IV. Related Work

Several researchers have tried to evaluate advantages and disadvantages of employing AOP to modularize crosscutting concerns [3][5][9][13][15][16][21]. The seminal paper on AOP [18] already referenced the exception handling concern as a good candidate to be modularized with aspects. There are specific studies that focus on implementing exception handling with aspects [3][4][21]. Lippert and Lopes [21] performed a study with a medium size reusable infrastructure (the JWAM framework) in which the goal was to modularize exception handling, extracting the exception detection and handling code to aspects. This study used the AspectJ language. This study found out that AOP promotes better tolerance to changes in the specification of the error handling behavior of a system, better support to incremental development, better reuse, and decreased system size.

Castor Filho et al. [3] conducted another study about the adequacy of the AspectJ language to modularize exception handling. They refactored four full-fledged applications (three object-oriented and aspect-oriented), moving exception handling to aspects. This study employed a set of static code metrics to compare the refactored and original versions of each target system. Similarly to the Lippert and Lopes study, Castor Filho and colleagues concluded that the degree of tangling and scattering decreased with the aspectization. Moreover, despite the growing number of operations, these new operations were simpler and responsible solely for implementing error handling. Furthermore, this study found out that reusing exception handlers is difficult and that previous findings pertaining to the amount of reuse that could be achieved by using AOP were not generally applicable. A limitation of both of these studies is that they assumed a correlation between exception handler reuse and number of LOC. As evidenced by our study, this is not the case and large amounts of reuse can be achieved without a reduction in the number of LOC of the system.

Recently, Cacho and colleagues [2] 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. The evaluation of EJFlow has shown it promotes increased reuse of exception handling code. Similar results were obtained by Hoffman and Eugster [16], with the use of explicit join points (EJP), a means to reduce the amount of obliviousness of aspect-oriented programs. These two works represent important improvements over AspectJ and their results are complementary to ours. In fact, the results of Setion 3 might serve as a motivation for the development of new language constructs aiming to improve AspectJ, in particular, and AOP languages, in general.

V. Concluding Remarks

In this paper, we presented a study aiming to assess the potential of different modularization approaches to promote intra-application reuse of exception handling code. More specifically, we compared aspect-oriented and object-oriented implementations of three different applications and employed a set of metrics specifically targeting the exception handling concern. The study results have shown that aspect-oriented programming promotes reuse of exception handling code. Even though a certain amount of reuse can also be obtained by employing pure OO programming, the study provides evidence that AOP allows for a larger amount of reuse while resulting in a better textual separation of concerns and less concern scattering. At the same time, for the three systems, there was considerably more code related to the error handling concern in the aspect-oriented versions of the target systems than in the object-oriented versions. These results contradict two previous well-known studies that evaluated the suitability of AOP to modularize error handling [21][3].

It is important make it clear that the modularization and reuse of exception handling was performed exclusively within each application. In the future, we plan to assess the possibility of reusing aspectized exception handling code across different applications, ideally based on a common underlying platform. Applications from different development platforms might use very different exceptions and handling strategies. Another thread of future work that we intend to follow consists of evaluating how the reuse promoted by AspectJ aspects affects software maintenance activities. Finally, our ultimate goal is to develop a tool capable of automatically detecting duplicated exception handlers and combining them into error handling aspects, so as to maximize code reuse

Acknowledgments

We would like to thank the anonymous referees, who helped to improve this paper. Júlio and Juliana are supported by CAPES/Brazil. Cristiane and Hítalo are supported by FACEPE/Brazil. Sérgio is partially supported by CNPq, grants 309234/2007-7 and 480489/2007-6. Fernando is partially supported by CNPq/Brazil, grants 308383/2008-7,

481147/2007-1, and 550895/2007-8. Emanoel and Nathalia are supported by CNPq/Brazil. This work is partially supported 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.

REFERENCES

[1] Cabral, B.; Marques, P. (2007), "Exception Handling: A Field Study in Java and .NET", in Proc. of the 21st European Conference on Object Oriented Programming, Berlin, Germany.

[2] Cacho, N; Dantas, F; Garcia, A; Castor, F. (2009), "Exception Flows made Explicit: An Exploratory Study", in: Proceedings of the 23rd Brazilian Symposium on Software Engineering. Fortaleza, Brazil. October 2009. To appear.

[3] Castor Filho, F. et al. (2006), "Exceptions and aspects: The devil is in the details", in: Proceedings of the 14th SIGSOFT FSE, pages 152–162.

