Designing Robust Java Programs with Exceptions

Martin P. Robillard and Gail C. Murphy
Department of Computer Science
University of British Columbia
2366 Main Mall, Vancouver, BC
Canada V6T 1Z4
{mrobilla,murphy}@cs.ubc.ca

ABSTRACT

Exception handling mechanisms are intended to help developers build robust systems. Although an exception handling mechanism provides a basis for structuring source code dealing with unusual situations, little information is available to help guide a developer in the appropriate application of the mechanism. In our experience, this lack of guidance leads to complex exception structures. In this paper, we reflect upon our experiences using the Java exception handling mechanism. Based on these experiences, we discuss two issues we believe underlie the difficulties encountered: exceptions are a global design problem, and exception sources are often difficult to predict in advance. We then describe a design approach, based on work by Litke for Ada programs, which we have used to simplify exception structure in existing Java programs.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming—*Java, exception handling*; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

DESIGN, EXPERIMENTATION

Keywords

Exception handling, error handling, exception structure design, software compartments

1. INTRODUCTION

> For programs to be reliable and fault tolerant, each program module must be defined to behave reasonably under a wide variety of circumstances. An exception handling mechanism supports the construction of such modules. [8, p.546]

Most modern programming languages, including object-oriented languages such as C++ [13] and Java [6], provide an exception handling mechanism. Syntactically, these mechanisms provide a means to explicitly raise an exceptional condition, and a means of expressing a block of code to handle one or more exceptional conditions. As described by Liskov and Snyder above, the intent of these mechanisms is to make it easier to reason about and build robust software systems.

Although an exception handling mechanism provides a basis for structuring source code that deals with unusual situations, little information is available to help guide a software developer in the appropriate application of the mechanism. In our experience, this lack of information about how to design and implement with exceptions leads to complex and spaghetti-like exception structures. Even when we set out to carefully design and implement the exception handling code in one of our programs, we still ended up in the same predicament. Speaking with other Java developers and analyzing other Java source code [11], we have found that the situation is far from uncommon.

In this paper, we reflect upon our experiences trying to build a robust program analysis tool in Java with a "good" exception structure (Section 2). We describe where and when we made trade-offs that eventually led to an overly complex exception structure. We then discuss the two issues we believe underlie the difficulties we encountered (Section 3). The first issue is that exception handling is a global design problem, making it difficult to decompose the problem to handle complexity. The second issue is that it is often extremely difficult to predict the sources of exceptions in advance, complicating the design and implementation of exception structure.

We believe that the difficulties we encountered can be mitigated through the development and use of better design methods for exceptions. To provide an example of how design methods can help, we describe an approach we have used to simplify the exception structure in existing systems (Section 4). This approach is based on a proposed method for designing fault-tolerant Ada systems [9]. We discuss the outcome of applying this approach to three different Java systems (Section 4 and 5) and analyze its costs and benefits (Section 6). We also compare the work presented in this paper to previous efforts (Section 7).

By describing the difficulties we have encountered working with exceptions and a method we have used to address these difficulties, this paper makes two contributions. First, it identifies some reasons why and how exception structure becomes complex. Second, it demonstrates how a straightforward approach to exception structure design and implementation can simplify exception structure, improving the robustness and changeability of the program.

FSE 2000, San Diego, CA.

2. DESIGNING WITH EXCEPTIONS

To illustrate the difficulty of designing and implementing a robust system with exceptions, we describe the case of the Jex exception analysis tool [11]. Jex is a static analysis tool that determines exception flow information in 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 of the program.

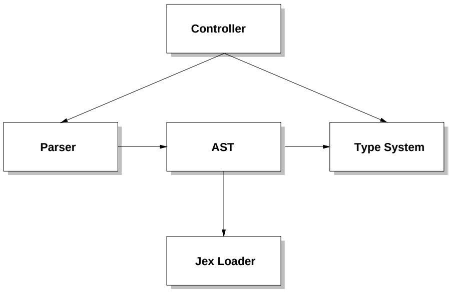

Figure 1: The Architecture of Jex

Jex consists of five components (Figure 1) comprising 136 classes and approximately 22 kLOC.[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.

