Implementing Modular Error Handling with Aspects: Best and Worst Practices

Fernando Castor Filho *Alessandro Garcia*
Cecília Mary F. Rubira

Technical Report - IC-06-22 - Relatório Técnico

November - 2006 - Novembro

Implementing Modular Error Handling with Aspects: Best and Worst Practices

Fernando Castor Filho[1], Alessandro Garcia[2], and Cecília Mary F. Rubira[1]

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

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

Resumo One of the fundamental motivations for employing exception handling in the development of robust applications is to lexically separate error handling code so that the latter and the normal code can be independently modified. Also, this scheme makes it possible for the normal code to be reused across various applications, using different error handling strategies. Since error handling is an inherently crosscutting concern, it is usually assumed that it can be better modularized by the use of aspect-oriented programming (AOP) techniques. However, recent studies argue that the ad hoc use of AOP can be detrimental to the quality of a system. When applying AOP to modularize exception handling, developers need to follow clear and simple practices in order to obtain a well-structured system design. If developers do not receive proper guidance, software systems whose exceptional behavior is modularized through aspects can exhibit problems, such as unexpected flows of control and swallowed exceptions, that stem from poorly designed/implemented error handling code. In this paper, we address this problem by providing some design/implementation guidance regarding the use of AOP to modularize exception handling. We propose a classification for error handling code based on the factors that we found out have more influence on its aspectization. Moreover, we present a scenario catalog comprising combinations of these factors and analyze how these scenarios positively or negatively affect the task of aspectizing exception handling.

1 Introduction

Exception handling [12] mechanisms were conceived as a means to improve modularity of programs that have to deal with exceptional situations [6]. Ideally, an exception handling mechanism should enhance attributes such as reliability, maintainability, and understandability, by making it possible to write programs where: (i) the code for error detection, error handling, and the normal behavior are lexically separate so that they can be modified independently [21]; (ii) the impact of the code responsible for error handling in the overall system complexity is minimized [22]; and (iii) an initial version that does little recovery can be evolved to one which uses sophisticated recovery techniques without a change in the structure of the system [21]. The use of exception handling in the construction of several real-world, large-scale systems and its inclusion in many modern mainstream programming languages, such as Java, Ada, C++, and C#, attest its importance to the current practice of software development.

In spite of the pivotal role of modular error handling in the overall system quality, there is not much implementation/design guidance in the literature on how to use exception handling mechanisms to develop error handling code that is modular. Most of the software development methodologies used in practice pay little attention to the design of a system's exceptional behavior [23]. Furthermore, most of the good programming/design practice cookbooks/catalogues, like the refactoring [8] and design pattern [9] catalogues, either concentrate on the system's normal behavior or are too generic to be successfully applied to modular exception handling.

The popularization of new mechanisms for separation of concerns, such as aspect-oriented programming (AOP) [14], further aggravates this problem as new decompositions of the system's exceptional behavior become possible. Although 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 [16,17], the ad hoc use of this new paradigm is sometimes detrimental to the quality of a system [4]. If developers do not receive the proper guidance, software systems whose exceptional behavior is modularized through aspects can manifest a number of problems that stem from poorly designed/implemented error handling code. Examples include swallowed exceptions [18] and resource leaking [3,25]. As pointed out by Bodkin [2], the use of AOP to implement modular error handling is valuable, but there are definitely situations where this is not the case. However, to the best of our knowledge, there are not any works in the literature that help developers in deciding when to aspectize.

In this paper, we address this problem by providing some design/implementation guidance regarding the use of AOP to modularize exception handling. We propose a classification for error handling code based on the factors that we found out have more influence on the aspectization of exception handling. We use the proposed classification to devise a set of scenarios that comprise combinations of these factors and indicate whether aspectization is beneficial or harmful in each of these scenarios. Our goal is twofold: (i) to assist developers of new systems to modularize the exceptional behavior with aspects, so that they can focus on the beneficial scenarios and avoid the harmful ones; and (ii) to assist maintainers of existing systems in the task of refactoring error handling code to aspects by showing when aspectization is worth the effort and when it is not.

This paper is organized as follows. The next section briefly describes the AOP paradigm and the AOP language we have used, AspectJ [16]. Section 3 provides an overview of a study we have conducted to evaluate the benefits and drawbacks of using AOP to modularize error handling. Section 4 presents the proposed classification for error handling code. It also includes a catalog of scenarios that can be derived from the interactions among the elements of the classification. Section 5 presents an evaluation of the proposed catalog. Section 6 reviews some related work. The last section rounds the paper and points directions for future work.

```
1 public aspect ConnectionPoolHandler {
2   pointcut setAutoCommitEH() : call(void Connection.setAutoCommit());
3   after(Connection con) throwing (SQLException e) : setAutoCommitEH()
4       && target(con){
5     con.close();
6   }
7   ...
8 }
```

Figura 1. A simple exception handling aspect.

