On Inter-Application Reuse of Exception Handling Aspects

Júlio César Taveira
University of Pernambuco
Recife, Pernambuco, Brazil
jcft2@dsc.upe.br

Hítalo Oliveira
University of Pernambuco
Recife, Pernambuco, Brazil
hos@dsc.upe.br

Fernando Castor
Federal University of Pernambuco
Recife, Pernambuco, Brazil
castor@cin.ufpe.br

Sérgio Soares
Federal University of Pernambuco
Recife, Pernambuco, Brazil
scbs@cin.ufpe.br

ABSTRACT

In this paper we describe a study whose goal is to answer the question: "Is exception handling a reusable aspect"? To this end, we have systematically attempted to reuse exception handling AspectJ aspects across seven different real applications, some of them from the same domain, some based on the same development platform. Our preliminary results indicate that, due to a number of reasons, inter-application reuse is very difficult to achieve in practice. It was only possible to reuse handlers for particular cases, mainly empty handlers and trivial logging handlers. No handler with more than one line of code could be reused.

1. INTRODUCTION

Exception handling [7] is a mechanism to develop robust software that allows one to structure a program to deal with exceptional situations. Currently, many mainstream programming languages, such as Java, Phyton, and C#, define exception handling mechanisms. An exception handling mechanism supports the separation between the code responsible for handling errors and the code implementing the main system functionality [12]. Ideally, this separation should result in systems that are less complex and, hence, easier to understand and more evolvable and reliable [13].

In spite of all its potential benefits, the use of exception handling is a non-trivial 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 situation 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 [14]. Often, exception handling is implemented in ad hoc way. As a consequence, exception handling code is, in general, not well-understood, tested, and documented [4]. Another common problem in programs written in mainstream programming languages is that exception handling code exhibits a large amount of duplication, i.e., 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 [8], including maintainability and reliability.

With the use of separation of concerns techniques such as aspect-oriented programming (AOP)[9], developers might be able to tame the crosscutting nature of error handling. 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. Previous studies discussed the intra-application reuse of exception handling code promoted by AOP. Lippert and Lopes [11] 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 [2] conducted an independent study targeting four deployable systems and found out that very little reuse could be achieved through the aspectization of exception handling. The first one of the aforementioned studies 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. A recent study conducted by Taveira et al. [15] addressed some of the limitations of the Lipert and Lopes [11] and Castor et al. [2] studies. It had the goal of assessing the extent to which AOP promotes intra-application reuse of exception handling code. It found out that aspects do promote reuse in a number of realistic situations, though this reuse does not translate into reduced code size.

None of the aforementioned studies have attempted to understand whether it is possible to reuse error handling code across applications and whether aspects would be an appropriate tool for this task. This paper is an initial attempt to fill in this gap and continue the study of Taveira et al. [15]. We attempt to assess the extent to which AOP and, in particular, AspectJ, promote inter-application reuse of exception handling code. Our study targeted seven medium-size real applications whose original implementation was in Java. These applications were chosen so as to represent situations where reuse might be easier to achieve, e.g., four of them

are Eclipse Plugins, two of them have similar requirements (both are metrics collection tools for Java programs). Our analysis of the results is qualitative. We found out that, unlike reusing error handling within the same application, inter-application reuse is very difficult to achieve in practice and, for most cases, requires non-trivial, system-wide refactoring or a priori planning.

2. INTRA-APPLICATION REUSE

In this section we present the study of Taveira et al. [15], which assessed the suitability of AspectJ to reuse exception handling code within applications. In this previous study, three applications implemented originally in Java were refactored. They are significant for our study because they are real, real applications, with different domains and platforms, and have some relevant characteristics: (i) all of them have at least 15K LOC and at least 1% of LOC pertaining to exception handling; (ii) the applications are from different domains and written in Java; (iii) the applications have many non-trivial exception handlers. For each application, two new versions were developed, one where exception handling is modularized using AspectJ aspects ("refactored AO" version) and another one where exception handling is modularized using Java classes ("refactored OO" version). In the latter case, exception handlers (the code within the `catch` blocks) were moved to special-purpose classes in an attempt to reuse the handlers leveraging only the mechanisms available in the Java language.

