6
The Approach

In Chapter 2 we mentioned that the exception handling code is often the least well understood (Sinha and Harrold, 1998; Robillard and Murphy, 2000; Robillard and Murphy, 2003) and tested (Sinha and Harrold, 1999; Fu and Ryder, 2005) part of the system. Since it is not the *primary concern* to be implemented, it does not receive much attention during system design, implementation and test phases. Moreover, testing the exception handling code is inherently difficult. Firstly, because it is tricky to simulate some exceptional conditions during tests (e.g., network problems). Secondly, because the large number of different exceptions that can happen in a system in runtime may lead to *test-case explosion* problem (Hanenberg et al., 2003).

The lack of verification approaches for the exception handling code of AO systems and the difficulties mentioned above – related to the exception handling code testing - stimulated the present work. This chapter presents a verification approach based on static analysis, to check the reliability of the exception handling code of AO applications. This approach is based on SAFE, the exception-flow analysis tool presented on the previous chapter. SAFE is used in this approach to help developers to reason about the exception flow and find *bugs* on the exception handling code of AO applications developed in AspectJ.

6.1.
Checking the Reliability of Exception Handling Code

The goal of the proposed approach is to help developers to prevent the exceptions signaled or handled by aspects from threatening the robustness of the application they are composed to. As illustrated in Figure 24, the aspects that compose an AO application may be of two kinds: *application aspects* and *library aspects.*

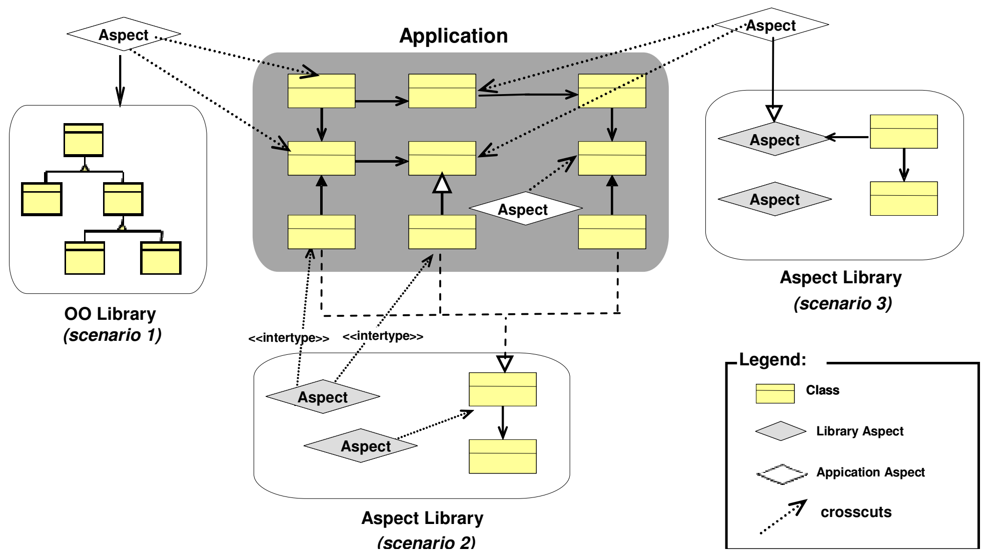

Figure 24. Types of aspects in an AO application.

The *application aspects* are developed in the application core, and may (i) implement a crosscutting concern by their own, (ii) integrate existing functionalities of the base code, or (iii) integrate the base code with functionalities available in OO libraries (see scenario 1 in Figure 24). The *library aspects* are pre-compiled aspects, implemented by third party developers, that implement traditional crosscutting concerns (i.e., performance monitoring, security, and transaction management) that would be spread in many application modules otherwise (Bodkin, 2006). Reusing an aspect library often means: (i) extending an abstract aspect with a concrete aspect that redefines the points on the application where the aspect should apply (see scenario 3 in Figure 24); or (ii) using `declare parents` AspectJ construct[28] to add a *tag interfaces* (Colyer, 2004) to some of the elements in the application (see scenario 2 in Figure 24) – which will identify which elements will be intercepted by the library aspects.

In this chapter we focus on the robustness and consistency issues pertaining to exceptional flow of programs using such *aspect libraries* or *application aspects.* When developing *application aspects* and reusing *library aspects* in exception-aware applications, the AO developer faces common difficulties. As mentioned on previous chapters, when an aspect adds a new functionality to a

[28] The `declare parents` AspectJ construct enables the developer to change the type hierarchy of a system.

system, this additional functionality can also bring new exceptions (e.g., access of null references, a noisy user input or faults on underlying middleware or hardware). Consequently, even simple aspects used for logging or monitoring (Coelho et al., 2006b) purposes, for instance, may throw exceptions that will flow through the system. If not adequately handled such exceptions may remain *uncaught* or may be handled by an existing handler on the base code (i.e., *unintended handler action* (Miller and Tripathi, 1997)).

One may argue that such problems also occur in OO development: we do not know which runtime exceptions that may flow from OO code as well and, consequently, cannot prepare the code to deal with them. Indeed, this is a real problem in OO development, and some static analysis tools, e.g., as proposed in (Robillard and Murphy, 2003; Fu et al., 2005), could be used to deal with it. However, some characteristics of aspect-oriented compositions *strengthens* these problems, such as: (i) *the ability to externally modify the behavior of the base code* (Krishnamurthi et al., 2004; Aldrich, 2005), (ii) some developers and approaches advocating an *oblivious development process* (Filman and Friedman, 2005), (iii) *the load-time weaving* (Colyer, 2004; Bodkin, 2005; Bodkin, 2006) available in some AO languages, (iv) and the *quantification* property (Filman and Friedman, 2005) (as detailed in Section 6.1.1).