2 AOP and AspectJ

AOP was proposed as a means to improve the modularization of systems that include *crosscutting concerns*. A crosscutting concern can affect several units of a software system and usually cannot be isolated by traditional OO programming techniques. A typical example of crosscutting concern is logging. The implementation of this concern is usually scattered across the modules in a system, and 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 [16].

AspectJ [16] 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. A *pointcut* picks out certain join points and contextual information at those join points. AspectJ includes operators for the definition of pointcuts formed by the combination of various join points. Examples of join points include method call (caller's context), method execution (callee 's context), and field access. Line 2 of Figure 1 declares a pointcut, named `setAutoCommitEH`, that selects calls to method `setAutoCommit()` of class `Connection`. *Advice* are pieces of code executed *before*, *after*, or *around* a selected a join point. In the latter case, the advice may alter the flow of control of the application and replace the code that would be otherwise executed in the selected join point. Lines 4-7 declare an advice that is executed *after* the join points selected by `setAutoCommitEH` when their execution ends by throwing `SQLException` (Line 4). This advice specifies that the target of the calls to `setAutoCommit()` can be referred to in the advice body through variable `con` (Lines 4-5). Finally, *aspects* are units of modularity for crosscutting concerns. They are similar to classes, but may also include the AspectJ-specific elements. Figure 1 defines an aspect named `ConnectionPoolHandler`. Aspects are associated with pure Java code (the "base code" in AOP terminology) by means of a process called weaving.

3 Background

The ideas we present in this paper are derived from lessons learned from a study we have conducted to assess the benefits of modularizing exception handling code with aspects.

The study consisted of refactoring existing applications so that the code responsible for implementing heterogeneous error handling strategies was moved to separate aspects. We have performed quantitative and qualitative assessments of four systems, three written in Java and one in AspectJ. These systems were developed by third parties from industry and academia. We describe the study and its results in detail elsewhere [4].

Our study has focused on the handling of exceptions. We have moved `try-catch`, `try-finally`, and `try-catch-finally` blocks in the four applications to aspects. Hereafter, we refer to these types of blocks collectively as `try-catch` blocks. Moreover, we use the terms `try` block, `catch` block (or handler), and `finally` block (or clean-up action) to explicitly refer to the parts of a `try-catch` block. Handlers in the aspects were implemented using *after* and *around* advice. Whenever possible, we have used *after* advice, since they are simpler. Clean-up actions were implemented exclusively as *after* advice. New advice were created on a per-`try`-block basis. In situations where multiple `catch` blocks are associated with a single `try` block, we have created a single advice that implements all the `catch` blocks. Pointcuts were employed to select the exception-throwing code. We did not aspectize method signatures (`throws` clauses) and the raising of exceptions (`throw` statements) because they are part of the error detection concern. In several occasions, we have modified the implementation of a method in order to expose join points that AspectJ could select more directly or contextual information required by exception handlers, for example, the values of local variables. We restricted the "allowed" modifications to well-known refactorings [8] and their aspect-oriented counterparts [20] and did not use refactorings that modified more than one class.

Quantitative evaluation of the results was based on the application of a metrics suite [11] to both the original and refactored versions of the systems. This suite includes metrics for separation of concerns, coupling, cohesion, and size. Qualitative evaluation leveraged the quantitative results and included a multi-perspective analysis of the refactored systems, including (i) the reusability of the aspectized error handling code, (ii) the beneficial and harmful aspectization scenarios, and (iii) the scalability of AOP to modularize exception handling in the presence of other crosscutting concerns.

One of the most important lessons we learned from the aforementioned study was that, although the use of AOP to separate exception handling from base code can be beneficial, that depends on a combination of several factors. In many common situations, in order to modularize error handling using aspects, some a priori redesign is necessary. If the exception handling code in an application is non-uniform, strongly context-dependent, or too complex, ad hoc aspectization might not be possible or lead to code which: (i) worsens the overall quality of the system, as measured by the employed metrics suite; (ii) does not represent the original design of the system; and/or (iii) exhibits "bad smells" [8].

4 Aspectizing Exception Handling

The existence of many situations where aspectization is not a good idea motivated us to try to understand precisely what factors make it easier or harder to modularize error handling with aspects. Based on our experience in refactoring the four target systems of our study, in this paper we propose a simple classification for exception handling code. This classification emphasizes factors that have the strongest impact on the aspectization of exception handling. Analyzing the interactions among these factors, we present a catalog of error handling scenarios and, for each scenario, indicate whether aspectization is beneficial or harmful. Section 4.1 describes the proposed classification and Section 4.2 presents the scenario catalog.