For the refactored OO and refactored AO versions, all the exception handling code that could be extracted to classes or aspects was put in a single class/aspect per package. Next, we conducted a search for similar exception handlers within the handler class or aspect of each package. Methods or advice implementing duplicated exception handlers were then eliminated. References to handler methods were modified to use the appropriate handlers. After this, the next step was to combine duplicated handlers appearing in different packages. The last was to compare handler classes or aspects across unrelated packages. In this case, duplicated handlers are moved to a global parent handler class or aspects and the handler classes are modified to inherit from it.

The main goal of the Taveira et al study was 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. Some results presented expected values. For instance, there was no variation between original version and refactored OO versions in terms of the number of code blocks pertaining to exception handling, `try` and `catch` blocks. However, these number were very different when we compared the original and refactored AO versions, although not as much as reported by Lippert and Lopes [11]. In general, AOP promoted a large amount of reuse of error handling code. Notwithstanding, the code overhead imposed by AspectJ is non-trivial. As a consequence, the overall size of the target systems of the study did not decrease, even though error handler reuse was considerable in all of them. The number of handlers was sensibly lower in the refactored AO versions but the amount of error handling code was much higher (for one specific system, the refactored AO version exhibit 103% more lines of error handling code than its OO counterpart).

3. STUDY SETTINGS

3.1 Target Systems

Our study targeted seven applications. These are real Java applications that have some relevant characteristics. The first important characteristic is their size. Almost all target systems have at least 10 kLOC and exception handling code accounts for at least 1% of this amount. Moreover, all of them have at least 80 components (classes and interfaces). These values vary widely between applications, though. The second important characteristic is the heterogeneity of the target systems. Most of the applications stem from different domains. However, we have specifically selected some applications that belong to the same domain because this might present some reuse opportunities. In the same vain, amongst the seven target systems, four of them are plugins for the Eclipse plataform. The target systems of this study are: Checkstyle[1], TeXlipse[2], EIMP[3], JHotDraw[4], Java Pet Store[5], AOPMetrics[6] and Metrics[7]. Table 1 lists the target systems of the study and, for each one of them, provides some important numbers and a short description. All the target systems of our previous study [15] were also employed in this new study. For each application, a new version was developed where the exception handling code was extracted to aspects.

3.2 Extracting Exception Handling

This study compared two different versions of each target system: (i) an original version, implemented in Java; and (ii) and AO version, with the exception handling concern implemented in AspectJ. To obtain the two versions of each target system, we extracted exception handling code from the original versions to new aspects whose sole responsibility is to implement the exception handling concern. We consider that exception handling code is all the code that exists only due to the need to handle exceptions, accordance to the ideas of Eaddy et al. [5]. 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. The extraction of error handlers to aspects was performed as a refactoring. Great care was taken to avoid modifying application behavior. After refactoring, we ran all the automated tests bundled with the target systems (two of them had such tests). For the ones that did not include tests, we executed them and compared the results of the original and AO versions. Moreover, in this activity, as well as when combining similar handlers (Section 3.3), modifications to the code were kept localized and did not change application semantics beyond the extraction or combination

[1] http://eclipse-cs.sourceforge.net/
[2] http://texlipse.sourceforge.net
[3] http://eimp.sourceforge.net
[4] http://www.jhotdraw.org/
[5] http://java.sun.com/developer/releases/petstore/
[6] http://aopmetrics.tigris.org/
[7] http://metrics.sourceforge.net/

Application	LOC	LOC EH	#Components	Dev. Plataform	Description
Checkstyle	19.2K	1211	290	Eclipse Plugin	Checks coding style rules in Java programs.
TeXclipse	31.7K	906	489	Eclipse Plugin	LaTeX editor plugin for Eclipse.
EIMP	9K	440	218	Eclipse Plugin	Eclipse instant messaging plugin supporting MSN, GTalk and other communication systems.
JHotDraw	23K	293	400	Java SE	Java GUI framework for technical and structured graphics.
Java Pet Store	17.5K	1709	330	Java EE	Demo for the Java platform, Enterprise Edition.
AOPMetrics	3K	61	80	Java SE	Metrics collection tool for Java and AspectJ systems.
Metrics	16K	733	292	Eclipse Plugin	Metrics collection tool for Java systems.

Table 1: Applications used in this work.

of handlers. For example, we did not modify library dependencies (e.g., changing a logging library), even though such modifications might have yielded better reuse.

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 [2]. In this phase of the extraction, 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. When a handler depends on local variables of the enclosing method, we first employ the Extract Method [6] 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. The `finally` blocks were moved to separate advice to promote the conceptual separation between exception handlers and clean-up actions.

