

Aspects of Exceptions at the Meta-Level (Position Paper)

Ian S. Welch¹, Robert J. Stroud, and Alexander Romanovsky

Department of Computing, University of Newcastle upon Tyne,
United Kingdom NE1 7RU

{i.s.welch, r.j.stroud, alexander.romanovsky}@ncl.ac.uk
<http://www.cs.ncl.ac.uk/research/dependability/reflection/>

1 Introduction

This paper describes the design and usage of a metaobject protocol that explicitly includes support for handling exceptions. We do not propose implementing exception mechanisms anew [3, 5] or proposing a unified meta-level software architecture for exception handling [4]. To make our discussion concrete we describe an extension of the Kava [9] metaobject protocol that includes exceptions as first class values, provide examples of Kava’s use, and compare Kava with related Java extensions.

We believe that insufficient attention has been paid to exceptions by designers of metaobject protocols for object-oriented languages. Most metaobject protocols provide a way of intercepting method execution but these protocols are usually discussed solely in terms of arguments and results. Signalled exceptions are rarely discussed. However, in order to successfully implement non-functional requirements using metaobject protocols it is important that exceptions are explicitly considered. For example, consider using a metaobject protocol approach to implement distributed objects. It is not sufficient just to convert method calls into remote method calls, exceptions must be converted into remote exceptions as well. Therefore, a metaobject protocol should be designed to treat method arguments, method return values and exceptions equally. This means that if the behaviour of method execution is reflected upon, then any signalled exception should be reified and be manipulable at the meta-level. Note that such a “exception-aware” metaobject protocol should not lead to base-level programmer’s expectations being confounded, as doing so would make both programming and verification very difficult. For example, the exception model should not be able to be changed dynamically, say from a termination model to a resumption model (as opposed to [2]).

2 Meta-Level Requirements

What facilities should a “exception-aware” metaobject protocol have? We propose that such a metaobject protocol needs two facilities: meta-level interception

of exceptions signalled from the base-level, and meta-level raising of exceptions at the base-level.

Meta-level interception is required to handle new exceptions introduced as a side-effect of the implementation of non-functional requirements. It may also be required to reinterpret existing base-level exceptions in the context of new non-functional requirements. An example of this was given in the introduction where distribution requires that local exceptions are reinterpreted as remote exceptions.

Meta-level raising of exceptions at the base-level is required to allow metaobjects to raise new types of exceptions and maintain the transparency of the meta-layer. For example, a metaobject may enforce a security policy by raising a security exception whenever the security policy is violated. Since we normally wish to implement non-functional requirements transparently this exception should appear to be raised at the base-level. If it appears to have been raised by the metaobject then the meta-level becomes visible to any clients of the base-level and then transparency is shattered.

These two features allow the metaobject protocol to support the following mappings between exceptions and values: from one exception to another, from one exception to a value, or from a value to an exception. In the remainder of this section we provide examples of how these mappings can be used.

Exception to exception. Adding debugging information to exceptions requires that one exception is mapped to another. Here, we want to add meta-information to an exception such as the time it was signalled. An extended version of the exception class could be defined that encapsulates the base-level exception and the meta-information. At the meta-level the signalled exception is replaced by an instance of the extended exception class.

Exception to value. Logging and then ignoring an exception requires that an exception is mapped to a value. Here, the base-level exception is suppressed and the method terminates normally returning a value specified at the meta-level.

Value to exception. Assertion checking [7] requires that a value is mapped to an exception. Here, a value of a member variable or argument of the method causes an exception to be raised. This exception will appear to be raised at the base-level to preserve transparency.

3 Kava

Kava is a reflective Java implementation [9]. It uses byte code transformations to make constrained changes to the binary structure of a class in order to provide a metaobject protocol that brings object execution under the control of a meta-level. These changes are applied at the time that classes are loaded into the runtime Java environment. The meta layer is made up of metaobjects that are written using standard Java. The binding between classes and metaobject classes are specified in an XML configuration file called a *binding specification*. Kava

brings the sending of invocations, initialisation, finalization, state update, object creation and exception signalling under the control of a metaobject protocol.