Therefore, while aspect-oriented compositions allow the development and reuse of crosscutting concerns, they might render less useful if they introduce new exceptions that may lead to potential faults. Currently, there is no approach or supporting tool[29] to help application developers to: (i) discover which exceptions may flow from an *application aspect or* a *library aspect*, (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 application aspects and especially third party aspects (from *aspect libraries*), to applications robustness in *exceptional scenarios.*

This chapter focuses on the exceptions that may flow from aspects, the consequences that they may bear, and how to deal with them. The main questions we seek to address here are the following:

[29] The static analysis tools that find exception paths in OO programs cannot be used in a straightforward fashion in AO programs, because they cannot interpret the effects on *bytecode* after the aspect weaving process.

- *What are the potential consequences of developing or reusing aspects 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. In the following sections we firstly discuss about challenges associated with aspect reuse in the presence of exceptions and detail the relatively new concept of *aspect libraries*. Then we present a verification approach based on static analysis that allows programmers to statically look for bugs on the exception handling code of aspect-oriented development systems. This verification approach takes into account exceptional situations neglected by the verification approaches for AO programs proposed so far. Section 6.3 shows how SAFE tool can be used to support some steps of the approach. Section 6.4 summarizes our experience when applying this approach to different aspect reuse and development scenarios. Section 6.5 presents an analysis of the precision of the SAFE tool – since the approach's effectiveness is strongly related to the precision of the tool output. And finally in Section 6.6 we provide further discussion of lessons learned.

6.1.1. The Characteristics of AO Compositions x the Development of Exception-ware AO Systems

Some characteristics of aspect-oriented development *strengthens* existing kinds of failure (e.g., uncaught exceptions, unintended handler action) on OO development concerning the way exceptions are handled and thrown inside the system. The *modification* (Krishnamurthi et al., 2004; Aldrich, 2005) performed by aspects 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 most often[30] 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 25 illustrates a `before` advice that intercepts an application method and

[30] When intertype declararions are used some editing may be necessary.

throws an exception. This exception will flow through the base code and interrupting the normal control flow of the advised code. 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. On the other hand, on the AO system presented in Figure 25, since the pointcut language available in most AO languages does not allow the developer to intercept the execution of the specific before advice that threw the exception, the AO developer cannot to avoid the exceptions (thrown by the `before` advice) from flowing though the system.

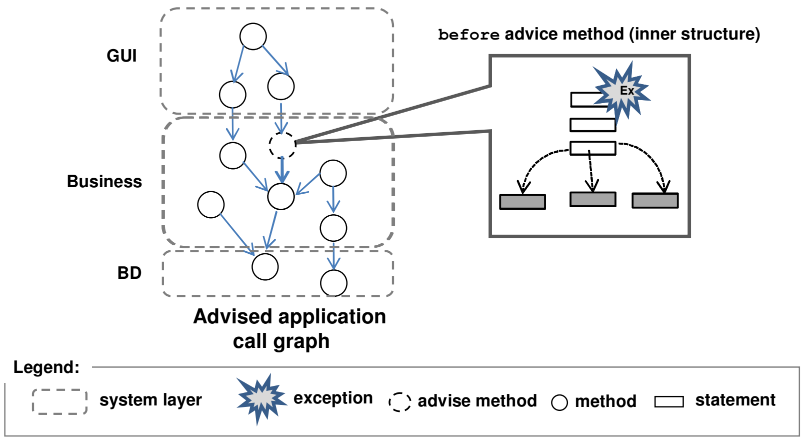

Figure 25. Consequences of aspectual modifications on the base code

Moreover, some AO development approaches rely on the *obliviousness* property (Filman and Friedman, 2005). 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 cannot prepare the code to deal with exceptions that may escape from aspects. Even if the aspect developer creates an aspect responsible for handling the exceptions thrown by crosscutting concerns, the AO developer cannot assure that this exception will not be mistakenly handled by a catch clause on the base code (see Inactive Handler Aspect in Chapter 4).

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.

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 in an application, new handlers should be defined in different places within the base code (one for each path in the call graph that may reach the affected method – see Figure 26). Even if the AO developer creates specific e*xception handling aspects* (Filho et al., 2007)) for handling the new exceptions, the s/he cannot assure that such exception will not be mistakenly handled by a catch clause on the base code (see Inactive Handler Aspect in Chapter 4). Moreover, since the way the exception should be handled o these points may differ, one e*xception handling aspect* per exception would not be sufficient.

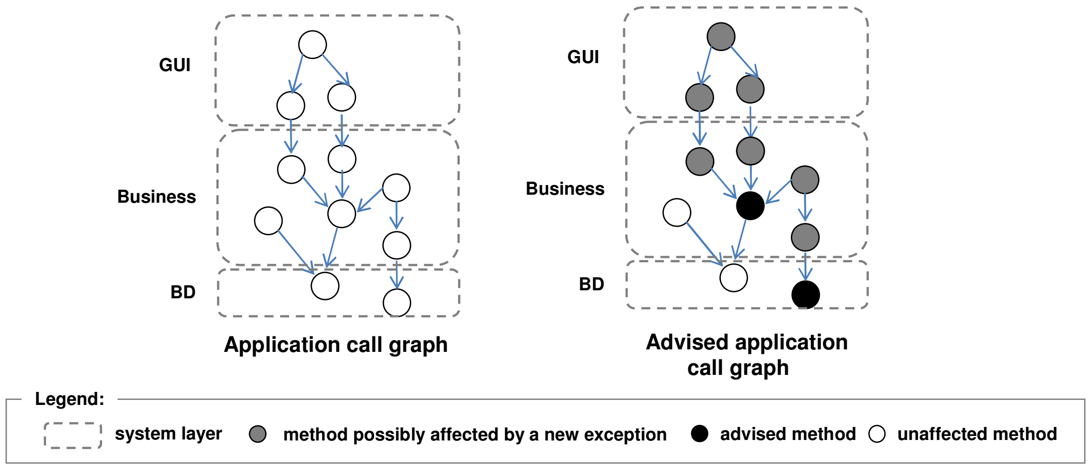

Figure 26. Consequences of quantification property in the presence of exceptions.

