Unveiling and Taming Liabilities of Aspect Libraries Reuse

Roberta Coelho[1, 2] Uirá Kulesza[2] Awais Rashid[3] Arndt von Staa[1] Carlos Lucena[1]

[1] Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Brazil

[2] Informatics and Applied Mathematics Department (DIMAp), Federal University of Rio Grande do Norte, Brazil

[3]Computing Department, Lancaster University, Lancaster, UK and Ecole des Mines de Nantes, France

{roberta,arndt,lucena}@inf.puc-rio.br
uira@dimap.ufrn.br awais@comp.lancs.ac.uk

Abstract. *Aspect Libraries have introduced new possibilities of application composition, facilitating the reuse of crosscutting functionalities. However, if on the one hand this facilitates the independent development of reusable aspects, on the other hand it can bring additional complexity to software reuse, especially, in exception-aware systems. In this work, we present a set of potential fault scenarios associated with aspect libraries reuse. We show that the reuse of aspect libraries is indeed fault prone in exceptional conditions and present a principled reuse approach supported by a static analysis tool that leads to significant improvements.*

1. Introduction

Although prebuilt aspect libraries are a relatively new reuse artifact, several useful collections have already become available, including: the Spring AOP aspect library [33], the Glassbox Inspector [15], the JBoss Cache [19], and GOF patterns aspect library [18]. Such libraries typically involve collaboration among aspects, classes, and interfaces, and enable the developer to extend the functionalities of existing applications avoiding significant coupling and large re-investments [3]. They implement crosscutting functionalities (e.g., performance monitoring, security, and transaction management) that would be spread in many application modules otherwise. According to Bodkin [3] "*reuse will prove to be the most important benefit from adopting Aspect Oriented Programming*."

While aspect libraries introduce new possibilities for application composition, in some circumstances they may threaten the client application's robustness and design consistency. In this paper we focus on the robustness and consistency issues pertaining to exceptional flow of programs using such aspect libraries; the additional behavior injected by aspect libraries may also bring new exceptions. Exceptions are abnormal computation states that arise as a consequence of, for instance, faults in the application itself (e.g., access of null references), a noisy user input or faults in underlying middleware or hardware. Aspect libraries developers may do their best to ensure that library functions do not create faults that impact client applications. However, unexpected behavior in

aspect library code (e.g., unanticipated null values, undocumented runtime exceptions thrown by libraries) is often present [9]. In order to know which exceptions may flow from aspect libraries, programmers must rely on library documentation – which, very often, is neither complete nor precise [6, 29, 34]. As a consequence, the developer may only notice the existence of unexpected exceptions thrown by an aspect library, by observing the failures caused by them on the application (e.g., uncaught exceptions [26]: exceptions that are not caught on the application code and lead to a software crash; or unintended handler actions [26, 28]: exceptions that are mistakenly handled by an existing handler in the base code).

One may argue that the problems associated with aspect libraries reuse also occur when OO libraries are reused: we do not know the exceptions that may flow from OO libraries as well and, consequently, cannot prepare the code to deal with them. Indeed, this is a real problem in OO libraries reuse, and some static analysis tools, e.g., as proposed in [14, 28], could be used to deal with it. However, some characteristics of aspect compositions strengthens this problem, such as: (i) the *invasive modification* of the base code [1], (ii) some developers and approaches advocating an *oblivious* development process [12], (iii) the *load-time weaving* [11] available in some Aspect Oriented (AO) languages, (iv) and the *quantification property* [12] (as detailed in Section 2.3).

While the invasiveness of aspect library compositions often allows the reuse of crosscutting implementations, they might render less useful if they introduce new exceptions that may lead to potential faults. Currently, there is no approach or supporting tool to help application developers to: (i) discover which exceptions may flow from an aspect library, (ii) prepare the code to deal with them; or (iii) check whether such exceptions were adequately handled. Such approach would reduce or altogether avoid the threats posed by aspect libraries to the application's robustness in exceptional scenarios.

In a previous study [9] we assessed the error proneness of AOP mechanisms on exception flows of programs. In the present work we focus on the exceptions that may flow from *library aspects*, the consequences that they may bear, and how to deal with them. Thus, the main questions we seek to address are the following:

- What are the potential consequences of reusing aspect libraries that may throw exceptions?
- How can one reduce the number of potential faults associated with them?

We believe the answers to these questions are of interest to a broad audience, due to the increasing number of AO developers. The contributions of this work are as follows:

- We explore the potential faults associated with aspect libraries reuse – showing a concrete example.
- We propose an approach for principled aspect reuse that takes into account exceptional situations neglected by existing approaches. The approach is based on an exception analysis tool called SAFE [8] developed to support this approach. This tool aims at finding the exceptions that may flow from aspect

libraries (*exception interfaces* [26]) and how such exceptions flow on the base code (exception paths).

The remainder of this paper is organized as follows. Section 2 presents some background on exception handling, and discusses the characteristics of aspects compositions and the challenges they can bring when reusing aspects in the presence of exceptions. Section 3 presents the reuse approach and how the static analysis tool can be used to support some steps in the approach. Section 4 summarizes our experience when applying this approach to reuse two different aspect libraries. Section 5 provides further discussion of lessons learned. Finally, Section 6 presents our conclusions and directions for future work. Due to space limitation, throughout this article we assume that the reader is familiar with AOSD terminology (e.g., aspect, join point, pointcut, and advice) and AspectJ language constructs [11, 20].

2. Background

2.1 Exception Handling

Exception handling [16] is a technique for structuring the error recovery code of a system. It promotes (i) the explicit separation between normal and abnormal code; and (ii) the explicit declaration of modules' *exception interfaces*. In modern languages such as Java and AspectJ the error recovery measures are encapsulated into handlers (`try-catch` blocks), and exceptions are represented as objects that are raised when an exceptional condition is detected. Raising an exception results in the interruption of the normal activity of the program; followed by the search for an appropriate *exception handler* to deal with the signaled exception. After the execution of an exception handler, control returns to the code that immediately follows the handler code.