4.1 The Proposed Classification

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

Tangling of `try-catch` Blocks. The first category describes where in the body of a method a `try-catch` block appears. We consider a `try-catch` block *tangled* if it is textually preceded or followed by a statement in the body of the method where it appears. Declarations and statements that appear textually before a `try-catch` block make it *tangled by prefix*. Accordingly, statements that appear after a `try-catch` block make it *tangled by suffix*. A `try-catch` block appearing within a loop is considered tangled by prefix and suffix, independently of the placement of other statements. A `try-catch` block whose `try` block surrounds the whole method body is considered *non-tangled*. Tangling of `try-catch` blocks impacts the selection of the exception-throwing code as a join point and might complicate the flow of control of the method when a handler is implemented as an advice. The shaded `try-catch` block in Figure 2(a) is tangled by suffix because it is follwoed by a call to `doSomething()`. The `try-catch` block in Figure 2(b) is tangled by prefix, because the declaration of variable `i` precedes it.
Nesting of `try-catch` Blocks. A `try-catch` block can be also classified as *nested* or *non-nested*. A nested `try-catch` block is one that is contained within a `try` block. The shaded code snippet of Figure 2(c) is an example of nested `try-catch` block. Nesting of `try-catch` blocks can make it difficult to understand the flow of exceptions in a given context. Moreover, implementing nested `try-catch` blocks as advice may require these advice to be ranked, so that they mimic the behavior of the original code. This is necessary if some of these advice are associated with the same join points. We do not consider nested a `try-catch` block that appears within a `catch` or `finally` block. Such a `try-catch` block is part of the system's exceptional behavior. It does not make sense to

```
class C1 {
  void m1(){
    try{ m2();
    }catch(E e){...}
    doSomething();
  } //m1()
} //C1
```

(a)

```
class C2 {
  void m3(){
    int i = 0;
    try{ m4(i);
    }catch(E e){...}
  } //m3()
} //C2
```

(b)

```
class C3 {
  void m5(){
    try{
      try{ m6();
      }catch(E1){...}
    }catch(E2 e){...}
  } //m5()
} //C3
```

(c)

```
class C4 {
  void m7(){
    try{ m8();
    }catch(E1 e){
      try{ ...
      }catch(E2){...}
    } //catch
  } //m7()
} //C4
```

(d)

```
class C5 {
  void m9(){
    try{
      m10(); //throws E
      doSomething();
    }catch(E e){...}
  } //m9()
} //C5()
```

(e)

```
class C6 {
  void m11(){
    try{
      doSomething();
      m12(); //throws E
    }catch(E e){...}
  } //m11()
} //C6
```

(f)

Figura 2. Tangling and nesting of `try-catch` blocks, and placement of exception throwing code.

aspectize it separately, as we are interested in separating normal and exceptional behavior using aspects and not indiscriminately aspectizing each and every `try-catch` block in an application. The shaded `try-catch` block in Figure 2(d) is **not** nested for the purposes of aspectizing error handling.

Placement of Exception-Throwing Code. We consider exception-throwing code any statement within a `try` block that can throw exceptions that some handler within the same method catches. Such handler can be directly associated with the `try` block or with some enclosing `try` block. An exception-throwing statement is *terminal* if it is not followed by any statements in the `try` block. When referring to a `try-catch` block, we consider its exception-throwing code terminal if all of its exception-throwing statements are terminal. In the general case, terminal exception-throwing code includes a single terminal exception-throwing statement. However, there are special cases where a `try-catch` block may have more than one terminal exception-throwing statement, such as when it has `if` statements. In Figure 2(e), the call to method `m10()`, which throws exception `E`, is *non-terminal* because it is followed by a call to `doSomething()`. In Figure 2(f), the call to `m12()` is terminal because it is the last statement of the `try` block.

Dependency of Exception Handlers on Local Variables. This category classifies exception handlers in two groups: those that do and those that do not depend on local variables. By "local variables" we mean variables defined within the containing context of a given `try-catch` block. Dependency on local variables hinders aspectization, as the joint

```
class C1 {
  void m1(){
    int x = 0;
    try{ x = m2();
    }catch(E e){
      log(‘‘error:’’  + x);
    }
  } // m1()
} //C1
```

(a)

```
class C2 {
  void m3(){
    int x = 0;
    try{ x = m4();
    }catch(E e){
      x = m5();
    }
    doSomething(x);
  } // m3()
} //C2
```

(b)

```
class C3 {
  void m6(){
    try{ m7();
    }catch(E e){
      Logger.log(e);
    }
    doSomething();
  } //m6()
} //C3
```

(c)

```
class C4 {
  void m8() throws E2{
    try{ m9();
    }catch(E1 e){
      throw new E2(e);
    }
  } // m8()
} //C4
```

(d)

```
class C5 {
  void m10(){
    try{ m11();
    }catch(E e){
      return;
    }
    doSomething();
  } // m10()
} //C5
```

(e)

```
class C6 {
  void m12(){
    for(int i=0;i++;i<10){
      try{ m13();
      }catch(E e){
        break;
      }
    } //for
  } //m12()
} //C6
```

(f)

Figura 3. Dependency on local variables and flow of control after handler execution.

point models of most AO languages (AspectJ included) cannot capture information stored by these variables. If a handler reads the value of one or more local variables, moving it to an aspect often requires the use of the Extract Method refactoring [8], in order to expose the variables as parameters of a method. In some cases, it is possible to avoid such refactoring by selecting the join point where the value is generated (for example, the return value of a method) and saving it for later retrieval during errror handling. In Figure 3(a), the `catch` block reads the value of local variable `x`. If a handler performs assignments to local variables, more radical refactoring may be necessary. In general, it is even harder to aspectize exception handling in this case. The resulting code almost always exhibits bad smells such as “Temporary Field” [8]. The handler in Figure 3(b) performs an assignment to local variable `x`.

