Regaining Control of Exception Handling

Martin P. Robillard and Gail C. Murphy
{mrobilla,murphy}@cs.ubc.ca

Technical Report Number TR-99-14
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC
Canada V6T 1Z4

December 1st, 1999

ABSTRACT
Just as the structure of the normal operations of a system tends to degrade as the system evolves, the structure of exception handling also degrades. In this paper, we draw on our experience building and analyzing the exception structure of Java programs to describe *why* and *how* exception structure degrades. Fortunately, we need not let our exception structure languish. We also relate our experience at regaining control of exception structure in several existing programs using a technique based on software containment.

Keywords
Exception Handling, Program Structure, Software Structure, Error Handling, Design.

1 INTRODUCTION

To help manage complexity, software developers use various types of structure to organize the source code for a system. Programming language constructs, such as procedures and procedure calls, provide a means of expressing fine-grained structure. Architectural styles [14], such as layers or pipes and filters, provide a means of describing coarse-grained structure. Selecting and implementing appropriate structures for a system can provide many benefits; for example, the source may be more modifiable or reusable as a result.

Despite the best intentions of software developers, the structures desired for a system tend to degrade over time. The addition of a new feature into the system may introduce additional calls between procedures that increase the coupling in the system. The need for improved performance may lead to violations of a desired architectural style.

The degradation of structure in software systems as they evolve has been studied for several years [3]. Overwhelmingly, the work on structure and structural degradation has focused on those parts of the source code that represent the normal flow of control in the system. Source code written in modern programming languages (e.g.,Ada [1], C++ [16], Java [9]) also includes exception handling code dedicated to representing actions to take in unanticipated or undesired cases. Not surprisingly, it has been our experience that the structure of exception handling code in a system also tends to degrade as the system evolves. Although the structure of both kinds of code degrade, the kind of degradation is quite different.

In this paper, we draw upon our experience at building and analyzing the exception structure of Java programs to describe *why* and *how* exception structure degrades. We then describe an approach that we have found helpful to regain control of the exception structure in an existing system. This approach is based on a proposed method for designing fault-tolerant Ada systems [10]. We discuss the outcome of applying this approach to three different existing Java systems.

We begin by describing problems we encountered when evolving the exception structure of a program analysis tool (Section 2). These problems occurred despite the care that was taken during design to make the exception structure of the program evolvable. Next, we draw on our experience at analyzing the exception structure of existing programs to propose a description of why and how the problems occur. Then, we describe the technique we have used to improve exception structure (Section 4), and describe the application of this technique to three systems (Section 5). Finally, we survey related work on exception structure design (Section 6), and summarize the paper (Section 7).

2 AN EXAMPLE OF EXCEPTION STRUCTURE DEGRADATION

To illustrate how the exception structure of a program can degrade, we describe the evolution of the Jex excep-

Table 1: Component-level exception specification

Component Name	Exceptions Raised
Controller	None
TypeSystem	TypeException
JexLoader	ParseException IOException
AST	AnalysisException ExceptionGenerationException
Parser	ParseException

tion analysis tool [13]. The Jex tool is a static analyzer that extracts exception flow information from Java programs. Specifically, Jex determines the list of exceptions that can be raised at every program point and presents this information in the context of the exception handling structure in the program.

Jex consists of five components (Figure 1). Each component consists of a set of highly related classes that are accessed through a restricted interface. The arrows in the figure represent calls between the components.

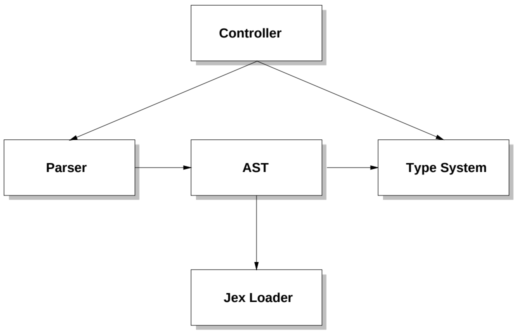

Figure 1: The Architecture of Jex

The tool was designed with exception handling in mind. Care was taken to design a meaningful and restricted set of exceptions for each component. Components were to signal their failures using mostly user-defined exceptions (e.g., `TypeException` for the Type System component). Table 1 shows the initial specification for the exceptions raised at the component level.