The list of exceptions that may flow from a method and is associated with its signature is the method's *exception interface* [26]. The exception interface should provide complete and precise information to the method user. The user can, therefore, prepare the code for the exceptions that can flow from it. However, some languages, such as Java and AspectJ, allow the developer to bypass this mechanism. In such languages exceptions can be of two kinds: *checked* exception – that needs to be declared on the method's signature that throws it – and *unchecked exception* – that does not need to be declared on the signaler method's signature. As a consequence, the client of a method cannot know which *unchecked* exceptions may be thrown by it, unless s/he recursively inspects each method called from it. For convenience, in this paper we split this concept of *exception interface* into two categories:

- the *explicit exception interface* that is part of the module (method or method like construct) signature and explicitly declares the exceptions; and
- the *complete (de facto) exception interface* which captures all the exceptions signaled by a module, including the implicit ones not specified in the module signature. For the rest of the paper, unless it is explicitly mentioned otherwise, exception interface refers to the complete (de facto) exception interface.

In AspectJ programs the *explicit exception interfaces* of advices (i.e. method like constructs that encapsulates the crosscutting behavior of an aspect) must be based on the

explicit exception interfaces of the advised methods. They should follow a rule similar to the "*Exception Conformance Rule*" [26] applied during inheritance, when methods are overridden. As a result an advice can only throw a checked exception if it is thrown by "every" advised method. To overcome this limitation, most of the advices throw *unchecked* exceptions (e.g., `SoftExceptions` in AspectJ) which do not need to be specified by every advised method. In other AO languages such as Spring AOP [33] and JBoss [19] aspect advices are represented as regular Java methods which can throw any exception (*checked* or *unchecked*).

2.2 Exception Occurrences in AO Libraries

The additional expense that is required to account for the effects of exception occurrences on aspect libraries may not be justified unless exceptions occur frequently in practice. A recent study [6] shows that the amount of code dedicated to exception occurrences (LOC EH) - code dedicate to exception raising and handling - in OO libraries is approximately 7% of the total number of LOC. We conducted a similar study to determine the frequency with which aspect libraries use exception-related constructs. In this study, we examined the assets of aspect libraries from different sources, and obtained the information summarized in Table 1. In the observed subjects from 2,09% to 8,82% of the total lines of code were dedicated to exception raising and handling concerns. We can observe that exceptions also occur frequently in aspect libraries.

Aspect Libraries	LOC	LOC EH	% LOC EH	# try	# catch	# throw
GlassBox Inspector (monitor)	3621	90	2,49%	16	16	14
Spring AOP	98976	8731	8,82%	975	1105	1847
JBoss Cache	41582	870	2,09%	363	363	494

Table 1. Characterisics of EH code on aspect libraries.

Furthermore, the addition of exception-related constructs in several main stream programming languages (e.g., Java, C++, C#) attests the importance of exception handling mechanism in the development of current systems [13,7, 30].

2.3 Characteristics of AO Compositions and their Consequences to Aspect Reuse

As mentioned before some characteristics of aspect compositions bring new challenges to aspect-oriented development and reuse in the presence of exceptions. Firstly, aspects perform *invasive modification* of programs [1] which allow a developer to externally modify the behavior of a method. This way of composition works *by reverse*, also known as the *inversion of control*: the aspect declares which classes it should affect rather than vice-versa. This means that adding and removing aspects from a system does not require editing the affected class definitions.

Consequently, when (re-)using aspects we cannot easily protect the advised code from the exceptions that may flow from them. Figure 1 illustrates a `before` advice that intercepts an application method and throws an exception. This exception will flow through the base code and interrupt the normal control flow of the advised code. On the other hand, in OO system development we can simply add a `try-catch` block surrounding the reused piece of code to avoid exceptions from flowing from it and affecting the normal application control flow.

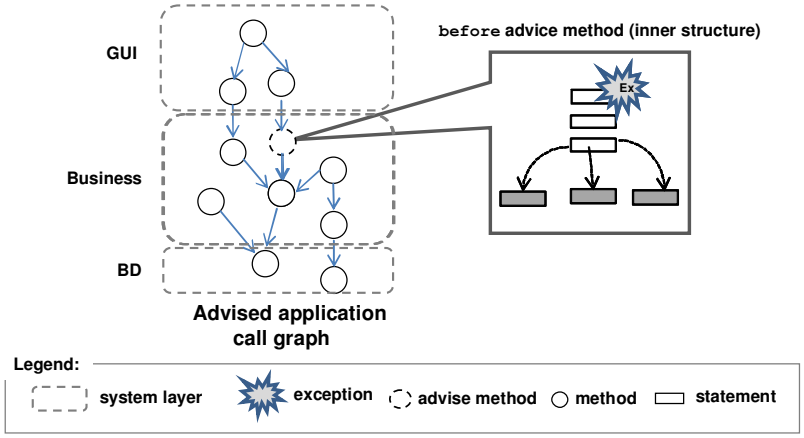

Figure 1. Consequences of invasiveness modifications.

Moreover, some AO development approaches rely on the *obliviousness* property [12]. According to it the developer of the base code does not need to know that the code will be affected by aspects. As a consequence, the application developer does not prepare the code to deal with exceptions that may escape from aspects. Third, some AO languages enable *load-time weaving*. The class loader reads a configuration file that specifies the aspects to be woven when applications are loaded. Thus, the developer only needs to deploy the aspect *bytecode* together with the application to be advised. This is the scenario that occurs most often when *aspect libraries* are reused. Not having access to the source code of imported aspects also has its drawbacks: the impact of aspects on the exceptional flow of applications is only discovered at runtime.

