4
The Empirical Study Results

When manually inspecting the code of each target system we found out three categories of bug-patterns related to the exception handling code of AO systems: (i) *aspects as handlers*; (ii) *aspects as signalers*; and (iii) *declare soft construct* – a construct only available in AspectJ language. These three categories are in a certain degree related, for instance: an *aspect signals* an exception that should be handled by a *handler aspect* which does not catch the exception, due to the wrong use of `declare soft` construct. However, we explain each bug pattern in separate, to focus on the problems that can be the cause of each one.

This chapter details the faults associated with bug pattern discovered in our work and presents further discussions and lessons learned concerning the empirical study as a whole.

4.1.
Bug Patterns in the Exception Handling Code of AO Systems

Bug patterns are recurring correlations between signaled errors and underlying bugs in a program (Allen, 2002). They are related to design anti-patterns, but bug patterns are directly correlated with faults at source code level. The inspection of exception paths allowed us to identify several exception-handling bug patterns that can be classified into three categories. First, the use of *aspects as handlers* led to some scenarios in which the `catch` clauses were moved to aspects, the so-called exception handling aspects or handler aspects. However, these aspects did not catch the exceptions they were intended to handle. Second, the application of *aspects as signalers* often implied aspects signaling exceptions to which no handler was defined. Such exceptions flew through the system and became uncaught exceptions or were caught by an existing handler in the code (usually by *subsumption*). Third, the use of `declare soft` construct was often problematic: due to its complex semantics, almost all developers performed similar mistakes when using this construct in almost all the analyzed releases.

Table 7 summarizes the bug pattern distribution in relation to the analyzed systems. Next sections describe the bug patterns shown in this table. For each of them, we provide a description and code examples extracted from target systems.

	Health Watcher AO		Mobile Photo AO		JHotDraw AO
Bug patterns	**V1**	**V9**	**V4**	**V6**	**V1**
Aspects as Handlers					
Inactive Handler Aspect	✓	✓	✓	✓	
Late Binding Handler Aspect		✓			
Obsolete Handler in the Base Code.					✓
Aspects as Signalers					
Solo Signaler Aspect	✓	✓			✓
Unstable Exception Interfaces.	✓	✓	✓	✓	
Exception Softening					
Handler Mismatch.	✓				
Solo Declare Soft Statement.			✓	✓	
Unchecked Exception Cause					✓
The Precedence Dilemma.			✓	✓	

Table 7. Distribution of the bug patterns per system.

4.1.1. Aspects as Exception Handlers

The role of aspects as handlers can be classified into two: (1) the aspect can handle its own internal exceptions; and (2) and it can handle external exceptions thrown by other aspects or classes. Aspects can be used to modularize the handlers of external exceptions relative to other crosscutting concerns implemented as aspects. The latter occurred in both Health Watcher and Mobile Photo systems. It can also be used to modularize part of exception handling from the base code (as in Mobile Photo). Such exception handling aspects are implemented using `around` and `after throwing` advice. The first two bug patterns presented next are related to aspects that act as external exception handlers, the last one is related to aspects as internal handlers.

Inactive Handler Aspect. This kind of fault happens when an handler aspect does not handle the exception that it is intended to handle. The cause is a fault on the pointcut expression. Such fault prevents the handler of advising the join point in which an exception should be handled. This exception either becomes *uncaught* or is mistakenly caught by an existing handler (*unintended handler action*). This bug pattern was only detected in Health Watcher and Mobile Photo systems, since the exception handling concern was not *aspectized* in JHotDraw. The fragility of the pointcut language, and the number of different and very specific join points to

be intercepted by the handlers aspects leads to such bug pattern. The code snippet extracted from Mobile Photo illustrates this problem:

```
aspect UtilAspectEH{
    // the pointcut was
    pointcut readImageAsByteArray(String imageFile):
    (call(public void Class.getResourceAsStream(String))
    &&(args(imageFile)));

    // the pointcut should be
    pointcut readImageAsByteArray(String imageFile):
    call(public java.io.InputStream Class.
    getResourceAsStream(String))&&(args(imageFile)));
    ...
}
```