Since the exception handling policy[31] (Robillard and Murphy, 2000) of an application is almost always based on its architecture, adding exceptions on unrelated points in the code may potentially break the exception handling policy.

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

The combination of these characteristics, therefore, results in fault-prone exception handling scenarios in AO systems.

6.2. Status of Current Aspect Libraries

Although pre-built aspect libraries are a relatively new reuse artifact, several useful collections have already become available, including: the Spring AOP aspect library (Johnson, 2007), the Glassbox Inspector[32], the JBoss Cache[33], and GOF patterns aspect library[34]. Such libraries implement traditional crosscutting concerns (i.e., performance monitoring, security, and transaction management), and enable the developer to extend the functionalities of existing applications avoiding significant coupling and large re-investments (Bodkin, 2006). According to Bodkin (2006) "*reuse will prove to be the most important benefit from adopting Aspect Oriented Programming*."

While aspect libraries introduce new possibilities for application composition, the circumstances mentioned above (see Section 6.1.1) may threaten the application's robustness and design consistency. Aspect libraries developers may do their best to ensure that libraries' functions do not create faults that impact the applications in which they are composed. However, unexpected behavior in aspect library code (e.g., unanticipated null values, undocumented runtime exceptions thrown by libraries) is often present (Coelho et. al, 2008).

In order to know which exceptions may flow from aspect libraries, programmers must rely on the libraries documentation - that is, more than often, neither complete nor precise. As a consequence, the developer may only realize the existence of unexpected exceptions thrown by an aspect library, by observing the failures caused by them on the application in *runtime*, such as: *uncaught exceptions* – which lead to unpredictable software crashes; and *unintended handler actions* – which lead to wrong recovery actions (e.g., wrong error messages presented to the user).

The additional expense that is required to account for the effects of exception occurrences on aspect libraries may not be justified unless exceptions

[32] https://glassbox-inspector.dev.java.net/
[33] http://labs.jboss.com/jbosscache/

occur frequently in aspect libraries in practice. A recent study (Cabral and Marques, 2007) shows that the amount of code dedicated to exception occurrences (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 12. In the observed subjects from 2,09% to 8,82% of the total lines of code were dedicated to exception raising and handling concerns (see LOC EH in Table 12).

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 12. Characteristics of the exception handling (EH) code on aspect libraries.

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

6.3. The Verification Approach

In this section, we present an approach to help AO developers to check the reliability of the exception handling code. 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 *uncaught exceptions* and *unintended handler actions*. Figure 27 illustrates the main steps that compose our approach.

[34] http://hannemann.pbwiki.com/Design+Patterns

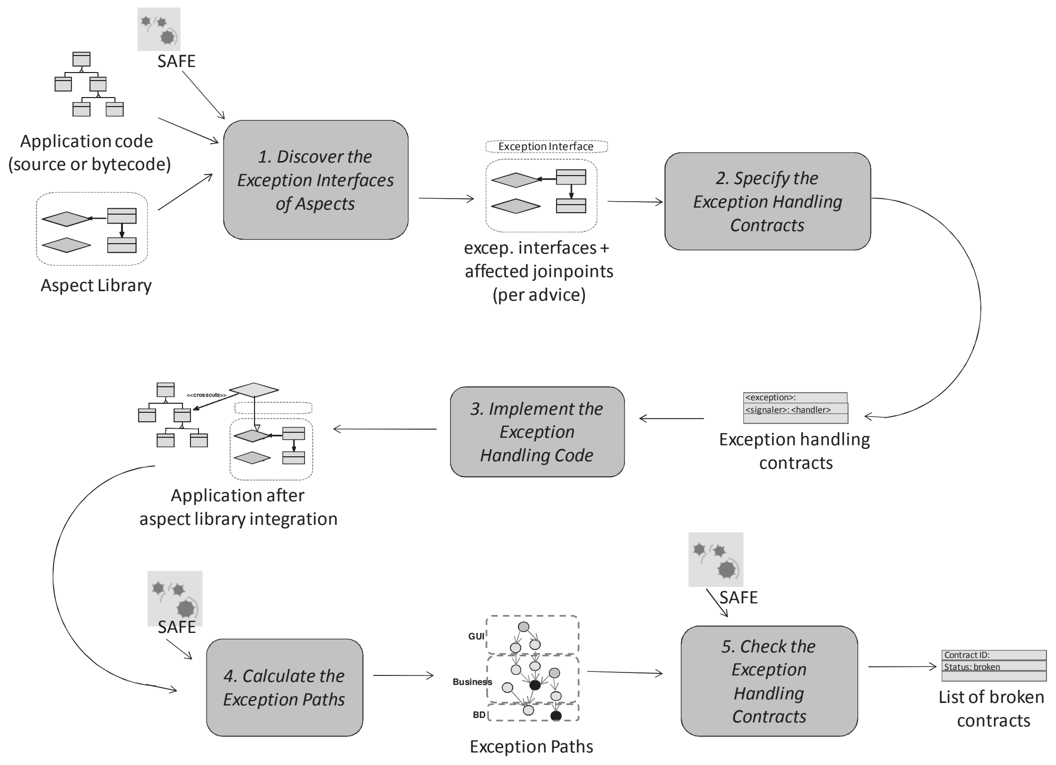
SAFE
Application code (source or bytecode)
Aspect Library
1. Discover the Exception Interfaces of Aspects
Exception Interface
excep. interfaces + affected joinpoints (per advice)
2. Specify the Exception Handling Contracts
<exception>:
<signaler>:<handler>
Exception handling contracts
3. Implement the Exception Handling Code
<<crosscuts>>
Application after aspect library integration
SAFE
4. Calculate the Exception Paths
GUI
Business
BD
Exception Paths
SAFE
5. Check the Exception Handling Contracts
Contract ID:
Status: broken
List of broken contracts

Figure 27. The proposed approach.