As the Jex tool evolved, we realized that the simplistic model above did not correspond to the actual behaviour of exceptions in the application. As one example of the actual behaviour, Figure 2 represents the propagation paths, at the class-level, leading to the entry point of Jex for a *single* exception, `ClassCastException`. Obviously, reasoning about the behaviour of the system with respect to this exception is difficult! Figure 3 presents a stylized representation of the flow of *all* exceptions in Jex at the component level. The width of each arrow is roughly proportional to the number of different exceptions flowing on the path; the numbers are also indicated on the figure. Clearly, there are more exceptions flowing in the system beyond component boundaries than were anticipated or desired.

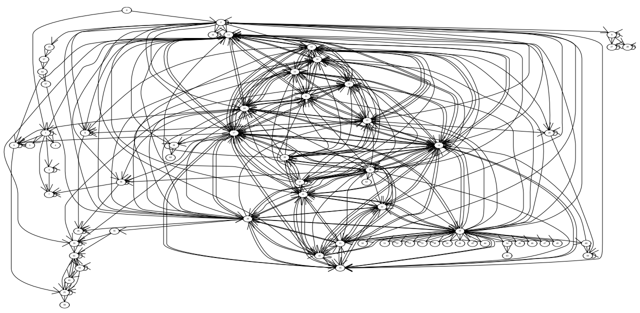

Figure 2: Propagation graph of `ClassCastException`

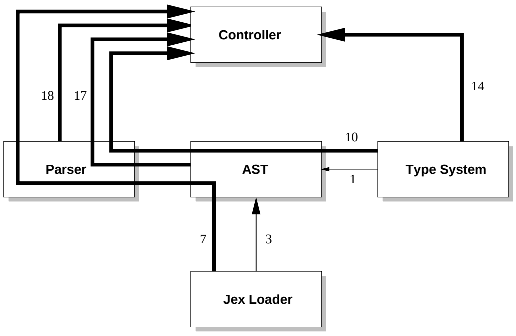

Figure 3: Exception Propagation in Jex

Problems with the exception structure

To gain insight into the degradation of the exception structure in Jex, we studied those potential exceptions that did not conform to our initial design. This study suggests several categories of problems that lead to the degradation.

Our exception design was too general. Our initial exception design was based on a component raising a small number of exceptions. The practical consequence of this design is that other exceptions that might arise in a component are re-mapped to the small set of exceptions defined for the component. For example, the implementation of the Type System component re-mapped the exceptions that could occur in classes contributing to the Type system to a generic `TypeException`.[1] As it turned out, this re-mapping was too general because it could

[1] The re-mapping did not include *unchecked* exceptions.

represent both some IO problems related to initializing the type system component, as well as lookup problems related to using the component. Since this distinction was necessary to write handlers that would provide users with useful feedback, the component was modified to propagate the `IOExceptions` raised during its operations. However, during the initialization of the component, it turns out that the `TypeExceptions` explicitly raised by the Type System and the `IOExceptions` that could propagate through carried the same semantic. As a result, the handling of both types of exceptions was identical. Clearly, there existed some confusion with determining when the semantics of the exceptions were the same and when they differed.

Conformance to the exception design caused a loss of information. In the case described above, we increased the set of exceptions designed to be raised by a component to help clients distinguish between different kinds of conditions. All too often, as the implementation is modified and new conditions under which exceptions may be generated emerge, it is easier to map the condition to the existing set of designed exceptions, rather than to refine the design. In the long term, this led, in the case of Jex, to exceptions that were very difficult to interpret. As an example, the Jex Loader component is a parser for files containing information about exception flow for all the methods of a class. As described in Table 1, the only two exceptions that were to escape the Jex Loader were an `IOException`, if the file could not be opened and a `ParseException` if it could not be parsed correctly. During the implementation of the system, it was determined that the situation in which a file does not contain information requested should be signaled as a `ParseException`. However, after the system had been in operation, it was realized that the difference between a parsing error and a method not being found (i.e., information was missing) should be distinct. To differentiate between the two situations, the value of the exceptions—information stored in the exception object—was used. However, this was a short-term solution because we found that the use of the exception value to distinguish between exceptions complicates the writing of handlers, and complicates maintenance of the exception structure.

Propagating system-defined exceptions made handlers difficult to write. The Jex Loader raised a system-defined `IOException` under some conditions, such as when a documentation file could not be found. It turns out that `IOException` could also be raised by the methods of the Java API classes when some low-level IO problem occurred. The result was that it was difficult for the client of the Jex Loader, the AST, to handle the exception because the cause of the exception could be so varied.