Late Binding (or Starved) Handler Aspect. This bug pattern occurred in the 9th release of Health Watcher AO version. The concurrency control concern was implemented as an aspect, which could possibly throw an instance of `TransactionException`. A specific aspect, called `HWTransactionExceptionHandler`, was defined to handle this exception (see Figure 4), and although the *pointcut* expression was correctly specified, the handler intercepted a point in which the exception was caught beforehand by a "*catch all clause*" in the base code as illustrated in Figure 14. We can observe from this example that the exception did not reach the correct handler due to a catch clause (present in the mB method) which captures the exception occurrence in the base code. This problem is difficult to diagnose, the compiler will give no warning to the developer since the `HWTransactionExceptionHandler` aspect intercepts the correct join points in the code (where the exception should be caught). This explains why this fault remained until version V9 of the HW system. Moreover, even if there is no "catch all" clause between signaler and handler aspects during development, such clause may be added during a maintenance task [14].

[14] It the handler was defined in the base code, and if `TransactionException` was a checked exception the compiler would warn the developer that the handler on the base code was inactive.

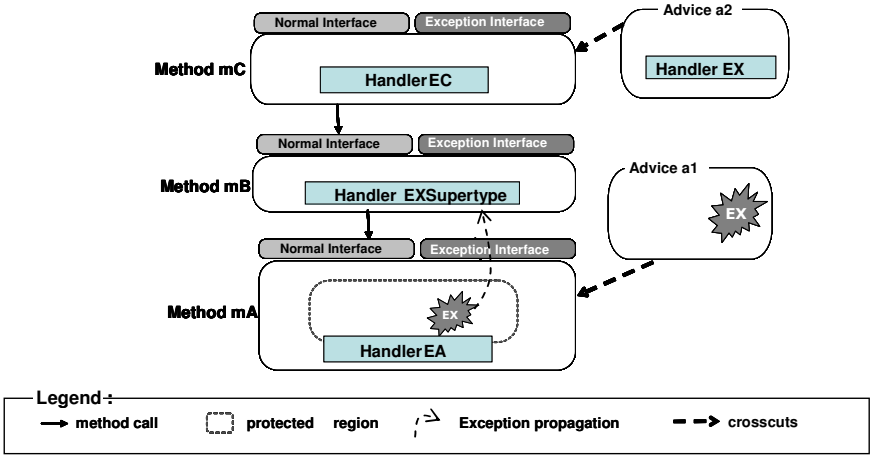

Figure 14. Schematic view of the Late Binding Handler.

Obsolete Handler in the Base Code. When an aspect handles or softens an exception *e,* thrown by an application method, if there was a handler previously defined for it on the base code, such handler will become *obsolete* - since *e* can no longer be reach it. In this study four exceptions handled by aspects generated obsolete handlers on the base code. The code snippet below was extracted from the JHotDraw AO version:

```
 private void readFromStorableInput(String filename) {
     try {
        …
        String fDrawing = input.readStorable();
        …
     }
     catch (IOException e) {
        initDrawing();
        showStatus("Error:" + e);
     }
     catch (org.aspectj.lang.SoftException e) {
        showStatus("Error: " + e.getWrappedThrowable());
     }
   }

public aspect IOExceptionHandling{
     declare soft: IOException : call (String StorableInput.
         readStorable() throws IOException);
      ...
 }
```

In this scenario the method `readStorable` signals `IOException`, but this exception is softened by `IOExceptionHandling` aspect. As explained previously, when an exception is softened, the AspectJ weaver handles it and throws an instance of `SoftException` which wraps the original exception. As a consequence the previous handler defined to `IOException` on the base code is no longer used. Since it was not removed from the base code and it became an obsolete handler.

Notice that an obsolete handler may lead to *unintended handler actions* after a maintenance task. A handler that was obsolete can mistakenly handle a new exception that flows from the protected code after a maintenance task. Such handler may present a wrong error message to the user or, in some cases, silence the exception (i.e., does not report any information that the exception was handler) – which is one of the most difficult bugs to diagnose (Miller and Tripathi, 1997).

4.1.2. Aspects as Exception Signalers

During the manual inspections we found potential faults that can occur when aspects signal exceptions. They are detailed below.