From the start, we wanted Jex to be robust and not to fail in unanticipated ways. To achieve this goal, we chose to build upon the exception handling mechanism supported by Java, the language used to implement Jex. Specifically, we used a two-level approach. At the first level, the component-level, we designed a restricted set of general, high-level, exceptions for each component. A component was to explicitly signal only this limited set of exceptions. By limiting the exceptions that could be raised, we hoped to limit the number and complexity of handlers for those exceptions in clients to a manageable level. Table 1 shows the initial specification for the exceptions raised at the component level; each was chosen to represent a failure that was meaningful from an architectural viewpoint. Our intent was to make reasoning about the exception flow a manageable task and to limit the number of unanticipated exceptions that could cross component boundaries and cause the system to crash.

Having a specification of the exceptions which can propagate from a component was also intended to simplify the design of intra-component exception handling, the second level. Specifically, for each intra-component exception, a decision had to be made whether to handle the exception locally, or to re-map the exception to a component-level exception and propagate it. To make the option of re-mapping feasible, we tried to name the component-level exceptions using terms general enough that intra-component exceptions could be meaningfully re-mapped. The idea was to be able to focus independently on intra-component or component-level exceptions, depending on whether we were implementing or integrating components. Specifying the component exception interfaces early supported this focus.

After producing an architectural design with detailed interfaces, the Jex development proceeded incrementally. The components were developed in a roughly bottom-up order: Type System, Jex Loader, Parser, AST, and Controller.

Table 1: Component-level exception specification

Component Name	Exceptions Raised
Controller	None
Type System	TypeException
Jex Loader	ParseException IOException
AST	AnalysisException ExceptionGenerationException
Parser	ParseException

2.1 Creeping Exception Complexity

Even before the Jex tool was complete, we realized that the exception structure was gaining rapidly in complexity. This increase in complexity occurred despite the fact that the structure of the code handling normal conditions remained straightforward. Our "loss of control" of the exception structure was due to the following causes.

Ambiguous exception semantics. As mentioned above, in our design approach, intra-component exceptions could be re-mapped to the smaller set of exceptions that might be raised by a component. In practice, this led to problems interpreting the meaning of a component-level exception. Consider the following example. The implementation of the Type System component re-mapped the exceptions that could occur in classes contributing to the Type System to the generic `TypeException`. In most cases, this re-mapping was appropriate because the precise meaning of the errors raised in the component did not enable clients to vary or improve their recovery code. However, in one case, the re-mapping approach led to a `TypeException` representing either some I/O problems related to initializing the type system component, or lookup problems related to using the component. Treating these exceptions the same made it difficult to write useful handlers; for instance, it was difficult to write a handler to inform the user that their environment was not setup properly. To enable this separate treatment, we modified the interface of the Type System component to propagate `IOException`. Unfortunately, this change led the explicitly raised `TypeException` and the propagated `IOException` to carry the same semantics during the initialization of the Type System. In this case, the handling of both types of exceptions was identical. Clearly, this situation made it confusing and difficult for a client to determine when the semantics of the exceptions were the same and when they differed.

Distinguishing exceptions with exception values. To enable clients to distinguish the specific reason a particular component-level exception was raised, we sometimes used information stored in the exception object—the value of the exception. This situation arose, for instance, in the client of the Jex Loader component. The Jex Loader parses files containing information about exception flow for all methods of a class. As described in Table 1, only two exceptions were to escape the Jex Loader: an `IOException` if the file could not be opened, and a `ParseException` if the file could not be parsed correctly. When the system was implemented, it was determined that the situation in which a file does not contain information requested about a method be signaled as a `ParseException`. However, when we integrated the Jex Loader with its client, the AST, we realized that the difference between a parsing error and a unfulfilled request for method information should be distinct. To differentiate between the two situations, we set the value of the exception differently in each case. Although this allowed us to write handlers to deal with each case separately, we found that the approach complicated both the writing of handlers, and the maintenance of the exception structure. The problem is that it is almost

[1] The kLOC figures in this paper include generated code and comments.

impossible to trace which program entities are allowed to create or modify the exception value.

Confusion over the use of system-defined exceptions. Under some conditions, such as when a documentation file could not be found, the Jex Loader raised a system-defined `IOException`. An `IOException` could also be raised by the methods of the Java API classes when some low-level I/O problems occurred. As a result, `IOException` ended up being more pervasive than most user-defined exceptions. It was difficult for the client of the Jex Loader, the AST, to handle the exception because the cause could be so varied.