Flow of Control After Handler Execution. The fifth category is related to how a `catch` block ends its execution. After the execution of a *masking* handler, control passes to the statement that textually follows the corresponding `try-catch` block. Figure 3(c) presents a masking handler. After its execution, method `doSomething()` is invoked. A *propagation* handler finishes its execution by throwing an exception, as shown in Figure 3(d). In this case, control is transferred to the nearest handler that catches the exception. A *return* exception handler ends its execution with a `return` statement, as shown in Figure 3(e). After the handler returns, system execution resumes from the site where the handler's method was called. Finally, a *loop iteration* handler is declared inside a loop and executes a statement such as `break` or `continue`. Figure 3(f) presents an example that ends with a `break` statement. If a `catch` block includes alternation commands (e.g. `if` statements) and, as a consequence, can end its execution in different ways, we assume the worst case scenario in terms of ease of aspectization. We consider loop iteration handlers the hardest to aspectize, followed by masking, return, and propagation handlers.

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

4.2 The Proposed Scenario Catalog

Using the proposed classification, it is possible to describe several exception handling scenarios representing recurring situations in software development. Table 1 consolidates the most frequent ones we have identified in our study. Each row represents one or more scenarios. To avoid repetition, if a category is marked more than once in the same row (e.g. tangled by suffix and non-tangled marked), the row represents more than one scenario and an OR semantics is adopted. For example, row #1 describes situations where a `try-catch` block is non-tangled, independently of nesting and placement of exception-throwing code. Also, the corresponding `catch` block does not depend on local variables and can end its execution by masking, propagating an exception, or returning. For simplicity, hereafter we use the term "scenario" to refer to all the scenarios that a row in Table 1 represents.

The rightmost column of Table 1 indicates whether it is beneficial ("Yes") or harmful ("No") to modularize exception handling with aspects in a given scenario. In general, we considered aspectization to be beneficial in a scenario if: (i) it has a positive effect on the quality attributes of the system, namely, coupling, cohesion, size, and separation of concerns, as measured by the employed metrics suite (Section 3), when comparing the original and aspectized code for instances of the scenario; and (ii) it generally does not exhibit bad smells. In some rows, the rightmost column indicates that the choice of aspectizing exception handling in a given scenario depends on factors that are not taken into account by the proposed classification. These cases are marked as "Depends". In the rest of this section we explain each scenario of Table 1.

Scenario #1. Since the `try-catch` block is not tangled, it is possible to select the entire execution of the method as the join point of interest. Moreover, it is straightforward to ex-

#	Tangled `try-catch` blocks			Nested `try-catch` blocks		Placement of exception-throwing code		Handler depends on local variables			Flow of control after handler execution				Should extract to aspects?
	pre.	suf.	no	yes	no	non-t.	term.	read	write	no	mask.	prop.	ret.	loop	
1			X	X	X	X	X			X	X	X	X		**Yes**.
2	X			X	X	X	X			X	X	X	X		**Yes**.
3	X	X			X	X	X			X		X	X		**Yes**.
4	X	X		X		X	X			X		X	X		**Yes**.
5	X	X			X		X			X	X				**Yes**.
6	X	X		X			X			X	X				**Yes**.
7	X	X			X	X				X	X				**Depends**.
8	X	X		X		X				X	X				**Depends**.
9	X	X	X		X	X	X	X			X	X	X		**Depends**.
10	X	X	X	X		X	X	X			X	X	X		**No**.
11	X	X	X	X	X	X	X		X		X	X	X		**No**.
12	X	X	X	X	X	X	X	X	X	X				X	**No**.

Tabela 1. Exception handling scenarios according to the proposed classification

tract the handlers to an aspect. It does not matter whether the exception-throwing code is terminal or not. Flow of control after handler execution is not a problem, as long as it is not loop iteration. If there are nested `try-catch` blocks, it may be necessary to rank the handler advice (i.e. advice implementing exception handlers after aspectization has taken place) if some of them are associated with the same join points. The examples in Figures 2(c), 2(d), 2(e), 2(f), and 3(d), and are instances of this scenario.

Scenario #2. A `try-catch` that is tangled by prefix is easy to aspectize because the tangling code (1) is not part of the join point of interest, namely, the exception-throwing code, and (2) does not influence the flow of control of the program after an exception is handled. Figure 2(b) presents an example of a tangled by prefix `try-catch` blocks. For the following scenarios, even though a `try-catch` block might be tangled by prefix, we do not represent this in Table 1 because, in practice, tangling by prefix has the same effect as no tangling.

Scenario #3. In this scenario, the `try-catch` block is tangled by suffix but it is possible to specify a pointcut that captures the execution of the whole method and associate the handler advice with it. In the original code, the handler ends its execution by either returning or throwing an exception, thus ignoring whatever comes after the `try-catch` block. Hence, the tangling does not matter because the code that textually follows the `try-catch` block is never executed when an exception is thrown from the `try` block. Figure 3(e) presents an example. Since the `catch` block returns, after exception handling the call to `doSomething()` is not executed. The exception-throwing code does not matter in this case because the execution of the whole method is the join point of interest.