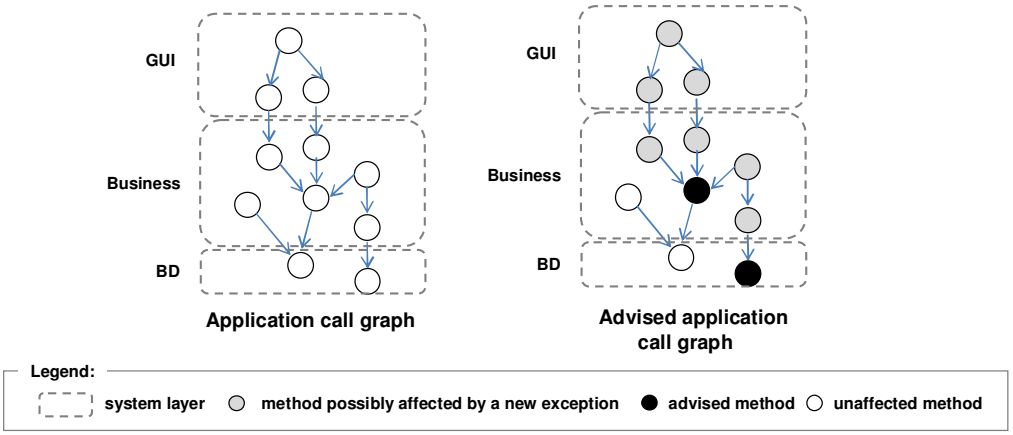

Figure 2. Consequences of quantification property in the presence of exceptions.

Last but not least, AO development is often based on the *quantification* property which refers to the desire of programmers to write programming statements with the following form: "*In programs* P, *whenever condition* C *arises*, *perform action* A". As a result aspects have the ability to affect *semantically unrelated* points in the code. Therefore, when a new exception is introduced by an aspect within an application, new handlers should be defined in different places within the base code (one for each path of the call graph that may reach the affected method – see Figure 2).

Since the exception handling policy[1] [27] of an application is almost always based on its architecture, adding exceptions on unrelated points in the code may potentially break the existing exception handling policy. The combination of these characteristics, therefore, results in fault-prone exception handling scenarios in AO systems.

3. The Proposed Approach

In this section, we present an approach and a supporting tool to help developers when developing an exception-aware aspect-oriented application. Section 3.1 briefly describes the SAFE tool (*Static Analysis for the Flow of Exceptions*) developed in this work to statically discover the *exception paths* [8, 9] (i.e., the path in a program call graph that links the signaler and the handler of an exception) of each exception thrown within an AO program. Section 3.2 presents each step of the proposed approach which covers both the development of application aspects and the reuse of aspect libraries in the presence of exceptions – the steps that should be performed to preserve the system exception handling policy after AO weaving.

3.1. The SAFE Tool

The discovery of exceptions that may flow from aspect libraries can be very complex if not infeasible to do manually, especially because of the use of `Runtime` exceptions in Java and AspectJ. The exceptions that escape from an aspect advice may come from different sources: (i) they can be explicitly thrown by a throw statement; (ii) they can also be implicitly thrown from specific operations such as division by zero (`ArithmeticException`), and when the developer accesses a null reference (`NullPointerException`); (iii) there are also exceptions that can be thrown by JVM environment when an abnormal situation occurs (`OutOfMemoryException`); (iv) the aspect-oriented frameworks (AspectJ, JBossAOP, SpringAOP) can also throw specific exceptions, that come from the additional code included by their respective weavers; and finally (v) calls to APIs/library methods may also implicitly throw exceptions.

The SAFE tool [8] statically analyses the woven bytecode in order to discover: (i) the *exception interfaces* of each advice; and (ii) the *exception paths* of each exception that escapes from an aspect library. To mine such information, the SAFE tool performs an *exception-flow analysis* similar to the one presented in [13] for OO systems[2]. The *exception-flow analysis* is a dataflow analysis, resembling the analysis based on *def-use* pairs [38] but instead of running on the control flow graphs, it runs on the program call graph. The graph used by the exception-flow analysis tool is an extension of the program call graph (PCG) with additional information in each node (program method). Such information comprises the statements where exceptions may be thrown and handled (by an enclosing `try-catch` block) [8].

---

[1] The exception handling policy comprises a set of design rules which define the system elements responsible for signaling, handling and re-throwing the exceptions; and the system dependability relies on obedience to such rules.

[2] Current exception flow analysis tools [13, 25] do not support AOP constructs. Even the tools which operate on Java bytecode level [13] cannot be used in a straightforward fashion. They cannot interpret the effects on bytecode after the weaving process of AspectJ.

After constructing the extended PCG the *exception-flow analysis* algorithm identifies every statement that may throw an exception, and for each of them, it traverses the program call graph backwards looking for the handlers defined for it. If no handler is found, the algorithm reaches the program entrance point and the exception is classified as an *uncaught exception*. During the propagation process, each method (a node in the call graph) in which the exception propagates is recorded as part of the *exception path* - it is one of the exceptions that compose the *exception interface* of the method. Our tool is based on Soot framework [32] for static analysis of *bytecode*. It uses Spark, one of the call graph builders provided by Soot. Spark is a field-sensitive, flow-insensitive and context-insensitive points-to analysis [32], used by other static analysis tools [13]. The SAFE tool considers all checked and unchecked exceptions, explicitly thrown by the application or implicitly thrown (e.g. via library method) by aspects in the library aspect.

3.2. Steps of the Proposed Approach

When implementing *application aspects* or re-using *library aspects* (i.e., aspects developed by third party developers), the developer should account for the exceptional conditions that may arise from them. Otherwise, exceptions may cross aspects boundaries and impair the system's integrity and robustness due to u*ncaught exceptions* and *unintended handler actions*. Much of the effort in our approach involves working out how to effectively integrate aspects into the base code without the risk of introducing the faults in the exception handling code – which may represent causes of potential system crashes.

The verification approach presented here complements the available AO testing approaches that focus on the system's normal control flow [22, 36, 21, 2, 37, 23]. These approaches perform unit and integration tests and look for faults in the AO code (e.g., too general or too specific pointcuts, conflicting aspect interactions). These approaches neglect the exception handling code, firstly due to the difficulty of simulating exception occurrences during tests and, secondly, because the large number of possible exceptions can lead to a *test case explosion problem.*

The approach based on the SAFE tool, that we have developed, comprises of the steps described below, which are also illustrated in Figure 3.

