Exception Handling Bug Patterns in Aspect Oriented Programs

Roberta Coelho[1,3] Awais Rashid[2] Uira Kulesza[3]
Arndt von Staa[1] Carlos Lucena[1] James Noble[4]

[1]Informatics Department
Pontifical Catholic University of Rio de Janeiro
Brazil
{roberta, arndt, lucena}@inf.puc-rio.br

[2] Computing Department, Lancaster University, Lancaster, UK and Ecole des Mines de Nantes, France
awais@comp.lancs.ac.uk

[3] DIMAp
Federal University of Rio Grande do Norte
uira@dimap.ufrn.br

[4] Victoria University of Wellington, New Zealand
kjx@mcs.vuw.ac.nz

Abstract

Aspects often impact the exceptional control flow of a program by *signaling* and *handling* exceptions signaled by other aspects or classes. The exceptions signaled by aspects may flow through the program execution in unexpected ways leading to failures such as uncaught exceptions and exceptions being caught by the wrong handlers. We identified a set of *bug patterns* via an empirical study of exception handling code in AO systems. These patterns are presented here in the form of a *bug patterns catalogue* containing bugs where aspects act as *exception handlers*, and bugs where aspects act as *exception signalers*.

Keywords Aspect-oriented programming, exception handling, bug patterns, dependable systems

1. Introduction

The term bug has often been used in computer science as a synonym for fault, "*a specific construction in the program code that may lead to a failure*". According to [1] it can also be used as a synonym for code bad smell[1], "*a piece of code that does not represent a fault by itself but that contributes to a difficult understanding of the code, and as a consequence to the introduction of faults*". It has been empirically observed that, due to the predictability of people's fallibility, many *bugs* often fall into known *categories* (or patterns) [2] - as people tend to the repeat similar mistakes. B*ug patterns* are, therefore, recurring characteristics of program code that *may* lead to failures.

Some *bug patterns* have been proposed so far to support the testing and debugging of OO programs [3, 4, 5]. As good software design skills involve knowledge of architectural and design patterns good debugging skills involve knowledge of bug patterns. Since many bugs follow one of several patterns. once a developer can recognize these patterns, s/he will be able to diagnose the cause of a bug and correct it more quickly, as well as learning to avoid them.

[1] The use of this term as a synonym for *code bed smell* is adopted by the *bug patterns* community [1], since they aim at documenting not just the pieces of code that represent a fault by themselves, but also code constructions (code bed smells) that may lead to spaghetti code and, as a consequence, to the introduction of faults.

Since the last decade, aspect-oriented programming (AOP) [11] has been increasingly used as a means to modularise crosscutting concerns, such as persistence, distribution [15], security and monitoring. A number of industrial-strength aspect-oriented programming frameworks have been deployed (e.g., AspectJ [6], JBoss [7] and Spring [8]) and non-trivial applications of AO industrial applications have been developed such as IBM Websphere [9] and GlassBox [10].

On one hand, the AO constructs open a new realm of design possibilities: on the other hand, the new AO constructs represent new *sources of bugs*. There has been little work on cataloging bug patterns in AO programs. Zhang and Zhao [12] detailed a list of general bug patterns associated with the main AspectJ constructs. These bugs, however, focus on the "normal" control flow of programs, and do not consider potential problems related to the exception handling code in AO programs.

In a previous empirical study [13], we assessed the error proneness of exceptional control flow in AspectJ programs, and we observed a set of recurring bugs on the exception handling code. The analysis was based on the manual inspection of three medium-sized systems from different application domains (both in Java and AspectJ versions). For two systems, more than one release were examined. Overall, this corresponds to 10 system releases, 41.1 KLOC of Java source code of which around 4.1 KLOC are dedicated to exception handling, and 39 KLOC lines of AspectJ source code, of which around 3.2 KLOC are dedicated to exception handling.

This paper details the *recurring bugs* discovered during this study. These bugs are presented as a catalogue of *bug patterns* structured in two different categories: (i) bugs on scenarios where aspects act as *exception handlers*; and (ii) bugs on scenarios where aspects act as *exception signalers*. Figure 1 illustrates the bug patterns discovered in each category.

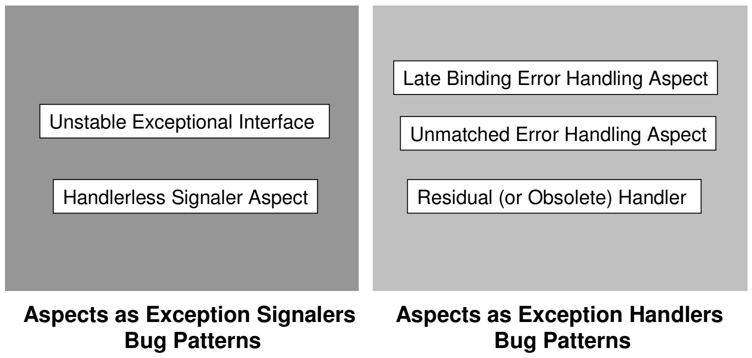
Unstable Exceptional Interface
Handlerless Signaler Aspect
Late Binding Error Handling Aspect
Unmatched Error Handling Aspect
Residual (or Obsolete) Handler
Aspects as Exception Signalers Bug Patterns
Aspects as Exception Handlers Bug Patterns

Figure 1. Bug Patterns categories.

