A Study on Exception Detection and Handling Using Aspect-Oriented Programming

Martin Lippert and Cristina Videira Lopes

Xerox Palo Alto Research Center Technical Report P9910229 CSL-99-1, Dec. 99

A Study on Exception Detection and Handling Using Aspect-Oriented Programming*

Martin Lippert
Computer Science Department, SE Group
University of Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg, Germany
lippert@acm.org

Cristina Videira Lopes
Computer Science Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304, USA
lopes@parc.xerox.com

ABSTRACT
Aspect-Oriented Programming (AOP) is intended to ease situations that involve many kinds of code tangling. This paper reports on a study 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.

We found that AspectJ supports implementations that drastically reduce the portion of the code related to exception detection and handling. In the best case scenario, we were able to reduce that code by a factor of 4. We also found that AspectJ provides better support for different configurations of exceptional behaviors, more tolerance for changes in the specifications of exceptional behaviors, better support for incremental development, better reuse, automatic enforcement of contracts in applications that use the framework, and cleaner program texts. We also found some weaknesses of AspectJ that should be addressed in the future.

Keywords: Exceptions, Contracts, Frameworks, Aspect-Oriented Programming

1 INTRODUCTION

The handling of exceptions in large software systems can consume a remarkable amount of development resources. Exceptions must be thought of throughout the whole development cycle and dealing with them is hard for many different reasons.

Exceptions and How They Appear in Programs

In early stages of development, developers must identify what the application should and should not do. For example, computing a distance given one parameter in meters and another parameter in feet results in a meaningless value, an error. Trying to compute such a value can be considered an "exceptional situation." If developers anticipate this situation, they can decide what to do if it ever occurs (maybe convert one of the values), or prevent it from occurring. This is part of the human effort of deciding exactly what is "normal" and what is "exceptional." Making wrong decisions or failing to identify exceptional situations can lead to catastrophic results (e.g. the notorious Mars climate orbiter), to be blamed on the analysts of the system. [1]

The normal and exceptional behaviors of a system that were identified in the early stages of development are then implemented in a program. This translation is a difficult and mostly manual process that is exposed to yet another layer of human errors, this time to be blamed on the programmers.

Focus and Scope of the Paper

This paper focuses on the process of combining the application's normal and exceptional behaviors in the program texts.

The study in the paper 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 this paper lead to programs with fewer implementation errors and that can be changed easier, is still an open research topic that will require serious usability studies as AOP matures. This paper targets the step before those usability studies can be done. It shows that there are some advantages in using as-

* This work was supported in part by the Defense Advanced Research Projects Agency under contract number F30602-97-C-0246.

[1] For the purpose of this paper, an *exception* is a behavior of the system indicating that the operation in process cannot be successfully completed, but from which other parts of the system can try to recover or chose to ignore. An *error* is an unwanted behavior of the system, from which the system by itself cannot recover. Exceptions may result in errors. Exceptions and errors can be generated by the system or by the underlying machine.

Having established these distinctions, we will from here on use the word *exception* throughout the paper, even though, in cases such as the violation of contracts, the exceptional behaviors result in errors.

pects to express the detection and handling of exceptions. It also exposes some limitations of AspectJ that should be addressed before doing the usability studies.

2 MOTIVATION AND SYNOPSIS OF THE STUDY

We took an existing framework written in Java and partially reimplemented it with AspectJ. This section frames this study by presenting the problem that it focused on. It then describes the materials used: AspectJ and the JWAM framework.

Some Background

A lot of work in programming languages has had the goal of providing better support for the detection and handling of exceptions, as they have been defined in the previous section. The work in type systems is, in many ways, one such example. Contracts, introduced by Hoare [13] are another 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. This concept evolved over the years into the well-known design methodology *Design by Contract* [21]. Some programming languages, of which Eiffel is the reference example, provide constructs to support contracts. [12] 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 errors, 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.

The Problem

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 exceptions. Figure 1 illustrates this issue with a small Java example. The example shows parts of a Game class, which is part of a distributed game. The code related to exceptional behaviors is underlined.