Solo Signaler Aspect. *Solo Signalers* are the aspects that signal an exception and no handler is bound to it. Such an aspect may lead to the same failures caused by the Inactive Handler Aspect defined in the previous section: an *uncaught exception* or an *Unintended Handler Action*. The *Unintended Handler Action* (Miller and Tripathi, 1997) is usually characterized by the exception signaled by an aspect being handled by *subsumption* via classes. The code snippet below was extracted from Health Watcher system, and illustrates the code of an aspect that handles an exception, but while handling the exception it invokes the `out.close()` method which may throw an `IOException`, which will become uncaught.

```
public aspect HWTransactionExceptionHandler {

   void around(HttpServletResponse response) :
    execution(* HWServlet+.do*(HttpServletRequest,
          HttpServletResponse)) && args(.., response) {

       try {
```

```
            proceed(response);
          } catch (SoftException e) {
                …
              PrintWriter out = response.getWriter();
                 out.println(HTMLCode.errorPage(…));
              out.close();
              …
          }
      }
    }
```

Unstable Exception Interface. In this study we have observed that aspects had the ability of *destabilizing the exception interface* of the advised methods. Every time a static or dynamic scope is used and the advice may signal an exception, the exception interface of the method will vary according to the scope in which a method is called. As a consequence, the same method could raise a different set of exceptions, even when the method arguments were the same, depending on the static (e.g. which class called it) or dynamic (information on the execution stack) scopes. The next code snippet, extracted from AJHotDraw, exemplifies an *unstable exception interface.*

```
pointcut commandExecuteCheckView(AbstractCommand command):
this(command)
   && execution(void AbstractCommand+.execute())
   && !within(*..DrawApplication.*)
  && !within(*..CTXWindowMenu.*)
  && !within(*..WindowMenu.*) && !within(*..JavaDrawApp.*);

 before(AbstractCommand command) :
              commandExecuteCheckView(command) {
    if (command.view() == null) {
       throw new JHotDrawRuntimeException("execute should NOT
                   be called when view() == null");
    }
  }
```

In this case, the `execute()` method will throw an instance of `JHotDrawRuntimeException` depending on the static scope where it is executed. This exception was not handled in any of the contexts and became uncaught. Most of the exceptions that were thrown by methods presenting an *unstable exception interface* were not adequately handled.

4.1.3. Softening Exceptions

In AspectJ an advice can only throw a checked exception if "every" intercepted method can signal it (i.e. declaring it on its `throws` clause). In other words, concerning checked exceptions, an advice should follow a rule similar to the "*Exception Conformance Rule*" (Matsuoka and Yonezawa, 1993; Miller and Tripathi, 1997) applied during inheritance, when methods are overridden. As a result an advice can only throw a checked exception if it is thrown by every intercepted method.

To bypass this restriction, AspectJ offers the `declare soft` statement, which converts (wraps) a given checked exception (in a specific scope) into a specialized unchecked exception, named `SoftException`. The syntax is: `declare soft : <someException> : <scope>`. The `scope` is specified by a pointcut that selects a subset of join points in which the `someException` exception will be wrapped. AspectJ is the only AO language that provides a `declare soft` construct. As detailed in Section 4.2.1, in Spring AOP and JBoss AOP, advices are allowed to throw any kind of exception, either checked or unchecked. It is possible because during the weaving process the weaver converts the exception interface of every advised method to allow every kind of exception to flow from it – including a `Throwable` in its `throws` clause. This section presets some bug patterns and also potential error-prone scenarios on the exception handling code when `declare soft` statement is used.

Solo Declare Soft Statement. According to the AspectJ documentation (Colyer, 2004), every time an exception is softened by an aspect, the developer should implement another aspect that will be responsible for handling the softened exception. However, this solution is very fragile, since it is up to the programmer to define a new aspect to handle the exception that was softened, and no message is shown at compile time to warn the programmer in case s/he forgets to define this *handler aspect*. In the Health Watcher and Mobile Photo systems, exceptions were softened and no handler was defined for them. This led to uncaught exceptions and *unintended handler actions* - exceptions caught by *subsumption* on the base code. In the code snippet presented in previous section to *Solo Signaler Aspect* bug pattern the `IOException` signaled inside

`HWTransactionExceptionHandler` was softened and no handler was defined for it.

```
public aspect HWTransactionExceptionHandler {
   // Makes soft all IO exceptions raised in this aspect
   declare soft : IOException :
   within(HWTransactionExceptionHandler+);
      void around(HttpServletResponse response) :
}
```