The remainder of this paper is organized as follows. Section 2 presents some background on exception handling in AO programs. Section 3 describes a simple AO system that will be used to exemplify the bug patterns. Finally, Section 4 details each of the bug patterns presented in Figure 1. The bug patterns are structured using the following form (borrowing some terminology from Allen [3]):

- pattern name;
- summary;
- symptoms;
- cause(s);
- cures and prevention; and

- related patterns (when necessary).

Although we present *cures and preventions* for the bug patterns, the focus of this paper is on the bug patterns' *symptoms and causes* – which are useful to support *debugging* and *testing* tasks. Due to some limitations of current AspectJ languages and tool support for reasoning about exceptional flow, some of the proposed solutions act as a *palliative* while better language and tool support are proposed. Therefore, this paper allows developers and testers of aspect-oriented applications to diagnose bugs on the exception handling code, and also designers of AOP languages and static analysis tools to consider pushing the boundaries of existing mechanisms to make AOP more resilient to such bugs. Throughout this article we assume that the reader is familiar with AOSD terminology and AspectJ language constructs. Appendix I presents brief explanation about AOSD terminology.

2. Background

2.1 Exception Handling Mechanism in AO Programs

This section presents the main concepts of exception handling mechanisms, and relates them to the constructs available in AO languages. An exception handling mechanism is comprised of four main concepts: *exceptions*, *exception signaler*s, *exception handler*s, and the *exception model* which defines how signalers and handlers are bound to each other.

Exception Raising. An exception is raised by an element – a method or method-like construct e.g., advice - when an abnormal computation state is detected. Whenever an exception is raised inside an element that cannot handle it, it is signaled to the element's caller. The exception signaler is the element that detects the abnormal state and raises the exception. In Figure 1, the advice a1 detects an abnormal condition and raises the exception EX. Since this advice intercepts the method mA, such exception will be included into method mA together with the additional behavior encapsulated on the advice.

Exception Handling. An exception handler is the code invoked in response to a raised exception. A handler can be attached to protected regions, e.g. methods, classes and blocks of code, or specific exceptions [19]. Exception handlers are responsible for performing the recovery actions necessary to bring the software system back to a normal state and, whenever this is not possible, to log the exception and abort the system in an expected safe way. In AO programs, a handler can be defined in either a method or an advice. Specific types of advice (e.g. around and after advice [20]) have the ability to handle the exceptions thrown by the methods they advise.

Handler Binding. In many languages, the search for the handler to deal with a raised exception occurs along the dynamic invocation chain. This is claimed to increase the software reusability, since the invoker of an operation can handle it in a wider context [21]. In AO programs, the handler of one exception can be present:

(i) in one of the methods in the dynamic call chain of the signaler; or

(ii) in an aspect that advises any of the methods in the signaler's call chain.

Figure 1 depicts one exceptional scenario in which one advice (a1) is responsible for signaling the EX exception, and other advice (a2) is responsible for handling EX, i.e.

a2 intercepts one of the methods in the dynamic call chain and handles this exception.

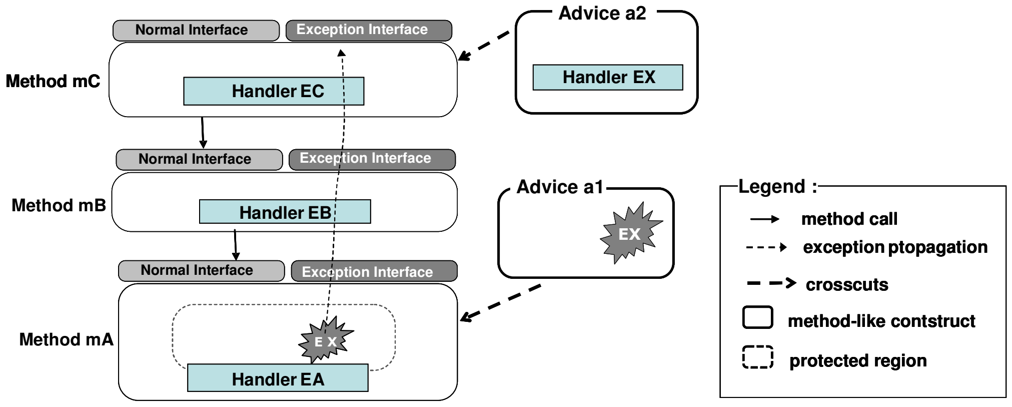

Figure 2. Exception-aware method call chain in AO programs.

In this paper we call *exception paths* the paths in a program call graph that link signalers and handlers of exceptions. Notice that if there is no handler for some exception, an exception path will star from the signaler and finish on the program entrance point. In Figure 1, the exception path of EX is <a1→mA→mB→mC→a2>. Therefore, each *exception path* comprises three main moments: the exception signaling, the exception flow through the elements of a program, and the moment in which the exception is handled or leaves the bounds of the software system without being handled (becoming an uncaught exception).