Scenario #4. This scenario bears strong resemblance to Scenario #3, but the nesting of `try-catch` blocks introduces some complications that hinder aspectization. Since the

```
aspect A1 {
  pointcut callM7():
    call(* C3.m7()) &&
    withincode(* C3.m6());
  around() :  callM7()  {
    try { proceed();}
    catch(E e) {Logger.log(e);}
  } //advice
  ...
} //A1
```

(a)

```
class C1 {
  void m1(){
    try{
      m2(); //throws E
      m3(); //throws E
      m4(); //throws E
    }catch(E){...}
    doSomething();
  } //m1()
} //C1
```

(b)

Figura 4. An aspectization for Scenario #4 and an instance of Scenario #7.

`try-catch` block is nested, it may be necessary to order handler advice associated with the same join points. Moreover, many (3+) levels of nesting combined with tangled by suffix `try-catch` blocks often result in complex code and special care should be taken in order to avoid the introduction of subtle bugs. In spite of these complications, aspectization is still beneficial in this scenario. In the general case, no a priori refactoring has to be applied to the original code and the aspectized code often exhibits good structuring that reflects the original design of the exceptional behavior.

Scenario #5. Figure 3(c) presents a general example of this scenario. It is easy to aspectize exception handling in this case because it is possible to directly associate a handler advice with the exception-throwing statement, in the example, the call to method `m7()`. Since the exception-throwing statement is terminal and the handler has a termination behavior, after execution of the aspectized handler, control passes to the statement that follows the `try-catch` block in the original implementation. In Figure 3(c), that would be the call to method `doSomething()`. Figure 4(b) shows an aspect implementing this solution. The shaded code defines a pointcut that selects calls to method `m7()` made within the code of method `m6()`. An *around* advice implements the handler. The `proceed()` statement is defined by AspectJ and executes the selected join point from an *around* advice, in this example, the call to `m7()`. In this scenario, it is not feasible to associate a handler advice with the execution of method `m6()`because, after the execution of this advice, the method would return and ignore the call to `doSomething()`.

Scenario #6. This scenario is similar to Scenario #5, the difference being the fact that the `try-catch` block is nested. As is the case with Scenario #4, nesting makes it harder to modularize error handling with aspects in Scenario #6 but, in general, does not negatively affect the quality of the final code. Therefore, aspectization is beneficial in this scenario.

Scenario #7. Moving the error handling code to an aspect in this scenario is hard because of the combination of non-terminal exception-throwing code, tangled by suffix `try-catch` block, and a handler that has a masking behavior. Because of these factors, it is difficult to mimic the behavior of the original code using an aspectized handler. Figure 4(b) presents an instance of this scenario.

Associating a handler advice to the exception-throwing code (as in Scenarios #5 and #6) is not an adequate solution. For example, if we associated a handler advice with the call to method `m2()` in Figure 4(b), after exception handling the call to method `m3()` would be executed. However, in the original implementation, control should be passed to the statement following the `try-catch` block, in the example, the call to `doSomething()`. Selecting the whole method (as in Scenarios #1-#4) as the join point of interest is also not adequate. In the example, if we associated a handler advice with the execution of method `m1()`, control would return to the method's caller after exception handling. This implies that the call to `doSomething()` would not be executed. The bottom line is: we would like to associate a handler advice to a block containing more than one statement, just like a `try` block, instead of a single statement or a whole method. Unfortunately, AspectJ does not include mechanisms for directly selecting a block of statements.

A possible workaround to these complications is to extract the code within the `try` block to a new method and associate a handler advice to the execution of the extracted method. This solutions has drawbacks, however: (i) it modifies the original design of the system's normal behavior; (ii) it might result in the creation of methods that do not make sense by themselves; and (iii) it is not effective in all cases because it is not always possible to extract a piece of code to a new method [8]. In general, in this scenario, aspectization is only beneficial if it is possible to extract the code within the `try` block to a new method and this method makes sense by itself, i.e., it could have been created by the developers of the system.

Scenario #8. This scenario presents the same complications as Scenario #7, but the nesting of `try-catch` blocks makes it even harder to modularize error handling with aspects. Except for simple cases with a single level of nesting and just a few `try-catch` blocks, aspectization is harmful in this scenario.

Scenario #9. Aspectizing handlers that use local variables of the enclosing method in AspectJ is difficult because the language does not make it possible for advice to access the local variables visible at a join point of interest. To try to address this problem, we analyzed four general ways of exposing a local variable to an advice: (i) to transform the variable into a field of the enclosing class; (ii) to capture some use of the variable and store this information for later retrieval; (iii) to extract the piece of code where the variable is used to a new method and expose this variable as a parameter of the extracted method; and (iv) to extend the class that defines the thrown exception so that its instances can store the values of the variables.