Unchecked Exception Cause. When a checked exception is softened, it is wrapped in a `SoftException` object. As mentioned before, in Java-like languages the type of an exception is used to make the binding between an exception and its handler. Thus, when wrapping an exception, we are also wrapping useful information in order to provide a fine-grained action for each exception. To overcome this limitation, every place that needs to handle a softened exception, catches the `SoftException` and unwraps it (through its `getCause()` method) in order to compare its cause with every possible exception that has potentially thrown inside the handler's context. Such "wrapping" solution is documented as one of the exception handling anti-patterns (McCune, 2007).

Handler Mismatch. In a version of the Health Watcher system, some exceptions were softened. However handlers were defined for the exceptions primitive types (i.e. types before being wrapped in a `SoftException`). Due to this *Handler Mismatch* almost all exceptions signaled by the crosscutting concerns became uncaught or were caught by unintended handlers. The code snippet below was extracted from Health Watcher and illustrates this problem. The `HWTransactionManagement` aspect softens the exception, and the `HWTransactionExceptionHandler` aspect tries to capture the primitive exception. This bug pattern illustrates an emergent property of a particular combination of aspects woven into the base program.

```
public aspect HWTransactionManagement {
    ...
    declare soft: TransactionException:
        call(void IPersistenceMechanism.beginTransaction())…;
}
```

```
public aspect HWTransactionExceptionHandler {
     void around(HttpServletResponse response) :
          execution(* HWServlet+.doGet(HttpServletRequest,
       HttpServletResponse)) && args(.., response) {
          try {
             proceed(response);
          }catch (TransactionException e)
     }
}
```

The Precedence Dilemma. This bug pattern occurs when an `after throwing` advice is used in combination with a `declare soft` statement referring the same pointcut. Only the code related to the `declare soft` construct is included on the bytecode. Since both constructions work converting one exception into another the weaver cannot decide which one should happen first and as a consequence includes on the bytecode only the code relative to the `declare soft` construct. This bug in the language design generates a `SoftException` exception that will not be adequately caught. The code snippet below was extracted from Mobile Photo and illustrates a scenario in which this problem will occur.

```
pointcut addImageData():
       execution(public void ImageAccessor.addImageData(String,
       String, String));

declare soft: RecordStoreException : addImageData();

after()throwing(RecordStoreException e)
       throws  PersistenceMechanismException: addImageData(){
       throw new  PersistenceMechanismException();
}
```

4.2. Discussions and Study Constraints

This section provides further discussion of issues and lessons learned while performing the empirical study described in this chapter and in Chapter 3.

4.2.1. Representativeness

A first question to be asked is to what extent our findings can be applied to other systems implemented in other AOP languages. We have investigated other AOP technologies such as: CaesarJ[15] (Mezini and Ostermann, 2003), JBoss AOP [16] (Burke and Brock, 2003) and Spring AOP[17] (Johnson et al., 2005; Johnson, 2007). Basically, they follow the same join point model as AspectJ, which allows an aspect to add or modify behavior on join points, potentially adding new exceptions.

	declare soft	advice can signal exceptions		advice types that can act as exception handlers		
		checked	unchecked	After throwing	after	around
AspectJ	yes	partially	yes	yes	yes	yes
CaesarJ	no	partially	yes	yes	yes	yes
JBoss AOP	no	yes	yes	yes	yes	yes
Spring AOP	no	yes	yes	yes	yes	Yes
* we considered only the ones that can handle exception from the base code						

Table 8. AO languages characteristics: advice types and exceptions that may be thrown from advice.

Table 8 summarizes our analysis regarding exception throwing and handling mechanisms available in such technologies. This analysis was mainly based on available documentation, and the development of "toy programs" in each AO language.

