A Catalogue of Bug Patterns for Exception Handling in Aspect-Oriented Programs

Roberta Coelho
DIMAp, Federal University of Rio Grande do Norte, Natal, Brazil
roberta@dimap.ufrn.br

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

Arndt von Staa
Informatics Department, Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
arndt@inf.puc-rio.br

James Noble
Victoria University of Wellington, New Zealand
kjx@mcs.vuw.ac.nz

Uirá Kulesza
DIMAp, Federal University of Rio Grande do Norte Natal, Brazil
uira@dimap.ufrn.br

Carlos Lucena
Informatics Department, Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
lcena@inf.puc-rio.br

ABSTRACT
Aspects allow a developer to externally add new functionality to a program. This additional functionality may also throw new exceptions that will flow through the program execution until they are handled. Moreover, aspects can also be used to handle exceptions thrown by base code or even other aspects. Unfortunately, exceptions thrown by aspects — or exceptions that should be handled by them — may flow through the program execution in unexpected ways leading to failures such as uncaught exceptions or exceptions being caught by the wrong handlers. In a previous empirical study we investigated the causes of such failures in Aspect-Oriented programs. In this paper we present causes of such failures as a catalogue of bug patterns for exception handling in Aspect-Oriented programs.

Categories and Subject Descriptors
D.2.5 [**Testing and Debugging**]: *Error handling and recovery. Debugging aids.*

General Terms
Aspect-oriented programming, exception handling, bug patterns, dependable systems.

Keywords
Dynamic Analysis, Monitoring, Aspect-Oriented Programming.

1. INTRODUCTION

The term bug has often been used in computer science as a synonym for fault or error, a specific construction in the program code that may lead to a failure [1]. It can also be used as a synonym for a code smell, 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 [1]. 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 repeat similar mistakes. Bug patterns are, therefore, recurring characteristics of program code that may lead to failures.

Some bug patterns have been proposed 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 developers can recognize these patterns, they will be able to diagnose the cause of a bug and correct it more quickly, as well as learning to avoid them.

Since the last decade, Aspect-Oriented Programming (AOP) [11] has been increasingly used as a means to modularize 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 exception handling code of AspectJ programs, and observed a set of recurring bugs on the way exceptions were thrown and caught inside the AO systems. Such analysis was based on the manual code inspections of a set of different releases of three medium-sized systems — from different application domains. Overall, this corresponds to 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 are used to catch exceptions — in these scenarios *Error Handling Aspects* [18] are used to *aspectize* the exception handling concern; and (ii) bugs on scenarios where aspect advice throw exceptions while adding new behavior to specific points in the program execution. Figure 1 illustrates the bug patterns discovered in each category.

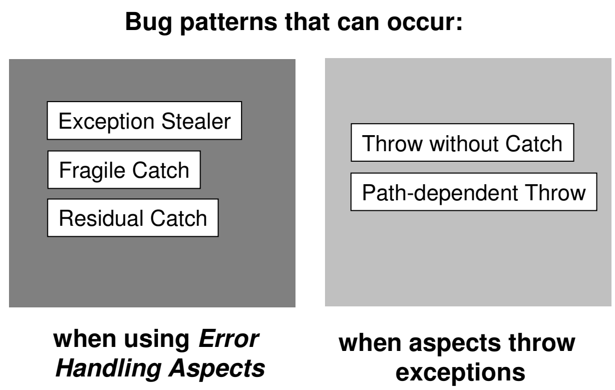

Figure 1: Catalogue of bug patterns.

The remainder of this paper is organized as follows. Section 2 presents some background on exception handling and AspectJ. Section 3 describes a simple AO system that will be used to exemplify the bug patterns. Section 4 details each of the bug patterns presented in Figure 1. Finally, Section 5 presents some discussions concerning the commonalities between the bug patterns presented here. 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 in 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

Everybody hates thinking about exceptions, because they're not supposed to happen

Brian Foote

Exception handling is a technique for structuring the error recovery code of a system in a way that errors can be easily detected and handled. Most mainstream programming languages provide constructs to signal the occurrence of an error (*throw an exception*) and to associate a set of recovery measures with the error, in order to deal with the problem (*catch and handle the exception*).

When a program throws an exception, the programming language's exception handling mechanism is responsible for changing the *normal control flow* of the computation within the program to its *exceptional control flow*. Thus, when an exception is thrown, the normal activity of a component is interrupted, and a search for an appropriate handler begins. When the appropriate exception handler is found, it is executed (the handler usually define a set of actions to remedy the exception) and control returns to the code that immediately follows the handler.

2.1.1 Exception Handling in AspectJ

AspectJ [6] is the most used aspect oriented programming language. AspectJ incorporates the aspect-oriented software development concepts into the Java programming language. The main concepts are the following: (i) join points – well-defined locations within the base code where a concern can crosscut the application (e.g. method calls); (ii) pointcuts – a collection of join points; and (iii) advice – a special method-like construction defined on aspects which are used to attach new crosscutting behaviors along the pointcuts. In AspectJ, as in Java, exceptions are represented in a hierarchical structure, on which each exception is an instance of the `Throwable` class (see Figure 2).

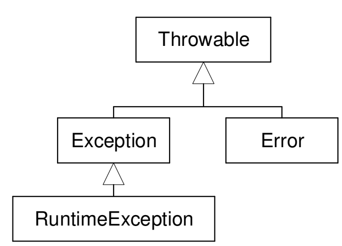

Figure 2: Exception Hierarchy in Java

Exceptions are either checked (`extends Exception`) or unchecked (`extends RuntimeException`) (see Figure 2). Errors represent asynchronous exceptions that can be thrown by Java platform and cannot be handled inside the program. A checked exception must be declared in the signature of every method that propagates it. Thus, the use of checked exceptions allows the compiler to statically check whether or not handlers are provided on the system to handle such exceptions. On the other hand, the unchecked exceptions do not need to be declared in the method signatures. As a consequence, there is very little that can be checked at compile time.