The first solution is conceptually bad because local variables only make sense in the method where they appear. Transforming a variable into a field implies that the variable is relevant to the class as a whole. Moreover, explicit precautions have to be taken in order to avoid inconsistencies caused by multi-threaded programs. Finally, this solution suffers from the “Temporary Field” bad smell. The second solution is conceptually better, but it cannot always capture the join point of interest. For example, an aspect can capture the value of a variable to which the result of a method is assigned or a variable provided as an argument in a method call, because it is possible to select the method call as a join point of interest. However, it cannot get the value of a variable that is simply used as a placeholder for the result of a mathematical expression. Another difficulty associated with the second solution is that it imposes a large implementation overhead. In the best case, it is necessary to create a new pointcut, a new advice, and a field just to obtain the value of the variable and store it. Also, it is necessary to guarantee that this field is thread-safe. It is important to stress that this solution is different from the first one, since the field is only visible to the aspect. The third solution has all the limitations and drawbacks highlighted in the description of Scenario #7. The fourth solution is conceptually a reasonable option. On the one hand, the capability of storing contextual information from the throwing site is one of the useful features of using objects to represent exceptions [10]. On the other hand, it might make the class that defines the exception too dependent on a specific situation. Another disadvantage is that this solution requires changes to a program that are non-local (they involve another class). Finally, this solution is only applicable to cases where the exception object is constructed in the same context where the variable is declared.

The decision of refactoring the handler to an advice in this scenario depends on: (i) whether it is possible to expose the values of the local variables of interest in a way that is conceptually reasonable and does not impose a large implementation overhead; and (ii) whether it would be beneficial to aspectize the handler if we ignored the dependency of the handler on the method’s local variables and classified the code according to Scenarios #1-8. The code in Figure 3(a) is an example of this scenario.

Scenario 10. Combinations of nested `try-catch` blocks and handlers that read values of local variables are, in general, difficult to aspectize and not worth the effort. Code adhering to this scenario is often very complex and includes handlers accessing local variables defined various nesting levels above. In order to modularize exception handling, it is almost always necessary to intensively refactor the OO implementation. The resulting aspect-oriented code often includes a sensible implementation overhead.

Scenario 11. In this scenario the `catch` block performs assignments to local variables of the containing method. The code in Figure 3(b) is an instance of this scenario. Amongst the four solutions we presented for exposing local variables to advice, only the first one makes it possible for the aspect to change the value of the variable without duplicating code. As pointed out above, this solution has some severe problems that make it inappropriate for practical use. Another possible solution is to transform the local variable into an array of its type comprising a single element and then use the fourth approach described above to store this array so that the handler can obtain it. In this manner, the handler becomes capable of modifying the variable directly. Besides the limitations highlighted above, this approach is obviously a hack that uses arrays as a means to simulate a pass-by-result

Attributes	Metrics	Definitions
Separation of Concerns	Concern Diffusion over Components	Number of components that contribute to the implementation of a concern.
	Concern Diffusion over Operations	Number of methods and advice which contribute to a concern's implementation.
Coupling	Coupling Between Components	Number of components declaring methods or fields that may be called or accessed by other components.
Cohesion	Lack of Cohesion	Measures the lack of cohesion of a class or an aspect in terms of the in Operations amount of method and advice pairs that do not access the same field.
Size	Lines of Code (LOC)	Number of lines of code.
	Number of Operations	Number of methods and advice of each class or aspect.

Tabela 2. Metrics Suite

semantics. The task of moving a `catch` block to an advice is described by the Extract Fragment to Advice [20] refactoring, an aspect-oriented adaptation of the Extract Method refactoring. Hence, similar constraints apply. In general, it is not possible to extract a code snippet to a new method if this snippet performs assignments to local variables of the containing method [8]. The same holds when one tries to extract a code snippet to an advice. Consequently, as a rule it is harmful to modularize exception handling using aspects in this scenario.

Scenario 12. Loop iteration handlers are usually too strongly coupled with the context where they appear. In this scenario, the problem is that commands `break` and `continue` appear inside a `catch` block and are part of the error handling concern. However, the loop where the corresponding `try-catch` block appears is part of the normal behavior of the system. Since these commands have to be associated with a loop, otherwise a compile-time error occurs, it is not possible to extract the `catch` block to an advice "as-is". As a consequence, loop iteration handlers inevitably require some redesign of the original code before they can be aspectized. Figure 3(f) presents an example of this scenario.

5 Evaluation

To evaluate the efficiency of the proposed catalog, we conducted a case study. Our goal was to assess whether a developer following the recommendations of the catalog would produce code of higher quality than a developer not using it. We measured the quality of a system using some of the metrics in the metric suite mentioned in Section 3. The metrics we used are briefly described in Table 2. A detailed description is available elsewhere [11]. Based on these quality attributes, we compared three versions of the same system: (i) an object-oriented version, implemented in Java; (ii) an aspect-oriented version where all the `try-catch` blocks were refactored to aspects; and (iii) an aspect-oriented version where all the instances of "Yes" scenarios (Section 4.2) and some instances of "Depends" scenarios were refactored to aspects. We considered it worth aspectizing an instance of a "Depends" scenario if this did not require any refactoring to be applied to the code im-

	"Yes"						"Depends"			"No"			
Scenario	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**9**	**10**	**11**	**12**	**Total**
Number of instances	5	4	25	1	2	5	3	0	4	0	4	0	53

Tabela 3. Number of instances of each scenario in the original version of Telestrada.

plementing the normal behavior. As pointed out by Parnas [21], the ability to modify error handling code without having to change the normal code is one of the benefits of using exception handling.