Unbounded unchecked exceptions. In Java, exceptions can be either checked or unchecked (*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 typically checked exceptions because they correspond to conditions which 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. Although checked exceptions have many benefits, they can also be expensive to implement. For instance, it would have been desirable to declare the AST's `AnalysisException` and `ExceptionGenerationException`[2] as checked exceptions. However, since the AST component is implemented as a hierarchy of roughly 100 different classes, and since the problems corresponding to the two exceptions can arise anywhere in the AST, exceptions could potentially propagate through most of the methods of most of the classes. If we had defined these exceptions as checked exceptions, we would have had to declare them in approximately two to ten methods in each of 100 classes. An engineering decision had to be made: declaring the exceptions as checked exceptions was not deemed cost-effective and the two exceptions 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 they could be handled effectively. Second, to limit the number of different unchecked exceptions flowing in the AST and thereby simplify the exception structure, some other exceptions were recast either as an `AnalysisException` or an `ExceptionGenerationException`, leading to ambiguity.

Many of the the low-level exceptions in Java, such as `ArrayIndexOutOfBounds`, are also 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. Since these exceptions are so general, it was impossible to provide handlers at the entry point to perform any recovery or to provide any useful error message. All we could do is simply catch all unchecked exceptions at the program entry point to avoid crashing the program.

3. DIFFICULTIES IN EXCEPTION DESIGN

Looking back at the Jex development, it may seem as if some of the problems described above could have been avoided by making better design decisions. Indeed, given an unbounded amount of time and resources, the design and implementation of a system can always be improved.

Few developments, though, proceed in an environment with unbounded resources. Reflecting on the causes presented above and analyses of other Java programs [11], we believe there are two main factors which contribute significantly to the difficulty of designing exceptions when time and resources are bounded.

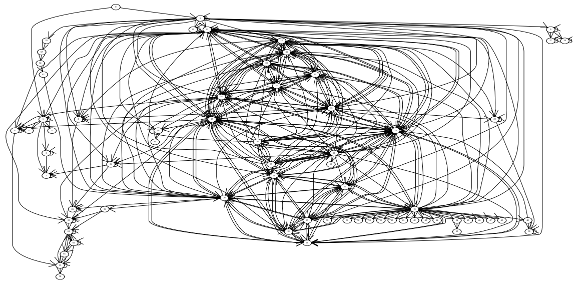

Figure 2: Propagation graph of `ClassCastException`

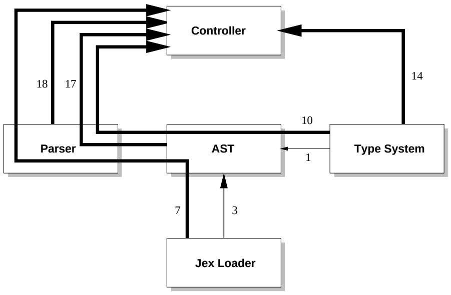

Figure 3: Exception propagation in Jex

1. Exception handling is essentially a global phenomenon.
2. It is difficult and costly to anticipate all categories of problems, and how they should be reported, during the design phase.

3.1 A Global Phenomenon

Exceptions are propagated automatically in Java: when a method exits with an exception, control is not necessarily transfered back to the direct caller of the method. Instead, it can transparently jump any number of levels up the call chain. This feature makes it possible for the raise and handle points for an exception to be separated by numerous method calls.

One difficulty related to automatic propagation is that reasoning about exception control paths can quickly become an intractable task for developers. As one example, Figure 2 represents the propagation paths, at the *class-level*, leading to the entry point of Jex for a *single* unchecked exception, `ClassCastException`. Obviously, reasoning about the behavior of the system and handling the exception reasonably to produce a robust system is difficult! Figure 3 presents a stylized representation of the flow of *all* exceptions between components in Jex. 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.

A second difficulty introduced by the global nature of exception handling is the cost of modifying exception interfaces, which requires more effort than modifying normal method interfaces. For example, the task of adding a parameter to a method signature requires tracking down where the method is called and ensuring that a meaningful value is assigned to the parameter. For the Java language, in cases where the method is not overloaded, the com-

[2] An `AnalysisException` results from any problem related to the syntactic analysis of the AST, while an `ExceptionGenerationException` corresponds to any problem related to the exception information generation phase of the AST.

piler will catch calls that have been “forgotten”. In contrast, when changing the list of exceptions a method can raise, how can one ensure that all the right handlers are identified and inspected? The handlers will not necessarily be in the callers of the method. And a name search on the type of the exception to be changed could be misleading since this exception could be raised by other methods, or some handlers could catch the exception by subsumption.[3] Tools like Jex [11] can help, but in our experience, tracking down handlers remains a daunting task. Furthermore, in the case of checked exceptions, changing the exception interface potentially requires modifying the interfaces of all the methods along the propagation path of the exception.