This tangling is, in part, a consequence of the programming language and, in part, a consequence of 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

```
class Game implements RemoteGame {
  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;
  }
  /**
   * readStatusFromFile uploads the game status from a file
   */
  protected void readStatusFromFile(String fname) {
    Contract.require(fname != null, this);
    try {
      ObjectInputStream in=new ObjectInputStream(new FileInputStream(fname));
      _items = (Map) in.readObject();
      _missingCards = (Map) in.readObject();
      _setsOfNumbers = (Map) in.readObject();
      in.close();
    }
    catch (Exception e) {ErrorLog.print("Error at: readStatusFromFile: " + e); }
  }
}
```

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 preconditions.

an operation is the same abstraction of "how to detect and react to exceptions" in that operation. This implicit unification has a number of problems:

- reuse: subclasses cannot reuse what a superclass's method does and redefine its own reaction to that method's exception handling (or vice-versa) without redefining the whole method.
- evolution: what a method does cannot be used in a context with different application exceptions specifications without changing the method
- documentation: the program texts can become seriously cumbersome, making it hard to understand what exactly a method is supposed to do
- loss of abstraction: behavior specifications of the kind "all methods that call the registry server must be prepared for network failures, and should retry calling it 3 times before giving up" cannot have a correspondent single abstraction in the program text. Therefore the implementation of this kind of specifications is scattered throughout the code, making its consistent maintenance hard

Aspect-Oriented Programming and AspectJ

Aspect-Oriented Programming was first proposed in [15] as a programming technique for modularizing concerns that cross-cut the basic functionality of programs. Exception handling was referred to as one of those cross-cutting concerns, and the paper suggested that it should be possible to achieve a relative separation between the functional code and the exception handling code.

AspectJ [19] 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 examples throughout the paper. Other features of AspectJ will be explained whenever necessary. For detailed information, we refer the reader to the documentation that can be found on the web [2]. This work used AspectJ version 0.4 beta 6.

The JWAM Framework

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 [14]. This work used JWAM version 1.4.

JWAM contains more than 600 classes and interfaces as well as about 150 test classes. The framework is structured in layers, as described in [3], and it includes some libraries that emulate "language extensions" such contracts and a metaobject protocol. This study targeted the whole framework.

3 REENGINERING JWAM WITH ASPECTJ

We have studied the use of AOP in the framework by trying several different aspect designs. We found that each one had advantages and disadvantages with respect to the others. One of them was clearly the best with respect to LOC; another one was clearly the best with respect to the "unplugability" of the exceptional behaviors. Section 4 contains an analysis of the results.

This section explains the work involved in designing and implementing some exceptional behaviors of the framework using AspectJ. It starts by giving an overview of what those behaviors are. It then presents some examples of reimplementations.

Overview of the Exceptional Behaviors in JWAM

All 600 classes of the framework had been originally designed and implemented using the Design by Contract methodology, briefly mentioned in section 2. The detailed description of this methodology falls out of the scope of this paper, and we refer the reader to [21].

JWAM uses contracts to ensure that callers do not misuse the methods, and that the methods' implementations preserve some of their basic specifications. As the next section will show, the designers of JWAM have made the decision that a broken contract always results in a runtime error 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. Tables 1 and 2 summarize the types of exceptions that are

caught in JWAM and the frequency by which they are caught. Table 1 shows the exceptions that are thrown by the Java libraries and that JWAM handles. Table 2 shows the exceptions that are defined and thrown in the framework and that are handled in it. The handling of these three types of exceptions shown in Table 2 is concentrated in the test classes, which are also part of the framework. For example, ContractBrokenException is caught and handled in a class that tests if the contracts infrastructure is working properly.

Type of exception caught	Number of catch statements
Exception	77
SQLException	46
IOException	38
RemoteException	29
NumberFormatException	22
ClassCastException	14
ClassNotFoundException	11
IllegalAccessException	8
SecurityException	5
InterruptedException	5
InstantiationException	5
FileNotFoundException	4
MissingResourceException	4
ParseException	4
Throwable	3
InvocationTargetException	3
NoSuchMethodException	2
NullPointerException	2
DocletAbortException	2
IntrospectionException	1
MalformedURLException	1
IllegalArgumentException	1

Table 1. Java library exceptions caught inside JWAM.

