EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming

Nelio Cacho[1], Fernando Castor Filho[2], Alessandro Garcia[1], Eduardo Figueiredo[1]

[1]Computing Department, Lancaster University, United Kingdom

[2]Department of Computing and Systems, University of Pernambuco, Brazil

{n.cacho, garciaa, e.figueiredo}@lancaster.ac.uk, fernando.castor@dsc.upe.br

ABSTRACT

Many of the problems that stem from the use of exception handling are caused by the local way in which exceptions are handled. It demands that developers understand the source of an exception, the place where it is handled, and everything in between. As a consequence, exceptions easily get "out of control" and, as system development progresses, exceptional control flows become less well-understood, with potentially negative consequences for the program maintainability and reliability. This paper presents an innovative aspect-oriented model for exception handling implementation. In contrast to other exception handling mechanisms, our model provides abstractions to explicitly describe global views of exceptional control flows. As a result, this new model makes it possible to understand exception flows from an end-to-end perspective by looking at a single part of the program. Also, it leverages existing pointcut languages to make the association of handlers with normal code more flexible. The implementation of our proposed model, called *EJFlow*, extends the AspectJ programming language with the aim of promoting enhanced robustness and program modularization. We evaluate qualitatively and quantitatively the proposed exception handling model through a case study targeting a real mobile application.

Keywords

Aspect-oriented programming, exception handling, modularity, exception control flow, metrics.

1. INTRODUCTION

Exception handling [24] is a technique for improving software modularity and reuse in the presence of exceptional conditions. The importance of exception handling is attested by the fact that many mainstream programming languages, such as Java, Ada, C++, and C#, implement exception handling mechanisms. These languages provide constructs to indicate the occurrence of an error (*raise* or *throw* an exception) and to associate a set of recovery actions with the error in order to remedy the problem (*handle* the exception). Error recovery measures are implemented within *exception handlers*. However, we argue that conventional exception handling mechanisms introduce two problems: (1) implicit exceptional control flows, and (2) limited support for reusing normal behavior and exception handlers.

The first problem is based on the assumption followed in existing exception handling models that it is enough to specify the places where a program raises exceptions and the places where it handles them. It means that there is no specification of the global effect of exception occurrences, and *exception control flows* are not explicitly declared. When program module raises an exception, the underlying exception handling mechanism is responsible for changing the normal control flow of the computation within the program to its exceptional control flow [23]. An exception is implicitly propagated through many levels, before it reaches an appropriate handler [23], potentially affecting the program behavior along the way. As a result, traditional exception handling mechanisms provide a solution that is inherently local to a problem that is intrinsically global [38].

The second problem has been recently tackled by aspect-oriented programming (AOP) languages [8, 28]. AOP has provided new composition mechanisms that help to reap the promised benefits of existing exception handling mechanisms. The use of AOP should make it possible to textually separate handler code from the normal code, so that the latter can be reused independently. However, the current AOP languages have some limitations when employed to structure exception handling code. First, they do not provide appropriate join point models for capturing certain exception handlers without leading to program anomalies in certain cases [7]. Second, they do not help much in making the interface between normal and error handling code explicit [8]. Finally, AspectJ-like languages bring software maintenance problems when artificial changes need to be introduced in order to expose contextual information to the aspectized exception handlers [25].

In this context, the contributions of this paper are threefold. First, we discuss the liabilities of conventional and AOP models for exception handling (Section 2). Second, we propose a novel aspect-oriented programming model for exception handling (Section 3) that supports both: (i) explicit exception flows, and (ii) improved support for exception handler reuse. The proposed model exploits existing pointcut languages to make the association of handlers with normal code more flexible. Third, we also present *EJFlow*, an implementation of our exception handling mechanism that extends AspectJ (Section 4). We evaluate our exception handling proposal through a quantitative and qualitative comparison with Java and AspectJ.

AOSD'08 March 31 -- April 4, 2008, Brussels, Belgium

2. LIMITATIONS OF EXISTING EXCEPTION HANDLING MECHANISMS

This section discusses the liabilities of exception handling mechanisms in terms of their limited reuse support (Section 2.1), and implicit exceptional control flows (Section 2.2). It also discusses how existing AOP models ameliorate or exacerbate these problems (Section 2.3).

2.1 Limited Support for Software Reuse

In Java, `try` blocks define exception handling contexts (EHCs) and `catch` blocks define handlers. EHCs are regions in a program where the same exceptions are always treated in the same way. An EHC can have a set of associated handlers, among which a handler is chosen when exceptions are raised within the context. An exception can be re-signaled (propagated) through many levels, before it reaches an appropriate handler. If no handler is found, the program terminates with an error.

Separation of concerns is one of the overarching goals of exception handling. In fact, one of the main motivating factors for the appearance of exception handling mechanisms was to separate the normal and abnormal behavior [35, 36]. Nonetheless, the kind of separation promoted by the exception handling mechanisms of most mainstream programming languages brings only limited advantages [8, 12, 28]. In particular, it hinders reuse of both error handling code (within the same application) and normal code (across applications) [8, 28]. The following code snippet, extracted from an Eclipse plugin, illustrates this.

```
public class CRLFDetectInputStream
  extends FilterInputStream {
        ...
  protected CRLFDetectInputStream(InputStream in,
      ICVSStorage file) {
    super(in);
    try {
      this.filename = getFileName(file);
    } catch (CVSException e) {
      this.filename = file.getName();
    }
  }
}
```

The example above defines the constructor for class `CRLFDetectInputStream`. This class is responsible for detecting the carriage return and line feed characters in input streams. The constructor attempts to obtain the full name of `file` by retrieving it from the file system through method `getFileName()`. If something goes wrong, e.g. the file could not be found, and `CVSException` is raised, the handler simply gets the name stored in variable `file`. In order to reuse class `CRLFDetectInputStream` in a different system, it might be necessary to change this policy, for example, to interrupt program execution when the file cannot be accessed in the file system. To achieve this, it would be necessary to directly modify the `catch` block in the constructor. This kind of accidental reuse is generally considered a bad practice in the software development community [13]. A much more desirable approach would be to simply "unplug" the error handling strategy associated to the constructor and "plug" the new one using only language-provided constructs for exception handling. However, this is currently not possible in any mainstream programming languages. The following code snippet illustrates another undesirable situation:

```
public class EclipseSynchronizer
    implements IFlushOperation {
  ...
  public void endBatching(...)
      throws CVSException {
    try {...} catch (TeamException e) {
      throw CVSException.wrapException(e);}
  } ...
  public IResource[] members(...)
      throws CVSException {
    ...
    try {...} catch (CoreException e) {
      throw CVSException.wrapException(e);}
  } ...
}
```

In this second example, two different methods within the same class, `endBatching()` and `members()`, implement identical exception handling strategies. `TeamException` is a subtype of `CoreException`. In languages such as Java, C++, and C#, it is not possible to implement a single handler and associate it with more than one method, to avoid code duplication. In order to implement a system such that its normal behavior and its error handling code can be reused independently, the latter has to be defined by means of separate methods that the exception handlers invoke. This solution is obviously not ideal, as it imposes an implementation overhead (due to the implementation of `try-catch` blocks) and requires the actual error handling code to be implemented outside of the exception handlers.