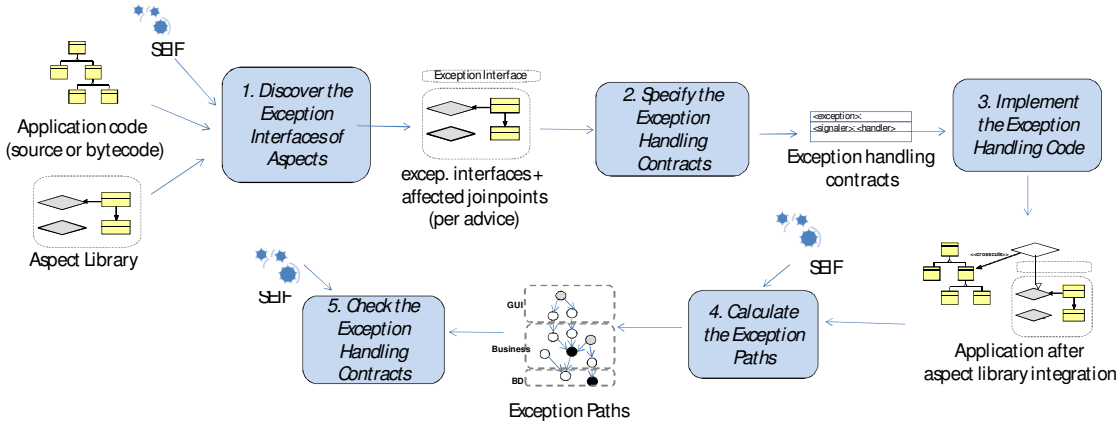

Figure 3. The proposed approach.

1. *Discover the Exception Interfaces of Aspects.* In this step the SAFE tool is used to discover the *exception interfaces* of each aspect advice defined on application aspects or library aspects that crosscut the base code - as detailed in Section 2.1, the *exception interface* of a method-like construct such as an advice comprises the list of exceptions that can be thrown from it. Since the SAFE tool works on the *woven* bytecode, at this step the aspect library needs to be combined (i.e., woven) with the base code. In case the elements to be affected by the aspect were not already implemented, a set of stubs may be defined and combined with the aspect to enable the analysis[3].

2. *Specify the Exception Handling Contracts.* Once the exception interfaces of aspect advices have been discovered, the developer should specify which elements are responsible for handling them. In our approach, this information is represented as a set of *exception handling contracts*. The *exception handling contracts* specify the *Signaler-Handler* relation per exception thrown by aspects and are specified by the SAFE tool in XML format as illustrated below:

```
<exception type="">
   <signaler signature="" />
   <handler signature="" type=""/>
</exception>
```

In case the application has already defined an exception policy, the *exception handling contracts* defined for aspect-signaled exceptions should be defined in accordance to it. The advantage of representing the *exception handling contracts* in a semi-structured way is that they can be used afterwards to partially automate the contract checking step.

3. Implement the Exception Handling Code. In this step, the developer should implement exception handling solutions according to the specified contracts. Examples of exception handling solutions to be implemented in this step are:

- *Error isolation*: This strategy avoids the exception signaled by an aspect from flowing to the client application. The handler can be defined in an exception handling aspect (i.e., an aspect defined to handle exceptions [7] that directly intercepts library aspects or application aspects). Or in the case the developer has access to the aspect source code, handlers can be defined within every advice that signals the exception.
- *App-specific error handling*: the developer can also define catch clauses inside application elements that will be responsible for handling the exceptions thrown by aspects (handling on the base code), or define an exception handling aspect that intercepts specific join points in the application code (handling on aspects).

4. Calculate the Exception Paths. In this step, the SAFE tool analyses the *bytecode* of the advised application, and calculates the *exception path* of each exception that may be thrown by aspects. Notice that if there is no handler for a specific exception, the *exception path* starts from the signaler and finishes at the program entrance point. After the exception paths are calculated, if the exception handling contracts were defined on

[3] Some approaches as the one proposed by Xie et al [36] automatically generate a set of stubs to be intercepted by aspects.

the structured way, the SAFE tool analyzes every exception path in order to discover whether the *handling contracts*, defined in Step 2, were obeyed by them.

5. Manually Inspect the Exception Handling Code. In this step the developer should inspect the exception handling code related to the broken exception handling contracts. By doing so, the developer will diagnose the cause of errors on the exception handling code. Moreover, the developer may gain a fine-grained view of how exceptions are handled (e.g., logging, presenting an error message to the user, or swallowing). After discovering the cause of the exception handling error the implementation steps 3, 4, and 5 should be repeated until every exception is adequately handled on the advised application.

In the following section, we show how our approach can be used to assure the quality of the exception handling code in real implementation scenarios involving *application* and *library* aspects.

5. Worked Example

The Health Watcher (HW) [31, 17] system is a web application that allows citizens to register complaints regarding issues in health care institutions. HW adopts the Layer architectural pattern which enables the separation of the persistence, business, and graphical user interface concerns. Several design patterns were also used in HW to refine each layer, such as: the Facade and the Persistent Data Collections (PDC) patterns [24]. To illustrate our approach, we selected two real change scenarios to be applied to the HW system. In this case study, we need: (i) to *monitor* the performance of http requests; and (ii) to add the *transaction management* support to the persistence operations.

A common strategy for *monitoring* an application is to include instrumentation code around system operations. However, this approach requires scattering duplicate code in many places in the code, which can be tedious, error-prone, and quite difficult to maintain. In our case study, we reused the Glassbox Inspector [4, 15], an *aspect monitoring library*, in order to implement the monitoring concern in the HW system. In our case study, the transaction management concern was also implemented using an aspect library. This concern also represents a typical crosscutting concern. To implement it, we reused the transaction management *aspect library* built on top of Hibernate [11] – an open source object relational mapping tool for a Java. As we can see, both aspect libraries used in this case study were developed by third party developers.