3.3 Reusing Exception Handling

Before attempting to reuse error handlers across applications, we started out by reusing it within each system. In this manner, the job of finding reuse opportunities amongst the target systems becomes more manageable because there are fewer cases to inspect, without loss of generality. To this end, we adopted the procedure described in Section 2 to reuse exception handling code. Some complications inevitably arise when one tries to combine different but similar exception handlers. For example, handler advice that has different return types can be combined when they do not really need to return anything (only the join point to which they are associated does). In general, developers must consider a large number of issues 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 application, we proceed to attempt to reuse handlers amongst the target systems. We employ a disciplined approach to

```
public privileged aspect FiguresExceptionHandler {
... //Mode code
   declare soft : CloneNotSupportedException:
         FA_cloneHandler();
   pointcut FA_cloneHandler():
         execution(Object FigureAttributes.clone(..));
   Object around() throws InternalError: FA_cloneHandler(){
         try { return proceed(); }
         catch (CloneNotSupportedException e) {
             throw new InternalError();
         }
    }
}
```

(a) Aspect complete in Intra-Application reuse.

```
public abstract aspect CloneErrorAbstractExceptionHandling{
        declare soft : CloneNotSupportedException :
                cloneNonSuportedGeneral();
        abstract pointcut cloneNonSuportedGeneral();
        Object around() throws InternalError:
                cloneNonSuportedGeneral(){
        try { return proceed();}
        catch (CloneNotSupportedException e){
              throw new InternalError();
        }
    }
}
```

(b) Reusable abstract aspect for reuse Inter-Application.

```
public privileged aspect CloneErrorImplEH extends
            CloneErrorAbstractExceptionHandling{
    pointcut cloneNonSuportedGeneral():
          execution(Object FigureAttributes.clone(..));
}
```

(c) Concrete aspect that implement only the pointcut in the project.

Figure 1: An example of intra-application reuse.

search for reuse opportunities in different systems (Section 3.4). Whenever we spot a candidate for reuse, we create a new abstract aspect containing an abstract pointcut and the advice corresponding to the handlers that can be reused. After this, the next step is to create, for each application where the handler can be reused, a new aspect extending the reusable abstract aspect, implementing only the abstract pointcut. Figure 1 presents an example. Figure 1(a) shows an initial case where only one aspect is responsible for handling exception `CloneNotSupportedException`. This solution is adequate for a single application. Figure 1(b) shows how this approach can be adapted to work with multiple applications without duplicating code. Each application that uses the reusable handler needs only to implement am aspect with a concrete pointcut, as shown in Figure 1(c).

3.4 Searching for Reuse Opportunities

The search for reuse opportunities was performed by searching the code textually. For this search we conducted a disciplined inspection according to the categories of exception handlers proposed by Cabral and Marques [1]. We have chosen this classification because it is recent and is the result of examining a large number of real systems implemented in Java and C#. In general, we attempted to reuse handlers

that adopted the same handling strategy. This approach was complemented by ad hoc inspection based on the type of the caught exception and similarity between handlers.

The search for similar exception handlers based on the Cabral and Marques [1] classification is more realistic than performing a completely ad hoc search, since it reduces the need for knowledge of all handlers in the target systems and fits most of them within a small number of categories. The set of handler categories that we considered in this study is presented in Table 2.

Category	Description
Empty	The handler is empty, it has no code and does nothing more than cleaning the stack.
Log	Some kind of error logging or user notification is carried out.
Alternative/ Static Configuration	In the event of an error or in the execution of a finally block some kind of pre-determined (alternative) object state configuration is used.
Throw	A new object is created and thrown or the existing exception is re-thrown.
Return	The handler forces the method in execution to return or the application to exit.
Close	The code ensures that an open connection or data stream is closed. Another action that belongs to this category is the release of a lock over some resource.
Others	Any kind of action that does not correspond to the previous ones.

Table 2: Exception handling strategies proposed by Cabral and Marques [1].

4. RESULTS

The results obtained in this study were very different from our previous study focusing on intra-application reuse. We found out that reusing error handling across applications is not possible in most of the cases and requires some a priori planning. However, a priori planning does not make sense when we consider multiple applications developed independently and at different points in time. In this section we present our preliminary findings. Our analysis is qualitative, unlike some previous studies targeting aspects and error handling [2, 11, 15], which employed a variety of metrics. Since only a very little amount of inter-application reuse could be achieved, the code of each target system prior to reusing the exception handlers and afterwards did not differ much. Therefore, the metrics would not show interesting results.