Some "research" programming languages, such as Guide [27] and Extended Ada [29], promote the reuse of normal code across applications and the separation of concerns between normal and error handling code by supporting alternative EHCs, such as methods, classes, objects, and exceptions. This is clearly desirable when all the exceptions within one such context, e.g. in the same class, are handled in the same way. Nevertheless, these languages do not provide straightforward mechanisms to reuse exception handling code within the same system. For example, there is no direct way to create a handler for a certain exception and associate it with a subset of the methods of a class without: (i) duplicating the handler code; or (ii) incurring in the overhead of implementing handlers that simply delegate the handling duties to a specialized method. Therefore, in this sense, these languages are not much different from Java, C#, and C++.

2.2 Implicit Exception Control Flows

In our view, the most serious problems with exception handling stem from the fact that it is a global design issue [38]. Existing exception handling mechanisms do not appropriately take this into account [4]. They are based on the implicit assumption that it is enough to specify the places where a program raises exceptions and the places where it handles them. The main consequence of this limitation in existing exception handling mechanisms is that exceptions introduce implicit control flow [6, 30]. If a programmer changes exception-related code, the control flow in apparently unrelated parts of the program may change in surprising ways [38]. This creates two direct complications:

- It becomes difficult to discover where the exceptions raised within a given context will be handled.
- It is also difficult to trace a handled exception to the place where it was originally raised.

In other words, traditional exception handling mechanisms provide constructs for raising and handling exceptions [23]. However, not much support is provided to the task of understanding the paths the exceptions take from the raising site to the handling site [30, 37, 38].

Some languages, e.g. Java, try to alleviate these problems by supporting the definition of an explicit *exception interface*. They require programmers to state the list of unhandled exceptions that each method signals to its clients, otherwise, the compiler will report an error. The main problem with this approach is that it hinders software maintenance [14]. If a new exception is added to the exception interface of a method at the bottom of the method call chain, the exception interfaces of all methods through which the new exception will be propagated also have to be updated. For programs with long method call chains, this is a time-consuming and error-prone task [14, 38], causing severe problems on the system architecture evolution [32].

Another problem with the exception interface approach is that it promotes a well-known undesirable phenomenon named "swallowed exceptions" [37], where programmers introduce empty handlers in a program in order to "silence" the compiler. One last problem is that exception interfaces do not make it possible to link the raising site of an exception to its handling site. They indicate potential propagation paths, but not the one that an exception raised at a certain point in a program takes on its way to an exception handler. As pointed out by Robillard and Murphy [38], even when extreme care is taken during earlier phases of development, exception propagation paths are created in ways that are very hard to predict and control. Moreover, even the limited help exception interfaces provide only applies to some of a program's exceptions. For example, in Java, the rules for exception interfaces do not apply to the so-called *unchecked* exceptions.

Another approach to mitigate the problems created by the global scope of exception handling is to use static analysis tools. There are some tools [9, 19, 39] that, based on the call graph of a program, track the flow of exceptions along the paths in this graph. They provide valuable help for developers trying to understand how a program behaves in the presence of exceptions, and how the exceptions themselves behave. These tools have noteworthy limitations, though. First, they are not as useful when the system is still under development, because the propagation paths might still not be complete (albeit they might be envisioned). Second, they help developers in understanding exception flow paths, but not in enforcing them. In general, a developer cannot, using such a tool, specify expected exception flow paths so that the tool verifies whether the exceptions are handled in the right places. This kind of inspection has to be performed "manually" and *a posteriori* by the developer, with the help of the tool. This is specially problematic during software evolution. A third problem is that there is no direct integration between the tool and the employed programming language. Therefore, in the best scenario, any knowledge acquired through the use of the tool has to be documented by means of comments in the code or some other informal, non-automatically verifiable form of documentation.

The bottom line is that, in general, outside of the raising and handling scopes, exceptions work as an undesirable (and difficult to track) side-effect to the normal code. Many of the problems reported in the literature pertaining to exception handling are related to the inability of properly modularizing it, so that the interface between exceptions/exception handling and the normal code is well-documented. Approaches such as exception interface and static analysis tools alleviate this problem, but only to a limited degree. Besides, they create new problems of their own. To the best of our knowledge, there are no general approaches to modularize error handling code, so that it can be understood and maintained separately from the normal code.

2.3 AOP Liabilities

Aspect-oriented programming languages solve some, though not all, of the problems raised in the previous subsections. They make it possible to textually separate handler code from the normal code, so that the latter can be reused independently. Also, they provide constructs to select multiple points in program execution and associate a single handler to all these points, thus promoting handler reuse within an application. In fact, the kinds of exception handling contexts that can be selected using aspect-oriented programming languages are only limited by the expressiveness of their join point models [8].

Nevertheless, in spite of the growing power of AOP languages, they have some limitations when employed to structure exception handling code. First and foremost, they do not provide appropriate pointcut designators (PCDs) for selecting the raising of an exception. An existing extension to AspectJ acknowledges this problem by proposing a new PCD named `throws` that selects the throwing of an exception as a join point of interest [1]. This new PCD can only be used to select the cases where an exception is explicitly thrown by means of Java's `throw` statement, though. In our view, the raising of an exception should be treated as a runtime event that cannot always be directly traced to `throw` statements. In this manner, handlers can be associated to the normal code in a more abstract (and less fragile) way.

Furthermore, existing aspect-oriented languages do not address the problems raised in Section 2.2, i.e. they do not help in making the interface between normal and error handling code explicit [8]. These languages do not alter the existing paradigm for error handling, where an exception handling mechanism should only provide constructs for raising exceptions and handling them. AOP languages do not provide means for developers to explicitly indicate and enforce at compile time the paths that exceptions should make from the point where they are raised to the point where they are handled. Even though AOP aims to address the implementation of concerns that are inherently non-local, due to their crosscutting nature, the approach it supports for structuring exception handling is still local.

A minor problem is that the `declare soft` construct implemented by some AspectJ-like languages has maintenance problems [8, 25] similar to the ones created by the `throws` clause of Java. If a new exception is propagated throughout a long call chain, a significant number of join points might have to be softened as a result. This problem is less severe, though, because it is possible to reduce its impact by localizing all the join points where a certain exception must be softened in a single `declare soft` statement. This solution has the obvious disadvantage that, if the number of softened join points is large, the code becomes less comprehensible.

3. A NOVEL EXCEPTION HANDLING MODEL

As discussed earlier, it is necessary to provide an approach to (i) properly separate normal code from exception handling code and (ii) make it possible to understand exception flow from an end-to-end perspective, ideally by looking at a single part of the program. In this context, an exception handling model is the core of our approach since it defines the interaction between raising sites with the handling sites. The model that we propose was mostly designed to facilitate the development of robust and modular error handling code. It is an extension of the model that Java and AspectJ support and introduces only two new concepts: *Explicit Exception Channels* and *Pluggable Handlers*. We describe these concepts in the following two subsections. In Section 4 we illustrate the proposed model with a language-agnostic example.

3.1 Explicit Exception Channels

An *explicit exception channel* (*channel*, for short) is an abstract duct through which exceptions flow from a raising site to a handling site. More precisely, an explicit exception channel *EEC* is a 5-tuple consisting of: (1) a set of exception types *E*, (2) a set of raising sites *RS*; (3) a set of handling sites *HS*; (4) a set of intermediate sites *IS*; and (5) a function *EI* that specifies the channel's exception interface. *Exception types*, as the name indicates, are types that, at runtime, are instantiated to exceptions that flow through the channel. For simplicity, we use the term "exception" to refer to both exceptions (runtime elements) and exception type (compile-time element). When necessary, we make the distinction explicit. The *raising sites* are loci of computation where exceptions from *E* can be raised. The actual erroneous condition that must be detected to raise an exception depends on the semantics of the application and on the assumed failure model. For reasoning about exception flow, the fault that caused an exception to be raised is not important, just the fact that the exception was raised. The *handling sites* of an explicit exception channel are loci of computation where exceptions from *E* are handled, potentially being re-raised or resulting in the raising of new exceptions. In languages such as Java, both raising and handling sites are methods, the program elements that throw and handle exceptions.