One advantage of mapping the exceptions into the type system is that handlers for one type of exception can also handle exceptions of its subtypes. Unfortunately, this characteristic can be the cause of some failures: a very general handler can catch an exception even though it does not have sufficient contextual information to handle it.

AspectJ reuses plain Java constructs to raise (`throw` statement) and handle exceptions (`try-catch-finally`) and to specify exceptions in method signatures (`throws` clause). AspectJ also contains new constructs that enable aspects to *throw* and handle exceptions, as follows:

- *The Declare Soft construct:* in AspectJ an advice can only throw a checked exception if it is thrown by "every" intercepted method [20]. To overcome this limitation, AspectJ offers the `declare soft` construct, which converts a given exception into a specialized runtime exception, named `SoftException`. The syntax is as following:

  ```
  declare soft : <someException> : <scope>;
  ```

  The <scope> is specified by a pointcut expression which selects a subset of joinpoints in which the <`someException`> will be wrapped in an instance of `SoftException`.

- `Handler` *Pointcut Designator*: One of the well-defined points during the execution of a Java program is the execution of an exception handler. AspectJ provides a pointcut designator that allows an aspect to advise the places where specific exceptions are handled through the `handler` pointcut designator [20].

- `After` *and* `After Throwing` *advice*: These kinds of advice allow aspects to be invoked when an exception is thrown by a method. This allows extra code to be executed when an exception is signaled.

Figure 3 presents a code snippet showing the use of some of these constructs. The `ErrorHandlingAspect` defines two pointcut expressions: one that intercepts the execution of every method defined in class `Foo` (line 2); and another that intercepts the execution of every method defined in class `Bar` (line 3). The `after throwing` advice (lines 4-6) catches every exception of type `E1` - that can be thrown during the execution of any method of class `Foo` – it then throws an instance of exception `E2` which stores the original exception message. The around advice (lines 7-13) catches any instance of exception E3 thrown by the execution of any method defined in class `Bar`. It then performs a recovery action to remedy the effects of the exception.

```
1. public aspect ErrorHandlingAspect {

2.   pointcut pointsToConvertExceptions() :
                    execution(* Foo.*(..));
3.   pointcut pointsToHandleExceptions() :
                    execution(* Bar.*(..));

4.   after() throwing (E1 e1) throws E2 :
5.            pointsToConvertExceptions() {
        throw new E2(e1.getMessage());
6.   }

7. void around():pointsToHandleExceptions(){
8.      try{
9.         proceed();
10.      }catch(E3 exc){
11.          recoverAction(exc);
12.      }
13.   }
14.
15. }
```

Figure 3: A simple exception handling aspect.

Besides being able of handling exceptions, through the use of *Error Handling Aspects* similar to the one presented above, every AspectJ advice has the ability to throw instances of any `RuntimeException`. Although we used AspectJ to exemplify the bug patterns the bug patterns described next can also be found on systems developed in such languages, and AOP frameworks that follow the same join point model as AspectJ (e.g., : CaesarJ [14], JBoss AOP [7] and Spring AOP [8])

3. EXAMPLE

This section presents an illustrative example of an information system, called Health Watcher (HW). Health Watcher is a web-based information system that allows citizens to register complaints regarding issues in health care institutions. This system is structured according to the Layered architectural pattern (see Figure 4). The GUI Layer comprises a set of Java Servlets responsible for receiving user requests (`ServletComplaint` and `ServletEmployee` in Figure 4). The Business layer contains a set of domain specific classes and the system facade. And finally the Data Layer contains the set of Data Access Objects (DAOs) responsible for persisting system data.

As we can see in Figure 4, the HW system implements some concerns as aspects:

(i) the `Monitoring` aspect is responsible for monitoring the performance of every Servlet request;

(ii) the `TransactionManagement` intercepts a set of methods defined by the Facade and by some DAOs, adding the transaction management concern. In other words, if an error occurs within the scope of a method intercepted by the `TransactionManagement` aspect, the aspect detects the error and executes a transaction rollback.