3.2 Unanticipated Exception Sources

In earlier work on exception analysis [11], we identified that reasoning about exceptions in Java programs was difficult because precise information about the types of exceptions that could be raised at various program points was not easily available to developers. Using very detailed design practices (down to the implementation level), it is possible for software engineers to accurately predict all the potential sources of exceptions originating in a module or component. While some organizations use such fine-grained detailed design as part of their software development process, it is not typical. Even when implementation information is available, it is still costly to trace the sources of all exceptions [11]. For these reasons, the specification of component-level exceptions tends to be based on incomplete knowledge of how a component can fail. As the design and implementation is iterated, new, unanticipated exception sources can emerge.

The end result is that software developers have incomplete knowledge about the exceptions that may occur when designing an exception structure for their program. This lack of knowledge leads to a tension between trying to design specific interfaces to enable appropriate handling of unusual situations, and designing general interfaces to enable emerging exceptions to be handled within the bounds of the current chosen scheme. This problem shares many similarities with designing module interfaces. However, as noted in the previous section, the global nature of exceptions makes the task more challenging.

4. SOFTWARE COMPARTMENTS

Our initial approach to designing exception handling was intended to make it tractable to understand our exception structure by dividing it into two levels: the component-level and the intra-component-level. Unfortunately, this two-level approach was not enough to keep control of the exception structure as it evolved during the incremental development of our system. One possible reason why our approach did not prove entirely successful is that it lacked the formal conceptual elements necessary for a systematic application, and guidelines to help make decisions when tradeoffs arose.

To investigate how a more thorough method could help, we tried applying the the principles of *compartmented software*, initially described by Litke for designing fault-tolerant systems in Ada [9]. Litke describes the approach as follows.

> An important effect of compartmented software design is to provide a clear specification and a ready understanding of error-tolerating behavior at the compartment boundaries. This property makes reasoning about the program behavior easier by reducing the complexity of relationships and makes modification of error-tolerating code easier [9, p. 405].

This property is exactly what we were trying to achieve. To apply the technique to Jex, we needed to adapt the technique for an object-oriented language, in this case, Java. This section describes our refinement of the technique and its application to Jex. Section 5 describes the application of the technique to two other systems.

Litke’s technique consists of the following steps.

1. Determining software *compartments.*
2. Defining precise and complete exception interfaces for each compartment.
3. Automatically verifying the conformance of the actual program to the compartment specification.

Clearly, this technique overlaps with our original concept of component-level and intra-component-level exceptions. However, in comparison, the technique of Litke is more explicit about the definition of compartments, provides better guidance for determining interfaces, and includes a verification aspect.

4.1 Compartments

The idea behind software compartmenting is that “compartmented programs have identifiable boundaries within them that contain the propagation of specific error classes” [9, p.405].

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. Litke suggests choosing boundaries at the software architecture level so that the functional and exceptional interfaces are minimal. Practically, aligning compartments with the program structure provides a basis for reasoning about the exception structure. In Jex, the compartments we chose mostly aligned with the architectural components in 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 comprised a high number of public methods with extensive 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 sub-component compartment around a class, `Resolver`, which was responsible for resolving simple names to fully-qualified Java names. This refinement is compatible with Litke’s approach, which suggests allocating “internal compartments” if the initial boundaries are too large [9, p. 460]. We defined this compartment for two reasons: the precise error semantics of the `Resolver` class were necessary to perform some specialized recovery in our program, and the cost of defining this compartment was negligible since the interface consisted of only a few methods.

4.2 Exception Interfaces

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 these exceptions is the most difficult step. 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.

[3] Subsumption is the action of implicitly upcasting an object when assigning it to a variable of a type corresponding to one of its supertypes.

Litke states four important guidelines in establishing exception interfaces [9, p.406].

1. "Use Ada exceptions to signal all detected errors."
2. "Enumerate all exceptions propagated across the defined boundaries."
3. "Decide [. . .] the precise semantics of all error signals propagated across the boundary."
4. "Determine appropriate transforms [re-mappings] for exceptions, including specifying which are completely handled, and which are partially handled before propagation to provide the specified semantics. This step may require a definition of a new exception to carry the specified semantics across the boundary."