The case study targeted one of the systems we used in our previous study (Section 3). We selected Telestrada, a traveler information system developed for a Brazilian national highway administrator. For the case study, we selected some self-contained packages of one of its subsystems comprising approximately 3350 LOC (excluding comments and blank lines) and more than 200 classes and interfaces. The system includes 53 `try-catch` blocks, excluding the ones that appear within `catch` blocks. As pointed out in Section 4.2, the latter are aspectized as part of their cointaining `catch` blocks. We consider each `try-catch` block a scenario instance. Table 3 shows the number instances of each scenario in the OO version of Telestrada. Amongst the 7 instances of "Depends" scenarios, we extracted error handling code to aspects in 5 of them, the four instances of Scenario #9 and one of the instances of Scenario #7.

The results of the evaluation for the three systems are depicted in Table 4. For all the metrics, lower values imply in better results. The "Partially Aspectized" version of the system is the one where we used the catalog to guide aspectization. For some of the metrics, the three versions exhibit similar results. The Number of Operations is higher in the two aspect-oriented versions because each handler advice counts as a new operation, unlike `try-catch` blocks in the original implementation. Moreover, Concern Diffusion over Components is better in the two aspect-oriented versions, being more than 18% lower than the original version. This difference is expected, as aspects help in localizing the implementation of crosscutting concerns. The most interesting result presented by Table 4 pertains the cohesion metric. In our previous studies, systems where exception handling was modularized through aspects consistently exhibit worse results for this metric [4]. In some cases, the measure of this metric for the aspect-oriented version was more than 25% higher than the OO implementation. This trend persists when we compare the original and entirely aspectized versions. The measure of Lack of Cohesion in Operations for the latter is 8% higher than for the OO version. Moreover, it was almost 15% higher than the measure obtained by the partially aspectized version. This result came to a surprise to us, as we did not expect the partially aspectized version to perform better than the OO implementation.

Metric Version	Concern Diffusion over Components	Concern Diffusion over Operations	Coupling between Components	Lack of Cohesion in Operations	Lines of Code	Number of Operations
Original	22	42	188	434	3424	432
Entirely Aspectized	18	42	182	469	3368	471
Partially Aspectized	18	42	185	409	3431	465

Tabela 4. Resulting of measuring the three versions of Telestrada.

6 Related Work

Since the publication of the first design pattern catalog [9], several catalogs that try to amass design/implementation time-tested knowledge regarding OO software development activities have appeared. Most of them focus on general purpose refactorings [8] and design patterns [9]. A few describe specific solutions and guidelines, based on practical experience, concerning error handling [7,18].

In recent years, some works have surfaced which try to gather together similar knowledge regarding aspect-oriented software development activities [5,13,16,20]. Many authors have devoted attention to developing refactoring catalogs [5,13,15,20] for aspect-oriented software. A few of them [15,5] include specific procedures for moving exception handling code to aspects. Laddad presents the "Extract Exception Handling" refactoring. The description of the refactoring centers around the effects of using it to extract trivial error handling code, but does not explain when it is useful (or possible) to apply it in practice. Cole and Borba describe a set of behavior-preserving programming laws that can be combined in order to specify a refactoring that works along the same lines as "Extract Exception Handling". These laws include preconditions that indicate when it is possible to apply them. While refactorings targeting error handling code concentrate on the mechanics of moving a `try-catch` block to an aspect, the work we present identifies situations where this is beneficial and where it is not. We use different strategies to refactor out exception handling code into aspects with the goal of assessing the ease of performing this task and the quality of the final code.

Some authors describe their experience in the application of exception handling to real OO software systems in the form of best practices [7] and anti-pattern [3,18] catalogs. An early paper by Cargill [3] describes some of the problems that often happen in C++ programs due to the complex flows of control that exception handling creates. Doshi [7] presents two small catalogs of best practices: one for specifying APIs whose services can throw exceptions and another one for developing the client code of these APIs (the actual exception handlers). A recent article by McCune [18] presents a list of error handling anti-patterns, some of them very well-accepted (e.g. "catch and ignore" and "throwing `Exception`"), and others that seem to be based on the personal opinions of the author (e.g. "log and throw"). Best practices and anti-pattern catalogs provide guidelines on what exception handlers should and should not do. Our work complements these catalogs by providing design/implementation guidance to developers using AOP techniques to enhance the modularity of their error handling code.