If an explicit exception channel has no associated handlers for one or more of the exceptions that flow through it, it is necessary to define its *exception interface*. The latter is a statically verifiable list of exceptions that a channel signals to its enclosing context, similarly to Java's `throws` clause. In our model, the exception interface is defined as a function ($Ex_1 \mapsto Ex_2$) that translates exceptions flowing (Ex_1) through the channel to exceptions signaled (Ex_2) to the enclosing EHC.

Raising and handling sites are the two ends of an explicit exception channel. Handling sites can be potentially any node in the method call graph that results from concatenating all maximal chains of method invocations starting in elements from *HS* and ending in elements from *RS*. To keep the model simple, we do not consider a handling site that throws exceptions a raising site. All the nodes in such graph that are neither handling nor raising sites are considered *intermediate sites*. Intermediate sites comprise the loci of computation through which an exception passes from the raising site on its way to the handling site. Intermediate sites in Java are methods that indicate in their interfaces the exceptions that they throw, i.e. exceptions are just propagated through them, without side effects to program behavior. Note that the notions of handling, raising, and intermediate site are purely conceptual and depend on the specification of the explicit exception channel. They are also inherently recursive. For example, an intermediate site of an explicit exception channel can be considered the raising site of another channel. In fact, a single Java `try-catch` block can be seen as a trivial explicit exception channel where the raising site is the same as the handling site.

3.2 Pluggable Handlers

A *pluggable handler* is an exception handler that can be associated to arbitrary EHCs, thus separating error handling code from normal code. A single pluggable handler can be associated, for example, to a method call in a class `C1`, two different method declarations in another class, `C2`, and all methods in a third class `C3`. In this sense, they are an improvement over traditional notions of exception handler. For example, the exception handling model of Guide [27] allows one to bind handlers to blocks, methods, classes, and exceptions, but not all of them at the same time! Another difference is that a pluggable handler exists independently of the EHCs to which it is associated. Therefore, these handlers can be reused both within an application and across different applications. Also, pluggable handlers can be associated to a rich assortment of EHCs, some of them atypical. For instance, besides all the kinds of EHCs that Guide supports, a pluggable handler can also be bound to an execution path within an application, to runtime objects, and to explicit exception channels.

3.3 Making the Proposed Model Concrete

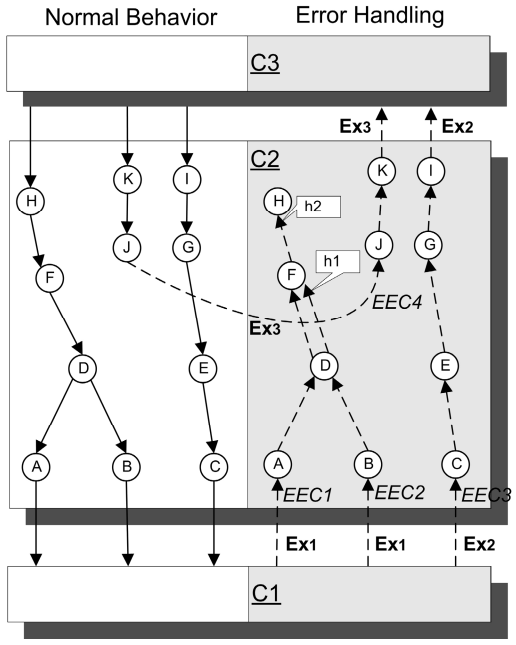

Figure 1: Control flow among components

For the sake of illustration, Figure 1 depicts part of the architecture of a software system, comprising three components: C1, C2, and C3. Component C3 requires services from

component C2 which, in turn, request services from component C1. Each component is divided in terms of its normal behavior (white part) and its error handling code (grey part). The circles inside C2 represent methods. A black arrow from circle a to circle b indicates that method a invoked method b. A dashed arrow indicates that, during the execution of b, an exception can be raised and this exception will be signaled to a. As would be expected, each arrow also indicates that control flow is passed from one method to the other. We also explicitly indicate the types of exceptions that the component C2 encounters and signals (*Ex1-3*).

We can define several different explicit exception channels in terms of the elements of Figure 1. For example, let EEC_1 be the explicit exception channel defined by the tuple *({Ex1},{A},{H,F},{D},{})*. Only exception *Ex1* flows through this channel. Explicit exception channel EEC_1 has method *A* as its sole raising site (even though, according to the figure, *A* receives this exception from the bottommost component) and *D* as its only intermediate site. Methods *H* and *F* are handling sites, since pluggable handlers *h2* and *h1* are bound to them, respectively. The latter catches exception *Ex1* and re-raises it, whereas the former stops its propagation. It is important to stress that method *F* is not an intermediate site because it is associated to a pluggable handler. EEC_1 does not define an exception interface, as it includes handlers for all of its exceptions. In total, the figure indicates four different explicit exception channels. Two of them, EEC_3 and EEC_4, have no handling site. For example, channel EEC_3 includes exception *Ex2* in its interface. It is defined by the tuple *({Ex2},{C},{},{E,G,I},{Ex2↦Ex2})*.

Basically, this model provides the means to specify, in a local manner, non-local information pertaining to exception flows. For example, in Java, to implement explicit exception channel EEC_1, it would be necessary to: (i) include *Ex1* in the exception interface of *A*, *D*, and *F*; and (ii) to implement `try-catch` blocks in methods *F* and *H*. This is not a large amount of work, but the information about the exception that *A* signals and that *H* and *F* handle is scattered throughout four different methods. If we consider a more complex program, where there might be sequences with dozens of method calls, having this information scattered throughout several methods can hinder maintenance and understandability. In this scenario, just looking at some method along the path from a raising to a handling site gives no clue whatsoever about either raising or handling sites.

4. EJFlow

This section presents EJFlow, an AspectJ extension that implements the proposed exception handling model. EJFlow provides means for developers to define explicit exception channels and pluggable handlers in terms of the abstractions supported by AspectJ, namely, pointcuts, advice, and inter-type declarations. More specifically, we define a new pointcut designator, a new kind of advice, and a new inter-type declaration. The pointcut (Section 4.1) defines explicit exception channels in terms of the exceptions that flow through them. The new advice (Section 4.2) implements pluggable handlers and, consequently, handling sites. The inter-type declaration (Section 4.3) allows one to specify the exception interfaces of explicit exception channels. Intermediate sites are computed automatically at compile time by the EJFlow compiler.