Figure 4 depicts some of the elements that compose the HW system after composing it with the aspect libraries described before (i.e., the *Monitoring Aspect Library,* and the *Hibernate Aspect Library*). The HW elements presented in Figure 4 will be detailed on the next subsections which describe the approach's steps. Each step aims at assuring that the aspects integrated with the base code will not threaten the application robustness in the presence of exceptions.

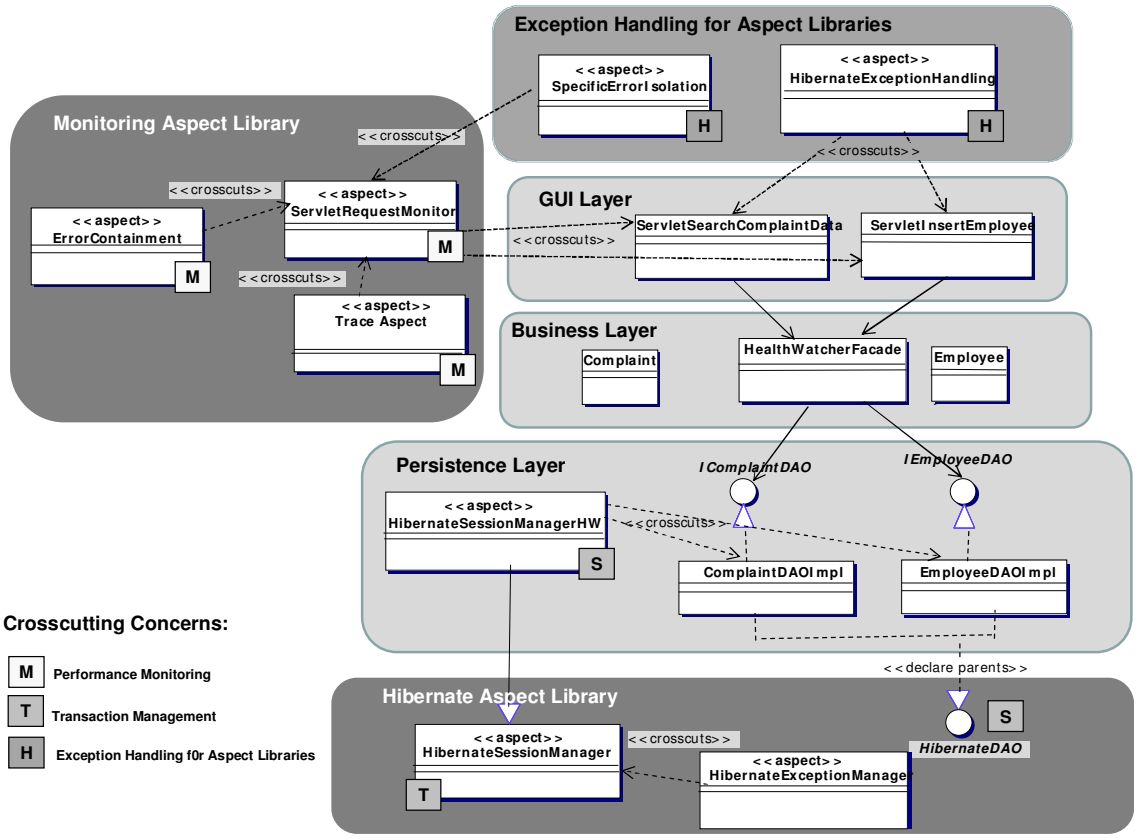

Figure 4. The Health Watcher system composed with Aspect Libraries.

5.1 Discover the Exception Interfaces of Aspects

The first step of our approach is to discover the *exception interfaces* of every aspect advice that affect the base code in each change scenario. In our case study, the *aspects* that intercept the base code are: (i) the `ServletRequestMonitor` – a *library aspect* that directly intercepts every application request operation (see Figure 4); and (ii) `HibernateSessionManagerHW` – an *application aspect* that extends the `HibernateSessionManager` abstract *library aspect* in order to specify the join points in the base code that needs to be intercepted to demarcate a system transaction (see Figure 4).

The SAFE tool recursively analyzes every aspect advice (analyzing every method called from them and every advice that may intercept them) and calculates their *exception interfaces*[4]. Listings 1 and 2 illustrate, respectively, the partial code of the `ServletRequestMonitor` and `HibernateSessionManagerHW` aspects and the *exception interfaces* of some advices defined on them.

[4] The SAFE tool runs on the woven bytecode. When building the woven bytecode, AspectJ converts every aspect into a standard Java class (called aspect class), and each piece of advice into a public non-static method in the aspect class whose signature is automatically generated [39]. (as illustrated by the Advice Method Signature in Listing 2).

```
public aspect ServletRequestMonitor {

//Intercepts every servlet request operation
public pointcut servletRequestExec():
   within(HttpServlet+) &&
   (execution(* HttpServlet.do*(..)) ||
   execution(*HttpServlet.service(..)))…;

   after() returning:monitorEndAllCases()
   {        …
     Response response =
       responseFactory.getLastResponse();
     if (response != null) {
       response.complete();
     } else {
       logError("Monitoring problem:
            mismatched monitor calls");
     }
  }
…

}
```

(a)

```
public aspect HibernateSessionManagerHW
            extends HibernateSessionManager {

 //Intercepts every DAO operation
 Object around() : operationDAO() {
   …
   Transaction tx = null;
   try {
     tx = session.beginTransaction();
     ret = proceed();
     tx.commit();
   } catch(HibernateException ex) {
      if (tx != null) tx.rollback();
      throw getHibernateExceptionManager()
             .createDomainException(ex);
    } finally {
       session.close();
    }
    return ret;
  }
 …
}
```

(c)

Aspect: ServletRequestMonitor

Advice Type: after returning

Sequence: 01

Advice Method signature:

glassbox.monitor.ui.ServletRequestMonitor: void ajc$afterreturning$glassbox_monitor_ui_ServletRequestMonitor1a7406ba1(…)

Exception Interface:

org.aspectj.lang.SoftException

(b)

Aspect: HibernateSessionManagerHW

Advice Type: around

Sequence: 01

Advice Method signature:

hibernate.HibernateSessionManagerHW: Object ajc$around$hibernate_HibernateSessionManagerHW1a506ba1(…)

Exception Interface:

org.aspectj.lang.SoftException

(d)

Listing 1. The code snippets of library aspects and their *exception interfaces*.