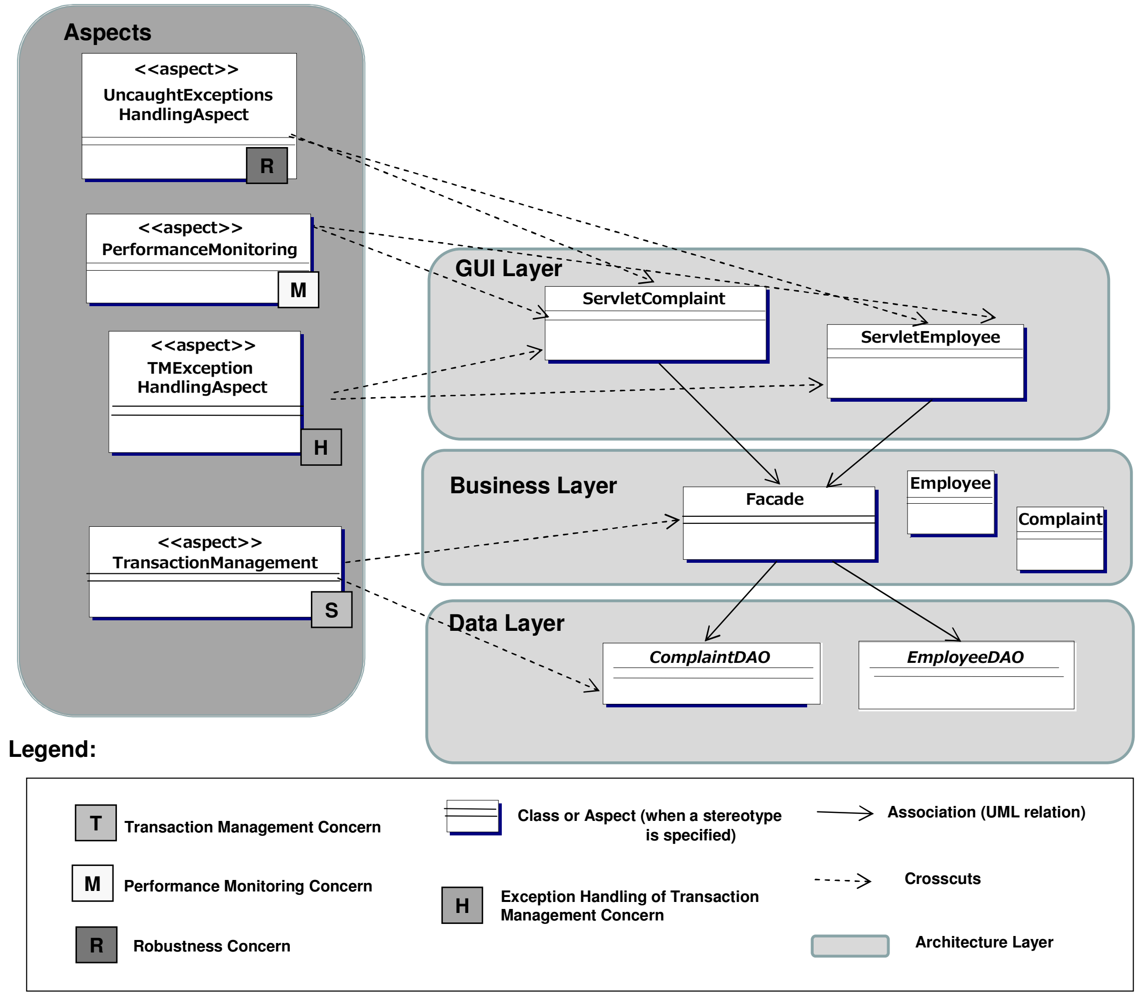

Figure 4. The architecture of HW system.

(iii) the `ExceptionHandling` aspect is responsible for catching the exceptions thrown by the transaction management aspect. It intercepts `Servlet` methods and catches the exceptions thrown by the `TransactionManagement` concern (instances of `TMException`). When catching such exceptions the exception handling aspect presents a pop-up message describing the failure.

(iv) The `AvoidUncaught` aspect was defined to avoid uncaught exceptions in the HW system. An uncaught exception is any instance of a `RuntimeException` that is not caught inside the system, and transparently propagates back to the program entry point, causing the Java virtual machine to terminate. To avoid such exceptions this aspect intercepts the set of methods from the GUI layer that receive user requests, and handles every exception that was not caught inside the system. To do so it uses a very general catch clause (`catch Exception`).

Each one of these aspects will be used in different scenarios to illustrate the bug patterns presented below. In this paper we do not cover bug patterns that can arise from the interaction between aspects.

4. THE CATALOGUE OF BUG PATTERNS

This catalogue of bug patterns is structured in two categories: (i) bugs on scenarios when aspects are used to catch exceptions, and (ii) bugs that can occur when aspects throw exceptions. 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 of AO systems, and therefore can help developers and testers to avoid and detect tem. The list of bug patterns can also be used to implement static checkers that could be used to automatically locate faults or potential faults in the source code.

4.1 Bug Patterns that can happen when using aspects to catch exceptions

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

4.1.1 Bug Pattern: Exception Stealer

The Exception Stealer bug pattern happens when an aspect is created to catch an exception, but some other catch clause defined in the base code catches (or "steals") the exception before it gets to the right point where it should be caught - by the aspect advice defined to handle it.

Symptoms

The symptom of this bug pattern is an *Unintended Handler Action* [22]. Consider an exception thrown by a base code method or an aspect advice, and which is supposed to be caught by an Error Handling Aspect. This exception can be handled by mistake by a catch clause in the base code – before the exception can reach the correct handler. As a consequence, the exception will not be adequately handled.

Causes

An exception could not be caught by an *Error Handling Aspect* that was defined to handle it because a there is a catch clause on a method in the call chain, between the method that threw the exception and the method that should handle it. The catch clause's type is the same type or a supertype of the exception that has been thrown. Figure 5 illustrates a scenario where this bug pattern occurs.

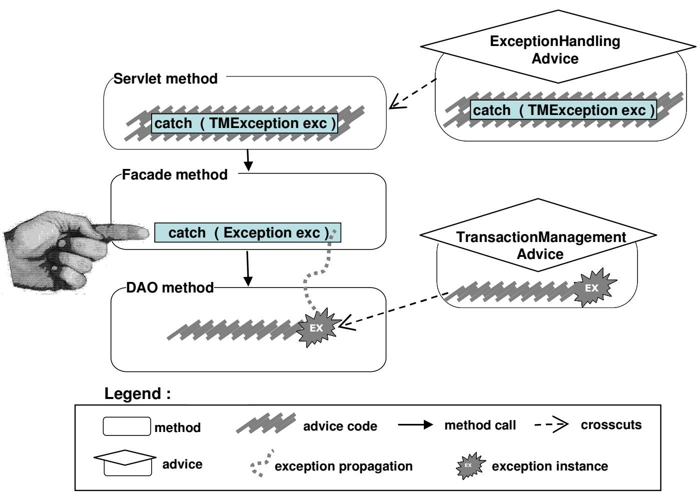

Figure 5: Schematic view of Stealer Exception bug pattern.