Applying these guidelines to Java has led to the following refinements.

1. *Only use exceptions.* Similar to Litke's first guideline, we have found it useful to limit the error handling structure to the use of the Java exception handling mechanism. Global error code variables and local exit instructions should be avoided. Adherence to this guideline ensures a simpler structure, and facilitates reuse by allowing clients to control 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.

2. *Document exhaustive interfaces.* The description of the abstract exceptions should be complete and precise. By complete, we mean that every possible abstract 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.

3. *Specify precise error semantics.* Since Ada does not support exception hierarchies, Litke suggests that the precise semantics of *all* abstract exceptions be specified in advance. As we described earlier, meeting this condition is almost impossible given the emergence of unanticipated exception sources as design and implementation proceeds. To try and manage this situation, we propose an enhanced version of this guideline.

3a. *Design exception interfaces for change.* When a hierarchical exception mechanism is available, as is the case in Java, it can be used to help manage the evolution of exception interfaces. Specifically, abstract exceptions should be chosen to be as general as possible while still being meaningful. If a new source of exception is uncovered that has a more specific meaning, often, a subtype of the original abstract exception can be added to the compartment's interface. This approach gives the client the option of either handling only the more general supertype exception or handling the more specific subtype exception. This change can be made without modifying the interfaces to all the methods propagating the exception. The difficulty lies in choosing the original abstract exception to be sufficiently general.

4. *Determine re-mappings for exceptions.* We interpret Litke's guideline simply as determining in advance which low-level exceptions should be handled locally, and which exceptions should be remapped as abstract exceptions. Again, because of unanticipated exception sources, this can be a difficult task. Instead, we have found it useful to distinguish between two classes of intra-component exceptions: *system* exceptions and *internal failures*. System exceptions typically correspond to broken assumptions about the system. Such assumptions can include constraints about the sequence of operations on a component, the parameters to an operation, or the environment. On the other hand, internal failures relate to an internal inconsistency. 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 used a single abstract exception, `AlgorithmicException`, to model internal failures.[4]

In addition to the first four guidelines suggested by Litke, we have also found the following guidelines to be helpful.

5. *Avoid using system-defined exceptions as abstract exceptions.* Even though reusing a pre-defined exception, such as `IOException`, as an abstract exception saves writing an exception class, this does not generally pay off as it makes the interpretation of abstract exceptions more difficult.

6. *Do not propagate abstract exceptions.* Abstract exceptions are meant to be semantically associated with the compartment that raises them. For instance, the `Resolver` compartment can raise a `NameException` when a name cannot be resolved to a fully-qualified Java name. To avoid weakening the semantic connotation of the exception, it should not be propagated past the client compartment. This is not to say that abstract exceptions should not necessarily be reused where appropriate. However, the use of abstract exceptions should be limited to compartment boundaries.

7. *Do not raise abstract exceptions except in a compartment's entry points.* Abstract exceptions should not be raised by methods that are not entry points to a compartment. If a method which is not an entry point raises an abstract exception, then it becomes difficult to reuse that method in another compartment. It is preferable to catch all intra-compartment exceptions at the compartment boundaries, and to re-map these exceptions into the abstract exceptions at the compartment's entry points.

Table 2 shows the complete list of abstract exceptions for each compartment in Jex. The Controller, being the entry point of the application, cannot raise any exception. 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 in a hierarchy. The `AlgorithmicException` is the abstract exception representing internal failures in each compartment other than Controller. This exception is reused in most compartments, and thus is allowed to propagate through compartments. We feel this "exception" to guideline 6 is acceptable since it represents internal failures, which we can rarely handle, except at the application entry point.

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; each is a subtype of `JexFileException` since each 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`.

4.3 Compartment Verification

The means of establishing conformance suggested by Litke is based on a proposed automated tool. This tool, using an informal model of exception control flow in Ada, is intended to locate and classify exception raise points, and to trace the propagation of exceptions up to either their handling point or a program exit point.