We could not easily bound unchecked exceptions. Another difficulty we had in establishing the structure of exceptions in Jex was related to the cost of some implementation decisions. In Java, exceptions can be either checked or unchecked (or *runtime*). Checked exceptions must be declared in the header of all methods which propagate them; unchecked exceptions need not be declared. User-defined exceptions are often checked exceptions because they correspond to conditions that developers find useful to signal as a specific potential cause of exiting a method. Enabling the compiler to check such exceptions makes clients aware of these specific exit conditions.

It would have been desirable to declare the `AnalysisException` and `ExceptionGenerationException` for the AST component as checked exceptions. However, the AST component is implemented as a tree of specialized nodes where the nodes are implemented as roughly 100 different classes. Since the problems corresponding to the two exceptions mentioned above could arise anywhere in the AST, the exceptions could potentially propagate through most of the methods of the node classes. Had we declared the exceptions as checked, we would have had to declare the exceptions in many places; specifically in approximately two to ten methods in 100 classes. This strategy was not deemed cost-effective and the two exception were defined to be unchecked. This decision had two main consequences. First, because they are unchecked, extra care and inspection was needed to identify the propagation paths of the exceptions so that the exceptions could be handled effectively. Second, to limit the number of different unchecked exceptions flowing in the AST, some other exceptions were recast either as an `AnalysisException` or an `ExceptionGenerationException`. As before, this led to a reliance on the value of the exception to provide useful information, creating a fragile situation.

Uncaught exceptions caused our system to crash. Many of the low-level exceptions in Java, such as `ArrayIndexOutOfBounds`, are unchecked exceptions. Because they are unchecked, these low-level exceptions, which typically represent a problem with the implementation of a component, tend to propagate out of the component through most of the call chain to the entry point of the application. Apart from indicating that the application is not operating correctly anymore, these exceptions were useless because they did not provide sufficiently precise or contextual information to perform any recovery or to provide any useful error message to users. The practical consequence of this situation is that our program would exit and dump a stack trace whenever a component would fail in an unanticipated way. This behaviour was not desirable.

We did not design the value of the exceptions. Although we had initially specified which exceptions would be thrown by a component, we did not specify what the value of the exceptions would be. As a result, using this value required significant inspection in modules that raised the exception.

3 WHY DOES EXCEPTION STRUCTURE DEGRADE?

The analysis of the exception structure of Jex and of numerous other Java programs has allowed us to identify a number of potential causes of complex or difficult to maintain exception structure.

Unpredictable Exception Sources

The full set of exceptions that may arise in a system cannot be determined until the implementation is complete, since the concrete implementation choices made in building the system affect the exceptions that are raised. The late determination of exceptions in the system makes it challenging to design and implement a well-structured set of handlers and propagation policies for the various kinds of exceptions.

Unanticipated Exceptions

As described earlier, Java supports both checked and *runtime* (unchecked) exceptions. The use of unchecked exceptions leads to two problems. First, pervasive exception types, such as `NullPointerException`, can circulate freely in a program, sometimes reaching the entry point of the application thereby causing the program to crash.[2] Second, since unchecked exceptions may be subsumed by the more general type, `Exception`, handlers can become overloaded. Problems associated with handler overload are considered separately below.

Handler Overload

An exception handler in Java states the type of exception that it handles. Since Java exceptions are related by a hierarchical type system, a handler, through subsumption, may end up handling more than one kind of exception. Subsumption can enable a programmer to succinctly describe the handling of a set of related exceptions. However, sometimes a handler may end up unknowingly subsuming exceptions. One instance in which this happens is when a handler subsumes unanticipated exceptions. This situation may also arise when a program evolves: a module may be changed to emit additional subtypes of a previously used exception type. Unknowing subsumption can cause the actions of a handler to become incoherent.[3] Handler overload is also inevitable when a developer chooses to explicitly raise an exception of a type that is commonly defined as the supertype of many other exceptions (e.g., `RuntimeException`). In these case, it is impossible for clients to handle the exception specifically.

Propagation

Every time an exception is propagated, it loses context. Consider the following case in point. A method reads from some stream object received as an argument. If an `IOException` occurs, the method typically cannot recover since it did not create the stream object and thus likely does not know the name of the source file used to create the stream. In such a case, the method is likely to propagate the exception. However, a client is not often any better positioned to handle the problem. A client, for instance, may not be able to determine from the `IOException` whether the problem was raised because an IO stream could not be created, opened, closed, written, read, queried, reset, flushed, or whether it suffered from some other problem. Lack of sufficient context makes it difficult for clients to design good recovery or useful notification upon catching propagated exceptions.