4.1 Cases where Reuse is Possible

For the target systems of this study (Section 3.1), it was possible to reuse exception handlers across applications for only three exception handling strategies (Table 2): (i) empty handlers; (ii) throwing handlers; and (iii) logging handlers. Notwithstanding, except for the empty handlers, only a few instances of each strategy could be reused. Empty handlers are easy to reuse because the only factor that distinguishes two empty handlers is the type of exception that each handler catches. Therefore, the combination of two empty handlers for two arbitrary checked exceptions E_1 and E_2 requires only the definition of a new handler that catches the least general common supertype of E_1 and E_2. If the common supertype is `Exception`, the handler must be careful in rethrowing any accidentally caught unchecked exceptions, to avoid changing the system behavior. This approach is not applicable to unchecked exceptions in general because, since they are not subject to Java type checking, it might result in exceptions caught accidentally.

Handlers that simply throw a new exception could also be reused, although much less than we initially predicted. In fact, throwing handlers could only be reused across two applications. Even though there are many throwing handlers that catch the same exceptions, they usually throw different ones. Table 3 illustrates this phenomenon. The second column shows all the exceptions caught by throwing handlers in four target systems, whereas the third presents the exceptions these handlers throw. Only handlers that caught `CloneNotSupportedException` and threw `InternalError` (indicated in the table by "*") could be reused and only for two situations in two target systems. Figure 3 shows an example of this case. Moreover, contradicting our intuition that applications based on the same development platform (e.g., Eclipse plugins) would yield more reuse opportunities, reuse was only possible for two applications that did not share a development platform (one of them is a plugin whereas the other is a standard Java application).

```
Object around() : FileMatchPattern_cloneHandler() ||
        FileMatchPattern_cloneFileSetHandler() ||
        FileMatchPattern_cloneProjectHandler() ||
        FileMatchPattern_cloneWorkingCopyHandler() ||
        AbstractFilter_cloneHandler() ||
        cloneHandle(){
    try {
        return proceed();
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}
```

Figure 2: Example of throws strategy.

The third case, and the most significant, was the possibility to reuse the exception handling code that used the Log strategy (Table 2), in three applications. Initially, each application has a class that provides a logging method. However, the logging methods of different applications might have different parameters or names, which hinders reuse. To address this problem, we implemented an interface, named `ILogObject`, responsible for unifying access to logging classes/APIs. Then, for each application, we implemented the interface so as to interoperate with the subjacent logger method. The reusable aspect, `LogAbstractHandler`, needs only to know this interface. Figure 3 presents this aspect.

The code snippet in figure defines two abstract pointcuts, as well as two abstract methods. Concrete aspects must implement these abstract pointcuts and methods. The pointcuts capture points in a system where checked exceptions and all exceptions should be logged. The methods are responsible for getting appropriate text message, considering the join point where the exception was thrown, and for getting the object that has the log method (an instance of `ILogObject`). There are two handler advice that work along the lines of the two abstract pointcuts. The first advice handles only checked exceptions. This means that even though it catches `Exception`, it rethrows any instance of `RuntimeException` that it encounters. The second advice logs any exception, independently of whether it is checked or not. Using this ap-

Application	Exception Captured	Exception Raised
Checkstyle Plugin	NumberFormatException, CheckstyleException, CloneNotSupportedException*, CheckstylePluginException	IllegalArgumentException, CheckstyleException, CoreException, IOException, SAXException, InternalError*
TeXclipse Plugin	Exception, IOException, CoreException, BadLocationException, InterruptedException	SoftException, TexDocumentParseException, BuilderCoreException, CoreException, RuntimeException, OperationCanceledException
JHotDraw	CloneNotSupportedException*, ClassNotFoundException, InstantiationException, IllegalAccessException, NoSuchMethodError	InternalError*, JHotDrawRuntimeException, IOException
Pet Store	ServiceLocatorException, CreateException, RemoteException, GeneralFailureException, XMLDocumentException, Exception, FinderException, OPCAdminFacadeException, SQLException, ServiceLocatorException, CreateException, JMSException, CatalogDAOSysException, MailerAppException, NamingException, JspTagException	AdminBDException, OPCAdminFacadeException, EJBException, FormatterException, GeneralFailureException, PopulateException, ServletException, SAXException, SoftException, CatalogException, JspTagException TransitionException, RuntimeException,