When a meta-level programmer creates a new metaobject class, the programmer extends the default metaobject class and overrides those methods that control the behaviours the programmer wishes to redefine. In *Kava* we define *around* style meta methods, so for each behaviour there is a *before* and *after* method.

The following listing shows the methods relating to overriding method execution in the metaobject class interface,

```
public interface IMetaObject {
    ...
    public void beforeMethodExecution(IMethodExecution context)
        throws Exception;
    public void afterMethodExecution(IMethodExecution context)
        throws Exception;
}
```

A context object is passed as an argument to each of the meta-level methods. The context reifies the context of the meta-interception as a context object that implements the `IMethodExecution` interface. In earlier versions of *Kava* exceptions that were raised during the execution of a method were not included in the context. Now, any exceptions that have been raised is included in the context in addition to reified method, its actual parameters, and the result of the execution of the method. In addition to reifying exceptions the context API has been extended to support the reflection of the exception back to the base-level and the overriding of the exception signalling at the base-level.

We are currently examining how this extended metaobject protocol can be used to implement Java language extensions such as multi-level handlers for exceptions (statement, block, method, class and exception level), design by contract, and n-version programming).

4 Examples

In this section we show how to use *Kava* to implement the examples described in the meta-level requirements section.

First, we show how an exception can be intercepted and converted to another type. Here, the exception is converted to an instance of an exception class used for debugging which encapsulates the base-level exception, the key to this is using the `setException` method to change the exception raised at the base-level,

```
public DebugMetaObject extends MetaObject {
    public void afterMethodExecution(IMethodExecution context)
        throws Exception {
        if (context.isExceptionRaised()) {
            context.setException(new DebugException
                (context.getException())); }}}}
```

The next example shows how an exception can be mapped to a value and the exception raising at the base-level suppressed. This metaobject is used to log and suppress `IOExceptions` exceptions. It checks that the base-level method exited because an exception was raised. The base-level exception is suppressed through the use of the `overrideException` method,

```
public LogMetaObject extends MetaObject {
    public void afterMethodExecution(IMethodExecution context)
        throws Exception {
        if (context.isExceptionRaised()) {
            Exception e = context.getException();
            if (e instanceof java.io.IOException) {
                context.overrideException();
                log(e); }}}}
```

The final example shows a mapping from a value to an exception. We want to check that a method never returns a null value. First, we check using the convenience method `getReturnType` returns an object reference, then we check that the value of that reference is not null. If it is null then we throw an `AssertionFailed` exception,

```
public AssertMetaObject extends MetaObject {
    public void afterMethodExecution(IMethodExecution context)
        throws Exception {
        if (context.getReturnType() == Type.OBJECT) {
            if (context.getReturnValue() == null) {
                throw new AssertionFailed(); }}}}
```

5 Related Work

Although exceptions are an integral part of the Java language there has been little explicit attention paid to them by the Java reflection community with the exception of Garcia et. al. [4]. Garcia et al. have proposed a unified meta-level software architecture for sequential and concurrent exception handling that is described using a set of design patterns. The patterns cover: Exceptions, Handler, Exception Handling Strategy, and Concurrent Exception Handling Action. They are attempting to codify “best practice” with regard to the implementation of reflective exception handling. They have made an implementation using a custom Java VM (Guaraná [8]) which means it is non-portable. In contrast, our work is more narrow in focus but has resulted in a portable implementation.

Explicit support for exceptions has been introduced into some Java implementations of portable compile-time Java extensions for programming using advanced separation of concerns. Below we describe the approach to exceptions taken with `AspectJ`¹ [6] and `ComposeJ` [10].

