Using AspectJ™ For Programming The Detection and Handling of Exceptions

Cristina Lopes, Jim Hugunin, Mik Kersten
Xerox PARC, USA
{lopes,hugunin,mkersten}@parc.xerox.com

Martin Lippert
University of Hamburg, Germany
lippert@acm.org

Erik Hilsdale
Indiana University, USA
eh@acm.org

Gregor Kiczales
University of British Columbia, Canada
gregor@cs.ubc.ca

Abstract

We took an existing framework written in Java™, the JWAM framework, and partially reengineered some of its exception detection and handling aspects using AspectJ™, an aspect-oriented programming extension to Java. The results of this reengineering project are reported in [5].

This position paper summarizes the project and the results presented in that publication, and proposes them for discussion in the context of the Workshop on Exception Handling in Object-Oriented Systems.

1 This Work in the Context of This Workshop

This position paper is based on a study described in more detail in [5]. That study shows a number of properties of AspectJ that (1) facilitate the combination of normal and exceptional behaviors and (2) reduce the opportunity of implementation errors when doing that combination.

Whether the properties of AspectJ documented in that study lead to programs with fewer implementation errors and that can be changed more easily, is still an open research topic. We believe some of the findings in that study will raise interesting discussion topics related to the design of language mechanisms for better support of exceptions. We present some those topics in section 5. The core of this position paper is a summary of [5].

2 Motivation and Synopsis of the Reengineering Project

A lot of work in programming languages has had the goal of providing better support for the detection and handling of exceptions. Contracts, introduced by Hoare [3] are, in many ways, one such example, consisting in the definition of pre-conditions, post-conditions and invariants that determine how to use and what to expect from a computational entity. [2] introduced the idea of explicit language constructs for exception handling. A lot of complexity in propagating an exception – up to the part of the program that knows how to handle it – can be avoided by taking advantage of those mechanisms. Many programming languages in use today include constructs for defining application exceptions, and for throwing and catching those exceptions.

Coding the detection and handling of exceptions, is, however, still a difficult process that requires a strict discipline from programmers. And the reality is that most software being written today uses programming languages that provide very little help. Hopefully, that will change.

But there is a side effect of coding application exceptions that cannot be addressed by simply including more powerful exception detection and handling mechanisms in the programming language. That side effect consists of the tangling between the code for what the program should do – i.e. its normal behavior – and the code for detecting and handling situations that are considered exceptional. Figure 1 illustrates this issue with a small Java example. The example shows parts of a class from a distributed game. The code related to exceptional behaviors is underlined.

```
Registry _registryServer; // set by some other method
/**
 * newRegistrationNumber is provided by Game to remote clients.
 * In turn, this game object invokes the remote registry server.
 */
public RegistrationNumber newRegistrationNumber (String registrationName)
  throws FailedRegistrationException {
  Contract.require(registrationName != null, this);
  RegistrationNumber result = null;
  try { result = _registryServer.newRegistrationNumber(registrationName); }
  catch(RemoteException e) {
    ErrorLog.print("Error at: newRegistrationNumber: " + e);
    throw new FailedRegistrationException();
  }
  return result;
}
```

Figure 1. Illustration of the tangling between the code for what the program should do and the code for detecting and handling exceptional behaviors. The code related to exceptional behaviors is underlined. It includes the exceptions' generation, catching and handling, and the pre-conditions.

This tangling is, in part, a consequence of the programming language and, in part, a consequence of the programmer's design decisions. But this tangling also reflects an important design decision that is implicit and, for most part, imposed by the programming languages: that the abstraction of "what to do" in an operation is the same abstraction of "how to detect and react to exceptions" in that operation. This implicit unification has negative effects on the way normal and exceptional behaviors can be reused, evolved and documented.

Aspect-Oriented Programming (AOP) is intended to ease situations that involve many kinds of code tangling. Code tangling is usually a consequence of having to code crosscutting concerns in the classes. In order to investigate AOP's ability to ease tangling related to exception detection and handling, we took an existing framework written in Java, the JWAM framework, and partially reengineered its exception detection and handling aspects using AspectJ, an aspect-oriented programming extension to Java.