Exception Overload

The exceptions that may be potentially raised by a component can be considered as part of the interface to the component. When a client is aware that a component C may raise an exception E, the client may introduce a handler for exception type E. Over time, component C may be extended in ways that can fail differently from the cases that exception type E was meant to capture. To ensure compatibility with clients, the developer of C may choose not to introduce a new potential exception, but rather may choose to re-map new exceptions to the exception type E. In this way, exceptions can become semantically overloaded. This overloading may eventually degenerate the meaning of a particular exception type for a component, making it impossible for a client to write recovery code. Another kind of exception overload occurs when developers choose to explicitly raise a system-defined exception to signal a condition other than the one that is specified as the rightful cause for the exception. An example often encountered is when application code raises an `InternalError`, even though the Java API specifies that this errors correspond to a problem in the Java virtual machine.

Systematic Ignoring of Exceptions

We have also observed many cases where developers resolved the problem of where to handle an exception by simply catching the exception and doing nothing, sometimes commenting the `catch` clause with a message to the effect that "this should not happen". In the case where such an assumption turns out to be false, the program can end up in an inconsistent state.

Unspecified Exception Values

A Java exception object carries a value. One use of the value is to store an explanatory string that can be

[2] The problem of uncaught exceptions has been identified previously, and many specific approaches have been proposed to address it (e.g. [8, 12, 13, 18, 19, 20]).

[3] For a more detailed discussion of this problem, see [13].

used to describe to a user the exceptional condition that occurred. Since this value is set when the exception is created, its use depends on the developer who wrote the line of code creating the exception. If consistently set, this value can be helpful in simplifying handlers; for instance, several kinds of exception can be caught through subsumption in a single handler and the value of the exception simply output. Inconsistent use of the value means that multiple handlers might need to be used.

Inconsistent Use of Exception Handling
In some Java programs, exception handling is combined with other error handling strategies, such as setting termination codes. Use of other strategies for dealing with exceptional conditions makes the exception structure more difficult to understand and manipulate.

4 ADDING STRUCTURE TO EXCEPTIONS

Given that the exception structure of a program can degrade in many ways, the question becomes how to regain control of a program's exception structure. For the Java programs we have been studying, we have had success applying a technique described by Litke for designing fault-tolerant systems in Ada [10]. This technique consists of a strategy for determining software *compartments* and for defining exception interfaces for each compartment. To account for the differences between Ada and Java, we have had to make some small adaptations to the technique. In this section, we describe the technique and discuss how we applied it to simplify and regularize the exception structure in Jex. We believe the Jex system is more maintainable as a result. We describe additional experiences applying the technique in Section 5.

Compartments
The idea behind software compartmenting is that "compartmented programs have identifiable boundaries within them that contain the propagation of specific error classes" [10, p.405]. Once a compartment has been chosen, we want to define an interface to that compartment that includes an exhaustive description of the exceptions that may propagate from it. The intent of compartmenting a program is to improve its robustness. Robustness is enhanced because we end up specifying a constraint on the system that all the exit points out of a compartment, including exit points due to exceptions [6], are known. Enhanced robustness does not come without a cost: enforcing and verifying the constraints requires additional implementation effort.

There is no real restriction on what compartments can be. In theory, a compartment could be any set of entities that can raise exceptions, such as a set of methods. Practically, aligning compartments with the program structure provides a basis for reasoning about the exception structure. We have also found it helpful to choose compartments such that the interface to a compartment is minimal. This choice reduces the effort necessary to enforce and verify the compartments.

In Jex, the compartments we chose mostly aligned with the architectural decomposition of the system: the Controller, the Type System, the Parser and the Jex Loader. We did not specify the AST component to be a compartment for two reasons. First, the interface to the component was complex, including a high number of public methods with intensive use of polymorphism. Second, the AST component was accessed only through the Parser; the compartment specified for the parser could include the AST. In addition to these coarse-grained compartments, we found it useful to specify one subcomponent compartment around a class responsible for resolving simple names to fully-qualified Java names, (the `Resolver` class). We added this compartment because the precise error semantics of the `Resolver` class were necessary to perform some specialized recovery in our program.

Abstract Exceptions
The next step involves determining which exceptions will be allowed to propagate from a compartment. We refer to exceptions designed to propagate from a compartment as *abstract exceptions*. Determining the exceptions that should propagate from a component is the most difficult step in regaining control of exception handling. The difficulty stems from trying to determine a list of semantically coherent exceptions that describe the complete set of problems that can happen in a compartment.

The exceptions that are allowed to propagate from a compartment should describe a problem in the context of the compartment for which it is defined, and should not reveal any of the compartment's internal workings. When determining a propagation policy for a compartment, it is useful to consider two categories of exceptions: *system exceptions* and *internal failures*.