Table 3: Exceptions caught and thrown by throwing handlers

```
public abstract privileged aspect LogAbstractHandler {
    public abstract pointcut checkedExceptionLog();
    public abstract pointcut exceptionLog();
    Object around(): checkedExceptionLog() {
        Object result = null;
        try { result = proceed();
        } catch (RuntimeException re){
            throw re;
        } catch (Exception e) {
            String logText = getMessageText(
             thisEnclosingJoinPointStaticPart.getId());
            getLogObject().logGeneral(logText, e);
        }
        return result;
    }
    Object around(): exceptionLog() {
        Object result = null;
        try {
            result = proceed();
        } catch (Exception e) {
            String logText = getMessageText(
             thisEnclosingJoinPointStaticPart.getId());
            getLogObject().logGeneral(logText, e);
        }
        return result;
    }
    public abstract String getMessageText(int pointcutId);
    public abstract ILogObject getLogObject();
}
```

Figure 3: Abstract Aspect.

proach, it was possible to reuse the same handler 21 times in the Checkstyle Plugin, 42 times in TeXlipse, and 25 times in Metrics. However, this approach has a cost, as discussed in the next section.

One important observation about the situations where reuse was possible is the number of LOC in each handler. In the first case, empty handlers, each handlers has effectively zero LOC. In the other two cases, each handler has only one LOC. In other words, only extremely simple handlers could be reused across applications. This is intuitive, the higher the number of LOC within a handler, the more complex and (probably) more application-specific it is.

4.2 ...And where it is Not

In addition to factors discussed in previous studies [2, 11, 15], inter-application error handling reuse brings many new complications with it. Most of these problems do not make sense in the context of intra-application reuse, the main focus of previous studies. It is worth noting that these problems were found when aspectizing error handling in real, medium-size applications. None of them are toy examples and we believe that these problems could also surface during practical aspect-oriented software development.

The first and perhaps most obvious problem is the difference amongst the application domains of the target systems. Except for Metrics and AopMetrics, all the target systems belong to different domains. Different domains imply in different requirements and, potentially, different errors (and exception types) and ways of treating them. Therefore, even though in many cases handlers captured the same exceptions in different systems and adopted similar handling strategies, small differences, e.g., the type of a thrown exception (Table 3), made reuse infeasible. Moreover, even within the same domain, no reuse could be achieved. We could not reuse any handlers across Metrics and AopMetrics, which suggests that other factors also have a strong impact on reuse. This leads us to the next problem.

Different libraries and development platforms, which encompass sets of libraries, also hinder inter-application reuse. Many of the studied systems, in spite of differing domains, had common subsets of requirements whose realization produced code that could not be reused. An instructive example is the need to parse XML files. Some of the target systems use DOM, whereas others use SAX. These libraries have different features, offer incompatible APIs and signal errors using different exceptions. Similar problems were also observed in areas such as user interfaces and file/database access. Diversity of development platforms further complicated matters. As mentioned before, no reuse could be achieved between Metrics and AopMetrics, even though they belong to the same domain. Among the reasons for this phenomenon we include their differing platforms (Metrics is an Eclipse plugin and AopMetrics is a standard Java applications), which means that they use different libraries and have different sets of standard exceptions.

When considering inter-application reuse, the build environment where a reusable handler is compiled might not be the same as the build environment of the application where it will be used. As a consequence, different compilers or even different versions of the same compiler might have adverse effects on reuse. To compile a reusable error handling aspect

in source code format together with the rest of an application, in general, no special precautions are necessary, except that reusable exception handlers should not be in the same compilation units (files) as the non-reusable units. On the other hand, if the handlers are compiled beforehand, additional matters must be considered. In this case, the employed compilers and their versions must be observed (for both the reusable handlers and the rest of the application). We have experienced this problem in our study. We compiled reusable error handling aspects using the ajc compiler[8], version 1.6.5, beforehand and obtained an error when attempting to compile an application together with these precompiled handlers using ajc 1.5. Another option, which we also investigated, is the **abc**[9] compiler. However, abc does not cover all the current features of AspectJ. Some of these features, for example, the `thisEnclosingJoinPointStaticPart.getId()` method (Figure 3), are necessary for some of the reuse strategies that we have devised. Threfore, it was not viable to use this compiler for most of the target systems.