4.1 Defining Explicit Exception Channels

EJFlow provides a new pointcut designator, `echannel`, to support the definition of explicit exception channels. This pointcut designator takes a formal parameter, `et`, consisting of the name of an exception type. It matches any join point where the raised exception is type of type `et`. In the EJFlow pointcut language, named pointcut expressions built up using the `echannel` designator are considered explicit exception channels, where the name of the pointcut expression represents the name of the channel. The `echannel` designator can be used to define the explicit exception channels described in Section 3.3. As a first example, the following simple pointcut declares an initial version of channel `EEC1` (Figure 1):

```
pointcut EEC1():echannel(Ex1)
```

Here, the pointcut matches any statements that may signal exception *Ex1*. Furthermore, EJFlow performs a static analysis [39] on the program's method call graph, in order to identify the raising and intermediate sites of a given explicit exception channel. The implementation of `echannel` attempts to locate methods that raise the exception supplied as argument and considers them raising sites. A method can only be considered a raising site if the act of raising the exception is not a consequence of another exception, neither an implicitly propagated one nor an exception raised by a handler. The analysis then proceeds upwards, through the method call graph, considering every method to be part of the explicit exception channel, either as intermediate or handling sites. For the example of Figure 1, the methods *A* and *B* are identified as raising sites of *Ex1*, and *D, F*, and *H* as either intermediate or handling sites. In summary, `EEC1` matches all calls through which exceptions that were raised as a result of the execution of methods *A* or *B* flow, including calls to methods *D*, *F* and *H*.

As mentioned in Section 3, exhaustive definition of exception types `et` can impair the usefulness of our approach. Hence, `echannel` supports AspectJ patterns to match exceptions related to a single class, a full class hierarchy, a class with a wildcard, or a combination of classes using logical operators. Therefore, rather than defining channels for exceptions `SocketException`, `FileNotFoundException`, `EOFException`, and so on, one can use just `echannel(IOException+)` to match all subtypes of `IOException`.

An explicit exception channel like the one defined above is too general to be useful. It is possible to specify more clearly-defined channels by explicit indicating the raising site of a channel. The code snippet below illustrates the definition of two explicit exception channels that include their respective raising sites:

```
pointcut rSite1 : withincode(public void A());
pointcut EEC1() : echannel(Ex1, rSite1);
pointcut rSite3 : withincode(public void C());
pointcut EEC3() : echannel(Ex2, rSite3);
```

The second parameter of a channel definition identifies its raising site. The two examples above define the raising sites as separate pointcuts that the definitions of `EEC1` and `EEC3` use (Figure 1). Notice that the `echannel` designator only supports the specification of channels that have a single raising site. At the implementation level, we see explicit exception channels with multiple raising sites as compositions of simpler channels, each

containing a sole raising site. EJFlow supports the definition of multi-raising site channels by means AspectJ's union operator:

```
pointcut EECComposite() : EEC1() || EEC3();
```

In this manner, the specification of both simple and complex channels remains very simple, in accordance to AspectJ's semantics, and avoids too much syntax overload.

In some cases, a channel might fork at intermediate sites, resulting in two or more different propagation paths for the same exceptions. For example, suppose that there is an extra dashed arrow from *E* to *J* in Figure 1. If we wanted to define `EEC3` to be exactly the same as *EEC3* in Figure 1, it would be necessary to exclude this extra propagation "branch". In EJFlow, to be more specific about a channel, a developer can indicate some of its intermediate sites in its definition. In a similar vein, one can exclude some intermediate sites. In both cases, the semantics is to include or exclude the entire subtree of the channel whose root is the provided intermediate site. Intermediate sites (both included and excluded) are supplied as extra arguments to `echannel`. The following snippet presents a simple example:

```
pointcut rSite3 : withincode(public void C());
pointcut iSite1 : !withincode(public void J());
pointcut EEC3() : echannel(Ex2, rSite3, iSite1);
```

Pointcut `EEC3` above defines an explicit exception channel through which exception `Ex2` flows, that has `rSite3` as its raising site, and that **does not** include the branch that starts in method `J()` and continues to `H()` (notice the "!" symbol in the definition of `iSite1`).

Explicit exception channels defined using only `echannel` are obviously incomplete, as they do not include handling sites nor an exception interface. If one compiles a program that defines such a channel, the EJFlow compiler will indicate a compilation error because there are exceptions that should propagated to the enclosing EHC but are not part of the exception interface of the channel. Unlike Java, EJFlow verifies if exceptions flowing through an explicit exception channel are handled or declared in its exception interface regardless of the exception type, i.e., these rules apply to both checked and unchecked exceptions.

4.2 Plugging Handlers to Exception Channels

In order to specify the handling site of an explicit exception channel, EJFlow provides the `ehandler` advice. This advice is an implementation of pluggable handlers. It encapsulates the exception handling code that is executed when a certain point in an explicit exception channel is reached. Each `ehandler` advice consists of: (i) a set of parameters, like any other advice; (ii) a `boundto` clause specifying the explicit exception channel to which the handler is bound; (iii) an associated pointcut that determines the join point, within the channel, at which the advice executes; (iv) a `catching` clause that indicates the exception to be handled; and (v) a body, the actual handler implementation. To give an example of basic `ehandler` functionality, the following code snippet presents a useful handler in the context of channel `EEC1`:

```
void ehandler() boundto(EEC1()):
    withincode(public void F()){ ... }
```

This advice handles exceptions flowing through explicit exception channel `EEC1`. Specifically, the handler is activated when such exceptions are raised within method `F()`. A pluggable handler can be associated with multiple explicit exception channels by means of AspectJ's set union operator (||). The handler in the following code snippet is executed when exceptions from channels `EEC1` or `EEC2` are raised within method `F()`:

```
void ehandler() boundto(EEC1() || EEC2()):
    withincode(public void F()){ ... }
```

In addition, the `catching` clause further narrows the scope to which the handler is associated. This clause states that the handler should only be executed if the specified exception is caught. It also has the effect of making that exception available for the body of the handler. The following code snippet presents an example of the use of the `catching` clause:

```
void ehandler() boundto(EEC1() || EEC3())
    catching(E1 e):
    withincode(public void F()){ ... }
```

It is also possible to define pluggable handlers that are not associated with channels. In this case, they work as `after throwing` advice, with the difference that, like `around` advice, they can stop the propagation of an exception. The code snippet below shows a simple example:

```
void ehandler() catching(E1 e):
    withincode(public void F()){ ... }
```

4.3 Specifying Exception Interfaces

When a program cannot handle all the exceptions that flow through an explicit exception channel, it is necessary to declare these exceptions in the channel's exception interface. The `declare einterface` inter-type declaration serves this purpose. The following code snippet illustrates the definition of exception interfaces:

```
1:pointcut rSite4() : withincode(public void J());
2:pointcut EEC4(): echannel(Ex3, rSite4);
3:declare einterface : Ex3
4:    echannel EEC4(): execution(void K());
5:
6:pointcut rSite3() : withincode(public void C());
7:pointcut EEC3(): echannel(Ex2, rSite3);
8:declare einterface
9:    echannel EEC3(): execution(void I())
```

The first inter-type declaration (lines 3-4) explicitly indicates the exception to be declared in the exception interface of the channel. Alternatively, the second one (lines 8-9) specifies only the explicit exception channel to which the exception interface is associated. This second format is more general and states that every exception that flows through channel `EEC3` and is not handled is part of the channel's exception interface.

The `declare einterface` inter-type declaration avoids the need to specify the exception interface of each method that acts as a raising or intermediate site within a channel. Therefore, in the example of Figure 1, with the use of the two declarations above, exceptions `Ex2` and `Ex3` do not need to be declared in the `throws` clauses of methods `C`, `E`, `G`, `I`, `J`, and `K`. However, the compiler still performs the static checks. If one of these declarations is removed, the compiler issues an error message.

4.4 Implementation Details

We implemented EJFlow in AspectJ by extending the AspectBench compiler (abc) [1]. This compiler is based on a powerful optimization framework (Soot [42]) which provides the mechanisms to analyze and transform Java bytecode. To create

explicit exception channels, we first obtain from abc's `GlobalAspectInfo` class the list of pointcut declarations which contain the `echannel` designator. Based on this list, we introduce a shadow type to iterate through all the parts of a method where an exception can be raised. As soon as a join point shadow can occur, we construct, based on the algorithm proposed by Fu and Ryder [19], chains of exception-flow to link the corresponding raising site with its possible handling sites (join point shadow).