System exceptions typically correspond to a broken assumption about the system. Such exceptions often result when assumptions or constraints about the sequences of operations on a component, the parameters to an operation, or the environment, are not respected. Examples of system exceptions include popping an empty stack (sequence of operations), accessing an non-existent element (parameters to an operation), or writing to a protected file (environment). In many cases, it is useful to report such conditions, but only if the problems are represented at the compartment boundary in a meaningful way. For Jex, we found that the most effective way to report such faults was to associate them with a high-level concept pertaining to the functionality of the component. Three examples of such exceptions are `NoSuchElementException` when a request is made for an element that does not appear in a collection; `LookupException` when a lookup table is accessed with an invalid key; and `ParseException` when a parser is unable to parse input.

What about the internal failures? These exceptions relate to an internal inconsistency, independent from the sequence of operations on a component, the parameters of the operations, or the environment. These exceptions typically correspond to an implementation problem and generally do not indicate anything useful to a client, except a failure of the component. Example of internal failures can include dynamically casting an object to an invalid type, or accessing an array beyond its bounds. For Jex and the other programs we have analyzed, we have not found it necessary to distinguish between the different classes of failures. We have used a single exception, `AlgorithmicException`, to model internal failures.[4]

The design of the abstract exceptions should be complete and precise. By complete, we mean that every possible exception, runtime or checked, must be specified. By precise, we mean that, if the exceptions that can be propagated are organized in a hierarchy, all exceptions should be documented, not only the supertype. This way, all exit points out of the compartment are explicit.

General Guidelines

In addition to choosing compartments and abstract exceptions, we have found the following general guidelines helpful to follow.

First, we have found it useful to limit the error handling structure to the use of exceptions. Global error code variables and local exit instructions should be avoided. Adherence to this guideline will not only ensure a simpler structure, but will facilitate reuse by allowing clients of various components to decide how they should fail. In Jex, no component was allowed to terminate the program except the Controller, which is the entry point to the application.

Second, to make it easier to implement and verify whether compartmenting was successful, we have found it helpful to restrict the functional interface of the compartment as much as possible. In our work with Java programs, this means that any method that is not part of the functional interface should be declared private.

Finally, if, for a particular component, different exceptions can be raised by the same access point, it is typically useful to organize them into a hierarchy so that the client has the option of either recovering from a general component exception or from a particular one. However, if such a strategy is chosen, care has to be taken to ensure that the general exception type chosen as a supertype for the hierarchy is general enough to semantically represent all sub-exceptions.

[4] Some languages directly support the unified signaling of internal errors. For example, CLU has a single unchecked `failure` exception. C++ re-maps all undeclared exceptions to a single `unexpected` type.

Table 2: New Abstract Exception Specification for Jex

Component Name	**Exceptions Raised**
Controller	None
TypeSystem	TypeSystemSetupException TypeSystemLookupException AlgorithmicException
Resolver	NameException AlgorithmicException
JexLoader	JexFileLoadingException JexFileInterpretationException AlgorithmicException
Parser	ParseException AnalysisException ExceptionGenerationException AlgorithmicException

Redesigning Exceptions for Jex

Table 2 shows the complete list of abstract exceptions for each compartment selected in Jex. The Controller, being the entry point of the application, cannot raise any exception. With the compartmenting in place, this specification is meant to include unanticipated exceptions, so the application cannot terminate in an unexpected way. For the Type System, we chose two exceptions corresponding to the two orthogonal modes of operations on the component: initialization and lookup. Since these exceptions can never be raised by the same operation, they are not specified as a hierarchy other than the basic Java exception hierarchy. The `AlgorithmicException` is the abstract exception representing internal failures in every compartment. The other abstract exceptions all represent system exceptions. The Resolver propagates a `NameException` if a name cannot be resolved. The Jex Loader can raise two abstract exceptions (other than `AlgorithmicException`) that are subtypes of `JexFileException` since they both can be raised by the same operation. Finally, the Parser can raise three abstract system exceptions, `ParseException`, and two exceptions that can propagate from the AST: `AnalysisException` and `ExceptionGenerationException`.

Implementing Abstract Exceptions in Java

Since there is no direct support for compartmenting in Java, it was necessary for us to implement a mechanism to enforce compartment boundaries. The mechanism we defined was an *exception guard*. An exception guard enforces the specification of abstract exceptions by preventing any unanticipated exception from propagating out of a compartment.