In this figure, an advice from `TransactionManagement` aspect adds new functionality (related to the transaction management concern) to a method from a DAO object. This additional behavior will throw an instance of `TMException` (an exception related to the transaction management concern) if an error occurs within the scope of a transaction. An advice was defined to handle the exception thrown by the `TransactionManagement` aspect (see the `ExceptionHandling` advice in Figure 5). This advice intercepts the points in the base code (more specifically, a `Servlet` method) where such exceptions should be caught. This bug pattern occurs when the exception that should be caught by the `ExceptionHandling` advice is caught by other catch clause . In this example the "*exception stealer*" is the catch clause inside a Facade method. As a consequence, the exception does not reach the right point where it should be caught — the `ExceptionHandling` advice — but will be handled by the Facade instead.

We can observe that the advice defined to catch an exception intercepts the correct point in the base code — where the exception should be caught — which means that there was no mistake in its pointcut expression. The exception could not reach the right catch clause because it was stolen beforehand by a catch clause in the base code. As we can see this same problem can also happen in OO development: an exception may be prematurely caught by an existing handler in the base code. During our empirical study [13] we observed that the problem is aggravated in AO systems because base code is supposed to be *oblivious* of the aspects. Some AO development approaches rely on the *obliviousness property:* the developer of the base code should not need to know that the code will be affected by aspects [23]. Consequently, the application developer does not prepare the code to deal with exceptions that may escape from aspects. Section 5 for discusses in more detail the influence of AO properties on the bug patterns presented here.

Code Example

In the Health Watcher system the `TransactionManagement` aspect may throw an instance of `TMException` if something goes wrong within the scope of a transaction:

```
aspect TransactionManagement {
  public pointcut DAOperations():
        execution(public * *DAO(..)))…;

  void around() : dataBaseOperations()
  {   ...
       //manage transactions
     if (status==0) {
      throw new TMException(cause_description);
     }
      ...
  }
 }
```

The *Error Handling Aspect* called `TMExceptionHandling` was defined to handle this exception in the GUI layer (see Figure 4). It intercepts the execution of the Servlet methods where the exception should be handled (see the code snippet below).

```
aspect TransExceptionHandling{

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

      void around():servletRequestExec()
      {
           try{
             proceed();
          }catch(TransactionException exc){
          //handle exception
        // present a pop-up message to the user...
          }
      }
```

However, the exception thrown by `TransactionManagment` aspect could not reach the `Servlet` methods where it should be handled, because the exception was caught beforehand by a "catch all clause" defined in the `Facade` class defined in the business layer (see the code snippet below). This means that exceptions thrown by the `TransactionManagement` aspect will not be adequately handled within the application.

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

Cures and Prevention

Ways to prevent this bug pattern are the following: (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 base program, and the other(s) for exceptions signaled by aspects. However, 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.

4.1.2 Bug Pattern: Fragile Catch

The Fragile Catch bug pattern happens when an aspect is created to catch an exception but due to a mistake on its pointcut expression it does not intercept the correct point in the program execution where the exception should be caught.

Symptoms

The Fragile Catch bug pattern occurs, an exception that should be caught by an *Error Handling Aspect* will not be caught by it. As a consequence, the exception will transparently propagate back to the program entry point, and may either: (i) become uncaught – if it reaches the program entry point without being caught, causing the Java virtual machine to terminate; or (ii) be mistakenly caught by an existing catch clause on the way to the program entry point (a failure also known as *Unintended Handler Action* [22]).

Causes

The fragility of the pointcut language and the number of different and very specific join points to be intercepted by the *Error Handling Aspects* are the causes that lead to this bug pattern. Figure 6 presents a schematic view of the Fragile Catch bug pattern.

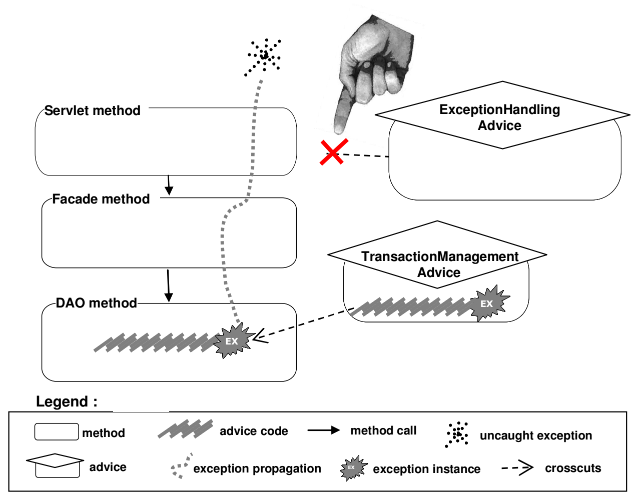

Figure 6: Schematic view of Fragile Catch bug pattern.