2.1 AspectJ

The reengineering project used AspectJ version 0.4. AspectJ [1] extends Java with the explicit concept of a "crosscut" and it supports actions on those crosscuts. For example,

```
crosscut setters(): Point &
         (void set(int x, int y) |
          void setX(int x) |
          void setY(int y))
```

The crosscut above denotes those times in the execution of the program when the message set(int x, int y) or the messge setX(int x) or the message setY(int y) are received by any point object. An associated action might be

```
static advice setters() {
  before { System.out.println("Entering a setter"); }
  after  { System.out.println("Exiting a setter");  }
}
```

The code above detects when one of those messages is received by a point, as defined in the crosscut setters; upon detection, something is printed out on the screen before and after the corresponding methods are executed. This simple example provides the basis for understanding the separation mechanisms that are the basis for the reengineering work. For detailed information about AspectJ, we refer the reader to the documentation that can be found on the web [1].

2.2 JWAM

The JWAM Framework is a Java-based object-oriented framework for interactive business applications, developed at the University of Hamburg. It contains components for supporting client/server computing, distribution and persistence, and it covers all the requirements for medium to large business applications. It

contains only the generic technical foundation, and can be used in different domains, for different purposes. Detailed information about the JWAM framework is available on the web [4]. This work used JWAM version 1.4.

JWAM contains more than 600 classes and interfaces as well as about 150 test classes. This study targeted the whole framework.

3 The Reengineering Work

JWAM uses contracts to ensure that callers do not misuse the methods, and that the methods' implementations preserve some of their basic specifications. The designers of JWAM have made the decision that a broken contract always results in a runtime exception of the thread where the contract was broken, and that no attempt should be made to recover from that.

Besides broken contracts, there are many other exceptional behaviors inside JWAM and in the applications that use it. Table 1 shows the five most-commonly caught types of exceptions in JWAM and the frequency by which they are caught inside the framework.

Type of exception caught	Number of catch statements
Exception	77
SQLException	46
IOException	38
RemoteException	29
NumberFormatException	22

Table 1. The five most-caught types of exceptions

When reengineering the exceptional behaviors of JWAM, contracts were immediately targeted as aspects. Besides contracts being related to the exceptional behaviors, there was another reason for seeing them as aspects: the documented optional nature of contracts at runtime suggested that sometimes developers may want to configure classes with contracts, but other times they may want to remove the contracts from the classes. The performance overhead introduced is a good reason for removing contracts. Using AspectJ we extracted the contract code from inside the classes into separate aspects.

Besides contracts, we also targeted the five most frequently caught exceptions as candidates for aspect-related implementations (see Table 1). In their total, they accounted for 212 catch statements, more than two thirds of the total number of catch statements in the framework. Interestingly enough, the number of different types of reactions to those exceptions was considerably smaller – 14. Some of the reactions that we found frequently were: "log and ignore," "set the return to a default value," and "throw an exception of a different kind." The fact that the number of reaction types is much smaller than the number of places where exceptions were caught shows that there was a fair amount of redundancy in the code related to exception handling.

Reengineering exception handling with AspectJ, however, presented a problem that AspectJ 0.4 could not handle elegantly. The problem is that, in principle, exceptions can be caught and handled in arbitrary parts of the methods' implementations, and AspectJ 0.4 does not provide support for capturing the catching of exceptions inside method boundaries. The work-around for this limitation consisted of wrapping exception catching in methods. This practice is, of course, unacceptable for any purposes other than this study. Hopefully, future versions of AspectJ will be able to address this problem properly, as this is not an inherent limitation of aspects. But this point is worth discussing.

Fortunately, there were some classes in JWAM that already had some sort of wrapping around exception catching. Those classes were part of the remote communication infrastructure, built with RMI. The handling of the `RemoteException` was concentrated in just a couple of classes. We found that there was only one kind of reaction to the `RemoteException`, namely to log the exception and ignore it. A typical piece of code for this is:

```
try { registry.put(name, this); }
catch (RemoteException e) {
  ErrorLog.print("registry call failed: " + e);
}
```

As Table 1 points out, there were 29 statements similar to this one in the two proxy classes, the only difference among them being the remote call itself.