Much of the effort in our approach involves working out how to effectively integrate aspect into the base code without the risk of introducing the faults on the exception handling code – which may represent causes of potential software crashes. The verification approach presented here complements the available AO testing approaches that focus on the system normal control flow (Zhao, 2003; Alexander et al., 2004; Lopes and Ngo, 2005; Coelho et al., 2006a; Lemos et al., 2006; Xie and Zhao, 2006; Xu and Xu, 2006; Lemos et al., 2007; Massicotte et al., 2007). These approaches perform unit and integration tests and looks for faults on 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 exception can lead to a *test case explosion problem*, as mentioned before.

The approach is based on SAFE tool, and comprises of the following steps, as illustrated in Figure 27:

1. ***Discover the Exception Interfaces of Aspects.*** In this step the SAFE tool is used to discover the list of exception types that may flow from each advice defined on application aspects or library aspects that crosscuts the base code. 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[35]. Moreover, once the exception interfaces of a set of aspects are discovered it can be reused along the development cycle (see Chapter 8, Section 8.3).

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*. As mentioned in Chapter 5, Section 5.3.4, the *exception handling contracts* specify the *Signaler-Handler* relation per exception thrown by aspects, and are specified in the SAFE tool in XML format.

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., as aspect defined to handled exceptions (Filho et al., 2007)) that directly intercepts *library aspects* or *application aspects.* Or in the case of application aspects for which the developer has

[35] Some approaches as the one proposed by Xie and Zhao (2006) automatically generate a set of stubs to be intercepted by aspects.

access to the source code, handlers should be defined on every advice method that signals an exception.

- *App-specific error handling:* the developer can define an *exception handling aspect* that intercepts specific join points in the application code (handling on aspects), or define catch clauses inside application elements that will be responsible for handling the exceptions thrown by aspects (although this design practice is not advocated by most of the current AO development approaches).

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 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. This step based on manual inspections is necessary in this approach, because the exception handling contracts (defined on SAFE tool) can only express simple constrains concerning the way the exception should be caught (e.g., which element should handle the exception). Thus, errors concerning the actions performed inside the handler after catching the exceptions can only be discovered by manual inspections in the current approach.

We can observe that the steps presented above - that compose the verification approach - can also be applied in OO development, with slight modifications. For instance, at step 1, instead of discovering the exception interfaces of aspects to be woven with the base code, SAFE tool could be used to discover the exception interfaces of classes (specially the *runtime* exceptions which are not defined on class methods signatures) to be composed with the basecode. However, in this chapter we focus on the development of aspects, and the impact of aspect weaving on the exceptional behavior of the system. In the following sections, we exemplify how our approach can be used to assure the quality of the exception handling code in real implementation scenarios involving *application* and *library* aspects.

6.4.
Worked Example

To illustrate our approach, we selected two real change scenarios to be applied to the first AO version of Health Watcher (HW) system (Soares, 2004; Greenwood et al., 2007) – the Web-based information system described in Chapter 3. The change scenarios are: (1) to monitor the performance of http requests, and (2) 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 of the source code, which can be tedious, error-prone, and quite difficult to maintain. In our case study, we reused the Glassbox Inspector (Bodkin, 2005), an *aspect monitoring library*, in order to implement the monitoring concern in the HW system. In our case study, the transaction management concern – which is also a typical crosscutting concern - was also implemented using an aspect library. We reused the transaction management *aspect library* built on top of Hibernate [36] (Colyer, 2004), an open source object relational mapping tool for a Java, to implement this concern.

Figure 28 illustrates the architecture of HW system after composing it with the aspect libraries described before (i.e., the *Monitoring Aspect Library,* and the *Hibernate Aspect Library*).