Each exception that composes the *exception interface* of each advice presented in Listing 1, originates from *internal library aspects* (`TraceAspect` and `HibernateExceptionManager` aspects in Figure 4) that intercept the advices under analysis. From the listings above, we can observe that trying to manually discover the exception interfaces can easily become an infeasible or error-prone task.

5.2 Specify the Exception Handling Contracts

After discovering the *exception interfaces* of every aspect advice or intertype declaration that affect the base code, we need to define the elements that should be responsible for handling them. In our approach, such information is represented in terms of a set of *exception handling contracts*. Listing 2 illustrates the *exception handling contracts* defined (i) to the instance of `SoftException` thrown by the `after returning` advice presented above, and (ii) to one advice of the `HibernateSessionManagerHW aspect` that may also throw an instance of `SoftException`. The `<signaler>` and `<handler>` elements contain an expression (similar to a *pointcut* expression) that will match the methods signature responsible, respectively, for signaling and handling the exceptions.

```
1. <contract id=1 description="Glassbox Contract">
2.   <exception type="org.aspectj.lang.SoftException">
3.     <signaler signature="glassbox.monitor.ui.ServletRequestMonitor.*"/>
4.     <handler signature="hw.handling.ErrorIsolation"
```

```
5.          type="same_exception"/>
6.   </exception>
7. </contract>

8. <contract id=2 description="Hibernate Contract">
9.  <exception type=" org.aspectj.lang.SoftException">
10.   <signaler signature="HibernateSessionManagerHW.*" />
11.   <handler signature="hw.handling.HibernateExceptionHandling.*"
12.       type="same_exception"/>
13. </exception>
14. </contract>
```

Listing.2. Exception Handling Contracts.

The first contract (id=1) defines that the `ErrorIsolation` aspect should handle any instance of `SoftException` signaled by any advice defined on the `ServletRequestMonitor` class. The second contract (id=2) states that the `HibernateExceptionHandling aspect` is responsible for handling the `SoftException` signaled by any advice defined on the `HibernateSessionManagerHW` aspect. Moreover, both contracts state that such exceptions should be caught by a `catch` clause whose argument is of the same type as the exception being caught (lines 5 and 12).

These handlers implement two different exception handling policies (see Section 4): one based on *Error-isolation*, and other based on *App-specific error handling*. The developer does not want exceptions that escape from the monitoring aspect library to affect the application normal control flow. On the other hand, if an exception occurs during data persistence (that relies on the transaction concern), the developer wants to notify that the requested transaction could not be performed. The exception thrown by the Hibernate aspect library should flow until the GUI layer, and each *servlet* should then handle the `SoftException` and present a proper error message to the user.

5.3 Implement Exception Handling Code

In this step, the exception handler aspects specified on the contracts defined above should be implemented. Listing 3(a) illustrates the partial code of the `ErrorIsolation` aspect.

```
public aspect ErrorIsolation {
  …
 public pointcut scope() :
     within(ServletRequestMonitor);

 void around():adviceexecution() &&
      scope()){
    try {
       proceed();
    } catch (SoftException e) {
      log(e);
    }
  }
}                (a)
```

```
public aspect HybernateExceptionHandler{

  public pointcut scope():
    within(hw.gui.* && HttpServlet+) &&
    (execution(* HttpServlet.do*(..)) ||
  execution(* HttpServlet.service(..)));

   void around():scope()){
     try {
            proceed();
     } catch (SoftException e) {
            presentUserMessage(e);
     }
  }                (b)
```

Listing 3. Code snippet for the `ErrorIsolation` aspect.

In order to isolate the exception that flows from the *monitoring aspect library*, the exception handling aspect (`ErrorIsolation` in Figure 4) needs to directly intercept the

aspect library *bytecode*[5]. Doing so, the `ErrorIsolation` aspect prevents the monitoring exception from affecting the flow of execution of the application. Similarly, the `HibernateExceptionHandler` aspect is implemented to handle the exceptions thrown by Hibernate aspect library. This aspect intercepts the base code, more specifically the `doGet(..)` and `doPost(..)` methods, using an around advice. This aspect handles instances of `SoftException` and presents a specific error message to the user. Listing 3(b) illustrates the partial code of `HibernateExceptionHandler` aspect.

5.4 Calculate Exception Paths

In order to assure that the *exception handling solutions* are correctly implemented, we use the SAFE tool to calculate the *exception paths* for the exceptions signaled by the *monitoring* and the *transaction management* crosscutting functionalities. Listing 4 illustrates one of the *exception paths* calculated by the SAFE tool for these exceptions[6]: Besides calculating the *exception paths*, the SAFE tool also checks whether they obey the exception handling contracts defined at the previous step. The automatic checking of exception handling contracts is useful when many *exception paths* should be analyzed. During the exception handling contract verification on the *exception paths,* we can observe that the `SoftException` is not handled by the element specified in the contract (see Listing 3(b)).

```
Exception: org.aspectj.lang.SoftException
Exception Path:
 (Signaler)<HibernateSessionManagerHW: Object ajc$around$ID(…)>
 (Intermediate)<ComplaintDAOImpl: search (…)>
 (Handler)<HealthWatcherFacade: searchComplaint (…)>
 (Action)org.aspectj.lang.SoftException captured by java.lang.Exception
                 Contracts: Hibernate Contract (id:2) broken
```

Listing 4. List of Exception Handling Contracts.