Let's say that we had removed the "general" catch clause that was defined in the Facade method (used to illustrate the previous bug pattern). Thus, the exception thrown by the `TransactionManagement` advice should be caught in the `Servlet` method intercepted by the `ExceptionHandling` aspect defined to handle it. However, due to a mistake on the pointcut expression associated to the `ExceptionHandling` advice, it does not intercept the `Servlet` method – where the exception should be caught. As a consequence the exception will flow throw back to the program entry point, until it is handled somewhere or becomes uncaught - terminating the system.

Code Example

The code snippet below illustrates a very simple version of this bug pattern. Due to a simple mistake in the pointcut expression – (a typo in at line 5). The `ExceptionHandling` aspect could not adequately handle the exception.

```
1.aspect ExceptionHandling{
2.
3.      public pointcut servletRequestExec():
4.          within(HttpServlet+) &&
5.          (execution(* HttpServlett.do*(..)) ||
6.        execution(*HttpServlet.service(..)))…;
7.
8.      void around():servletRequestExec()
9.      {
10.          try{
11.            proceed();
12.          }catch(TransactionException exc){
13.          //handle exception
14.     // present a pop-up message to the user...
15.          }
16.
17      }
```

This bug pattern usually happens an *Error Handling Aspect* should intercept a very specific part of the base program, as illustrated in the code snippet below:

```
// the pointcut defined on Error handling Aspect
1. pointcut getResource(String imageFile):
2.      (call (public void
3.             Class.getResourceAsStream(String))
4.             &&(args(imageFile)));

// the pointcut should be
5. pointcut getResource(String imageFile):
6.    (call(public java.io.InputStream Class.
7.        getResourceAsStream(String))
8.        &&(args(imageFile)));
```

Due to a subtle difference (a mistake on the method return type) the *Error Handling Aspect* that contain the pointcut defined at lines 1-4 will not handle the exception it was intended to handle, triggering the bug.

This bug pattern can be seen as an application of the general *fragile pointcut problem* to an exception handling scenario. In OO systems since the catch clauses are directly added in the base code, so this bug pattern can never be present.

Cures and Preventions

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 either a long term solution, nor a prevention to this problem.

Related Patterns

This bug pattern can occur when applying the *Error Handling Aspect Pattern* [18].

4.1.3 Bug Pattern: Residual Catch

The *Residual Catch* happens when a catch clause that was *aspectized* (refactored to an *Error Handling Aspect*) is left in the base code. As a consequence, the catch code will be duplicated.

Symptoms

A residual catch clause is a catch clause that used to handle exceptions before an AO refactoring. As a consequence, this residual catch can *mistakenly* handle an exception that should be handled by an *Error Handling Aspect*. This symptom is characterized as an *Unintended Handler Action,* which is a kind of failure on the exception handling code that is very difficult to detect [22].

Causes

This bug occurs when the exception handling code that was defined in the base code is refactored to an aspect. Usually, during an AO refactoring whose goal is to *aspectize* the exception handling concern, two main steps are performed. First, the developers creates an *Error Handling Aspect* that should intercept the point in the program execution where the exception should be handled. Secondly, they remove the *exception handling code* (the `catch` clause and its corresponding `try`) from the points in the base code where the exception handling concern was *aspectized*. The Residual Catch bug pattern occurs when a developer forgets to perform the second step (or performs it incompletely), and as a consequence there will two catch clauses for a single exception — one defined in the base code and other on the *Error Handling Aspect*. Figure 7 illustrates this bug pattern.

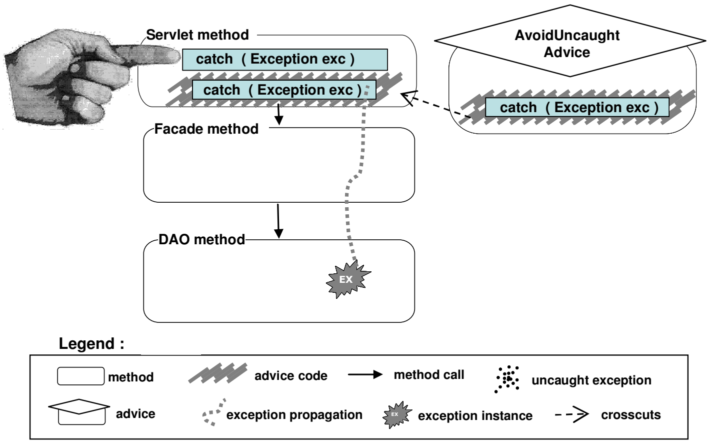

Figure 7: Schematic view of Obsolete Catch bug pattern.

In HealthWatcher, the `UncaughtExceptionHandling` aspect was created to catch every exception that was not caught inside the system – ensuring the system is more robust by avoiding the bad consequences of uncaught exceptions. Before this aspect was created the robustness concern was tangled in every `Servlet` method (as a set of `catch(Exception exc)` clauses). After refactoring this concern to the `UncaughtExceptionHandling` aspect a developer forgot to remove the corresponding catch clauses from the base code. The catch clause pointed out on the Figure is one such residual catch clause, and it catch exceptions that should be caught by the *Error Handling Aspect*. This residual catch clause should be removed.

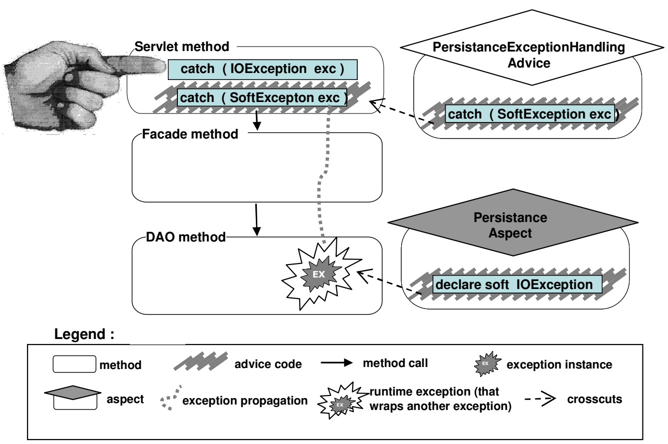

Figure 8: Schematic view of Obsolete Catch bug pattern.