[36] http://www.hibernate.org

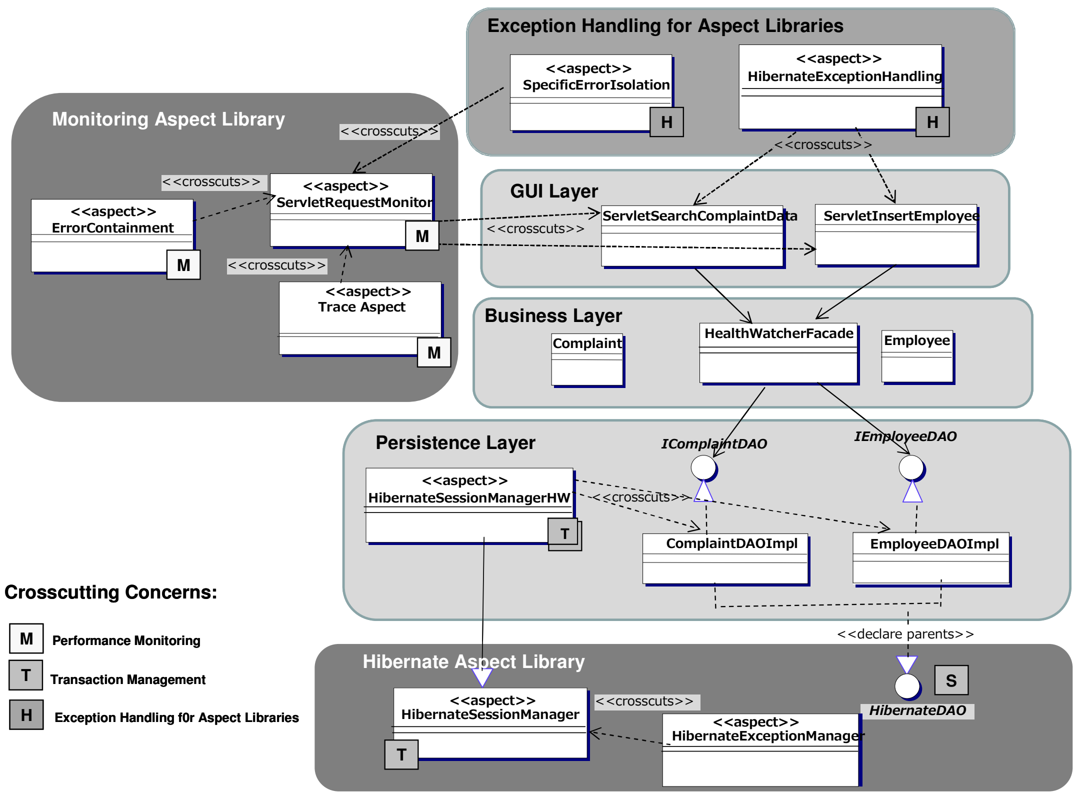
Exception Handling for Aspect Libraries
<<aspect>> SpecificErrorIsolation
H
<<aspect>> HibernateExceptionHandling
H
Monitoring Aspect Library
<<crosscuts>>
<<aspect>> ErrorContainment
M
<<aspect>> ServletRequestMonitor
M
<<aspect>> Trace Aspect
M
GUI Layer
ServletSearchComplaintData
ServletInsertEmployee
Business Layer
Complaint
HealthWatcherFacade
Employee
Persistence Layer
IComplaintDAO
IEmployeeDAO
<<aspect>> HibernateSessionManagerHW
T
ComplaintDAOImpl
EmployeeDAOImpl
<<declare parents>>
Crosscutting Concerns:
M Performance Monitoring
T Transaction Management
H Exception Handling f0r Aspect Libraries
Hibernate Aspect Library
<<aspect>> HibernateSessionManager
T
<<aspect>> HibernateExceptionManager
S
HibernateDAO

Figure 28. The extended architecture of Health Watcher system.

As presented in Chapter 3, the HW system also contains a set of *application aspects* that implement crosscutting concerns such as: concurrency control, persistence (partially) and exception handling (partially). For the sake of simplicity, however, Figure 28 focuses on the crosscutting concerns implemented in this case study and omits the others. The HW elements presented in Figure 28 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.

6.4.1. 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) `ServletRequestMonitor` – a *library aspect* that directly intercepts every application request operation (see Figure 28); 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 28).

The SAFE tool recursively analyzes the *woven bytecode* and calculates the exception interface of every aspect advice (analyzing every method called from them and every advice that may intercept them). Listings 20 and 21 illustrate the partial code of the `ServletRequestMonitor` and `HibernateSession-ManagerHW` and the *exception interfaces* reported by SAFE for some advices[37].

```
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");
       }
   }
    …
}
```

```
Aspect: ServletRequestMonitor
Advice Type: after returning
Sequence: 01
Advice Method Signature:
glassbox.monitor.ui.ServletRequestMonitor: void ajc$afterRetu
rning$glassbox_monitor_ui_ServletRequestMonitor$4$9166ad69(…)
Exception Interface: org.aspectj.lang.SoftException
```

[37] Some parameters are omitted for readability

Listing.20. The Partial code of `ServletRequestMonitor` and the exception interface of the `after returning` advice.

```
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;
  }
 …
}
```

```
Aspect: HibernateSessionManagerHW
Advice Type: around
Sequence: 01
Advice Method Signature:
hibernate.HibernateSessionManagerHW: Object ajc$around$hiber
nate_HibernateSessionManagerHW$1$a506ba1(…)
Exception Interface: org.aspectj.lang.SoftException
```

Listing.21. Partial code of `HibernateSessionManagerHW` and the exception interface of the `around` advice.

As detailed in Chapter 5 the SAFE tool runs on the woven bytecode. When building the woven bytecode, the AspectJ weaver 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 (see Chapter 5, Section 5.1.1). In the Listings above the "**`Advice Method`**

`Signature`" contains the automatically generated method signatures for the advices presented above. "**`Sequence`**" specifies the order in which the advice was defined on the bytecode.

At this step, the developer may optionally investigate the causes of the exceptions that flow from each aspect advice - since this information is also calculated by the SAFE tool during the exception-flow analysis[38]. Table 13 below presents the causes of the `org.aspectj.lang.SoftException`, which may be thrown by the `after returning` advice defined on the `ServletRequestMonitor` aspect.

We can observe from Table 13 that the `org.aspectj.lang.SoftException` may be caused by three different exceptions that are *softenized* (i.e., handled, wrapped in a `SoftException` and then re-thrown). The information presented in this table exemplifies how complex it can be to try to reason about the flow of exceptions without a tool support. A similar analysis can be performed to find out the possible causes of `SoftException` that can flow from the Hibernate aspect library.