Type of exception caught	Number of catch statements
ThingNotAvailableException	12
ContractBrokenException	11
ReflectionException	4

Table 2. JWAM exceptions caught inside JWAM.

Design by Contract

JWAM implemented its own support for design by contract. It did so with a very simple contract package that exports the class `Contract` and that defines and uses the runtime exception class `ContractBrokenException`. The `Contract` class is shown in Figure 2. This straightforward implementation reflects a number of decisions made by the designers of JWAM. According to that design, when any contract is broken, a runtime error occurs in the thread where that contract was broken, because `ContractBrokenException` is never caught.

JWAM sees the use of contracts in the framework, and in the applications that use the framework, as an *enhancement* of the design of the classes – something not mandatory, but useful for detecting programming errors.

When contracts are used in JWAM, their application should follow two guidelines. (1) The class's contracts are documented in comments preceding each of its methods and constructors. Those comments, once run through Javadoc, document what the contracts are for that class, so that the programmers use it appropriately. (2) Each method's pre- and post-conditions are checked by inserting calls to the methods of the `Contract` class in the beginning and in the end of the method. Figure 3 illustrates the use of contracts in the framework, before aspects were applied.

When reengineering the exceptional behaviors of JWAM, contracts were immediately targeted as aspects. Besides contracts being related to the exceptional behaviors, there was another symptom for *aspectification*: the documented optional nature of contracts at runtime suggested that sometimes developers may want to configure classes with their 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.

Section 4 elaborates on the lessons learned with implementing contracts with aspects. For now, we simply describe what was involved in their reengineering. We'll do so with

```
public class Contract  {
  static void require(boolean precondition,
                      Object contractor)
         throws ContractBrokenException {
    if (!precondition)
      throw new ContractBrokenException(
        "precondition of" + contractor +
        " violated");
  }
  static void ensure (boolean postcondition,
                      Object contractor)
          throws ContractBrokenException {
    if (!postcondition)
      throw new ContractBrokenException(
        "postcondition violated in " +
        contractor);
  }
}
```

Figure 2. The Contract class in JWAM

```
import de.jwam.system.contract.Contract;
/**
 * This example is adapted from the
 * cookbooks of the JWAM framework.
 * (simplified version of the Account class)
 */
public class Account {
  /**
   * @require owner != null &&
   *          owner.length() > 0
   * @require accountNumer > 0
   */
  public Account(String owner,int accNo){
    Contract.require(owner != null &&
                     owner.length() > 0, this);
    Contract.require(accNo > 0, this);
    _owner = owner;
    _accNo = accNo;
  }
  /**
   * @require amount > 0
   */
  public void deposit (float amount) {
     Contract.require(amount > 0.0, this);
     _balance = _balance + amount;
  }
 // ... other methods omitted
}
```

Figure 3. Use of contracts, before aspects.

the example of the Account presented in Figure 3. Figure 4 illustrates one type of reengineering that was made for the classes in the framework. In this example, the object class Account contains only "what an account does", and the aspect class AccountContract contains its contract.

The design of contracts as aspects opened some new possibilities for implementing contracts. Two of those possibilities are discussed next.

Crosscutting Specifications

We found that all methods of JWAM that return an object had the following post-condition: `@ensure result != null`. This reflected a simple but powerful specification that cannot be stated explicitly using design by contract, namely that whenever methods return an object, they should really return *some* object. Aspects make it possible to capture that specification more concisely. For example, a contract aspect contains the following part:

```
crosscut methodsReturningAnObject():
         * & Object *(..);
static advice methodsReturningAnObject() {
  after {
    Contract.ensure(thisResult != null,
                    thisObject);
  }
}
```

AspectJ details. **thisResult** is a special AspectJ variable that denotes the return value of the methods.