Exception Interfaces. The caller of a method needs to know which exceptions may be thrown by the called method. In this way, the caller will be able to prepare for any exceptional conditions that may happen during system execution. For this reason, some languages provide constructs to associate to a method's signature, a list of exceptions that this method may throw. Besides providing information for the callers of such method, this information can be checked at compile time to verify whether handlers were defined for each specified exception. This list of exceptions is defined by Miller and Tripathi [22] as the *exception specification or exception interface* of a method. Ideally, the *exception interface* should provide complete and precise information for the method user. 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[2]. 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 thesis we split this concept of exception interface in two categories:

(i) the explicit (de jure) exception interface that is part of the module (method or method like construct) signature and explicitly declares the exceptions; and

[2] In some situations, to list all the exceptions that may escape from a method in the throws clause may become unworkable. Some exceptions, for instance, cannot be adequately handled inside the program (e.g., out of memory exceptions). Forcing the developer to list all of them could lead to unnecessary work during development and maintenance tasks.

(ii) 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 thesis, unless it is explicitly mentioned otherwise, exception interface refers to the complete (de facto) exception interface.

In the rest of this paper, unless it is explicitly stated, we use the expression "exception interface" to refer to a complete (de facto) exception interface. Although both the normal interface (i.e. method signature) and the exception interface of a method can evolve along software life cycle, the impact of such change on the system varies significantly. When a method signature varies, it affects the system locally, i.e. only the method callers are directly affected. On the other hand, the removal or inclusion of new exceptions in an exception interface may impact the system as a whole, since the exception handlers can be anywhere in the code. As depicted in Figure 1, an aspect can add behavior to a method without changing the normal interface of that method. However, the additional behavior may raise new kinds of exceptions, hence impacting the exceptional interface of that method.

Exception Handling Contexts. Exception Handling Contexts (or Protected Regions) are regions in a program where the specific exception types are always treated in the same way [21]. Each region is associated with one or more handlers - from which a handler is chosen when exceptions are raised within that context.

2.2 Target Language: AspectJ

We used AspectJ as our target language to exemplify the *bug patterns* (see Appendix I). Firstly because the systems analyzed in our empirical study, where the *bug patterns* were discovered, were developed in AspectJ. Secondly, because nowadays it is the most used AO language. A first question to be asked is to what extent these patterns can be found in systems implemented in other AOP languages. To answer this question we have investigated other AOP technologies such as: CaesarJ [14], JBoss AOP [7] and Spring AOP [8]. Basically, they follow the same join point model as AspectJ, and as a consequence the bug patterns described next can also be found on systems developed in such languages.

3. Example

This section presents an illustrative example of an information system, called *Health Watcher*. *Health Watcher* is a web-based information system that allows citizens to register complaints regarding issues in health care institutions. Figure 3 illustrates slices of the AO design of this system structured according the Layer architectural pattern [16]. According to this pattern, the elements from each layer should communicate only through well defined layers` interfaces. The purpose of a layer interface is to define the set of available operations – from the perspective of interacting client layers - and to coordinate the layer response to each operation. Several design patterns have been proposed to refine each layer of this architecture. Some of them are: the Facade pattern, the Persistent Data Collections (PDC) pattern [17], and *Error Handling Aspect* pattern [18].

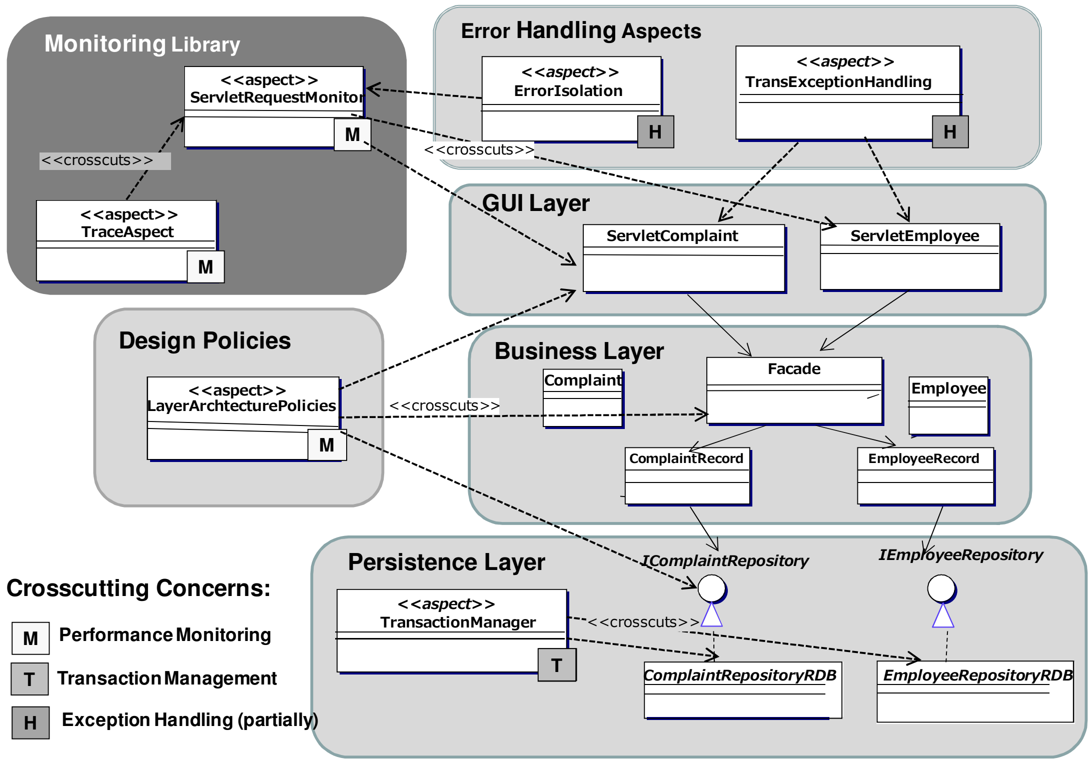

Figure 3. The AO design of *Health Watcher*.