Original Exception	**Exception Path before it is Softened**
java.lang.IllegalArgumentException	**`(Signaler)`**<glassbox.debug.TraceAdvice: void ajc$afterThrowing$ glassbox_debug_TraceAdvice$3$4c73158c(…)> **`(Handler)`**<glassbox.monitor.ui.ServletRequestMonitor: void ajc$ afterReturning$glassbox_monitor_ui_ServletRequestMonitor $4$9166ad69(…)> **`(Action)`** net.sf.hibernate.LazyInitializationException captured by java.lang.Exception (Converted to SoftException)
Net.sf.hibernate.LazyInitializationException	**`(Signaler)`**<glassbox.debug.TraceAdvice: void ajc$afterThrowing$glassb ox_debug_TraceAdvice$3$4c73158c(…)> **`(Handler)`**<glassbox.monitor.ui.ServletRequestMonitor: void ajc$ afterReturning$glassbox_monitor_ui_ServletRequestMonitor $4$9166ad69(…)> **`(Action)`** net.sf.hibernate.LazyInitializationException captured by java.lang.Exception (Converted to SoftException)
java.net.SocketException	**`(Signaler)`**<glassbox.debug.TraceAdvice: void ajc$afterThrowing$glassbox_debug_TraceAdvice$3$4c73158c(…)>

[38] The information presented on tables was extracted from the SAFE tool output, which can be presented in two formats: XML and .xls (Microsoft Excel) files.

	(Handler)<glassbox.monitor.ui.ServletRequestMonitor: void ajc$ afterReturning$glassbox_monitor_ui_ServletRequestMonitor $4$9166ad69(…)> **(Action)** Subsumption: java.net.SocketException captured by java.lang.Exception (Converted to SoftException)

Table 13. Exception paths calculated by SAFE tool.

Since the SAFE tool is based on the static analysis of Java *bytecode*, the information presented in Table 13 are not user friendly since it presents the automatically generated method signatures for each aspect advice. On the other hand, then advantage of working on 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 (see Section 6.6).

6.4.2. Specify the Exception Handling Contracts

After discovering the *exception interfaces* of every aspect advice 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 22 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 throw `HibernateException`. 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.

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

<contract id=2 description=”Hibernate Contract”>
 <exception type=”org.aspectj.lang.SoftException”>
   <signaler signature=”HibernateSessionManagerHW.*” />
```

```
11.   <handler signature=”hw.handling.HibernateException
12.             Handling.*” type=”same_exception”/>
13. </exception>
14. </contract>
```

Listing.22. 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 instances of `SoftException` signaled by any advice defined on the `HibernateSessionManagerHW` aspect[39]. 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).

Each **`handler`** defined on the contracts above implements a different exception handling policy (see Section 4): the `hw.handling.ErrorIsolation` is based on *Error-isolation*, and `hw.handling.HibernateException` is 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 (see Figure 28). 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 (see Figure 28).

[39] As illustrated in Section 5 the around advice has a different representation on the woven bytecode when compared to the before and after advices. The around advice is represented by a static method on the affected class. Therefore, the exception handling contract is not only checked besed on the string matching between the information specified on the contract and the advice method signature on the woven bytecode.

6.4.3. Implement Exception Handling Code

At this step the exception handler aspects specified on the contracts defined above should be implemented. Listing 23 illustrates the partial code of `ErrorIsolation` aspect.

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

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

Listing 23. 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 28) needs to directly intercept the aspect library *bytecode*[40]. 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 the `HibernateException` and presents a specific error message to the user. Listing 24 illustrates the partial code of `HibernateExceptionHandler` aspect.

```
public aspect HibernateExceptionHandler {

   public pointcut scope():
```

[40] Since version 1.2, AspectJ allows the weaving of aspects into bytecode (that may contain woven aspects) by using `inpath` compile option.

```
          within(hw.gui.* && HttpServlet+) &&
          (execution(* HttpServlet.do*(..)) ||
           execution(* HttpServlet.service(..)));

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

Listing 24. Code snippet for the `HibernateExceptionHandler` aspect.

6.4.4. Calculate Exception Paths

In order to assure that the *exception handling solutions* are correctly implemented, we use the SAFE tool to compute the *exception paths* for the exceptions signaled by the *monitoring* and the *transaction management* crosscutting functionalities. Listing 25 illustrates some of the *exception paths* computed by the SAFE tool[41]. 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 25).

```
Exception: org.aspectj.lang.SoftException
Exception Path:
 (Signaler)<ServletRequestMonitor: ajc$afterReturning…(…)>
 (Intermediate)<ErrorIsolation: void ajc$around$proceed…(…)>
 (Handler)<ErrorContainment: ajc$around$ErrorIsolation…(…)>
 (Action)org.aspectj.lang.SoftException capured by
         org.aspectj.lang.SoftException
Contracts: Glassbox Contract (id:1) obeyed


Exception: org.aspectj.lang.SoftException
```

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

```
Exception Path:
 (Signaler)<ComplaintRepositoryRDB: search_aroundBody1$advice(…)>
 (Intermediate)<ComplaintRepositoryRDB: search (…)>
 (Handler)<HealthWatcherFacade: searchComplaint (…)>
 (Action)org.aspectj.lang.SoftException capured by java.lang. Exception
Contracts: Hibernate Contract (id:2) broken


Exception: org.aspectj.lang.SoftException
Exception Path:
 (Signaler)<EmployeeRepositoryRDB: search_around Body1$advice(…)>
 (Intermediate)<EmployeeRepositoryRDB: search (…)>
 (Handler)<HealthWatcherFacade: Object searchEmplyee(…)>
 (Action)org.aspectj.lang.SoftException capured by java.lang. Exception
 Contracts: Hibernate Contract (id:2) broken
```

Listing 25. List of Exception Handling Contracts.