[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: Abstract exception specification for Jex

Component Name	Exceptions Raised
Controller	None
Type System	TypeSystemSetupException TypeSystemLookupException AlgorithmicException
Resolver	NameException AlgorithmicException
Jex Loader	JexFileException JexFileLoadingException JexFileInterpretationException AlgorithmicException
Parser	ParseException AnalysisException ExceptionGenerationException AlgorithmicException

```
public TypeSystem()
{
   try
   {
      // Initializing the Type System
      // Perform necessary exception
      // re-mappings
   }
   catch( TypeSystemSetupException e )
   { throw e; }
   catch( Throwable e )
   { throw new AlgorithmicException( e ); }
}
```

Figure 4: An example of exception guard implementation

Java eases the problem of establishing conformance through its support of exception hierarchies. Through subsumption, many types of exceptions can be caught in a single `catch` clause. Specifically, we use *exception guards* to enforce the exception interface, ensuring only abstract exceptions defined for a compartment are allowed to propagate.

Each entry point method of a compartment is wrapped in a `try` block with a `catch` clause for every abstract exception, followed by a catch clause declaring the type `Throwable`, the supertype of all exceptions that can be raised in a Java program. The `catch` clauses for the abstract exceptions simply re-throw the exceptions, 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 is 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. In this case, because of the complexity of the operations in the top-level `try` block, the re-mappings to abstract exceptions are performed within the `try` block and the abstract exceptions are filtered in the corresponding `catch` clauses. For methods with simpler operations, it is possible to perform the re-mappings directly in the `catch` clauses associated with the top-level `try` block.

Although they help enforcing conformance without requiring tool support, exception guards have two weaknesses. First, exceptions can theoretically be raised in the top-level `catch` clause. Second, the mechanism only enforces *syntactic* conformance. Specifically, an unchecked exception that was meant to be re-mapped as an abstract exception could be "forgotten", and automatically re-mapped as an internal failure.

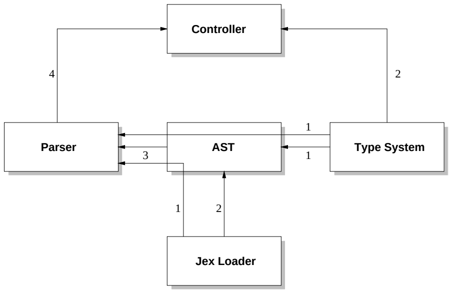

Figure 5: Exception propagation in the revised version of Jex

When necessary, these finer points can be dealt with using the Jex tool. Since Jex extracts exception information from Java programs, it can be used to report any exceptions raised in the top-level catch clause, and all uncaught exceptions propagating to the top-level `try` block. Running Jex on the modified Jex program allowed us to ensure that the desired containment properties were respected.

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

5. ADDITIONAL EXPERIENCES

In many respects, Jex was an ideal candidate for applying the compartmenting approach. Since compartmenting is a refinement of our original design approach for exceptions, most compartments aligned with Jex components. In addition, in choosing abstract exceptions, we could build upon our previous experience choosing component-level exceptions. To investigate whether the compartmenting approach applies in other situations, we applied the technique to two other existing software packages: GNU JTar version 1.1. and IBM's Bobby class library, version 7.[5]

We choose to apply the compartmenting approach to existing systems instead of experimenting with the development of new Java programs because our goal was to investigate how reasoning about compartmented exception structure compared to reasoning about other styles of exception structure. The use of existing programs allowed us to gather insight into how specific design decisions made by various developers influenced the overall exception structure. The tradeoff is that this case study approach prevented us from studying how compartment design interacts with traditional design activities. The investigation of this interaction is the subject of future work.

Each of the package investigated has unique characteristics. The JTar source demonstrates a layered architecture with no uniform error handling. The Bobby package is a class library with no overall architectural structure. In both cases, using compartmenting, we were able to simplify and improve specific aspects of the error handling structure without changing the structure of the program handling normal operations.

5.1 JTar

JTar is a command-line program implemented in Java which creates and extracts tape archive (tar) files. It comprises 48 classes in 6 packages (≈7 kLOC). Based on a manual inspection of the source code, we determined that JTar has a simple architecture consisting of three layers: Controller, Actions, and Buffered I/O (Figure 6).

[5] Bobby is now called the Jikes Bytecode Toolkit. It is available at IBM's Aphaworks website, `www.alphaworks.ibm.com`.

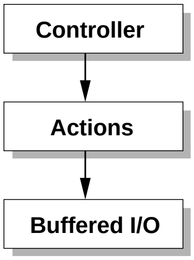

Figure 6: Simplified architecture of JTar

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 I/O layer to carry out specialized I/O tasks.