We can observe from the AO design that some concerns are represented as aspects in this system, such as: monitoring, transaction management, and some design policies (e.g., to assure that elements from one layer will not access services from superior layers). Moreover, the exception handling concerns of crosscutting concerns were also represented as aspects - as illustrated by the *Error Handling Aspects* layer in Figure 3. The *Error Handling Aspects* [18] intercept the points in the code where exceptions thrown by the corresponding crosscutting concerns should be dealt.

4. The Catalogue of Bug Patterns

As mentioned before, our catalogue of *bug patterns* is classified in two categories: (i) bugs on scenarios where aspects act as *exception signalers,* and (ii) bugs on scenarios where aspects act as *exception handlers*. This catalogue is a useful source of information for *debugging* and *testing* the exception handling (EH) code of AO systems. As it shows which kinds of bugs are most likely to happen in the EH code, it can help developers and testers to *avoid* and *detect* tem. The list of bug patterns can also be used to implement static checkers that can be used to automatically locate *faults or potential faults* in the source code.

4.1 Aspects as Signalers

When aspects add new behaviors to points in the code, this new behavior may bring new exceptions that will flow through the code. During the manual inspections, we found *recurring bugs* that can occur when aspects signal exceptions. These bug patterns are detailed below.

Unstable Exceptional Interface

The *Unstable Exceptional Interface* bug pattern affects the list of exceptions that can be thrown by a method, and can be the cause of *uncaught exceptions* and *unintended handler actions* in AO systems.

Symptoms

No handler was defined to catch an exception thrown by a method, as a consequence such exceptions: (i) will become *uncaught* – the exception thrown by the application method is not caught inside the system, that it may lead to a software crash; or (ii) will be mistakenly caught by an existing handler (a scenario also known as *unintended handler action*).

Causes

In general, a method may signal a set of exceptions as a consequence of: (i) boundary values of method's parameters (e.g., if we pass *-1* to a method that only works for positive numbers); and (ii) its internal behavior. Usually, such exceptions (that compose the exceptional interface of a method) do not depend on which client directly (or indirectly) invoked the method. In AO systems, however, aspects may modify any method's well-established behavior, and may create a situation where the exceptions that a method throws may depend on which clients are calling it.

These scenarios usually happen when an aspect advice is associated with *pointcut designators* specifically used for scoping purposes (e.g., within, withincode, cflow, cflowbelow). As a consequence, the same method will have different behaviors depending upon how it is called (even if the arguments passed to the method are always the same).

When the list of exceptions that can be thrown by a method varies according to the scope that it is executed, we may say that this method is the owner of an *Unstable Exception Interface*.

As this bug pattern *contributes to a difficult understanding of the exceptional behavior of a method*, and as a consequence to the introduction of faults (e.g., usually exceptions that are only thrown in very specific exceptional scenarios) they remain uncaught or are caught by unintended handlers. Figure 4 presents a schematic view of this *bug pattern*.

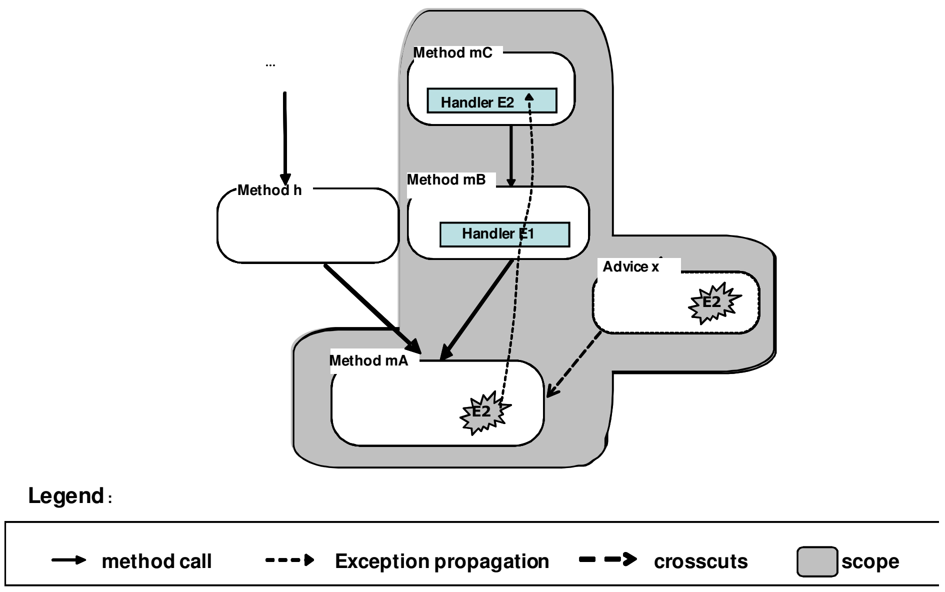

Figure 4. Schematic view of the Unstable Exception Interface.