Besides calculating the *exception paths*, the SAFE tool also checks whether they obey the exception handling contracts defined at the previous step. During the exception handling contract verification on the *exception paths,* we can observe that the `org.aspectj.lang.SoftException` is not handled by the element specified in the contract (see Listing 22).

6.4.5.
Manually Inspect the Exception Handling Code

During code inspection, we observed that the instance of `SoftException` that can[42] 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 24.

This kind of problem could not be anticipated, during the development of `HybernateExceptionHandler` and was only discovered by 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 Hibernate exception will flow until

[42] This exception can be signaled in some circunstances according to the static analysis information.

it reaches the join points intercepted by the exception handling aspect in the GUI layer.

This manual inspection step was performed on the empirical study presented in Chapter 3, in order to diagnose specific exception handling failures such as the causes the `incorrect user message` (i.e., the catch clause handled the exception and presents a message to the user that is not related to the failure that happened) detailed in Chapter 3, Section 3.1.4.

6.5. The Approach's Effectiveness x the SAFE Tool's Precision

As detailed in Sections 6.3 and 6.4 most of the approach's steps depend on the information computed by the SAFE tool. For this reason, the approach's effectiveness is strongly related to the precision of the tool output. During the implementation of the exception-flow static analysis algorithm of SAFE, we had to deal with a number of implementation trade-offs (detailed in Chapter 5) to balance between the tool's usefulness and the precision, such as:

Flow-Insensitive Analysis. In SAFE the *exception interfaces* of methods comprises the set of exception signaled (directly through the throw statement or indirectly through method calls) and not handled inside the method. The analysis performed by SAFE to discover the exception interfaces does not take into account contextual information - which could establish that a specific exception cannot actually be thrown at runtime. For instance, the method `nextElement()` defined in `java.util.Enumeration` class (Gosling et al., 1996) may signal `NoSuchElementException` if no element is found on the `Enumeration`. However, if a call to this method is *guarded* by a call to `hasMoreElements()` method (which returns false in case there is no element in the enumeration), the instance of `NoSuchElementException` will never be signaled in runtime. Though a context-sensitive analysis could yield more precision, but its cost needs to be further investigated.

The implementation tradeoffs together with the inherent limitations of any static analysis represent the sources of imprecision to the exception analysis algorithm, which can lead to *exception paths* that can never occur in runtime (*spurious paths*). Unfeasible paths are a common problem in every static analysis, and can be addressed by: (i) human examination - to identify spurious paths during manual inspections; and (ii) the definition of new techniques to automatically remove as many spurious links as possible[43].

In our approach, the spurious exception paths were identified during the manual inspections of the exception handling code - guided by the *exception paths* reported by the SAFE tool - and then skipped. Table 14 presents the results of applying the SAFE tool to find the exception paths of the AO versions of Health Watcher, Mobile Photo and JHotdraw systems[44] - collected during the empirical study presented in Chapter 3.

System	Reported	Spurious	Category 1	Category 2
Health Watcher AO v1	212	0	0	0
Health Watcher AO v9	338	0	0	0
Mobile Photo AO v4	117	11	0	11
Mobile Photo AO v6	132	18	0	18
AJHotDraw	281	37	37	0

Table 14. Reported and spurious exception paths.

The first column lists the system considered, the column 2 lists the number of reported exception paths, the column 4 lists the *spurious exception paths* that were detected during manual inspections, the column 5 lists the number of *spurious exception paths* that can be detected using a context-sensitive call graph construction algorithm (Category 1) and column 5 contains the number of *spurious exception paths* can be removed by a more precise analysis (Category 2).

The number *spurious exception paths* found in this study is sufficiently low to make the SAFE tool useful in practice. A detailed look at these *spurious exception paths* reveals the reasons why they occur and tell us how we can improve the SAFE tool[45]. The *spurious exception paths* found in Mobile Photo were caused after throwing advice that unconditionally threw an exception - overriding the exception previously thrown by the advised code. Listing 26

[43] Some unfeasible paths, however, cannot be automatically removed due to intrinsic limitations of static analysis.
[44] The information presented in this table does not include the exception paths that originated to libraries to which we did not have access to the source code.
[45] We believe that the same behaviour will happen in other AO systems, since the set of target systems studied englobes the different ways of handling exceptions using aspects as described in Chapter 3. However further studies still need to be performed to assure this idea.

presents the `after throwing` advice code, and Listing 27 illustrates the effect of this advice on the woven *bytecode.*

```
after(String mediaFile) throwing(Exception e) throws
  ImagePathNotValidException:readMediaAsByteArray(mediaFile){
        throw new ImagePathNotValidException(
               "Path not valid for this image:"+mediaFile);
}
```

Listing 26. Code snippet for an `after throwing` advice.

```
1. public void method() throws Exception{
2.   try{
3.    //original method body
4.     ...
5. } catch(Exception t){
6.     <AspectID>.aspectOf().ajc$afterThrowing$<Id>(…);
7.      throw t;
8. }
```

Listing 27. Effect of the `after throwing` advice on the woven bytecode.

We can observe that the exception path that starts in line 7 of the code snippet presented in Listing 27, can never happen in runtime because the `after throwing` advice unconditionally[46] throws an instance of `ImagePathNotValidException`, and consequently, the statement defined in line 7 will never be executed. We believe that detecting statements that *unconditionally throws exceptions* during the exception analysis would improve the tool precision in such scenarios.