The residual catch bug pattern can also occur when using the `declare soft` construct (only available in AspectJ language). This construct is often used during an AO refactoring, when a concern that can throw a *checked exception* is *aspectized* [15]. A common AO solution is to convert a checked exception thrown by the concern being aspectized into an unchecked `RuntimeException`. This new exception should be caught by an *Error Handling Aspect* at just those points where original checked exception was caught. In the example illustrated below the Persistence concern was *aspectized*, and the exceptions that used to be thrown by it (i.e., `IOException`) where converted to a specific `RuntimeException`[1]. In this case, since the exception type had changed, the residual catch will become dead code — no exception will be handled by it. Unfortunately, after more maintenance, dead residual handlers can come alive again, and then cause failures [10].

Code Example

The code snippets below illustrate the scenario described above. The Servlet `doGet` method used to handle instance of IOExceptions before the persistence concern was *aspectized*.

```
public class ServlerEmployee extendes … {
   public void doGet(HttpServletRequest req,
                           HttpServletResponse res){
       try{
        ...
       }catch(IOException exc){
       }
}
```

The persistence aspect converts the IOException on an instance of a SoftException through the use of the declare soft statement:

```
public aspect Persistence{

   declare soft: IOException : DAOScope();
   ...
}
```

An Error Handling Aspect was defined to handle the exception of the Persistence concern:

```
public aspect PersistanceExceptionHandling {

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

     void around():servletRequestExec()
     {
           try{
```

[1] This AO refactoring is known to promote the *unplugability* of aspects (another facet of *obliviousness*). If the IOException was not converted to a RuntimeException the signatures of every method potentially throwing the IOException would have to include it in its throws clause.

```
              proceed();
          }catch(SoftException exc){
            //handle exception
    // present a pop-up message to the user...
          }

     }
```

Cures and Prevention

During AO refactoring, every time an aspect is defined to catch some exception type, the base code catch clauses for this exception should be 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. AO refactoring tools should provide warnings to the developer suggesting removal of residual catch clauses, or even removing them automatically as part of refactorings.

Related Patterns

This bug pattern can occur when applying the Error Handling Aspect Pattern [18].

4.2 Bug Patterns that can happen when aspects throw exceptions

4.2.1 Bug Pattern: Throw without Catch

The Throw without Catch bug pattern occurs when an aspect advice throws an exception but no handler is defined to catch it.

Symptoms

The developer detects an uncaught exception – an exception that is not caught inside the system, and transparently propagates back to the program entry point, causing the Java virtual machine to terminate; or an exception that is *mistakenly* caught by an existing handler – this scenario is also known as *Unintended Handler Action* [22]. This is very difficult to diagnose since an existing handler may swallow the exception without logging or presenting any warning to the user.

Causes

This bug occurs when a method or AO construction, such as an aspect advice, an intertype declaration or the `declare soft` construction, throws an exception (or wraps and re-throws an exception in the case of `declare soft` construct) and no catch clause is defined to catch it, either in the base code or in an *Error Handling Aspect*.

We can observe that 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. If such exception is not documented, a developer cannot know that the aspect may throw an exception, and as a consequence will not define a catch clause to handle the exceptions that may flow from it. Figure 9 illustrates a scenario where a monitoring aspect throws an exception that becomes uncaught since no handler was defined to it.

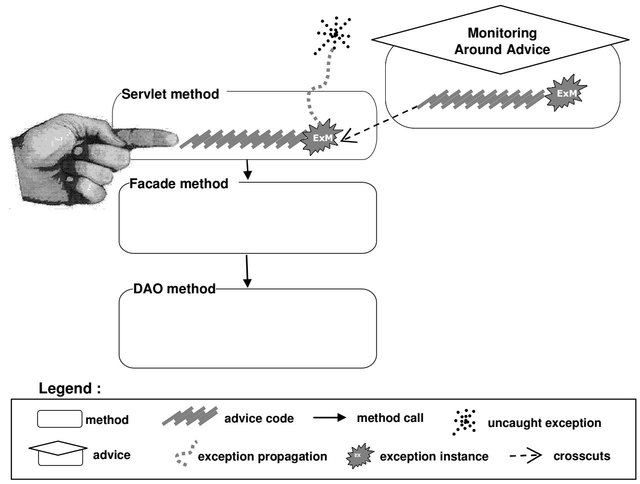

Figure 9: Schematic view of Throw-without-Catch bug pattern (scenario 1).

According to the AspectJ documentation [20], every time the `declare soft` construct is used (i.e., an exception is softened by an aspect) the developer should implement another aspect that will be responsible for handling the softened exception. This solution is very fragile, as (i) it is up to the programmer to define a new aspect to handle the exception that was softened, and (ii) no message is shown at compile time to warn a programmer if the *Error Handling Aspect* is forgotten. Figure 10 illustrates a scenario where the `declare soft` construct is used and no catch clause was defined to handle the instance of `SoftException` thrown by it.