This kind of situation also occurred for pre-conditions on

```
public class Account {
 /**
   * @require owner!=null &&
   *          owner.length()>0
   * @require accountNumer > 0
   */
  public Account(String owner, int accNo) {
    _owner = owner;
    _accNo = accNo;
  }
  /**
   * @require amount > 0
   */
  public void deposit (float amount) {
    _balance = _balance + amount;
  }
  // ... other methods omitted
}
```

```
import de.jwam.system.contract.Contract;
class AccountContract  {
  static advice Account &
                new(String s, int n){
    before {
      Contract.require(s!=null && s.length > 0,
                       thisObject);
      Contract.require(n > 0, thisObject);
    }
  }
  static advice Account & deposit(float f){
    before {
      Contract.require(f > 0.0,thisObject);
    }
  }
  //… other advice omitted
}
```

AspectJ details. This example uses what AspectJ calls *anonymous* crosscuts – note the advice apply to sets of events that are not named, but are pointed out very concretely. For example Account **&** new(String s, int n) is used anonymously; it denotes the instantiations of Account objects by the invocation of a constructor with that signature. **thisObject** is a special AspectJ variable that denotes, in this case, an Account object. It is used instead of **this**.

Figure 4. A class and its contract aspect.

input parameters. All methods getting objects as parameters had pre-conditions of the kind "make sure those object references are not null."

Contracts for Interfaces

The JWAM framework defines a number of different interfaces. These interfaces can be implemented in many different ways by the applications that use the framework. Some of these implementations are included in the framework as default implementations.

The concept of contracts applies not only to classes, but it applies especially to interfaces. But without aspects, the contract for an interface is necessarily reduced to the documentation part of the contract (i.e. the comments). The implementation of the contract for an interface is of the

responsibility of each class that implements that interface.

Using AspectJ, we were able to implement contracts for interfaces. In order to save space in the paper, we do not present an illustration of this. But consider the piece of aspect code shown in Figure 4. If `Account` is not a class but an interface, the contract aspect still works as intended. The semantics of AspectJ, in that case, is that the aspect affects every implementation of the interface `Account`.

Contract aspects defined for interfaces inside the framework can be automatically used by all classes that implement those interfaces.

Handling of Library Exceptions

Besides contracts, we also targeted the five most frequently caught exceptions as candidates for *aspectification* (see Tables 1 and 2). 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 (more in Section 4).

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 this limitation consisted in 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.

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 reminder of this subsection describes the reengineering work that was done for the `RemoteException`.

The framework has 2 different types of remote objects named `RegistryServer` and `RMIMessageBroker`. For each of these, the framework defines a proxy class that the clients are supposed to use. So, the calls from clients to remote objects should always go through these JWAM proxies. These proxy classes had been introduced for the purpose of encapsulating the distribution aspects as much as possible. The handling of the `RemoteException` was concentrated in those proxy classes.

```
public class LogAndIgnoreRemoteException {
  crosscut throwsRemoteException():
    RegistryServerProxy & * *(..) |
    RMIMessageBrokerImpl & private * *(..);

  static advice throwsRemoteException() {
    catch (RemoteException e) {
      ErrorLog.print("remote call failed in: "
                     + thisMethodName +":" + e);
    }
  }
}
```

```
public class RegistryServerProxy {
 // 23 public methods. We show one of them:
  public void removeSniffer(String registrarname,
                            DvIdentificator id){
    _registryServer.removeSniffer(registrarname,
                                  id);
  }
}
```

AspectJ details. "`* *(..)`" denotes all methods, regardless of their parameter types. **catch** is another kind of advice that refers to when the given message invocations return an exception of the given type. **thisMethodName** is a special AspectJ variable that denotes the name of the method that is being executed.

Figure 5. RMI exception aspect and a class to which it applies.

We found that there was only 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. It is likely that, as JWAM evolves, there will be some effort put into doing a more serious recovery from this exception. When that happens, and given the implementation without aspects, developers will have to manually modify those 29 statements, so that they implement the new handler specifications. Figure 5 illustrates the kind of reengineering that was done for RMI exception aspects.