The GUI layer of AJHotdraw system is based on the definition of a set of GUI components that inherits from the same abstract class and overrides some of its methods. Listing 28 illustrates *exception paths* that include method calls defined in GUI components. Some exception paths share the same method name (i.e., `mouseDown()`) – represented in bold in the listing below.

```
org.jhotdraw.standard.AbstractFigure.clone()
org.jhotdraw.contrib.GraphicalCompositeFigure.clone()
org.jhotdraw.standard.CreationTool.createFigure()
org.jhotdraw.standard.CreationTool.mouseDown()
org.jhotdraw.figures.TextTool.mouseDown()
```

```
        org.jhotdraw.figures.ConnectedTextTool.mouseDown()
        org.jhotdraw.util.UndoableTool.mouseDown()
        org.jhotdraw.standard.SelectionTool.mouseDown()
        org.jhotdraw.samples.javadraw.MySelectionTool.mouseDown()
        org.jhotdraw.standard.StandardDrawingView$DrawingViewMouseListener.
mousePressed()
```

Listing 28. List of method calls composing an exception path.

The partial code of `CreationTool.mouseDown()` is illustrated below. We can observe that this method does not directly calls the `TextTool.mouseDown()` – there is no instance of `TextTool` class in the scope of the method. This method only calls the method `mouseDown()` defined the superclass which is the same superclass of `TextTool`.

```
//Creates a new figure by cloning the prototype.
public void mouseDown(MouseEvent e, int x, int y) {
      super.mouseDown(e, x, y);
      setCreatedFigure(createFigure());
      ...
  }
```

Listing 29. Code snippet of `CreationTool.mouseDown()` method.

These spurious paths were, therefore, caused by the imprecision of the underlying call graph builder module that uses the context-insensitive points-to algorithm available in SPARK framework. *S*tudies have reported that the use of *Data-Reachability* and *context-sensitive* analysis algorithms can improve the precision of the call graph and consequently of the *exception paths* calculated when traversing it (Liang et al., 2005; Fu and Ryder, 2007). Such algorithms would check the values stored in program values, to decide, for instance, which methods could be called, when a method defined in a supertype is called.

6.6.Discussions and Lessons Learned

This section provides further discussion of issues and lessons we have learned while reusing aspect libraries and adopting our reuse approach in different scenarios.

[46] There is no if clause guarding the exception thrown by the after throwing advice.

Static analysis x Testing Exceptional Conditions. Our approach relied 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 library aspects 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 with the simulation of most of such scenarios (Bruntink et al., 2006).

Using Stubs. Before performing the static analysis, the SAFE tool requires the aspect library to be combined with a base code. Since aspects are intrinsically related to the concerns they affect. The same restriction is present in automated testing approaches for aspects (Lopes and Ngo, 2005; Xie and Zhao, 2006). Thus, if the developer wants to apply this approach before the base code is implemented, "stub implementations" should be provided to every advised method.

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 combining the *library aspects* and the application code at compilation time and then adopting the approach proposed here.

Documentation of Aspect Libraries. Current aspect libraries neither explicitly document which aspects will affect the base code, nor the exceptions that may flow from library advices that affects the base code. As we have discussed in this thesis, such information is very useful when developing robust systems: the developer wants to make sure that a piece of third party code will not threaten the robustness of the application (exceptional scenarios). The SAFE tool can also be used to automatically generate the exceptional interfaces of aspect libraries helping to document them.

Aspect-library Development. The case study presented in this chapter 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 *application aspects* and *aspect libraries*. Using a similar approach the aspect library developer could isolate the base code from exceptions that may flow from library code.

Static Analysis based on Java bytecode. The SAFE tool is based on the static analysis of Java *bytecode*. Then advantage of working on 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. On the other hand the SAFE tool output that the results of the bytecode static analysis, is not , such automatically generated method signatures are presented to SAFE users. 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.

Dependence on the User Input. The user needs to specify a set of information to be used by the SAFE tool during the analysis: (i) the application packages (see Chapter 5, Section 5.3.2.1), and the (ii) exception handling contracts. The SAFE tool relies on this information to classify the exception paths according to its *Signaler-Handler* relationship, and find out the broken *exception handling contracts* respectively. If the user fails to specify a relevant application package, SAFE may report a wrong *exception path* classification and consequently wrong *Signaler-Handler* relationships. If the user fails to define the exception handling contracts (e.g., specify a wrong contract), the tool may report a wrong broken contract. In cases where the user forgets to specify an exception handling contract, the contract violation will not be automatically reported by the tool, but can be eventually discovered during manual inspection of the exception handling code.

6.7.Summary

This chapter presents an approach, supported by SAFE tool, which aims at assisting the developer when checking the reliability of the exception handling code of AO applications. Such applications may contain *application aspects* and/or *library aspects* – a new reuse artifact available nowadays. This approach supports the reasoning about the exceptions that can flow from aspects; and provides brief guidelines to the developer of how such exceptions should be handled.

The limitations of this approach are strictly related with the limitations of the static analysis tool that supports it. During the implementation of SAFE we had to deal with a set of trade offs that involved the tool precision, the processing cost and the usefulness of the information provided by it. Therefore, due to some implementation decisions the SAFE tool may report *false broken exception handling contracts*.

However, the number of false broken contracts found in the case study presented in this chapter was sufficiently low assuring that the SAFE tool is in fact useful in practice. Moreover, the false broken contracts can be are easily identified during the manual inspections of the exception handling code guided by the *exception paths* reported by the SAFE tool (one of the approach's steps).

The analysis of the broken contracts, revealed the reasons why they occur and identified where we can improve the tool. The exception-flow analysis implemented by SAFE can be improved by using contextual information (Liang et al., 2005) or *Data-Reachability* algorithms (Fu and Ryder, 2007). Both would reduce the number of false broken contracts reported by the analysis.