¹ the version described here is 0.8

AspectJ allows programmers to use aspect-oriented programming techniques in Java. AspectJ like Kava can be used to map exceptions and values to each other. An `around` advice applied to a `receptions` pointcut can be used to implement the mapping of an exception to exception, exception to value, value to exception. This is because `around` advice selectively pre-empts the normal computation at the specified join point. AspectJ also has two new features related to exception handling. First, the advice `after throwing` allows aspects to be invoked when an exception is thrown (in Java `throw` is used to raise an exception). This allows extra code to be executed when an exception is signalled but does not allow the signalling to be overridden. This is roughly equivalent to the interception feature in Kava. Second, aspects can be woven into existing exception handler code through the use of the `handles` pointcut. This allows extra code to be invoked when an exception is handled, and it allows handling code to be overridden. This feature is not supported in our metaobject protocol as we currently choose to intervene only at the level of a method rather than within `try ... catch ... finally` clauses.

ComposeJ allows programmers to use composition filters in Java. Composition filters [1] allow messages sent and received by objects to be intercepted and manipulated. Filters can be composed with other filters to implement complex non-functional behaviour. There are different types of filters in the model, one of which has explicit support for exceptions. The `Error` filter allows predicates on base-level state to be evaluated and an exception to be raised that causes the system to halt. This allows the implementation of assertions and contracts in Java. In the current version it is not clear if signalled exceptions are considered to be message or not. If they are then other filters such as `Dispatch` could be used to implement mappings that are similar to Kava.

Kava could implement the same functionality as the `Error` filter. It cannot add behaviour to exception handlers like AspectJ although we believe that many useful extensions for dependability can be developed without that capability. In terms of implementation Kava differs from AspectJ and ComposeJ in that it is a load-time extension to Java and can be used to add non-functional behaviour to compiled code. This makes it useful for dealing with mobile or third-party code where the API may be understood but the source code might not be available.

6 Conclusions

Metaobject protocols must be “exception aware” so that they can be used to implement a wide range of non-functional requirements. Such a metaobject protocol requires two features to support the successful implementation of non-functional requirements. The first feature is the ability to intercept exceptions signalled from the base-level, and the second feature is the meta-level raising of exceptions at the base level. These two features allow the metaobject protocol to implement mappings between exceptions and values that can be used to improve the dependability of applications.

There is one reflective Java implementation that is “exception aware” but it is non-portable. There are portable extensions to Java that introduce “exception awareness” for advanced separation of concerns but these require access to source code. Our implementation in `Kava` is portable and applies reflection and load-time. This allows `Kava` to be used for a wide range of applications such as mobile code or third-party code.

Acknowledgements

We would like to acknowledge the financial support of the ESPRIT projects: MAFTIA project (IST-1999-11583), and DSOS project (IST-1999-11585).

References

1. M. Askit, L. Bergmans, and S. Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In *ECOOP*, volume LNCS 615, pages 372–395. Springer-Verlag, 1992.
2. A. Burns, S. Mitchell, and A. J. Wellings. Mopping up Exceptions. In *ECOOP’98 Workshop on Reflective Object-Oriented Programming and Systems*, pages 365–366, 1998.
3. Christophe Dony. Exception Handling and Object Oriented Programming : Towards a Synthesis. In *Proceedings of ECOOP/OOPSLA’90*, pages 322–330, Ottawa, Canada, 1990.
4. Alessandro F. Garcia, Delano M. Beder, and Cecilia M. F. Rubira. Unified Meta-Level Software Architecture for Sequential and Concurrent Exception Handling. *The Computer Journal (Special Issue on High Assurance Systems Engineering)*, 2001.
5. M. Hof, H. Mossenbock, and P. Pirkelbauer. Zero-Overhead Exception Handling Using Meta-Programming. 1338:423–431, 1997.
6. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffery Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP 2001*, volume LNCS 2072, pages 327–353, Budapest, Hungary, 2001. Springer-Verlag.
7. B. Meyer. Design by Contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice-Hall, 1991.
8. Alexandre Oliva and L. E. Buzato. The Design and Implementation of Guaraná. In *Usenix COOTS*, pages 203–216, San Deigo, California, USA, 1999. Usenix.
9. Ian Welch and Robert Stroud. Kava – Using Byte-Code Rewriting to Add Behavioral Reflection to Java. In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, pages 119–130, San Antonio, Texas, 2001.
10. J. C. Wichman. ComposeJ: The Development of a Preprocessor to Facilitate Composition Filters in the Java Language. Master’s thesis, University of Twente, 1999.