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.

```
abstract public class AbstractLogAndIgnore {

  abstract crosscut methods();

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

Figure 6. Abstract exception handler.

Unlike contract aspects, these exception handling aspects are not, and should not be, "unpluggable", but they can be mutually "replaceable."

Plug-and-Play Exception Handlers

AspectJ supports the notion of abstract crosscuts. An abstract crosscut is a set of events that is left undefined but that can be advised. Using this feature, we were able to design and implement exception handlers that can be plugged into many different applications, and not only to JWAM. Figure 6 shows one of those handlers, namely the one that corresponds to the reaction "log and ignore."

Abstract aspects are used by applications through subclassing. For example, the abstract aspect in Figure 6 can be used in JWAM – resulting in exactly the same behavior as that of Figure 5 – by concretizing the abstract crosscuts:

```
public class JWAMRemotExceptionHandler
       extends AbstractLogAndIgnore {

  crosscut methods():
    RegistryServer & * *(..) |
    RMIMessageBrokerImpl & private * *(..);
}
```

The `AbstractLogAndIgnore` aspect can be used by many other frameworks and applications. They simply need to subclass it and to bind the crosscut `methods` to the specific events to which the handler should be hooked.

4 RESULTS AND LESSONS LEARNED

About 11% 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."

Quantitative results

Table 3 presents an overview of the measurable results of this work. 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. The details are presented next.

	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 3. Overview of the results. The numbers for the implementation with aspects reflect the design that maximized the benefits in terms of LOC (best case scenario).

Detection of Contract Violations

Without aspects, the framework contains 375 post-conditions of the form "`result != null`". That is, 56% of post-conditions are redundantly reimplementing the same specification! With aspects, that number can be reduced to 1. In using an aspect for implementing this post-condition we also need the hooks into the methods that return an object. AspectJ 0.4 does not provide a designation mechanism for expressing "all methods that return an object reference" concisely. We had to list them one by one.

The framework also contains 1510 pre-conditions of the form "`arg != null`". With AspectJ, that number can be reduced to 10. This is greater than 1, because in the version of AspectJ that we used, there is no elegant way of expressing this pre-condition as concisely as it is specified. The best we could do was to make a *cut* at the methods according to the number of reference parameters. So, for example, methods with two reference parameters can be grouped as:

```
abstract crosscut methTwoRefParam(Object a,
                                  Object b);
static advice methTwoRefParam(Object a,
                              Object b) {
  before {
    Contract.require(a != null, thisObject);
    Contract.require(b != null, thisObject);
  }
}
```

This is not ideal, but it's also not as bad as it may seem. The semantics of crosscuts in AspectJ ensure that the crosscut `methTwoRefParam`, for example, can be used to hook into any methods with two reference parameters, independent of the parameters' positions in the method's parameter list. But, again, AspectJ 0.4 does not provide a designation mechanism for expressing "all methods that take two object references" concisely. We had to list them one by one.

The numbers for the implementation with aspects in Table 3 reflect one particular design of the contract aspects,

among the many that we have studied. This design extracted only the two specifications "result != null" and "argument != null" into separate aspects, and left all other pre and post-conditions inside the classes. This design maximized the benefits in terms of LOC. An alternative design that we studied was to adhere to the rule of one contract per class. In that case LOC is considerably higher, due to the extra LOC in each aspect. In the worst case scenario, there are no benefits in terms of LOC, but there are also no disadvantages – the LOC with aspects is of the same order of magnitude as the original implementation (12.5% of total LOC).

Another point we explored was contract aspect implementations for the interfaces of the framework. As described in section 3, using AspectJ, we were able to do that. In doing that, the applications that use the framework do not need to reimplement the contract checking code defined for those interfaces. This benefit is most important for so called "hotspots" inside the framework [23, 18]. A hotspot is a class of the framework that is subclassed in the applications, or an interface of the framework that needs to be implemented in the applications.

We studied the hotspots of JWAM, concentrating on the three most implemented and overridden methods. We also studied two applications of JWAM. The first application is an order handling application containing more than 200 classes. The second one is a note analysis application with about 100 classes.

method name	# of redefinitions inside the framework	Reduction in contract checking using aspects (inside the framework)	# of redefinitons inside the framework + two applications	Reduction in contract checking using AOP (inside the framework + two applications)
ThingDesc	33	97.0%	69	98.6%
CreateFP	28	96.4%	52	98.1%
CreateIP	28	96.4%	52	98.1%

The numbers shown above are only for the most reused methods of the framework, which are not that many. For the most part, the framework is used via black-box-reuse or via white-box-reuse and default implementations.

Exception Handling

Table 4 shows the results for the five most frequently caught exceptions in JWAM. The number of different reactions to each type of exception (i.e. the third column) corresponds directly to the number of different aspect handlers, where each aspect handler has exactly one catch statement.

Type of exception	**Number of original catch statements**	**Number of different reactions**	**Reduction in catch statements using aspects**
Exception	77	7	90.9%
SQLException	46	2	95.6%
IOException	38	3	92.1%
RemoteException	29	1	96.5%
NumberFormat-Exception	22	1	95.4%

Table 4. The five most frequently caught exceptions.

Summary

These numbers demonstrate that the use of aspects can significantly reduce the amount of code that deals with exception detection and handling. Using the design that maximized the benefits in terms of LOC, we were able to cut that code to 1/4 of its original size. The cut affected the redundant information that existed in the original code, and that was being imposed by the programming language. Less redundant code usually decreases the opportunity for programming errors. The worst case scenario with aspects is not much worse than the original implementation.

Qualitative results:
The following list describes some of the advantages we found of using aspects.

Better support for different configurations. The contract aspects can be plugged and unplugged as necessary at compile-time. Some languages (e.g. Eiffel) provide a built-in mechanism to support a similar thing, but Java still does not support that. Moreover, the unpluggability of aspects is not limited to what it can do for contracts. For example, using aspects we can also configure exception catching and handling at compile-time. We can have an aspect for making only minimal exception handling, such as catching the generic type `Exception` everywhere, and logging it, and we can have another aspect (or set of aspects) that distinguishes among different types of exceptions and reacts differently to them. The application can be configured with either of them.

Better tolerance for changes in the specifications. The implementation reflects the specifications more directly. For example, "methods that return an object should return *some* object" can be implemented by one single aspect module, even though that module affects many other parts in the code. Another example: "the reaction to the RemoteException should always be *log and ignore*" can also be implemented by one single aspect module. These better abstractions in the code are more robust to changes of the specifications. For example, if the specifications for handling the `RemoteException` change, programmers will have to