5.5 Manually Inspect the Exception Handling Code

During code inspection, we observed that the instance of `SoftException` that can be signaled by `HibernateExceptionHandler` is mistakenly handled by a *"catch all"* clause defined on the system Facade (see `HealthWatcherFachade` class in Figure 4) before it can reach the join point intercepted by the handler aspect (`HybernateExceptionHandler`) presented in Listing 3(b). This kind of problem could hardly be anticipated during the development of `HybernateExceptionHandler` because the exception handling aspect will intercept the correct join point (where the exception should be caught) and no warning will tell the user that no exception will reach this point. In this case study such problem could be detected with an appropriate tool support: the SAFE tool. One way of solving this broken exception handling contract is to replace the *"catch all"* clause defined in the Facade element by specific catch clauses (one per exception handled at this point). Doing so, the instance of SoftException will

[5] AspectJ allows the weaving of aspects into *bytecode* (that may contain woven aspects) by `inpath` compile option.

[6] We omit package names, return types and advice IDs for simplicity.

flow until it reaches the join points intercepted by the exception handling aspect (in the GUI layer).

6. Discussions and Lessons Learned

This section provides further discussion of issues and lessons we have learned while applying our approach to reuse scenarios of aspect libraries.

Static analysis x Testing Exceptional Conditions. Our approach relies on static analysis in order to discover which exceptions may flow from aspect libraries. To discover such exceptions, we could alternatively write integration tests to verify whether aspect libraries affect the application code as expected under exceptional conditions. However, the test of exceptional conditions is inherently difficult, due to the huge number of possible exceptional conditions to simulate in a system and the difficulty associated to simulate most of such scenarios [5].

Aspect Libraries Development. This paper focused on the reuse of *aspect libraries*, but we could observe that some approach's steps could be useful when the developer implements her/his own aspect library. Using a similar approach, the aspect library developer could: (i) isolate the client code from exceptions that may flow from library code (*Error-isolation* strategy in Section 3.2); (ii) or *explicitly document* the library advices that will affect the base code and the exceptions that may flow from them. As a *crosscutting interface* (XPI) [40] is a way of documenting the points of a system that can be affected by aspects, such *explicit documentation* could work by reverse. The *Exceptional Interface* (EXI) *of the aspect library* could contain which exceptions (*checked* or *unchecked*) can be signaled by every library aspect that will affect points in the base code. The SAFE tool can be used to automatically generate the EXI of aspect libraries – which could be directly used by the developer or used though an IDE [35]. As we discussed in this paper, such documentation would be very useful when developing robust systems.

Load-time weaving. As mentioned before, some aspect libraries can be reused in load-time. However, in order to assure that the aspect library reused at *load-time* will not impair system robustness, it is fundamental to prepare their code beforehand for the exceptions that may flow from aspect libraries in runtime. This can be accomplished by adopting the approach proposed here (possibly only during the first time an aspect library is reused).

Static Analysis based on Java bytecode. The SAFE tool is based on the static analysis of Java *bytecode*. The advantage of working with the Java bytecode, instead of the AspectJ source code, is that we can incorporate in our analysis the exceptions that flow from aspect libraries and OO libraries. However, sometimes the developer should deal with the SAFE tool output that may contain automatically generated *advice methods signatures* (e.g., void ajc$after$MonitorContextLoaderManagement$1$6e34821 (...)) [39]. We are currently devising a strategy to map the advice representation on the *bytecode* to its representation on the source code. This will make the tool output more user-friendly.

7. Related Work

So far, initial work has been developed which investigate the problems related to library aspects reuse [25, 41, 42]. These works focused on the context of incremental software development, and how an aspect may affect subsequent integrated elements, although it was implemented without being aware of them. These works discuss the unpredictable effects and errors that can arise from such scenarios. Although they discuss anticipated aspect composition problems, they do not tackle the problems that may arise when exceptions flow from re-used aspects. In our work we investigated the collateral effects of reusing aspects in the presence of exceptions. Moreover, we also proposed an approach to help developers to deal with them. Although some problems related to aspect libraries reuse are similar to the ones associated with OO libraries reuse, we have shown that some characteristics of AO compositions aggravated the problems.

8. Concluding Remarks and Future Works

This work presented an approach that aims at guiding the developer during the stage of aspect libraries reuse. It supports reasoning about the exceptions that can flow from aspects; and it provides brief and clear guidelines of how such exceptions should be handled. The contributions of this work, however, are not limited to developers of robust aspect-oriented applications who need to make more informed decisions when reusing aspect libraries in the presence of exceptions. But the approach is also useful to help developers when building their own reusable aspect libraries. It provides a way to identify potential problems that may happen on different reuse scenarios. Furthermore, the present work also allows for designers of AO languages to consider pushing the boundaries of existing mechanisms to make AOP more robust and resilient to exceptional conditions. There are several ways our work can be continued: (i) apply this approach to other reuse scenarios, in order to perform more extensive validation; (ii) investigate the usefulness of this approach in software evolution scenarios; and finally (iii) adapt the approach to aspect library development.

8. References

[1] Aldrich, J.: Open Modules: Modular Reasoning about Advice. In: Proc. of ECOOP'05.

[2] Alexander, R.T.; Bieman, J.M.; Andrews, A.A. Towards the Systematic Testing of Aspect-Oriented Programs. Technical Report CS-04-105. Dept. of Computer Science, Colorado State University Fort Collins/Colorado - USA, March 2004.

[3] Bodkin, R., Next steps with aspects, AOP@Work: http://www.ibm.com/developerworks/java/library/j-aopwork16/index.html 11/ (Mar 2006).