According to Table 8 only AspectJ provides a syntactic element to explicitly soften checked exceptions (column 1). Thus, the bug patterns related to this construct (Section 4.1.3) are peculiar to AspectJ. Concerning the nature of exceptions that may be thrown by advice, all languages allow advice to throw unchecked exceptions (column 3). In AspectJ and CaesarJ, an advice can only throw a checked exception if "every" intercepted method can throw it (declaring it on its throws clause) (column 2). In CaesarJ, only an `around` advice signature may throw checked exceptions. In Spring AOP and JBoss AOP languages, advices may throw checked exceptions, no matter the exceptions that can be

[15] http://caesarj.org/
[16] http://www.jboss.org/jbossaop
[17] http://www.springframework.org/

signaled by the advised methods[18]. Therefore, all bug patterns associated with *advice as signalers* (Section 4.1.2) may occur in systems developed in such languages. Moreover, all languages allow the definition of aspects that may handle exceptions thrown by another aspect, or elements of the base code (columns 4-6). As a consequence, all bug patterns associated with *advice as handlers* (Section 4.1.1) can also be found on systems developed in these languages.

Table 9 below summarizes the characteristics of the *pointcut* designators available in each language.

	Pointcut Designators						
	Method-related		Scope				Exception-handling
			Statically		Dynamically		
	call	execution	within-like	withincode-like	cflow-like	cflowbelow-like	handler-like
AspectJ	yes	yes	yes	yes	yes	yes	yes
CaesarJ	yes	yes	yes	yes	yes	yes	yes
JBoss AOP	yes	yes	yes	yes	yes	yes	no
Spring AOP	no	yes	yes	no	no	no	no

Table 9. Differences on available pointcut designators.

We can observe that the *pointcut* designators are very similar in almost all languages. All languages allow the definition of *pointcut* designators that delimits the static scopes (e.g., packages) in which new behavior will be added by aspects (column 3). The Spring AOP language, is the only solution from the analyzed set that does not allow the definition of dynamic scopes in a *pointcut* expression (column 4-5). Moreover *pointcuts* in Spring AOP can only intercept the method execution (column 1). The other languages allow a *pointcut* expression to intercept method calls as well as executions. The characteristics of the pointcut designators available in each language enable occurrence of *unstable exceptional interfaces* problem described in the next section.

[18] It is possible because the exception interface of every advised method is modified to allow any kind of exception to flow from it (`throws Throwable` defines the exception interface of the intercepted methods)

4.2.2. Exception Handling vs. AOP Properties

The goal of exception handling mechanisms is to make programs more reliable and robust. However, we could observe that some properties of AOP may conflict with characteristics of exception mechanisms. In this study we observed that *quantification* and *obliviousness* properties pose specific pitfalls to the design of exception handling code. Following, we explain and discuss these pitfalls.

Quantification Property. Aspects have the ability to perform modifications at specific join points in the program execution where a property holds – an ability also known as *quantification property* (Filman and Friedman, 2005). AspectJ supports quantification via *pointcuts* and advice. *Pointcuts* can for instance intercept the call and execution of methods, through call and execution *pointcut* designators respectively (see Chapter 2, Section 2.1.4.1). On exception-aware systems, such *pointcut* designators may cause different impact in the exceptional interfaces of methods. While the `execution` *pointcut* affected the exceptional interface of the advised methods themselves, the call *pointcut* affected the exceptional interface of the advised method's caller. Such impact can also be influenced by static and dynamic scopes associated with the *pointcuts*. Static scopes such as, `within` and `withincode` delimit the classes or packages on which the aspects will inject a new behavior. Dynamic scope constructs (i.e., `cflow` and `cflowbelow`) allow an aspect to effect (or not) a specific point in the code depending on the information available on the runtime execution stack.

The main consequence of the quantification property in exception-aware AO systems was that the *exception interfaces* of methods can vary depending on where the method was called, even when the method arguments are the same. Therefore, the same method of a class could raise a different set of exceptions depending on which object called it or on some information on the execution stack (in case of `cflow` and `cflowbelow`, for example).

The *unstable exception interface* cannot happen in OO programs since the set of exceptions thrown by a method cannot vary according to the scope where it is executed – regarding the arguments are the same. We observed in our study that in scenarios in which methods presented such *unstable exceptional interfaces*, the exceptions signaled on specific scopes by the advised methods often became

uncaught or were erroneously handled by an existing handler within the base code – characterizing the *unintended handler action*. A possible reason is that it is more difficult for the method's user to prepare the base code to handle the exceptions that will be thrown depending on the dynamic or static scope it is executed.