For each such handling site, the compiler checks each `ehandler` advice associated with that explicit exception channel in the system and determines if the advice's pointcut can match that handling site. If it can match, we add a new entry to the enclosing method's exception handler table for the bytecode corresponding to the handling site. The code for the handler is inlined whenever the `ehandler` advice body implements any `return` operation, otherwise the code is inserted at the end of the shadow as a call to the handler advice. At the same time, checked exception(s) propagated along the call chain are declared in the method interface until meet some handler or channel's exception interface. At the end, we perform a post-weaving analysis to ensure that all exceptions flowing through the explicit exception channel are handled or declared.

Special attention is dedicated to handling sites which belong to many explicit exception channels. In this case, an enhanced *catch-and-reraise* [2] approach is employed to separate into different nested `try`-blocks correlated `ehandler` advice. In addition, we use unique integer constants to distinguish exceptions of the same type that need to be handled differently. Dynamic residue is used to compare, after catching similar exceptions, the passed value with the desired binding; if equal, the exception can be handled, otherwise it is reraised.

5. EVALUATION

This section presents a comparison of the Java, AspectJ and EJFlow implementations for the MobileMedia case study [18] (Section 5.1), based on a suite of modularity metrics (Section 5.2). We present the results by means of charts that represent the data gathered in the measurement process. The Y-axis presents the absolute values gathered by the metrics. We break the results of our analysis in two parts: (i) modularity measures with respect to separation of exception handling concerns (Section 5.3); (ii) modularity measures from the system viewpoint based on coupling, cohesion, and conciseness metrics (Section 5.4). Throughout these sections, when appropriate, we also make a qualitative analysis of the results.

5.1 Target Case Study

We selected a real software application, called MobileMedia, to assess the feasibility and effectiveness of our approach. MobileMedia [18] is a mobile application that manipulates photo, music, and video on mobile devices, such as mobile phones. The application uses various technologies based on the Java ME platform, such SMS, WMA and MMAPI.

We believe that this application is representative of how exception handling is typically used to deal with errors in real software development efforts for two reasons. First, MobileMedia encompasses a large number of exception handlers that implement diverse exception handling strategies ranging from trivial to sophisticated. Second, they present heterogeneous crosscutting relationships involving the normal code, the handler code, the clean-up actions, and other crosscutting concerns.

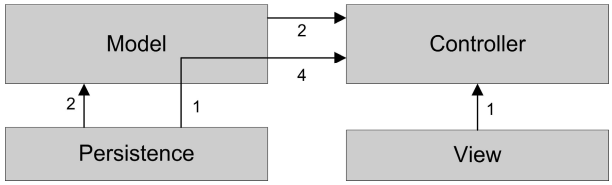

Figure 2: Exception propagation in *MobileMedia*

The design of the MobileMedia application is mainly determined by the use of the Model-View-Controller (MVC) architectural pattern [3]. The exception handling code for the Java implementation followed the design approach described in detail elsewhere [38]. Figure 2 presents a representative partial view of

Table 1. The Metrics Suite [11, 22]

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

the architectural design of the system and the main paths through which exceptions flow between its components. The numbers in the figure represent the number of exceptions flowing on each path. The AspectJ version followed the recommendations of a catalog of best practices [7, 8]. Finally, the details of the EJFlow implementation are described during the discussion of the metrics results. The same strategy was employed for organizing exception handling aspects in the AspectJ and EJFlow implementation. Four aspects were created in each version to contain the exception handling code.

5.2 Metric Suite

The quantitative assessment was based on the application of a metrics suite (Table 1) to the three versions of MobileMedia implemented in Java, AspectJ, and EJFlow. This suite includes metrics for separation of concerns, coupling, cohesion, and size [11,22]. We selected these metrics because they are the metrics which have mostly been applied in a number of empirical studies [5, 8, 18, 22, 25]. The coupling, cohesion, and size metrics are based on classic OO metrics [11]. The original OO metrics were extended to be applied in a paradigm-independent way, supporting the generation of comparable results. Furthermore, the metrics suite introduces four new metrics for quantifying separation of concerns. They measure the degree to which a single concern (exception handling, in our study) in the system maps to the design components (classes and aspects), operations (methods and advice), and lines of code. For all the employed metrics, a lower value implies a better result. Table 1 presents a brief definition of each metric, and associates them with the attributes measured by each one. We have used our tool [17] to collect the coupling, cohesion, and size metrics. The concern metrics required the manual 'shadowing' of the code, i.e. identifying which segment of code contributed to exception handling concern in the MobileMedia. Detailed descriptions of the metrics appear elsewhere [22].

5.3 Separation of Concerns Measures

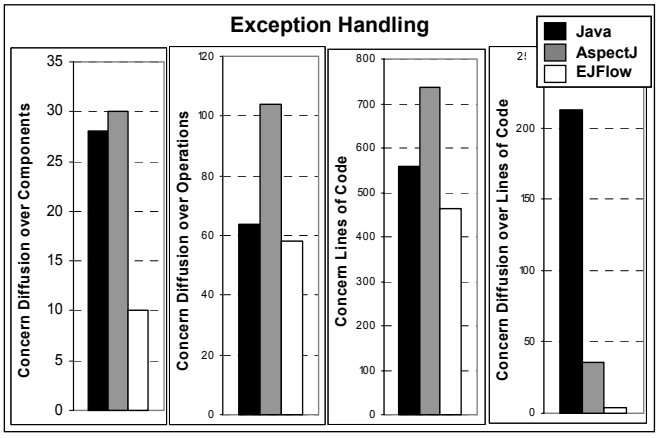

Figure 3: Concern measures for exception handling

This section presents an analysis of separation of concerns focusing on two exception-related concerns, namely exception handling [8] and exception interface (Section 2.2). Charts in this section support an analysis of how the three solutions for the case study affect the concern measures. Figure 3 shows how the exception handling concern was separated in MobileMedia in terms of the 4 concern measures. Hereafter, we refer to specific elements of the exception handling concern as `try-catch`, `try-finally`, and `try-catch-finally` blocks in the code of MobileMedia.

As can be observed in Figure 3, Java and AspectJ experience similarly high degrees of scattering and tangling in all the metrics for the exception handling concern. In particular, the AspectJ version presents the highest value for Concern Diffusion over Components (CDC) metric. The main reason for this result was due to the impossibility of entirely modularizing exception handling for some scenarios using AspectJ [7, 8].

```
public boolean handleCommand(Command c,
                             Displayable d) {
  String label = c.getLabel();
  if (label.equals("Exit")) {
      . . .
  else if (label.equals("Save Media"){
    try {
       MediaData mediaData = getAlbumData().
                          getMediaInfo(mediaName);
                . . .
     }catch (InvalidMediaDataException e) {
              . . .
          return true;
     }
    . . .
  }
 return false;
}
```

Figure 4:AspectJ limitation to modularize error handling code

Figure 4 depicts an example of such scenario in which a controller implements the Chain of Responsibility pattern [20] to manage the screen functionalities. When the "*Save Media*" option is chosen, the associated controller calls its `handleCommand()` method to see if it can satisfy the request. If so, the controller must return *true*. If not, then `handleCommand()` returns *false* and the control is passed to the next controller in the chain.