modify only one block of code, as opposed to 29.

Better support for incremental development. The previous points have one important consequence: in situations when there are several iterations of analysis/design/ implementation, the tolerance to changes in the specifications of the exceptional behaviors is particularly important. For example, the developers can quickly prototype an implementation that concentrates on the normal behaviors, by plugging in a handler aspect for the generic `Exception` type everywhere. Later, other handler aspects that perform more sophisticated recoveries can replace that aspect.

Better reuse. By extracting the exception detection and handling from the classes into separate aspects, we are able to reuse the contracts along a type hierarchy. For exception handling, we can implement abstract handlers that can be plugged into many different applications. This is a promising result that opens the possibility of having libraries of aspects – generic implementations of utilities that crosscut many classes.

Automatic enforcement of contracts in applications that use the framework. The previous point has one important consequence: contract aspects ensure that the contracts will be implemented by classes that use the framework through white-box reuse (subclassing or implementing an interface). There is less opportunity for implementation errors.

Less interference in the program texts. The coding of the normal and exceptional behaviors of the classes can be separated into different modules. This makes it easier to understand what those two behaviors really are, because their implementations do not collide in the same text.

The following list describes some of the weaknesses of AspectJ 0.4 and suggests some directions for future work.

Insufficient support for reconstruction of the local effects. The separation of normal and exceptional behaviors is a good thing, but in many occasions we felt the need to see the "whole picture" of a method implementation. Having to browse through different files and having to read what the aspect crosscuts were did not scale beyond a few dozen classes and half a dozen aspects. There needs to be some kind of support for reconstructing all the local effects that aspects have on classes.

Inadequate semantic support for inheritance of pre-conditions. AspectJ provides one semantics for the composition of aspects along a type hierarchy, roughly an "AND" semantics. That semantics conflicts with the standard composition of contracts along type hierarchies, which is, roughly, an "OR" for pre-conditions and an "AND" for post-conditions (subclasses may define weaker pre-conditions and stronger post-conditions). It is possible to work around this problem, but the result is not elegant. It should be noted, however, that this situation did not occur in JWAM – it was something that came up in discussing contract aspects.

Designation mechanisms still not expressive enough. We really needed support for advising message sends (caller-side events) in order to be able to catch exceptions deeper than method boundaries. We also needed a more expressive mechanism for denoting crosscuts of events. For example, "all methods with at least two object parameters."

5 RELATED WORK

A considerable amount of work has been done to enhance object-oriented programs with exception detection and handling mechanisms. [5, 9, 21], just to name a few, describe different flavors of those mechanisms for object-oriented systems. Other papers present exception handling mechanisms for specific purposes like database-intensive information systems [5], component-based real-time software [17], fault-tolerance [8, 20]. [22] presents a rich overview of literature about exception handling in software systems. Those papers deal with the basic mechanisms to enable an object-oriented program to throw and catch exceptions. This paper stands on all that work. Our study focuses on the new mechanisms introduced by AspectJ that open new possibilities for the composition of normal and exceptional behaviors.

There have been several proposals for integrating contracts in Java. Some of those proposals have an AOP flavor. The work described in [7] supports pre- and postconditions with a mixture of a library, the jContractor, and the use of a design pattern. Kramer's iContract [16] is a preprocessor-based tool to generate the contract checking code out of special tags like @pre or @post inside the method documentation. We think aspects provide a better story and solution for capturing all kinds of exceptional behaviors, including the violation of contracts.

That the use of exception handling can affect the reusability of code is described in [10]. They reported, that "concurreny and exception handling have substantial impact on software structures and they are important factors affecting reuse". To enable a higher factor of reuse they removed the exception handling code from the libraries. They separated the library code from the exception handling code. However, their study did not provide solutions for composing the exceptional behaviors with the classes. AOP, either through AspectJ or in some other form, could be that solution.

Walker et al. have made an initial usability study of aspect-oriented programming [26] using an earlier version of AspectJ. Their study had a different purpose and scope than this study. They measured how programmers perceived programs written AspectJ vs. programs written in plain Java. They did not focus on exceptions in particular. Our experience with our study reinforces some of their findings, namely that the aspects should have a well-defined scope of effect on core functional code. The contract aspects as well