The most well-known study focusing on the interplay between exception handling and AOP was performed by Lippert and Lopes [17]. The authors used AOP to modularize exception handling in a large OO framework. The authors also attempted to identify some situations where it was easy to aspectize error handling code and highlighted a few where it was not. However, since the study targeted a reusable infrastructure, instead of a full-fledged application, it is difficult to extrapolate their findings to the development of real-world applications where error handling is application-specific. In another paper [4], we present very preliminary versions of the classification and scenario catalog that appear in Section 4. This preliminary classification is much simpler than the one we present here and this reflects on the scenario catalog. Additionally, this previous work does not attempt to evaluate the use of the catalog.

7 Concluding Remarks

This paper makes two contributions to the current body of knowledge about the interplay between AOP and the error handling concern. First, a classification of exception handling code in terms of the factors that we found out have more influence on aspectization. Second, an analysis of the interactions amongst these factors, grouped as sets of scenarios, considering how they affect (i) the ease of aspectizing exception handling code, and (ii) the quality of the refactored code when compared to the quality of the original code. An initial evaluation has shown that the proposed catalog helps improving the quality of the final system, when AOP is used to modularize error handling.

This work focused on a single aspect-oriented language, namely, AspectJ. We have not attempted to assess whether the proposed classification and scenarios catalog apply to other aspect-oriented languages. More powerful join point models would make it possible to deal more appropriately with more complicated cases, such as those where handler blocks are tangled or nested, thus affecting the study results. In the future, we intend to experiment with other languages in order to understand: (i) to what extent the ideas we present here are generally applicable; and (ii) how other language features impact the aspectization of exception handling. A priori, we intend to evaluate the pros and cons of using CaesarJ [19] and HyperJ [24] to modularize error handling code.

We still have not evaluated the adequacy of AOP techniques to modularize code responsible for detecting errors. In the fault tolerance literature [1], it is usually argued that error detection should be part of a system's normal activity, since it is too strongly coupled with the normal application code. We believe that investigating if AOP can be employed to modularize error detection is an exciting direction for future work.

Acknowledgements

We would like to thank the anonymous referees, who provided several useful comments and suggestions. We would also like to thank Rodrigo Minetto for reading and commenting a preliminary version of the paper. Fernando is supported by FAPESP/Brazil under grant 02/13996-2. Cecília is partially supported by CNPq/Brazil under grant 351592/97-0, and by FAPESP/Brazil, under grant 2004/10663-8. Fernando and Cecília are also partially supported by FINEP/Brazil under grant 1843/04, project CompGov, which aims to develop a Shared Library of Components for e-Government. Alessandro is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008. Alessandro is also supported by the TAO project, funded by Lancaster University Research Committee.

Referências

1. T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 2nd edition, 1990.
2. R. Bodkin. Re: Exception handling done thru aop, August 23rd 2004. Email at the aspectj-users mailing list. Address: http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg02849.html.
3. T. Cargill. Exception handling: A false sense of security. *C++ Report*, 6(9), November-December 1994.
4. F. Castor Filho, N. Cacho, E. Figueiredo, R. Ferreira, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: The devil is in the details. In *Proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 2006. To appear.
5. L. Cole and P. Borba. Deriving refactorings for aspectj. In *Proceedings of the 4th ACM Conference on Aspect-Oriented Software Development*, pages 123–134, March 2005.
6. F. Cristian. A recovery mechanism for modular software. In *Proceedings of the 4th International Conference on Software Engineering*, pages 42–51, 1979.
7. G. Doshi. Best practices for exception handling, 2003. Address: http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html.
8. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software Systems*. Addison-Wesley, 1995.
10. A. Garcia, C. Rubira, A. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, November 2001.
11. A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. J. P. de Lucena, and A. von Staa. Modularizing design patterns with aspects: A quantitative study. *Transactions on Aspect-Oriented Programming*, 1:36–74, 2006.
12. J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.
13. S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *Proceedings of 4th Net.ObjectDays Conference*, pages 19–35, September 2003.
14. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, LNCS 1271, pages 220–242, 1997.
15. R. Laddad. Aspect-oriented refactoring, parts 1 and 2, 2003. The Server Side, www.theserverside.com.
16. R. Laddad. *AspectJ in Action*. Manning, 2003.
17. M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 418–427, June 2000.
18. T. McCune. Exception-handling antipatterns, 2006. Address: http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html.
19. M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd ACM Conference on Aspect-Oriented Software Development*, pages 90–99, March 2003.
20. M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th ACM Conference on Aspect-Oriented Software Development*, pages 111–122, March 2005.

21. D. L. Parnas and H. Würges. Response to undesired events in software systems. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 437–446, San Francisco, USA, October 1976.
22. B. Randell and J. Xu. The evolution of the recovery block concept. In *Software Fault Tolerance*, chapter 1, pages 1–21. John Wiley Sons Ltd., 1995.
23. C. M. F. Rubira, R. de Lemos, G. Ferreira, and F. Castor Filho. Exception handling in the development of dependable component-based systems. *Software – Practice and Experience*, 35(5):195–236, March 2005.
24. P. Tarr et al. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.
25. W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 419–433, October 2004.