Obliviousness property. The *obliviousness property* (Filman and Friedman, 2005), which was believed to be a fundamental property for aspect oriented programming, states that programmers of the base code do not need to be aware of the aspects which will affect it. It means that programmers do not need to prepare the base code to be affected by the aspects. However, since there are no mechanisms to protect the base code from the exceptions that will flow from aspects, a new exception signaled by the aspect may flow through the system, if no handler is defined to it. This exception may become uncaught and terminate the system in an unpredictable way. Even in cases when a handler aspect is defined for each aspect that can throw an exception (as implemented in the AO versions of Health Watcher), there is no guarantee that the exception thrown by an aspect will be handled by the handler aspect defined for it - such exceptions may be prematurely caught by a handler in the base code, as illustrated on the bug pattern *Late Binding Handler Aspect* (Section 4.1.1). Moreover, AspectJ and other existing AO languages allow the modifications caused by aspects to happen dynamically. Although this mechanism opens a new realm of possibilities in software development, it hinders the task of preparing the base code of the exceptions that can be thrown from aspects. During system execution, it is difficult to anticipate whether any unintended handler action or uncaught exception will be caused by the aspects.

4.2.3. Additional Lessons Learned

AO Refactoring Strategies in Exception-Aware Systems. Many AO systems nowadays are generated from an OO version in which some crosscutting concerns are detected and *Aspect Oriented Refactoring* techniques are used to convert some crosscutting concerns into aspects. Such *AO Refactoring* techniques aim at preserving the behavior while refactoring crosscutting concerns to aspects. However, we have observed in our study that the AO refactoring approaches

adopted in the target systems did not preserve the exceptional behavior of the crosscutting concerns in some situations. The catalogue of bug patterns presented in this study can be used by such *refactoring* approaches to prevent some bugs when *refactoring* crosscutting concerns to aspects.

Software Maintainability. It is difficult to define at the beginning of a project which exceptions should be dealt inside the system (Robillard and Murphy, 2000), the exception handling code is often modified along the system development and maintenance tasks. As a consequence, some bugs avoided during AO refactoring, such as the *Late Binding Handler Aspect* (Section 4.1.1), may accidentally be included during a maintenance task - breaking an existing exception handling policy. The exception handling policy comprises a set of design rules that defines the system elements responsible for signaling, handling and re-throwing the exceptions; and the system dependability relies on the obedience of such rules. Reasoning about the exceptional path, looking for potential-faults on the exception handling code, can quickly become unfeasible if carried out manually – due to the complexity and the huge number of exception paths to be followed. For this reason we need tools to help developers (i) understanding the impact of aspect weaving on the existing exception handling policy, and (ii) finding bugs in the exceptional handling code during maintenance tasks.

Finding Bugs in Exception Handling Code of AO Programs. Testing exception handling code is inherently difficult (Bruntink et al., 2006), due to the huge number of possible exceptional conditions to simulate in a system and difficulty associated with simulating most of such scenarios. Hence, a valuable strategy for finding faults in the exception handling code can be to *statically* look for them. The exception flow analysis tool developed in our work used to detect *uncaught exceptions*, could be extended in order to include some of the bug patterns described in this work. A similar strategy was adopted by Bruntink et al. (2006) to find faults on idiom based exception handling code.

***New Interactions between Aspects and Classes*.** The works presented so far on the interactions between aspects and classes, focus on the normal control flow and on information extracted from data-flow analysis. In this study we could observe that new kinds of interaction, between aspects and classes, emerged from the exceptional scenarios (e.g., one class catches one exception thrown by an

aspect). Such *Signaler-Handler* relationships between the elements of an AO system can be used as a coupling metric that exists between these elements on exceptional scenarios. We are currently refining the categorization of the *Signaler-Handler* relationships derived from this study.

4.3. Summary

In this chapter we presented the results of our empirical study. A set of bug patterns related to the exception handling code of AspectJ systems is presented. They are presented in three categories: bugs related to aspects that act as exception signalers, bugs on aspects that acts as exception handlers, and bugs on an AspectJ specific construct associated with the exception handling code (i.e. `declare soft` construct). This chapter also presented to which extent the bug patterns presented here can be generalized to other AO languages: Spring AOP, JBoss AOP and CaesarJ. Moreover, lessons learned during the empirical study were discussed in detail.