[4] Bodkin, R., Performance monitoring with AspectJ, Online: http://www.ibm.com/developerworks/java/library/j-aopwork10/ and ../j-aopwork12 (Sep 2005). [5]

[5] Bruntink, M.; Deursen, A.V.; Tourwe, T. Discovering faults in idiom-based exception handling. In: Proceedings of the International Conference on Software Engineering, 2006, p.242-251. [36]

[6] Cabral, B., Marques, P.: Exception Handling: A Field Study in Java and .NET. In: Proc. of ECOOP'07.

[7] Castor Filho, F., Garcia, A., Rubira, C.: Extracting Error Handling to Aspects: A Cookbook. In: Proc. of ICSM'07.

[8] Coelho, R, Analyzing the Exception Flows of Aspect-Oriented Programs, PhD Thesis, PUC-Rio, July 2008.

[9] Coelho, R, Awais, R., Garcia, A., Ferrari, F. Cacho, N., Kulesza, U., Staa, A., Lucena, C., Assessing the Impact of Aspects on Exception Flows: An Exploratory Study, In: Proc. of ECOOP´2008. (to appear)

[10] Coelho, R. et al, Assessing the Impact of Aspects on Exception Flows: An Empirical Study. Website: http://www.inf.puc-rio.br/~roberta/aop_exceptions
[11] Colyer, A., et al. Eclipse Aspectj: Aspect-Oriented Programming with Aspectj and the Eclipse Aspectj Development Tools. 2004: Addison-Wesley.
[12] Filman, R., Friedman, D., Aspect-oriented programming is Quantification and Obliviousness. In Aspect-Oriented Software Development. Addison-Wesley, 2005.
[13] Fu, C., A. Milanova, et al. Robustness testing of Java Server Applications. IEEE Transactions on Software Engineering.
[14] Fu, C.; Ryder, B. G.: Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In: Proc. of ICSE'07.
[15] Glassbox Inspector. https://glassbox-inspector.dev.java.net/
[16] Goodenough JB. Exception handling: Issues and a proposed notation. Communic. of the ACM 1975.
[17] Greenwood, P.; et al.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: Proc. of ECOOP'07. pp. 176–200
[18] Hannemann, J.; Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: Proc. of OOPSLA'02, 2002
[19] JBoss Cache: http://labs.jboss.com/jbosscache/
[20]Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W., "Getting Started with AspectJ", Communication of the ACM, 44(10), 2001, pp. 59-65.
[21] Lemos, O.; Ferrari, F.; Lopes, C.; Masiero, P. Testing aspect-oriented programming Pointcut Descriptors. In: Workshop on Testing Aspect-Oriented Programs, 2006, p.33-38.
[22] Lopes, C.; Ngo, T. Unit-Testing Aspectual Behavior. In: Proceedings of the Workshop on Testing Aspect-Oriented Programs 2005 (WTAOP 2005), 2005.
[23] Massiote, E.P.; Badri, L.; Badri, M. Towards a Tool Supporting Integration Testing of Aspect-Oriented Programs. Journal of Object Technology, 6(1), 2007, p.67-89.
[24] Massoni, T.; Alves, V.; Soares, S.; Borba, P. PDC: Persistent Data Collections pattern. In: SugarLoafPLoP 2001, p.311–326.
[25] McEachen, N., Alexander, R., Distributing Classes with Woven Concerns – An Exploration of Potential Fault Scenarios. In Proc. of AOSD' 05, pp.192-200.
[26] Miller, R.; Anand, T.: Issues with Exception Handling in Object-Oriented Systems. In: ECOOP'97.
[27] Robillard M. P., Murphy, G., Designing robust java programs with exceptions. In Proc. of FSE 2000.
[28] Robillard, M.; Murphy, G.: Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems. In: ACM Trans. Softw. Eng. Methodol (2003).
[29] Sacramento, P.; Cabral, B.; Marques, P. Unchecked Exceptions: Can the Programmer be Trusted to Document Exceptions? In: Proceedings of IVNET'06, 2006.
[30] Sinha, S. and Harrold M., Analysis of Programs with Exception-Handling Constructs, In: ICSM'98.
[31] Soares, S.; Borba, P.; Laureano, E.: Distribution and Persistence as Aspects. In: Software: Practice and Experience, Wiley, vol. 36 (7), (2006) 711-759.
[32] Soot: A Java Optimization Framework. http://www. sable.mcgill.ca/ soot, accessed 19/12/2007.
[33] Spring AOP aspect library:http://www.springframework.org/
[34] Thomas, D. The Deplorable State of Class Libraries. Journal of Object Technology, 1(1), 2002, 21-27.
[35] Saurabh Sinha, Alessandro Orso, Mary Jean Harrold: Automated Support for Development, Maintenance, and Testing in the Presence of Implicit Control Flow. ICSE 2004: 336-345.
[36] Xie, T.; Zhao, J. A framework and tool supports for generating test inputs of AspectJ programs. In: Proceedings of the In Proc. 5th International Conference on Aspect-Oriented Software Development, 2006, p.190–201.
[37] Xu, W.; Xu, D. State-Based Testing of Integration Aspects. In: Proceedings of the Second Workshop on Testing of Aspect-Oriented Programs (WTAOP'06). In conjunction with ISSTA'06,2006, p.7-14.
[38] Zhao, J. Data-flow-based unit testing of aspectoriented programs. In COMPSAC'2003.
[39] Hilsdale, E.; Hugunin, J. Advice weaving in AspectJ. In: Proceedings of the In 3rd International. Conference on Aspect-oriented Software Development (AOSD 2004), Lancaster, UK, 2004, p.26–35.
[40] Sullivan, K., Kevin Sullivan , William G. Griswold , Yuanyuan Song , Yuanfang Cai , Macneil Shonle , Nishit Tewari , Hridesh Rajan, Information hiding interfaces for aspect-oriented design, In FSE 2005.
[41]Lopez-Herrejon; Batory, D.; Lengauer, C. A disciplined approach to aspect composition. In: ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation, 2006, p.68 – 77
[42]Apel, S., Leich, T.; Saake, G. Aspectual Mixin Layers: Aspects and Features in Concert. In: ICSE 2006, p122-131.