In this figure, the *Advice x* adds a new functionality to *method mA* only when such method is called from *method mC* (i.e., the pointcut expression contains a dynamic scope delimiter) – this scope is represented in gray in Figure 4. Therefore, this additional functionality, and the new exception E2 that comes with it, will not be part of method mA when it is called from another method such as *method h.* As a consequence, when the *method mA* is called from *method mC*, it may throw E2 exception – and a handler should be defined for it. On the other hand, if it is called from *method h*, it will not throw the exception E2 (even if the method arguments are the same as the one passed on the previous scope) since *advice x* does not affect the *method mA* in this scope.

Code Example

The code snippet bellow illustrates a scenario where this bug pattern can be detected:

```
aspect LayerArchitecturePolicies {

  pointcut designPolicy (Facade fcd): this(fcd)
     && call(void Facade+.*())
      && !within(HttpServlet+.*);

  before(Facade fcd) : designPolicy(fcd) {
      String info = fcd.getCurrentContext();
      throw new DesignViolationException("…"+info);
  }
}
```

In this example, the pointcut expression defined in the `LayerArchitecturePolicies` aspect intercepts the execution of *any method defined on the Facade class, but only when it is* ***not*** *executed within a Servlet.* As a consequence, the advice associated to it only affect and throw a `DesignViolationException` if it is called from a method that is not defined on a Servlet.

In our illustrative example, another aspect (i.e., `TransExceptionHandling`) is calling a method defined on Facade class in order to prepare the error message to be presented to the user. The developer did not know that such method call would violate a design policy (and that, as a consequence, an exception would be thrown). Thus, he/she did not define a handler to the exception thrown in this context and such exception became *uncaught*.

```
aspect TransExceptionHandling {
  …
 void prepareErrorMessage(Exception ex){
     System.out.println("Error on " +
        Facade.getInstace().getApplicationName()"+
        ex.getMessage());
   }

}
```

Cures and Prevention

Detecting this bug pattern involves: (1) finding every advice that uses a *scope-specific* pointcut designator; (2) recursively inspecting such advice (i.e., inspecting every method called from it and every other advice that advises it, with or without tool support); (3) if the advice may throw an exception, inspect the methods in the program call graph that directly or indirectly calls the advised method (advised method) to verify if exception handlers were defined to handle the exception.

The most common way to prevent this bug is for the developer to create a *handler* to catch each exception that is thrown in each situation where the exception will be thrown – such handlers should be included in every method that calls the *intercepted method*. Ideally, a default handler could catch any exception that was not caught by other handlers: unfortunately, this is not possible in the current version of AspectJ.

Alternatively, the developer may be able to replace (dynamic) advice that throws the exception with a (static) `declare error` statement that will generate an error at compile time. For example, aspects that represent design policies should use `declare error`:

```
pointcut designPolicy ()  :
     execution(void Facade+.*())
     && !within(HttpServlet+.*);

declare error : designPolicy():
     "Design Violation Exception: calling Facade";
}
```

Handlerless Signaler Aspect

The *Handlerless Signaler Aspect* bug pattern occurs when an aspect advice signals an exception and no handler is defined to handle it.

Symptoms

An exception that is thrown in the system becomes (i) *uncaught* – the exception thrown by the application method is not caught inside the system, as a consequence it may lead to a software crash; or (ii) is mistakenly caught by an existing handler (a scenario also known as *unintended handler action*).

Causes

This bug occurs when an aspect advice or intertype declaration signals an *unchecked* exception, and no handler is defined to catch it. Even a very simple and naïve aspect (e.g., logging) may call a library that throws an undocumented *unchecked* exception that impacts the execution flow of the application.

Code Example

The code snippet below was extracted from *Health Watcher* system, and it shows an aspect that monitors the status of each HTTP request. This aspect calls an OO library (at logError()) that signals an *unchecked exception* when the log file is too `large`. As a consequence, when this exception is signaled it remains *uncaught*, and causes a software crash.

```
aspect ServletRequestMonitor {

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

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

Cures and Prevention

In languages such as Java that support *unchecked* exceptions, to know which exception may be signaled from a method a developer must recursively inspect every method called by it. Therefore, *preventing* this bug pattern involves: inspecting the code (manually, or using an exception flow analysis tool [13]) and checking if an exception handler was defined to handle the exceptions thrown by an advice. There are two possible ways of handling an exception thrown by an aspect: (i) *app-specific error handling*; or (ii) *error isolation*.

According to the *app-specific error handling* strategy, we can create an *Error Handling Aspect* that intercepts specifics points in the code where the exception thrown by the aspect should be handled.

According to the *error isolation* strategy an Error Handling Aspect is created to intercept the *every aspect* that may signal an exception, or a catch clause is included within every advice that signals the exception. Such aspects or catch clauses will capture and log the exception for off-line analysis so that the main application never sees the exception. One example of error isolation is the GlassBox monitoring aspect library [10]. The developers of GlassBox implemented an *error isolation* solution to prevent exceptions flowing from the monitoring code to affect the monitored application.

The code snippet bellow illustrates a handler aspect that implements the error isolation strategy.