Likewise, the handler associated with method `getMediaInfo()` needs to return *true* in order to skip subsequent invocations related to successful saving operations. Thus, the problem here is that to keep the semantics of the original handler, the whole `try-catch` block, including the invocation to `getMediaInfo()`, must be extracted to an *around* advice since the same exception can be raised by many invocations of `getMediaInfo()` in many different places of `handleCommand()` method. This solution is undesirable [7], as it keeps the normal and abnormal code tangled in the extracted advice body.

```
boolean ehandler()
  boundto(InvalidMediaDataChannel()):
    within(BaseController) {
      . . .
      return true;
}
```

Figure 5: Example of *ehandler* advice

On the other hand, EJFlow addresses this problem by inlining the `ehandler` advice code. This preserves the semantic of the original handler and allows increasing the applicability of our technique by extracting more exception handling code from the application's components. Figure 5 describes the handler responsible for handling exceptions flowing through the channel `InvalidMediaDataChannel` originated in the `Model`

component. This explicit exception channel captures the propagation of `InvalidMediaDataException`.

The obtained results for Concern Diffusion over Operations (CDO) and Concern Lines of Code (CLOC) were better for the refactored EJFlow version of the MobileMedia. The reduced number of operations and lines of code containing exception handling code was a direct consequence of the approach used by EJFlow to modularize handlers responsible only for remapping caught exceptions. Such handlers play an important role in the Java implementation since they enforce the exception interface between components [38]. In AspectJ, to modularize these handlers, one moves their code to `after throwing` advice. However, unlike `try-catch` blocks in Java, each handler advice is counted as a new operation which increases the CDO and CLOC values for the AspectJ implementation. In addition, the overhead in excessive use of AspectJ constructs, such as pointcut and declare soft, contributes to the high CLOC value in the AspectJ solution. Also, we believe that this makes the program more convoluted and less understandable.

```
public String[] getMediaNames(String mName)
   throws UnavailableMediaAlbumException,
            AlgorithmicException
{ String[] result = null;
  try {
     result = persistence.
                  loadMediaData(recordName);
  }catch(PersistenceMechanismException e){
      throw new
          UnavailableMediaAlbumException(e);
  }catch (Throwable e) {
      throw new AlgorithmicException(e);
  }
  return result;
}
```

Figure 6: Example of remapping exceptions

EJFlow uses `declare einterface` to enforce the exception interface. Thus, a huge amount of code dedicated only to implement such handlers can be replaced by two lines of interface declaration. This successful aspectization has a positive influence on the general result of any system once remapping handlers represent almost 30% of all handlers in real Java applications [4].

```
1:pointcut rs1(): withincode
     (public void ImageAccessor.addImageData(..));
2:pointcut UnavailableMediaAlbumChannel()
     echannel(PersistenceMechanismException,
                rs1, within(ModelData));
3:declare einterface
     echannel  UnavailableMediaAlbumChannel():
     winthin(ModelData);
4:pointcut AlgorithmicExceptionChannel():
     echannel(Throwable, within(ModelData));
5:declare einterface echannel
     AlgorithmicExceptionChannel():
     winthin(ModelData);
```

Figure 7: Remapping exceptions using EJFlow

Figure 7 shows the use of `declare einterface` to implement a remapping equivalent to the one illustrated in Figure 6. The second line captures exceptions propagated from the lower-level component (*Persistence*) to the *Model* component and defines the `UnavailableMediaAlbumChannel` channel. Line 3 then declares the exception interface for this channel to be the façade of the *Model* component (*ModelData)*. In addition, EJFlow captures all possible general exceptions that could be propagated from *Model* component to the higher-level component and remap them to `AlgorithmicExceptionFlow`.

In order to assess how much effort can be avoided by using EJFlow to manage the exception flow, Figure 8 displays metric results of separation of concerns regarding the exception interface concern. As a consequence of the utilization of explicit exception channels, the measure results presented a significant reduction in comparison with Java and AspectJ implementations. In terms of CDC, explicit exception channels concentrate the exception interface code in fewer components. Hence, adaptations in a channel can be easily achieved by just looking at four components rather than 10 or 14 in the Java and AspectJ implementations, respectively. In addition, EJFlow reduces the amount of code necessary to define the exception interface (CDO and CLOC in Figure 8).

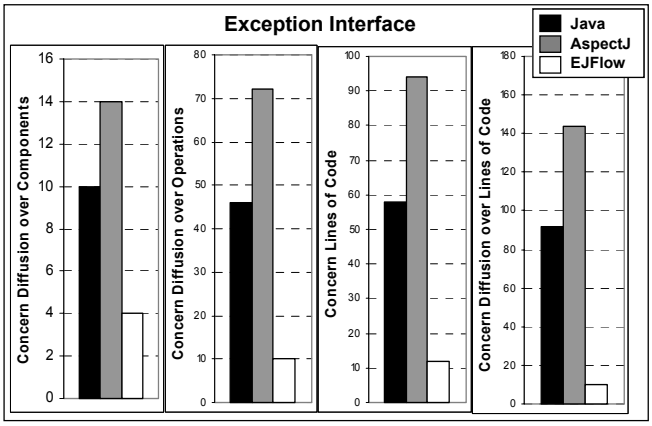

Figure 8: Separation of Concerns metric values for the exception interface concern

5.4 Coupling, Cohesion, and Size Measures

Empirical data presented in Figure 9 supports our claims that a global view of exception flow facilitates code reuse. Exceptions within the same channel are handled similarly in most of the methods. Hence, it becomes easier for the developer to identify which handlers can be reused.

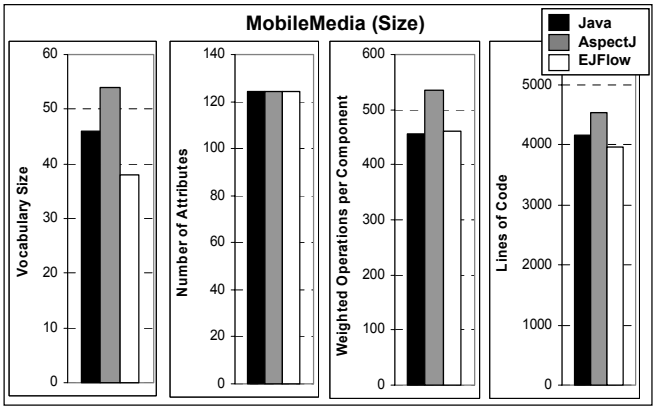

Figure 9: Results for size metrics

In addition, EJFlow allows reducing the number of components by removing exceptions used only for enforcing the exception interface between components. For instance, Figure 7 does not use any exception to enforce the exception interface. Thus, if the exceptions `UnavailableMediaAlbumException` and `AlgorithmicException` are not used in another context, they can be removed from the application. Therefore, the numbers of exception classes (Vocabulary Size metric) and lines of code are reduced.

The coupling between normal behaviour and error handling (CBC in Figure 10) is also reduced in the EJFlow version of MobileMedia. For the most parts, this happens because methods within a channel do not need to explicitly list the exceptions that they signal in their interfaces. Since many of these methods do not do anything with these exceptions (other than implicitly propagating them), the number of classes on which these methods depend will unavoidably decrease.