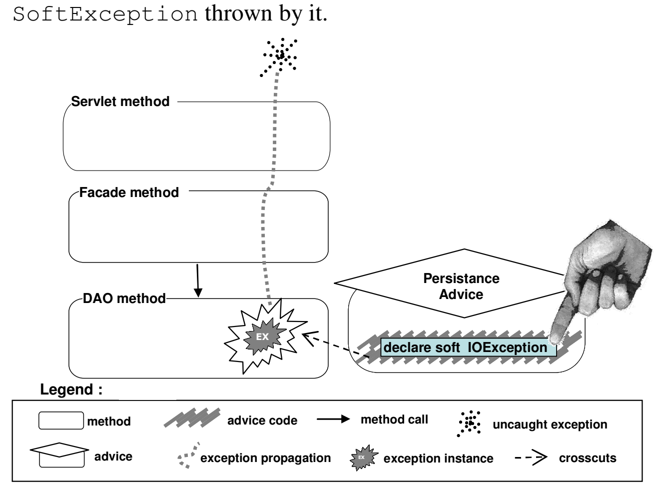

Figure 10: Schematic view of Throw-without-Catch bug pattern (scenario 2).

Code Example

The code snippet below was extracted from the Health Watcher system. It shows an aspect that calculates and logs the performance of each HTTP request; to do so it calls an OO library to log the performance – this OO library throws a runtime exception when the log file is too large. Since this exception is not documented not handler was defined to it.

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

      void around : calcPerformance()
       {
           …
           perf = calcPerformance();
           log (perf);

        }
  }
```

Cures and Prevention

In languages such as Java that support unchecked exceptions, in order to know which exception may be thrown from a method, developers 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) application-specific error handling; or (ii) error isolation.

According to the application-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 advise every aspect that may signal an exception and catch the exceptions signaled by it. This solution avoids the exceptions thrown by aspects from flowing to the program execution. Such error-isolation aspects 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 below illustrates a handler aspect that implements the error isolation strategy.

```
public aspect ErrorIsolation {
  ...
  public pointcut scope() :
       within(healthwatcher.aspects.*);
  void around(): adviceexecution() &&
     scope()){
   try {
       proceed();
    } catch (Exception e) {
      log(e);
    }
 }
}
```

The `ErrorIsolation` aspect defines the scope of the aspect via a pointcut expression that matches every element defined on the aspects package of HealthWatcher – excluding any aspects whose name follows the pattern "`*AroundAdvice`". This aspect intercepts every advice execution within the *scope* via the `adviceexecution` designator that intercepts the execution of every advice. The advice associated with this pointcut catches every instance of Exception that may be thrown by any advice execution.

This solution only works well for isolating the exceptions that come from before and after advice, however, since around advice may also contain a call to `proceed()` – that invoke the intercepted method - when handling exceptions that escape from `around` advice, we will also intercepting exceptions thrown by base methods intercepted by the `around` advice. There is no easy way to intercept executions thrown in around advice only — excluding the execution of the intercepted method [24]. In this solution, the exceptions thrown by the client application (calling proceed) will be caught and handled as an aspect exception – which may break the exception handling policy of the client application.

To solve this problem, we can improve the previous solution relying on a naming pattern to exclude the exceptions that come from around advice to be swallowed. We can write specific aspects whose name matches `*AroundAdvice` which will include every around advices, and exclude this advices from the `ErrorIsolation scope()` in the following way:

```
public pointcut scope() :
     within(healthwatcher.aspects.*)
     && !within(healthwatcher.
     aspects.*AroundAdvice);
```

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 introduced when solving the bug pattern presented here.

4.2.2 Bug Pattern: Path-dependent Throw

The Path-Dependent Throw bug pattern occurs when the exceptions thrown by a method depends on the path (in the program call chain) from which this method is executed. This bug pattern causes uncaught exceptions and unintended handler actions in AO systems.

Symptoms

The developer detects an uncaught exception – the exception thrown by an application method is not caught inside the system. This may lead to a software crash; or an exception being mistakenly caught by a handler — a scenario also known as *Unintended Handler Action* [22]*,* which is very difficult to diagnose since a handler in the base code may swallow the exception without logging or presenting any warning to the user.

Causes

This bug pattern usually happens when an aspect advice is associated with a pointcut expression that includes any of the scope designator used for scoping purposes (i.e., `within`, `withincode`, `cflow`, `cflowbelow`). Due to these scope designators, an aspect may or may not affect a method according to the call chain — the calling path used to reach the method. 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.

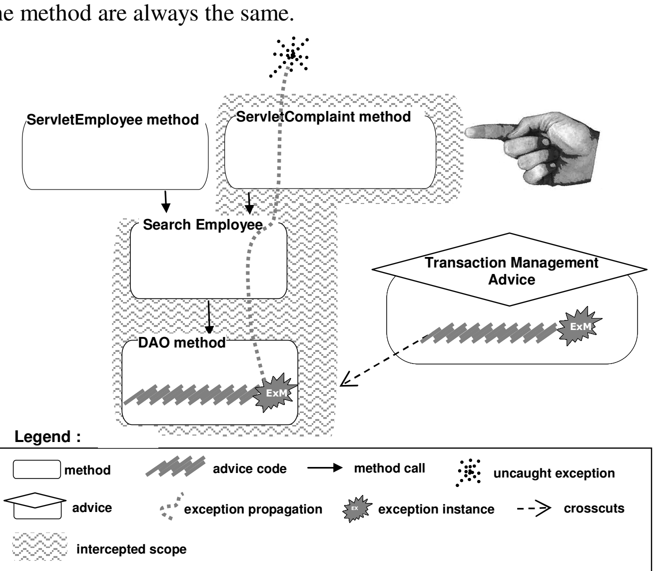

Figure 11: Schematic view of Path-Dependent Throw bug pattern.

When the list of exceptions that can be thrown by a method varies according to the scope within which it is executed (that is, the method call chain from which it was called) we say that this method suffers from the *Path-Dependent Throw* bug pattern. This bug pattern makes understanding the exceptional behavior of a method very confusing. As a consequence, exceptions thrown by such methods can easily remain *uncaught* or are caught by *unintended handlers*. Figure 11 presents a schematic view of this bug pattern.

In this figure, the `TransactionManagement` advice adds a new functionality to a method from a *DAO object*, but only when the method is called from the `SearchEmployee` method, when `SearchEmployee` was itself called by a method defined on the `ServletComplaint` — i.e., the advice is associated with a pointcut expression that contains a dynamic scope delimiter. This scope is represented in gray lines in Figure 11.

Therefore, this additional functionality, and the new exception (`TMException`) that comes with it, will be part of the *DAO method* when it is called from the call path: `Servlet Complaint` → `Search Employee`. When it is called on the call path `Servlet Employee` → `Search Employee` → `DAO method`, it will not throw an instance of `TMException`. We can observe that even if the DAO method arguments are the same, the set of exceptions thrown by it will differ.

Code Example

The code snippet below illustrates a scenario where this bug pattern can occur. We add the `LayerArchitecturePolicies` aspect to HW — this aspect is responsible for checking architectural policies, such as: *the methods defined by the system Facade can only be accessed by Servlets*. The code snippet below illustrates this aspect:

```
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 in the Facade class, but only when it is not executed within a Servlet. As a consequence, the advice associated to it only affects and throws 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). The developer did not define a handler to the exception thrown in this context and so the exception will not be caught inside the system: it may become uncaught or be mistakenly caught by an existing handler.

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

}
```

Finally, we can observe that in AO systems aspects may modify any method's well-established behavior, and may create a situation where the exceptions that a method throws may depend on the control-flow path used to reach the method (e.g., which clients are calling it).

5. DISCUSSIONS

This bug patterns catalogue presents some causes of the most common failures on the exception handling code in AO programs:

- *Uncaught Exceptions*: exceptions thrown and not caught inside the system, that it may lead to a software crash; or
- *Unintended Handler Actions:* exceptions being mistakenly caught by wrong handlers. This failure is very difficult to diagnose since a handler in the base code may swallow the exception without logging or presenting any warning to the user.

Moreover, analyzing the characteristics of the bug patterns, we can observe that some bugs can also happen in object-oriented systems. However, we had observed that some AO properties (*quantification* and *obliviousness*) pose specific pitfalls to the development of the exception handling code in AO systems, which further aggravates the failures mentioned above.

The *quantification* property of aspects allows programmers to write statements with the following form: "In program P, whenever condition C happens, perform action A". AspectJ supports this property by means of pointcuts and advice. An advice can be of two types —call and execution — that intercept the call and execution of a set of join points respectively. Call and execution advice have different impact on the exception behaviour of modules. The exception advice affects the exceptional behaviour of the advised methods, while the call advice affects the exceptional behaviour of the advised module's caller.

Such impact can also be influenced by static scopes such as `within` and `withincode` — which delimit the classes or packages into which aspects will inject new behavior – and dynamic scope constructs (i.e., `cflow` and `cflowbelow`) which 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 to the exception handling model of AO systems is that the exceptional behaviour of modules may vary depending on where they are used. Therefore, the same module can raise different sets of exceptions depending on which class called it or even if it is called within a specific call path. Consequently, it will be more difficult for the module's user to prepare the code to handle the exceptions that the module can throw.

The *obliviousness* property establishes that programmers of the base code – the classes which will be affected by the aspects – do not need to be aware of the aspects which will affect it. Obliviousness means that programmers do not need to prepare the base code to be affected by the aspects [23]. Since (in AspectJ at least) there are no mechanisms to protect base code from exceptions that will flow from the aspects added behavior, the exceptions thrown by aspects may be erroneously caught by modules from the base code, or become uncaught. Moreover, there are aspect oriented constructs that allow aspects to add new behaviors on modules at load time – which can make this kind of problems even more difficult to diagnose since it is not easy to reason about the effect of aspects on the exception flow of programs during system execution.

6. ACKNOWLEDGMENTS

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. Finally we thank the members of the PLoP 2008 writers' workshop for their comments on this paper: Steven Hill, Nobukaza Yoshioka, Takao Okubo, Hironori Nashizaki, Eduardo Fernandez, David Pearce, Brian Foote, Amir Raveh, Yuji Kobayashi, Robert Hanmer.

7. 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.

[23] R. Filman, T. Elrad, S. Clarke, M. Aksit, Aspect-Oriented Software Development, Addison-Wesley, 2005.

[24] C. Clifton G.T. Leavens, J. Noble. MAO: Ownership and Effects for More Effective Reasoning about Aspects. In: Proceedings of the European Conference on Object Oriented Programming (ECOOP), Springer, 2007, p.451-475.

8. APPENDIX A - Aspect Terminology

This appendix contains a brief overview of the terminology associated with aspect-oriented software development. We have used the terminology described by Kiczales et al [11] and adopted by aspect-oriented programming languages, such as AspectJ. We present below the main terms that are usually considered as a conceptual framework for aspect-orientated design and programming.

Aspects. Aspects are modular units that aim to support improved separation of crosscutting concerns. 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 execution of a system. Examples of join points are method calls, method executions, exception throwing and field sets and reads. Pointcuts are collections of join points and may have name.

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.