In our system, we have implemented abstract exceptions and exception guards using exception re-mapping, and exception subsumption, respectively. Conditions resulting in an abstract exceptions can either be raised explicitly using the `throw` keyword, or be raised implicitly by a method call or a language operation, such as a division by zero. In the latter case, we map the actual exceptions raised in the component to our declared abstract exceptions by identifying their origin, catching them, and re-throwing them as a type conforming to an abstract exception.

To guard against various types of internal failures propagating past components, we wrap the interface operations of each compartment in a `try` block with a `catch` clause for every abstract exception, followed by a catch clause declaring the type `Throwable`. The `catch` clauses for the abstract exception simply re-throw the exception, while the `catch` clause for the general type maps all exceptions to a new exception of type `AlgorithmicException`. As specified in Table 2, an `AlgorithmicException` is raised whenever an unanticipated exception was detected in the compartment, and thus this type of exception corresponds to the concept of internal failure. Figure 4 gives an example of the implementation of an exception guard for the constructor of the Type System.

Exception guards do not prevent the presence of other internal exception handlers. For example, the `try` block in Figure 4 could contain an internal handler to perform any necessary re-mappings.

```
public TypeSystem()
{
   try
   {
      // Initializing the Type System
   }
   catch( TypeSystemSetupException e )
   { throw e; }
   catch( Throwable e )
   { throw new AlgorithmicException( e ); }
}
```

Figure 4: An example of exception guard implementation

To verify the conformance of the implementation of compartments to their exception guard contracts, we use the Jex tool itself. For each method, Jex returns the list of exceptions that can propagate out of the method. Using Jex, it has been relatively easy to establish conformance.

With the exception guards in place and the conformance to our specification of abstract exceptions verified, the propagation interactions between the components of our systems are now simpler (Figure 5).

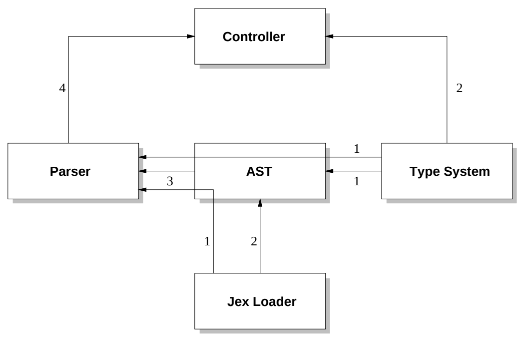

Figure 5: Exception Propagation in the Revised version of Jex

5 ADDITIONAL EXPERIENCE

We have applied the technique described in the previous section to two other software packages: GNU JTar version 1.1 and IBM's Bobby class library, version 7. In both cases, we were able to improve specific aspects of the error handling structure without reengineering the structure of the program handling normal operations.

JTar

JTar is a Java command-line program to create and extract tape archive (tar) files. It consists of 48 classes in 6 packages. Based on a manual inspection of the source code, we determined that JTar has a simple, 3-level layered architecture. This architecture is represented in Figure 6.

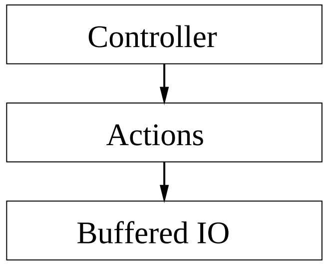

Figure 6: Layered Architecture in JTar

The Controller layer is the entry point to JTar and controls the application. It parses the command-line arguments and calls the Actions layer to perform actions requested by users through the command-line arguments, such as create, read, and extract tar files. The classes of the Actions layer use the Buffered IO layer to carry out specialized IO tasks.

Our inspection of the source code indicated that error handling was not uniform in JTar. In the classes of the Buffered IO layer, internal failures, such as `NullPointerException`, were propagated; other exceptions, such as `IOException`, resulted in termination of the program. In the Actions layer, exceptions were typically caught and an error code was set before the call returned normally, perhaps without having completed the action. As we have pointed out in section 3, we believe that these choices were not optimal. First, handling exceptions using a combination of exception handling and normal code makes it very difficult to reason about and alter the flow of the program under exceptional conditions. Second, exiting the application at various points in the program scatters the program termination code, making reuse of the components difficult.

We thus set about to improve the JTar code in two ways. One improvement was to ensure that the application exits only at the Controller layer. The second improvement was to report every fault using exceptions, rather than using error codes. This improvement is consistent with the idea that a point where it does not make sense to continue the execution of a program should be reported by an exception [6].

To make these improvements, we applied the approach described in the previous section. We decided to make each layer a compartment.