as the exception handling aspects have a clear defined scope that can be ensured using abstract and concrete aspects in combination.

6 CONCLUSION

We have studied how aspects ease the tangling related to exception detection and handling. We took an existing framework, JWAM 1.4, and partially reimplemented it using AspectJ 0.4. We tried several aspect designs that targeted the contracts and the handling of the five most frequently caught exceptions.

We found that AspectJ supports implementations that drastically reduce the portion of the code related to exception detection and handling. In the best case scenario, we were able to reduce that code by a factor of 4. That reduction reflects cuts in the redundant information that was being imposed by Java. We also found that AspectJ provides better support for different configurations of exceptional behaviors, more tolerance for changes in the specifications of exceptional behaviors, better support for incremental development, better reuse, automatic enforcement of contracts in applications that use the framework, and cleaner program texts. The study also exposed some weaknesses of AspectJ that should be addressed in the future.

ACKNOWLEDGEMENTS

The authors would like to thank PARC's Computer Science Lab, where most of this work was done. Special thanks to the members of the AspectJ team – Gregor Kiczales, Jim Hugunin, John Lamping, Eric Hilsdale, Mik Kersten and Chandra Boyapati. This work would not have been done without them. Thanks also to Bill Griswold for his comments on an earlier version of this paper.

REFERENCES

1. ANSI: *American National Standard Programming Language PL/I*, ANSI X3.53-1976. American Natioal Standards Institute, New York, 1976.
2. AspectJ Web Site, <http://www.aspectj.org/>.
3. D. Bäumer, G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, H. Züllighoven: Framework Development for Large Systems, in *Communications of the ACM*, Vol. 40, No. 10, 1997.
4. A. P. Black: *Exception Handling: The Case Against*, Technical Report 82-01-02, Department of Computer Science, University of Washington, 1982. Reprinted in May 1983.
5. A. Bordiga: Language Features for Flexible Handling of Exceptions in Information Systems, in *ACM Transactions on Database Systems*, Vol. 10, No. 4, pp. 565-603, Dec. 1985.
6. A. Bordiga: Exceptions in object-oriented languages, in *ACM SIGPLAN Notices*, Vol, 21, No. 10, pp. 107-119, October 1986.
7. J. Bruno, U. Hölzle, M. Karaorman: *jContractor: A Reflective Java Library to Support Design By Contract*, Technical Report TRCS98-31, Department of Computer Science, University of California, Santa Barbara, December 1998.
8. F. Cristian: Exception Handling and Software Fault Tolerance, in *IEEE Transactions on Computers*, Vol. c-31, No. 6, pp. 531540, June 1982.
9. C. Dony: Exception Handling and Object-Oriented Programming: towards a synthesis, in *Proceesings of OOPSLA/ECOOP '90, SIGPLAN Notices*, Vol. 25, No. 10. ACM Press, October 1990.
10. M. F. Dunn, J. C. Knight: *Software Resuse In An Industrial Setting: A Case Study*, IEEE, 1991.
11. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995.
12. J. B. Goodenough: Exception Handling: Issues and Proposed Notation, in *Communications of the ACM*, Vol. 18, No. 12, 1975, pp. 683-696.
13. 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.
14. JWAM framework Web Site, <http://www.jwam.de/>.
15. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Longtier, J. Irwin: Aspect-Oriented Programming, in *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, Finland, Springer-Verlag, 1997, pp. 220-242.
16. R. Kramer: iContract – The Java Design By Contract Tool, in *Proceedings of Tools 26 - USA '98*, IEEE Computer Society.
17. J. Lang, D. B. Stewart: A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology, in ACM Transactions on Programming Languages and Systems, Vol. 20, No. 2, pp. 274-301, March 1998.
18. R. E. Johnson, B. Foote: Designing Reusable Classes, in *The Journal of Object-Oriented Programming*, Vol. 1, No. 2, 1988, pp. 22-35.
19. C. Lopes, G. Kiczales: Recent Developments in AspectJ™, in *ECOOP '98 Workshop Reader*, Springer-Verlag, 1998.
20. P.M. Melliar-Smith, B. Randell: Software Reliability: the Role fo Programmed Exception Handling, in *Proceedings of ACM Conference Language Design for Reliable Systems, SIGPLAN Notices*, Vol, 12, No. 3, pp. 95-100, March 1977.
21. B. Meyer: *Object-Oriented Software Construction*, Prentice Hall, New Jersey, 1997.
22. A. Oberweis, W. Stucky: Exception handling in software systems: a literature survey, (in german), in *Wirtschaftsinformatik*, Vol. 33, No. 6, December 1991.
23. W. Pree: *Design Patterns for Object-Oriented Software Development*, ACM Press, Addison-Wesley, 1994.
24. D. Riehle, H. Züllighoven: A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor, in: J. O. Coplien, D. C. Schmidt (eds.): *Pattern Languages of Program Design*, Reading, Massachusetts, Addison-Wesley, 1995.
25. R. W. Sebesta: *Concepts of Programming Languages*, Third Edition, Addison-Wesley, Reading, Massachusetts, 1996.
26. R. J. Walker, E. L. A. Baniassad, G. C. Murphy: An Initial Assessment of Aspect-oriented Programming, in *Proceedings of 21st International Conference on Software Engineering (ICSE)*, IEEE Computer Society Press, 1999.