[4] Castor Filho, F.; Garcia, A; Rubira, C. (2007), "Extracting Error Handling to Aspects: A Cookbook", In: 23rd IEEE International Conference on Software Maintenance.

[5] Coelho, R; Rashid, A; Garcia, A; Ferrari, F. C.; Cacho, Nélio; Kulesza, Uirá; von Staa, A.; Lucena, C. J. P. (2008), "Assessing the Impact of Aspects on Exception Flows: An Exploratory Study", in : Proceedings of the 22nd European Conference on Object-Oriented Programming, pages 207—234, Paphos, Cyprus, July.

[6] Cristian, F. (1979) "A recovery mechanism for modular software", In: 4th ICSE, pages 42–51.

[7] Dooren, M. van; Steegmans, E. (2005), "Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations", In: Proceedings of OOPSLA'05, pages 455–471.

[8] Eaddy, M., Aho, A., Murphy, G. C. (2007), "Identifying, Assigning, and Quantifying Crosscutting Concerns". Proceedings of the 1st Workshop on Assessment of Contemporary Modularization Techniques, Minneapolis, USA.

[9] Figueiredo, E.; et al. (2008), "Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability", In: Proceedings of the 30th International Conference on Software Engineering, pp. 261-270.

[10] Fowler, M. (1999), "Refactoring: Improving the Design of Existing Code", Addison-Wesley.

[11] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995), "Design Patterns: Elements of Reusable Software Systems". Addison-Wesley.

[12] Garcia, A.,; C. Rubira, C.; Romanovsky, A.; Xu, J. (2001), "A comparative study of exception handling mechanisms for building dependable object-oriented software", In: Journal of Systems and Software, 59(2):197–222.

[13] Garcia, A. et al. (2006), "Modularizing design patterns with aspects: A quantitative study. In: Trans. AOSD 2006, 1:36-74.

[14] Goodenough, J. B. (1975) "Exception handling: Issues and a proposed notation." Comm. of the ACM, 18(12):683–696.

[15] Greenwood, P. et al. (2007), "On the impact of aspectual decompositions on design stability: An empirical study", In: Proc. of the 21st ECOOP.

[16] Hoffman, K.; Eugster, P. (2008), "Towards Reusable Components with Aspects: An Empirical Study on Modularity and Obliviousness" In: Proceedings of the 30th International Conference on Software Engineering (ICSE).

[17] Juergens, E; Deissenboeck, F.; Hummel, B.; Wagner, S. (2009), "Do Code Clones Matter?". In: Proc. of the 31st International Conference on Software Engineering (ICSE'2009), pages 485—495, Leipzig, Germany, May.

[18] Kickzales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.; Irwin, J. (1997), "Aspect-Oriented Programming", In: ECOOP.

[19] Kulesza, U. et al. (2006), "Quantifying the effects of aspect-oriented programming: A maintenance study", In: Proc. of the 22nd ICSM, pages 223–233.

[20] Laddad, R. (2003), "AspectJ in Action", Manning.

[21] Lippert M.; Lopes C. V. (2000), "A study on exception detection and handling using aspect-oriented programming", In: Proceedings of ICSE'2000, pages 418–427.

[22] Parnas, D. L.; W¨urges, H. (1976), "Response to undesired events in software systems", In: 2nd ICSE, pages 437–446, San Francisco, USA.

[23] Randell, B.; Xu, J. (1995), "The evolution of the recovery block concept", In: Software Fault Tolerance, chapter 1, pages 1–21. John Wiley Sons Ltd.

[24] Robillard, M.P.; Murphy, G.C. (2003) "Static analysis to support the evolution of exception structure in object-oriented systems". ACM Trans. Softw. Eng. Methodol. 12(2): 191--221.

[25] Sant'Anna, C.; Garcia, A; Chavez, C.; Lucena, C; Staa, A. von (2003), "On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework", in: XVII Brazilian Symposium on Software Engineering, Manaus, Brazil.

[26] Shah, H.; Görg, C.; Harrold, M. J (2008), "Why Do Developers Neglect Exception Handling?" Proceedings of the 4th International Workshop on Exception Handling , Atlanta, USA.

[27] Tarr, P. et al. (1999), "N Degrees of Separation: Multi -Dimensional Separation of Concerns". Proceedings of the 21st International Conference on Software Engineering.

[28] Wong, W. E.; Gokhale, S. S.; Horgan, J. R. (2000), "Quantifying the closeness between program components and features,"Journal of Systems and Software, 54(2):87-98.