We began with the Buffered IO compartment. First, we determined the entry points to the compartment. For Buffered IO, the entry points consisted of the 17 public methods of the two classes, `BufferRead` and `BufferWrite`.[5] At each entry point, we implemented exception guards for the following exceptions types `JTarCorruptedInputException`, `JTarFileIOException`, `JTarNoPermissionException`, `JTarNoArchiveNameException`, and `AlgorithmicException`. This list was determined by inspecting the conditions under which a termination occurred, and the message that was output at the termination point. The first four exception types correspond to conditions related to the parameters or the object of the various operations (typically, a tar file). To allow the possibility of handling all of these exceptions at once in a client, we have defined them as subtypes of a general `JTarException` type. Internal failures were represented using `AlgorithmicException`, just like they were in Jex. We also have identified the points in Buffered IO where an exit was present and replaced the exit instruction with a suitable `throw` statement. This redesign of the exception structure of the Buffered IO layer leads to a simpler failure mode because all exceptional conditions are reported to the Actions layer in the form of anticipated exceptions, and because the Buffered IO layer is unable to terminate the application.

[5] Originally, none of the methods were scoped `private`. However, we could easily determine a set of methods that was only accessed within the class. To make the interface to the Buffered IO layer more explicit, we have qualified these methods as private.

We proceeded in a similar way for the Actions compartment, identifying faulty points in the program that set an error code and replacing them with the raising of abstract exceptions newly defined for the Actions compartment. In doing so, we had to assume that no other useful computation occurred after the detection of the faulty point.

When implementing the guards, we have found that the main difficulty is in mapping existing exceptions and exit instructions to the abstract exceptions defined in the guard specification. This requires reasoning about the semantics of a program. We have found the Jex tool useful in determining the source of exception occurrences and in helping us to link those occurrences with corresponding catch clauses.

Bobby

Bobby is a Java class library for manipulating Java class files. It consists of 118 classes representing, among other things, the class file itself and various entities present in a class file, such as the constant pool, fields, and instructions.

As opposed to Jex and JTar, Bobby does not have an obvious architectural decomposition. All the Bobby classes are public and most of them are intended for inter-package use. There are numerous dependencies between classes. For these reasons, it was not possible to identify clean compartments within the Bobby package. However, since Bobby is a class library, it is meant to be used by client code. From the perspective of potential client code, two issues regarding exception handling arose:

1. Most problems internal to Bobby (as opposed to the class being manipulated) were reported either as `RuntimeExceptions` or as `InternalErrors`. These two conditions cause difficulties for a client wishing to recover from a problem in Bobby. A `RuntimeException` cannot be caught without catching all other `RuntimeExceptions`, restricting the granularity of the potential recovery. According to the Java API specification, an `InternalError` corresponds to a problem with the Java virtual machine (JVM). When Bobby uses an `InternalError`, it is only possible to distinguish if an InternalError was raised by Bobby or the JVM by inspecting the call stack.

2. Bobby also reports some system exceptions result-

ing from a sequence of operations on the class library as `InternalErrors`.

To give more flexibility to the clients of Bobby, we were interested in a version that would report system exceptions as such (as opposed to internal failures), and that would report actual internal failures in a way that was more easily usable by clients.

Applying the technique described earlier to restructure the exception structure was more difficult in the case of Bobby than the other two systems. Since almost every method in Bobby is public, implementing guards implied adding a `try` block in every method of the 118 classes. To manage the cost of this change, we decided to implement the guards only on a subset of frequently-used classes.

The guards consisted of the two existing user-defined exceptions, `BB_ClassFileException` and `BB_DuplicateClassException`. We also added `BB_IllegalOperationException` to signal illegal operations and `BB_InternalFailureException` to represent internal failures. The first two exceptions were checked exceptions. We decided to implement the two new exceptions as unchecked, runtime exceptions, because of their pervasiveness inside the Bobby package.

Applying the approach to Bobby, even in this partial way, allowed us to simplify the flow of exceptions within the Bobby classes. The revised Bobby is also easier to use because the exit points, both normal and exceptional, are explicit, and the exceptions are more meaningful.

6 RELATED WORK

Existing work on exception structure focuses on the initial design of exceptions for a system.

Seminal work on the design of robust and fault-tolerant programs using exception handling was done by F. Christian. His contributions include a study of how the failure occurrences related to specific classes of design faults can be addressed using default exception handling based on automatic backward recovery [5]. In a later paper, Christian also proposed "a programming language suitable for writing well-structured robust programs" [6, p.163]. The work, mostly theoretical, includes a deductive system for proving total correctness and robustness properties of programs with exceptions.