Unlike contract aspects, the added symptom for *aspectification*, in this case, was not the optional nature of the handler – catching `RemoteException` is not optional, it's mandatory – but the expected evolution of the handler specifications. Taking into account the ratio between reaction types and number of places where exceptions were caught, it is likely that the number of RMI exception handlers in the future will be much less than 29 (1 to 3 would be an educated guess). Therefore, there seemed to be some advantages in implementing the handler as an aspect that hooks into specific parts in the implementation of the classes. Unlike contract aspects, these exception handling aspects are not, and should not be, "unpluggable", but they can be mutually "replaceable."

Using AspectJ we extracted the contract code from inside the classes into separate aspects. Moreover, using abstract crosscuts, we were able to design and implement exception handlers that can be plugged into many different applications, not only to JWAM. The following code shows one of those handlers, namely the one that corresponds to the reaction "log and ignore."

```
abstract public class AbstractLogAndIgnore {
  abstract crosscut methods();
  static advice methods() {
    catch (RemoteException e) {
      ErrorLog.print("remote call failed in: "
                     + thisMethodName + ":" +e);
    }
  }
}
```

Abstract aspects are used by applications through subclassing. For example, the abstract aspect above can be used by concretizing the abstract crosscut:

```
public class JWAMRemotExceptionHandler
       extends AbstractLogAndIgnore {
  crosscut methods():
    RegistryServer & * *(..) |
    RMIMessageBrokerImpl & private * *(..);
}
```

4 Summary of Results

About 11% of LOC of the original implementation of the framework is about detecting and handling exceptional behaviors. This is a significant number, given that the framework's exception handling strategies were quite simple, the most frequent one being "log and ignore."

JWAM consists of 614 classes, and about 44,000 lines of code (LOC). The use of contracts is uniform throughout the classes of the framework, and it amounts to 2,786 LOC. The handling of exceptions is not uniform. 125 of those classes concentrate 100% of the implementation of exception handling – this corresponds to about 2,000 LOC. Table 2 presents a summary of the measurable results of this work.

	Without aspects	*With aspects*
Exception detection	2120 pre-conditions (2120 LOC)	620 pre-conditions (660 LOC)
	666 post-conditions (666 LOC)	291 post-conditions (300 LOC)
Exception handling	414 catch statements (2,070 LOC)	31 catch aspects (200 LOC)
% of total LOC	10.9%	2.9%

Table 2. Summary of quantitative results.

Besides these results, we also made a number of qualitative observations, covering both advantages and disadvantages of using aspects. The advantages were: (1) better support for different configurations of normal and exceptional behaviors; (2) better tolerance for changes in the specifications; (3) better support for incremental development; (3) more potential for reuse; (4) automatic enforcement of contracts in application that use the framework; (5) less interference in the program texts. The disadvantages were: (1)

insufficient support for reconstruction of local effects; (2) inadequate semantic support for inheritance of preconditions; (3) designation mechanism still not expressive enough.

5 Discussion Topics

Aspect-oriented programming has been developed to provide programmers with mechanisms to implement, abstract and compose crosscutting concerns. AOP language design is based on understanding the structure of common crosscutting concerns and developing mechanisms that can capture that structure and can be composed to develop more complex crosscutting structures.

By submitting to this workshop we are hoping to engender discussion on several topics: (i) does the description of error detection and handling as a concern that crosscuts the basic functionality seem viable to other researchers in this area; (ii) what are the basic crosscutting structures required to support sophisticated error detection and handling schemes; and (iii) what changes could be made to AspectJ to support those kinds of crosscutting.

References

1. AspectJ Web Site, <http://www.aspectj.org/>.
2. J. B. Goodenough: Exception Handling: Issues and Proposed Notation, in *Communications of the ACM*, Vol. 18, No. 12, 1975, pp. 683-696.
3. C. A. R. Hoare: An Axiomatic Basis for Computer Programming, in *Communications of the ACM*, Vol. 12, No. 10, pp. 576-580, 583, October 1969.
4. JWAM framework Web Site, <http://www.jwam.de/>.
5. M. Lippert and C. Lopes: A Study on Exception Detection and Handling Using Aspect-Oriented Programming, in *Proceedings of the 22nd International Conference on Software Engineering*, Limmerick, Ireland, June 2000 (to appear).