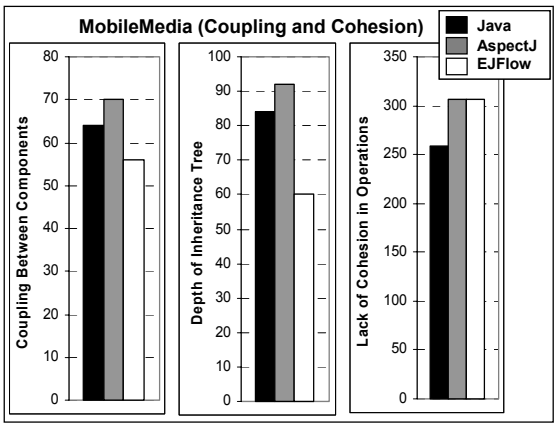

Figure 10: Results for coupling and cohesion metrics

6. RELATED WORK

Exception handling is widely regarded as a crosscutting concern in the literature [28, 39]. In fact, the seminal paper on AOP [26] already suggested that exception handling was one of the most promising candidates for the application of AOP. Hence, it is no accident that several works [7, 8, 16, 28] have employed AOP techniques to modularize exception handling code, with different goals. Moreover, a myriad of exception flow analyses [10, 15, 19, 39] and language constructs [12, 14, 27, 30] have been proposed in the literature with the goal of improving the quality of exception handling. In this section, we present work that, we believe, is directly related to our own. For simplicity, we place related work in four categories: (i) AOP and exception handling; (ii) exception flow analyses; (iii) language constructs for improved exception handling; and (iv) domain-specific aspect-oriented languages.

AOP and Exception Handling. In a seminal study, Lippert and Lopes [28] have employed an old version of AspectJ to refactor exception handling code in a large OO framework, called JWAM, to aspects. The goal of this study was to assess the usefulness of aspects for separating exception handling code from the normal application code. The authors found that, when applied to a reusable infrastructure that implements general (i.e. non-application specific) error handling policies, the use of aspects to modularize exception detection and handling brings several benefits, for example, better reuse, less interference in the program texts, and a decrease in the number of LOC. Castor Filho et al have conducted a complementary empirical study [8] whose goal was to understand the benefits and limitations of using aspects to structure error handling code in real-world applications. This study targeted four different systems, three object-oriented and one aspect-oriented, implementing various application-specific exception handling policies. It consisted of refactoring the error handling code of the four applications to aspects. The study revealed that some of the common knowledge pertaining to the use of aspects to modularize exception handling only applies to simple cases that are not always realistic. For example, the authors found out that reusing non-trivial exception handling code is a very difficult task that depends on several factors. None of the aforementioned studies proposes means to use aspects to help in tracking the flow of exceptions in a program. Moreover, even though both of them pointed out some of the limitations of AspectJ for handling exceptions, they did not propose suitable language extensions to remedy these problems.

Castor Filho, Garcia, and Rubira have elaborated a catalog of best and worst practices [7] pertaining to the aspectization of exception handling. This catalog aims at helping developers to decide when to and when not to extract exception handling code to aspects, based on the influence that this decision has on criteria such as coupling, cohesion, and separation of concerns. This work does not, though, consider the relation between aspects and exception flow; neither provides guidelines to explicitly delimit the interface between normal code and error handling aspects.

Some works have reported on the use of AOP to structure exception handling code without attempting to analyze the benefits and drawbacks of this technique, nor proposing extensions to existing AOP languages. For example, Soares, Laureano, and Borba [41] employed AspectJ to modularize handlers for exceptions introduced by other aspects, most notably distribution and persistence, in an information system from the healthcare domain. Fetzer et al [16] employed aspect-oriented programming to implement the concept of "atomic exception handling", exception handlers that perform a rollback before they throw an exception.

Exception Flow Analyses. Many static analyses of source code that generate information about exception flow have been described in the literature. Usually, this information consists of the exception propagation paths in a program and is used, for example, to identify uncaught exceptions in languages with polymorphic types, such as ML. Chang et al [10] present a set-based static analysis of Java programs that estimates their exception flows. This analysis is used to detect too general or unnecessary exception specifications and handlers. Fähndrich [15] and colleagues have employed their BANE toolkit to discover uncaught exceptions in ML. Schaefer and Bundy's [40] work describes a model for reasoning about exception flow in Ada programs. They also present a tool that uses this model to track down uncaught exceptions and provide exception flow information to programmers. The popular JEX tool, devised by Robillard and Murphy [39], analyzes exception flow in Java programs. It includes a GUI to display a program's exception propagation paths and detects handlers that are too general. Fu and Ryder [19] propose an extension to typical static analyses for

Java. Instead of trying to identify exception propagation paths, their analysis targets what they call "exception propagation chains", i.e. they attempt to discover causal relations between exception propagation paths in order to get a picture of the whole system, in terms of exception flow.

Our proposal builds upon many of these works but, to the best of our knowledge, is the first one to combine exception flow analysis and aspect-oriented programming. Also, it attempts to integrate the exception flow analysis with a language construct. In this manner, developers can indicate, during system implementation, specific parts of the program whose exception flow should be analyzed at compile time.

Language Constructs for Improved Exception Handling. There are several papers that propose extensions to existing programming languages in order to improve the quality of exception handling code and, as a consequence, the quality of the normal code. One the one hand, the Lore language [23] allows the definition of different levels of granularity for EHCs, based on static program elements such as classes, method declarations, and exception classes. On the other hand, Cui and Gannon [12] argue that, in object-oriented languages, it makes sense to associate exception handlers to runtime objects. Our approach generalizes these approaches by using the pointcut language of AspectJ, plus the extensions we propose, to select both fine- and coarse-grained EHCs to which exception handling advice can be associated.

A complementary idea is to extend existing languages with constructs that act as implementation-level specifications of error handling behavior [14, 30]. For example, anchored exception declarations [14] solve the maintenance problems posed by Java's checked exceptions that we discussed in Section 2.2. In this approach, it is possible to specify that a method signals the same exceptions as another method, usually one it invokes. In this manner, in a chain of method declarations, if the exception interface of the bottommost method changes, other methods in the chain do not necessarily have to be modified. ExnJava [30] is not really a language extension (though it implements a very lightweight modification to the semantics of Java's `throws` clause). Instead, it is a tool comprising several features, such as a module system and exception-specific refactorings, that aim to support the specification of exceptions. The design of EJFlow builds upon these proposals. However, to the best of our knowledge, it is the first one to include a construct for specifying non-local issues pertaining to exception flow in a programming language.

Domain-Specific AOP Languages. In its original incarnation [26], AOP was mainly based on domain-specific languages. For example, the seminal paper on AOP presents aspect-oriented languages for domains such as distribution and scientific computing. This trend was followed by many works in the literature, most notably in the area of distributed computing. Both AWED [33] and DJcutter [34] make it possible to select join points of interest that, besides being scattered throughout the application code, are also scattered across different nodes communicating through a network. Some domain-specific approaches to AOP focus specifically on issues pertaining to error handling. For instance, one of the first language extensions implemented using the abc[1] extensible compiler was a pointcut designator capable of selecting occurrences of the `throw` statement in Java programs. In our view, none of these domain-specific AOP approaches tackles the issues that we address in this paper.

7. CONCLUDING REMARKS

The novel exception handling mechanism we presented in this paper leverages AOP techniques to promote improved separation between normal and error handling code, while keeping track of exception control flows. In this manner, we also intend to minimize the impact that some typical problems that stem from the use of traditional exception handling mechanisms [6, 37, 39] have on the overall quality of software systems. We claim that the use of the proposed model brings three main advantages: (i) it makes exception flow explicit and understandable in a localized way, without the need to examine other parts of the program; (ii) it enhances program modularization by improving the error handling code reuse; and (iii) it promotes better maintainability of normal and error handling code by separating the handlers and eliminating annoying exception interface declarations. We have implemented the most part of EJFlow, with small syntactic additions to AspectJ. Our ongoing work encompasses the empirical evaluation of the error proneness on the use of EJFlow when compared to conventional proposals discussed in this paper. Furthermore, we intend to overcome a limitation of our evaluation by assessing the scalability of EJFlow in the presence of software maintenance and evolution scenarios.