Work on designing programs with exceptions also spans the areas of application-domain specific approaches, methodologies, and design tools. As described earlier, Litke [10] proposed an approach to designing fault-tolerant systems in Ada. Similarly, de Lemos and Romanovsky have suggested a framework for integrating exception handling into the early phases of the software life-cycle [7].

Tools and modeling techniques integrating exceptions have also been suggested. An example of a tool is OODREX [2], which helps take exceptions into account when designing C++ classes. An example of a notation that integrates exceptions is the Unified Modeling Language (UML) [4]. An extension to UML to model exceptions as pre- and post-condition constraints using the Object Constraint Language [17] has also been proposed recently by Soundarajan and Fridella [15].

The work described in this paper differs from these efforts as it focuses on our experiences reengineering exception structure into existing programs. Many of the problems we have identified with exception structure degradation, such as exceptions that arise because of the implementation path chosen, are not discussed in the literature.

More closely related to the problems discussed in this paper is an analysis by Miller and Tripathi [11] of why it is difficult to design exceptions in object-oriented systems. However, their analysis is focused on conceptual clashes between object-orientation and exception handling, such as abstraction, encapsulation, modularity and inheritance, rather than on the realities of programming with exceptions.

7 SUMMARY

Just as the structure of the normal operations of a system tends to degrade as the system evolves, the structure of exception handling also degrades. This degradation can result from many factors. We have identified and described a number of factors in this paper, including unpredictable exception sources, unanticipated exceptions, and handler overload.

Fortunately, the situation is not hopeless. We have shown how, in our experience, a straightforward extension of an approach to designing fault-tolerant systems in Ada can be used to regain control of the exception structure in an existing system. We presented the results of applying the technique to three systems, and described the guidelines we have employed in performing the reengineering. We believe the approach is useful because we have been able to improve the exception handling code of a system with minimal effort, and without changing the normal structure of the code.

The experiences described in this paper extend beyond Java programs. Developers working in other languages and researchers working on exception handling design can benefit from the elucidation of why exception structure degrades and the ability to straightforwardly reengineer exception structure.

ACKNOWLEDGEMENTS

We would like to thank IBM for providing us with the source code of Bobby. We are also grateful to A. Lai for useful comments on the paper. This work was funded by a Natural Sciences and Engineering Research Council of Canada (NSERC) graduate fellowship and research grant.

REFERENCES

[1] *Ada 95 Reference Manual: Language and Standard Libraries, version 6.0*, December 1994. Revised international standard ISO/IEC 8652:1995.

[2] C. Bamford and B. Dollery. OODREX: An object-oriented design tool for reuse with exceptions. In *Proceedings of the International Conference on Object-Oriented Information Systems (OOIS'95)*, pages 248–251, Berlin, Germany, 1995. Springer-Verlag.

[3] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[5] F. Christian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, 31(6):531–540, June 1982.

[6] F. Christian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10(2):163–174, March 1984.

[7] R. de Lemos and A. Romanovsky. Exception handling in a cooperative object-oriented approach. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, pages 3–13. IEEE Computer Society, May 1999.

[8] M. Fahndrich, J. Foster, J. Cu, and A. Aiken. Tracking down exceptions in standard ML programs. Technical Report CSD-98-996, University of California, Berkeley, February 1998.

[9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Longman, Inc., 1996.

[10] J. D. Litke. A systematic approach for implementing fault tolerant software designs in Ada. In *Proceedings of the conference on TRI-ADA '90*, pages 403–408. ACM, December 1990.

[11] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 85–103. Springer-Verlag, June 1997.

[12] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of the 26th Symposium on the Principles of Programming Languages*, pages 276–290, January 1999.

[13] M. P. Robillard and G. C. Murphy. Analyzing exception flow in Java™ programs. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 322–337. Springer-Verlag, September 1999.

[14] M. Shaw and D. Garlan. *Software Architecture. Perspective on an Emerging Discipline.* Prentice-Hall, 1996.

[15] N. Soundarajan and S. Fridella. Modeling exceptional behavior. In *Proceedings of the UML'99 Conference*, 1999.

[16] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

[17] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML.* Object Technology Series. Addison-Wesley, Reading/MA, 1999.

[18] K. Yi. An abstract interpretation for estimating uncaught exceptions in standard ML programs. *Science of Computer Programming*, 31:147–173, 1998.

[19] K. Yi and B.-M. Chang. Exception analysis for Java. In *ECOOP'99 Workshop on Formal Techniques for Java Programs*, June 1999.

[20] K. Yi and S. Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proceedings of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 98–113, September 1997.