```
1. public aspect ErrorIsolation {
2.   ...
3.   public pointcut scope() :
4.          within(<SignalerAspect>)
            && !within(*..*AroundAdvices);
5.
6.   void around():adviceexecution() &&
7.      scope()){
8.    try {
9.       proceed();
10.    } catch (<Exception> e) {
11.      log(e);
12.    }
13. }
14.}
```

The `adviceexecution` pointcut (line 6) matches join points where an advice is executing. This aspect handles every instance of *`<Exception>`* that may flow from the execution of any advice defined on the `<SignalerAspect>`. This strategy works well for isolating the exceptions that come from before and after advice only. The execution of an around advice may also contain the execution of the advised method (`proceed`). Since there is no way to intercept the execution of around advice, excluding the execution of `proceed`, if we used the same strategy for dealing with exceptions thrown from around advice execution, the exceptions thrown by the client application (calling `proceed`) would be swallowed or erroneously handled inside the aspect – breaking the exception handling policy of the client application. This solution relies on a naming pattern to exclude the exceptions that come from around advice to be swallowed: write static inner aspects whose name matches *AroundAdvices which will include the around advices. Relying on name patterns is a fragile solution, but is a palliative to deal with such situation while AO languages and tools are improved.

Related Patterns

The *Error Handling Aspect* pattern [18] can be used as one of the ways of solving this bug pattern. As a consequence, the *bug patterns Late Binding Error Handling Aspect* and *Unmatched Error Handling Aspect,* related to the use of *Error Handling Aspect,* may be included when solving the bug pattern presented here.

4.2 *Aspects as Handlers*

Aspects can be used to *modularise* the exception handling concern. In such scenarios the catch clauses defined on the base code can be moved to aspects called Error *Handling Aspects* [18], which are implemented using around and after throwing advice. The *bug patterns* presented next are related to the use of the *Error Handling Aspect pattern.*

Late Binding Error Handling Aspect

The *Late Binding Aspect Handler* bug pattern happens when an aspect is created to handle an exception, but the aspect intercepts a point in the program execution where the exception to be caught was already caught by a handler in the *method call chain* that connects the exception signaler to the aspect handler.

Symptoms

When (i) an aspect is defined to handle one exception, (ii) it intercepts the correct point in the base code where the exception should be caught, (iii) but the exception does not reach the handler.

Causes

Although the *pointcut* expression defined on the *Error Handling Aspect* is correctly specified, the handler may intercept a point in the program code in which the exception was caught beforehand by a "*catch clause*" on the base code as illustrated in Figure 5 - this *catch clause* could be defined to handle an exception of the same type of the exception to be caught or any of its supertypes.

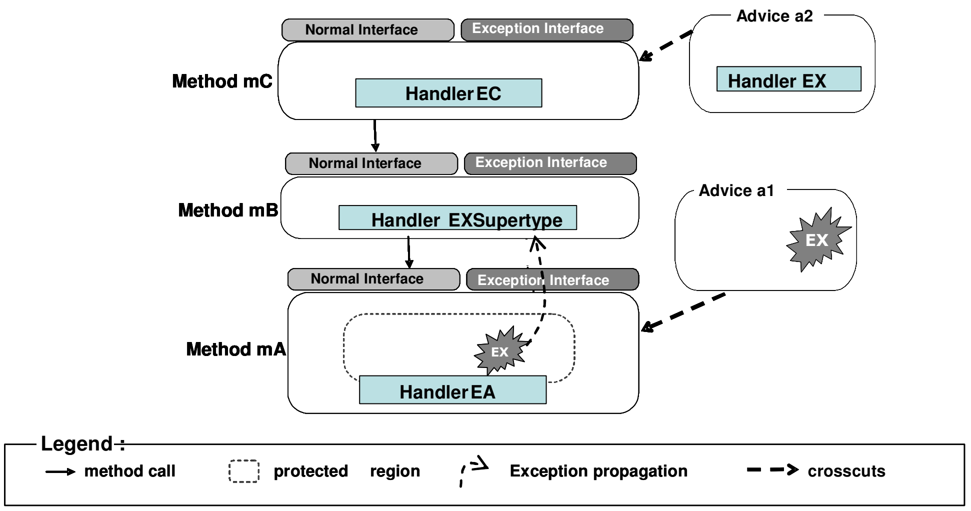

Figure 5. Schematic view of the Late Binding Handler.

In this figure, the *Advice a1* adds new functionality to *Method mA*. This additional functionality throws a new exception *EX*, which flows backwards through the advised method call chain. Another advice was defined to handle the exception (advice a2), which intercepts a point on the base code where the exception EX should be handled (*Method mC*). However, the exception EX was caught by the catch clause inside *Method mB* and as a consequence the exception will not reach the point in the code where it should be handled by advice a2.

We can observe that this problem can also happen in OO development: an exception may be prematurely caught by an existing handler in the base code. But the problem is aggravated in AO systems because base code is supposed to be oblivious of the aspects.

Code Example

In the HealthWatcher example the transaction management concern is implemented as an aspect which thows a `TransactionException` if something goes wrong:

```
aspect TransactionManager {

      public pointcut dataBaseOperations():
           execution(public * *RepositoryRDB(..)))…;

       void around() : dataBaseOperations()
       {   ...
          //manage transactions
         if (status==0) {
           throw new TransactionException(cause);
```

```
            }
          ...
     }
      …

  }
```

A specific aspect, called `TransExceptionHandling`, was defined to handle this exception (see Figure 4),

```
aspect TransExceptionHandling{

      public pointcut servletRequestExec():
           within(HttpServlet+) &&
           (execution(* HttpServlet.do*(..)) ||
           execution(* HttpServlet.service(..)))…;

      void around():servletRequestExec()
      {
           try{
             proceed();
          }catch(TransactionException exc){
              //handle exception
              ...
          }

      }
```

However, the exception thrown by `TransactionManager` did not reached the Error Handling Aspect that intercepted the GUI layer. The exception was caught beforehand by a "*catch all clause*" defined on the Facade class defined on the business layer - as illustrated in Figure 14. This means that exceptions from the Transaction concern are being handled by (and so are visible within) the application program.

```
  public class Facade {
     ...
   public Complaint searchComplaint(String id)
   {
      try{
         ComplaintRepositoryRDB.getInstance().search(id);
       }catch(Exception exc){
          //handle exception
           ...
       }
    }
```

Cures and Prevention

To prevent this bug pattern: (i) avoid "catch all clauses" during development, (ii) replace them (when possible) by specific catch clauses, (iii) create two (or more) exception hierarchies: one for exceptions signaled by the main program, and the other(s) for exceptions signaled by aspects.

Definitely curing this bug pattern in the context of evolving systems is still a challenge to current AO development technologies.

Related Patterns

This bug pattern can be found in scenarios where the Error Handling Aspect Pattern [18] is used.

Unmatched Error Handling Aspect

The *Unmatched Error Handling Aspect* bug pattern happens when an aspect is created to handle an exception but it intercepts the wrong point in the program execution.

Symptoms

An *Error Handling Aspect* is defined to handle an exception but does not handle it. As a consequence the exception will either become (i) *uncaught* – the exception thrown by the application method is not caught inside the system, as a consequence it may lead to a software crash; or (ii) will be mistakenly caught by an existing handler (a scenario also known as *unintended handler action*).

Causes

This kind of bug occurs when an *Error Handling Aspect* does not handle the exception that it is intended to handle, due to a mistake on the pointcut expression. Consequently, the exception will become *uncaught* or will be mistakenly caught by an existing handler (unintended handler action). The fragility of the pointcut language, and the number of different and very specific join points to be intercepted by the handler aspects lead to such bug.

Code Example

The code snippet below illustrates this problem. The `TransactionManager` needs to intercept a specific point in the code where an exception should be caught, but since this join point was very specific the developer made a mistake while defining it.

```
aspect TransactionManager{

   // 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)));

      ...
  }
```

Cures and Prevention

The only way to solve this problem is to correct the mistake in the pointcut expression. This is not a long term solution, since the required pointcut can change in any maintenance task. Currently, AspectJ does not allow a long term solution to this problem.

Related Pattern

- This bug pattern can be found when applying the Error Handling Aspect Pattern [18].
- Although both this bug pattern and the *Late Binding Error Handling Aspect* bug pattern describe cases where aspects fail to handle exceptions, the reasons for failure are different. In the *Late Binding Error Handling Aspect* bug pattern the pointcut expression intercepts the correct join point but the exception is mistakenly handled previously; in this bug pattern the pointcut expression is wrong.

Residual Handler

The Residual *Handler* bug pattern (also known as *Obsolete Handler* bug pattern) happens when a handler defined for a specific exception is no longer required, either because another handler was defined for the exception (in an aspect or in the base code), or because the operation that threw the exception was removed during a maintenance task.

Symptoms There is an inactive catch clause on the base code or inside an aspect.

Causes The handler associated with an exception on the base code becomes obsolete, because the exception handled by it is not signaled anymore. This obsolete handler is a source of problems during software maintenance if the program is changed to throw an exception that this handler will catch. Then, the obsolete handler may mistakenly catch exceptions and handle them incorrectly.

Code Example

The code snippet below was extracted from the HealthWatcher example. The `TransExceptionHandling` aspect handles an `IOException` thrown at specific points in the base code and wraps it on an instance of `TransactionException`. This instance is then re-thrown.

```
aspect TransExceptionHandling {

      public pointcut fileOperations():
           execution(public * File+.read(..));

     void around(): fileOperations(){
         try {
            proceed();
         } catch (IOException e) {
           throw new TransactionException(e);
         }
         …

     }
```

Unfortunately, older exception handlers that handled `IOExceptions` in the base were not updated. The code snippet below illustrates one residual handler that remained in the base code. This residual handler (lines 8-11) will prevent advice in the `TransExceptionHandling` aspect from handling this exception.