Acknowledgements. This work is supported in part by the European Commission grant IST-33710 - Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE), grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), grant 479395/2004-7: Brazilian Council for Scientific and Technological Development (CNPq). Eduardo is supported by CAPES-Brazil. Part of this work was conducted while Fernando was supported by FAPESP – Brazil, grant 06/04976-9

8. REFERENCES

[1] Avgustinov, P. et al. abc : An Extensible AspectJ Compiler. *Trans. AOSD I*, pages 293-334, 2006.

[2] Buhr, P. A. and Krischer, R. Bound Exceptions in Object Programming. In *Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object Oriented Systems*, pp. 20-26, Darmstadt, Germany, July 2003.

[3] Buschmann, F. et al Pattern-Oriented Software Architecture: a System of Patterns. John Wiley & Sons, Inc. 1996.

[4] Cabral, B and Paulo, M. Exception Handling: A Field Study in Java and .NET. *Proc. of ECOOP*, Berlin, Germany, 2007.

[5] Cacho, N. et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. *Proc. of AOSD.06*, Bonn, Germany, 2006.

[6] Cargill, T. Exception Handling: A False Sense of Security. *C++ Report*, vol. 6, no. 9,pp. 21-24, Nov.-Dec. 1994.

[7] Castor Filho, F., Garcia, A., Rubira, C. Extracting Error Handling to Aspects: A Cookbook. *In Proc. of the 23rd ICSM*, Paris, France, 2007.

[8] Castor Filho, F. et al. Exceptions and Aspects: The Devil is in the Details. *Proc. of Int'l Symp. on Foundations of Software Engineering (FSE)*, 2006.

[9] Chang, B., Jo, J. and Her, S. H. Visualization of exception propagation for Java using static analysis. *Proc. 2nd IEEE Workshop on Source Code Analysis and Manipulation*, pages 173-182, 2002.

[10] Chang, B.-M. et al. Interprocedural exception analysis for java. *In Proceedings of 16th ACM SAC*, pp. 620-625, 2001.

[11] Chidamber, S., and Kemerer, C. A Metrics Suite for Object Oriented Design. *IEEE Trans. on Software Engineering*, 1994, 476-493.

[12] Cui, Q. and Gannon, J. D. Data-Oriented Exception Handling. *IEEE Transactions on Software Engineering* 18(5), pages 393-401, 1992.

[13] Cusumano, M. A.. 'Systematic' vs. 'Accidental' Reuse in Japanese Software Factories. MIT Sloan School of Management Working Paper #3328-BPS-91, September 1991.

[14] Dooren, M. and Steegmans, E. Combining the Robustness of Checked Exceptions with the Flexibility of Unchecked Exceptions Using Anchored Exception Declarations. *In Proceedings of OOPSLA'2005*, pages 455-471, USA, 2005.

[15] Fahndrich, M. et al. Tracking down exceptions in standard ML. Technical Report CSD-98-996, University of California, Berkeley, 1998.

[16] Fetzer, C., Högstedt, K., and Felber,P. Automatic detection and masking of nonatomic exception handling. *IEEE Trans. Sw. Eng.*, 30(8):547–560, 2004.

[17] Figueiredo, E., Garcia, A. and Lucena, C. *AJATO: An AspectJ Assessment Tool*. *Proc. of ECOOP'06* (demo), 2006.

[18] Figueiredo, E. et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. To appear in *Proceedings of ICSE'08*, Leipzig, Germany, 2008.

[19] Fu, C. and Ryder, B. G. Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. *In Proceedings of ICSE'07*, pages 230-239, Minneapolis, USA, 2007.

[20] Gamma, E. et al. Design Patterns: Elements of Reusable Software Systems. Addison-Wesley, 1995.

[21] Garcia, A, Beder, D. and Rubira, C. An exception handling software architecture for developing fault-tolerant software, *Proceedings of the 5th IEEE High Assurance Systems Engineering Symposium*, USA, 2000, 311--320.

[22] Garcia, A. et al. Modularizing Design Patterns with Aspects: A Quantitative Study. *Transactions on AOSD*, 1, 2006, 36-74.

[23] Garcia, A. et al. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. *Journal of Systems and Software*, 59(2):197-222, November 2001.

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

[25] Greenwood, P. et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. *Proc. of ECOOP.07*, Berlin, Germany, 2007.

[26] Kiczales, G. et al. Aspect-oriented programming. In *Proceedings of ECOOP'97*, pages 220–242, 1997.

[27] Lacourte, S. Exceptions in guide, an object-oriented language for distributed applications. *In Proceedings of the 5th ECOOP'91*, LNCS 512, pages 268–287, 1991.

[28] Lippert, M and Lopes, C. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. *In Proc. of the 22nd ICSE*, pages 418-427, Limmerick, Ireland, 2000.

[29] Litke, J. D. A systematic approach for implementing fault tolerant software designs in Ada. In *Proceedings of the Conference on TRI-ADA '90* (Baltimore, Maryland, United States, December 03 - 06, 1990).

[30] Malayeri, D. and Aldrich, J. Practical Exception Specifications. *Advanced Topics in Exception Handling Techniques*, LNCS, 2006.

[31] Miller, M. and Tripathi, A. Issues with Exception Handling in Object-Oriented Systems. In *Proceedings of ECOOP*, pp.85-103, LNCS-1241, Finland, 1997.

[32] Molesini, A., Garcia, A., Chavez, C., Batista, T. On the Quantitative Analysis of Architecture Stability in Aspectual Decompositions. *Proc. 7th IEEE Conference on Software Architecture (WICSA)* 2008, Vancouver, BC, Canada.

[33] Navarro, L. D. et al. Explicitly Distributed AOP using AWED. *In Proc. of AOSD'2006*, Bonn, Germany, 2006

[34] Nishizawa, M., Chiba, S., and Tatsubori, M. Remote pointcut: a language construct for distributed AOP. In *Proceedings of the AOSD'2004*, pages 7-15, Lancaster, UK, 2004.

[35] Parnas, D. L. and Würges, H. Response to Undesired Events in Software Systems. *In Proceedings of the 2nd International Conference on Software Engineering*, pages 437-446, San Francisco, USA, 1976.

[36] Randell, B. and Xu, J. The Evolution of the Recovery Block Concept. In *Software Fault Tolerance* (M. Lyu, Ed.), Trends in Software, pages1-22, J. Wiley, 1994.

[37] Reimer, D., Srinivasan, H. Analyzing exception usage in large Java applications. In: *Proceedings of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems*.

[38] Robillard, M. P. and Murphy, G. C. Designing robust Java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT international Symposium on Foundations of Software Engineering: Twenty-First Century Applications.* ACM Press, New York, 2000, 2-10.

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

[40] Schaefer, C. F. and Bundy, G. N. Static analysis of exception handling in Ada. *Software: Practice and Experience*, 23(10):1157–1174, October 1993.

[41] Soares, S., Laureano, E. and Borba, P. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA'02*, pages 174–190, 2002.

[42] Vall´ee-Rai, R. et al. Optimizing Java bytecode using the Soot framework: Is it feasible? In David A. Watt, editor, *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.