Considering that different platforms are associated with little to no reuse, an important question is whether it is easier to reuse error handlers across applications based on the same development platform. As discussed in the previous section, we found out that the answer is "depends". In this study, all the Log handlers that could be reused were implemented within Eclipse plugins. However, this is more a consequence of the uniformity with which logging APIs are employed than of commonalities stemming from the development platform. Since almost all logging APIs, independently of the development platform, work similarly (one method responsible for logging whose parameters are a text message and complementary information), it was possible to wrap the differences amongst the three target systems using an interface and an abstract aspect. Furthermore, as shown in Figure 3, the type of the caught exception does not make much of a difference when logging errors. This eliminates one of the factors that hamper reuse for handlers implementing other strategies, such as throwing an exception (Table 3). Hence, even though there was some correlation between common development platforms and reuse, it would be preposterous to say that there was also causality.

Even when these problems of different domains, development platforms, and libraries do not occur, there is still another issue: The implementations of the handling strategies for the same exception in different points of the system. A single exception may have different strategies implemented in the same system, due to different members implementing them, or even to the time when the strategy is implemented, e.g., due to a delayed schedule or different team members. This problem is even more disparate if we consider different applications, with quite different teams, cultures, and environments.

5. CONCLUDING REMARKS

In this paper, we presented a study to assess the impact that exception handling code modularization can have on different applications. With this aim, the exception handling code of seven systems was modularized with AspectJ. This modularization task first internally reused the aspects in each application and then tried to reuse code between the seven applications. The study results indicate that we can reuse (some) exception handling aspect. However, the achieved level of reuse was well beyond what was initially expected and very different from the level of reuse obtained in other published studies [2, 11, 15].

It is important to emphasize that these are real applications, being relevantly used and developed by different teams. They also have different goals and platforms. Therefore, a possible future work is to investigate the exception handling modularization of different versions of a same system, maybe a Software Product Lines (SPL) [3]. Other future work is to assess how much reuse is possible if the system development considers reuse from scratch, again SPL is a candidate domain of investigation.

[8] http://www.eclipse.org/aspectj/doc/next/devguide/ajc-ref.html
[9] http://abc.comlab.ox.ac.uk/

6. ACKNOWLEDGMENTS

We would like to thank the anonymous referees, who helped to improve this paper. Júlio is supported by CAPES/Brazil. Sérgio is partially supported by CNPq, grants 309234/2007-7 and 480489/2007-6. Fernando is partially supported by CNPq, grants 308383/2008-7, 481147/2007-1, and 550895/2007-8. This work is partially supported by INES, funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.

7. REFERENCES

[1] Cabral, B. et al. *Exception Handling: A Field Study in Java and .NET.* In: 21st ECOOP, Germany, 2007.
[2] Castor Filho, F. et al. *Exceptions and aspects: The devil is in the details* 14th SIGSOFT FSE, 2006
[3] Clements, P. C., Northrop, L. *Software Product Lines: Practice and Patterns* Addison-Wesley, 2001.
[4] Cristian, F. *A recovery mechanism for modular software.* 4th ICSE, 1979.
[5] Eaddy, M., Aho, A., Murphy, G. C. *Identifying, Assigning, and Quantifying Crosscutting Concerns.* In: 1st ACoM Workshop, 2007.
[6] Fowler, M. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999
[7] Goodenough, J. B. *Exception handling: Issues and a proposed notation.* In: Comm. of the ACM, 1975.
[8] Juergens, E. et al. *Do Code Clones Matter?* In: 31st ICSE, 2009.
[9] Kickzales, G. et al. *Aspect-Oriented Programming.* In: 11th ECOOP, 1997.
[10] Laddad, R. *AspectJ in Action.* 1st ed., Manning, 2003.
[11] Lippert, M.;Lopes, C. *A study on exception detection and handling using aspect-oriented programming.* In: 22nd ICSE, 2000
[12] Parnas, D. L. and Würges, H. *Response to undesired events in software systems.* In: 2nd ICSE, 1976
[13] Randell, R. and Xu, J. *The evolution of the recovery block concept.* John Wiley Sons Ltd, 1995
[14] Shah, H. et al. *Why Do Developers Neglect Exception Handling?* In: 4th IWEH, 2008
[15] Taveira, J. C. et al. *Assessing Intra-Application Exception Handling Reuse with Aspects.* In: 23rd SBES, 2009.