```
1. private void updateEmployee(Employee) {
2.
3.    try {
4.      ...
5.      tStamp = (TimeStamp)input.readTimeStamp();
6.      ...
7.    }
8.    catch (IOException e) {
9.       printErrorMessage("Error:" + e);
10.   }
11. }
```

Cures and Prevention

Every time an aspect is defined to handle an exception, the catch clauses previously associated with that exception should be inspected and removed when possible (i.e., if they are not responsible for handling any other exceptions). Specific tool support (exception flow analyzers [13]) should help during this task.

Related Patterns

- This bug pattern can be found on the base code after applying the Error Handling Aspect Pattern [18].

Acknowledgements

We would like to thank our shepherd Fernando Castor Filho for his insightful suggestions that helped us a lot to improve this catalogue of patterns. We also would like to thank the PC member Paulo Borba who kept track of our discussions and also contributed to the ideas presented here.

References

[1] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer M. Schwalb. An Evaluation of Two Bug Pattern Tools for Java, 2008 International Conference on Software Testing, Verification, and Validation, 2008, pp. 248-257.

[2] P. Louridas. Static Code Analysis. IEEE Software 23(4), p.58-61, 2006.

[3] E. Allen. Bug Patterns In Java. APress, 2002.

[4] Y. Nir, E. Farchi, and S. Ur. Concurrent bug patterns and how to test them. In International. Parallel and Distributed Processing Symposium, IPDPS 2003.

[5] FindBugs™ - Find Bugs in Java Programs. On site: http://findbugs.sourceforge.net/bugDescriptions.html

[6] http://www.eclipse.org/aspectj

[7] http://www.jboss.org/jbossaop/

[8] http://www.springframework.org/

[9] A. Colyer, A. Clement, “Large-Scale AOSD for Middleware”, Proc. AOSD Conf., 2004, ACM, pp. 56-65.

[10] Glassbox Inspector. https://glassbox-inspector.dev.java.net/

[11] G. Kiczales; J. Lamping; A. Mendhekar; C. Maeda; C. Lopes; J. Loingtier; J. Irwin. Aspect-Oriented Programming. In: Proceedings of the European Conference of Object-Oriented Programming (ECOOP’97), Springer-Verlag, 1997, p.220-242.

[12] S. Zhang; J. Zhao. On Identifying Bug Patterns in Aspect-Oriented Programs. In: Proceedings of the Computer Software and Applications Conference (COMPSAC 2007), 2007, p.431–438.

[13] R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. von Staa, C. Lucena, Assessing the Impact of Aspects on Exception Flows: An Exploratory Study, ECOOP 2008.

[14] M. Mezini; K. Ostermann. Conquering Aspects with Caesar. In: Proceedings of the Proceedings of the 2nd International Conference on Aspect-oriented Software Development, Boston, Massachusetts, ACM Press, 2003, p.90-99

[15] S. Soares; P. Borba; E. Laureano: Distribution and Persistence as Aspects. In: Software: Practice and Experience, Wiley, vol. 36 (7), (2006) 711-759.

[16] F. Buschmann; R. Meunier; H. Rohnert; P. Sommerlad; M. Stal. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley, 1996.

[17] T. Massoni; V. Alves; S. Soares; P. Borba. PDC: Persistent Data Collections pattern. In: Proceedings of the In First Latin American Conference on Pattern Languages of Programming — SugarLoafPLoP, University of Sao Paulo Magazine - ICMC, 2001, p.311–326.

[18] F. Filho, A. Garcia, C. Rubira, The Error Handling Aspect Pattern, SugarLoafPlop 2007.

[19] A. Garcia; C. Rubira. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. Journal of Systems and Software, 59 (6), 2001, p.197-222

[20] A. Colyer, et al. Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison-Wesley, 2004.

[21] J. Goodenough. Exception Handling: Issues and a Proposed Notation. Communications of the ACM, 18(12), p.683–696, 1975.

[22] R. Miller; A. Tripathi. Issues with Exception Handling in Object-Oriented Systems. . In: Proceedings of the European Conference on Object Oriented Programming (ECOOP'97), Springer, 1997, p.85–103.

Appendix I – Aspect Terminology

Aspects. Aspects are modular units that aim to support improved separation of crosscutting concerns [11]. An aspect can affect, or crosscut, one or more classes and/or objects in different ways. An aspect can change the static structure (static crosscutting) or the dynamics (dynamic crosscutting) of classes and objects. An aspect is composed of internal attributes and methods, pointcuts, advices, and inter-type declarations.

Join Points and Pointcuts. Join points are the elements that specify how classes and aspects are related. Join points are well-defined points in the dynamic execution of a system. Examples of join points are method calls, method executions, exception throwing and field sets and reads. Pointcuts have name and are collections of join points.

Advices. Advice is a special method-like construct attached to pointcuts. Advices are dynamic crosscutting features since they affect the dynamic behavior of classes or objects. There are different kinds of advices: (i) before advices - run whenever a join point is reached and before the actual computation proceeds; (ii) after advices - run after the computation "under the join point" finishes; (iii) around advices run whenever a join point is reached, and has explicit control whether the computation under the join point is allowed to run at all.

Inter-Type Declarations. Inter-type declarations either specify new members (attributes or methods) to the classes to which the aspect is attached, or change the inheritance relationship between classes. Inter-type declarations are static crosscutting features since they affect the static structure of components.

Weaving. Aspects are composed with classes by a process called weaving. Weaver is the mechanism responsible for composing the classes and aspects. Weaving can be performed either as a pre-processing step at compile-time or as a dynamic step at runtime.