Based on an inspection of the source code, we determined that error handling was not uniform in JTar. In the classes of the Buffered I/O layer, internal failures, such as `NullPointerException`, were propagated; other exceptions, such as `IOException`, caused the program to terminate. 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. We believe that these choices were not optimal. First, handling exceptions using a combination of exception handling and normal code makes it difficult to reason about and modify how the program handles exceptional conditions. Second, exiting the application haphazardly complicates reuse of the components.

Our goal was to improve the JTar code in two ways. One improvement was to ensure the program could only exit at the Controller layer. The second improvement was to report every fault using exceptions. To make these improvements, we applied the refined compartment approach with each layer serving as a compartment. Our case study focuses on the interface between the Actions and Buffered I/O compartments.

We began with the Buffered I/O compartment. First, we determined the entry points to the compartment which consisted of the 17 public methods of the two classes, `BufferRead` and `BufferWrite`.[6] 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 the program terminated, and the message that was output at that point. The first four exception types correspond to conditions related to the parameters or the object (typically, a tar file) of the various operations. To allow the possibility of handling all of these exceptions at once in a client, we defined them as subtypes of a general `JTarException` type, which represents any problem related to the handling of a tar file. As before, internal failures were represented using `AlgorithmicException`. Next, we identified the points in Buffered I/O where an exit was present and replaced the exit instruction with a suitable `throw` statement. This redesign of the exception structure of the Buffered I/O layer led to a simpler failure model because all exceptional conditions are reported to the Actions layer in the form of anticipated exceptions, and because the Buffered I/O layer is unable to terminate the application.

Applying the compartmenting technique required reasoning about the semantics of the program. Much of the effort required involved tracking down the sources of errors and converting them into exceptions. When implementing the guards, we found that the main difficulty was in mapping existing exceptions and exit instructions to the abstract exceptions defined in the compartment specification.

5.2 Bobby

Bobby is a Java class library for manipulating Java class files. It consists of 118 classes (≈23.5 kLOC) representing such items as the constant pool, methods, and instructions of a class file.

In contrast 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 Bobby. 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.

First, most problems internal to Bobby were reported as either a `RuntimeException` or as an `InternalError`. These two conditions make it difficult for a client 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). For a client, it would only be possible to determine if an `InternalError` was raised by Bobby and not by the JVM by programatically inspecting the call stack.

Second, Bobby also reports some system exceptions which result from a sequence of operations on the class library as `InternalErrors`, making it difficult for clients to anticipate these failures.

To improve the flexibility of Bobby clients, we were interested in a version which would not report system exceptions as internal failures, and which would enable actual internal failures to be handled more effectively.

Applying the compartmenting technique 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 top-level `try` block and ensuring that exceptions raised in the method conformed to the interface in every method of the 118 classes. To manage the cost of this change, we decided to implement the guards on only a subset of frequently-used classes.

The abstract exceptions 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 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] Originally, none of the 32 methods of these classes 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 I/O layer more explicit, we have qualified these methods as private.

6. EVALUATION

The case studies show that compartmenting is applicable to a small set of diverse systems. In this section we discuss whether the approach is workable by considering its benefits and costs.

6.1 Benefits

From our original experience, we cited two difficulties related to designing and implementing "good" exception structure: reasoning about global exception flow, and dealing with emerging exception sources (Section 3).

With respect to dealing with global exception flow, our experience in the three case studies agrees with the intuition of Litke: bounding exception propagation at compartment boundaries has practical benefits. First, by limiting the scope of exception propagation at compartment boundaries, we reduce the intra-compartment exception design problem to deciding whether an exception should be handled locally or whether it should be propagated and remapped as an abstract exception. Second, by enforcing hard interfaces between compartments, we eliminate the possibility of unanticipated exceptions being raised by a compartment. The end result is that compartments and exception guards enable developers to reason about compartment-level exceptions in a straightforward and manageable way.

It is less obvious how compartmenting allows developers to better manage the emergence of new exception sources. Design guideline 3a, presented in Section 4, specifically caters to that effect, but whether it will work well in the general case, and how well it scales, is the object of future investigation.

For the three programs we studied, the guidelines were sufficient to address most of the problems initially identified with exception structure. As one example, we now revisit the causes we had identified for the Jex system (Section 2).

Ambiguous exception semantics. As we mentioned earlier, determining the ideal set of abstract exceptions is the true challenge of software compartmenting. To avoid the case of ambiguous exception semantics identified in Section 2, care must be taken to ensure that the meaning of the system abstract exceptions chosen for the new interface do not overlap. For example, in the refined version of Jex, we did not propagate the pre-defined `IOException`, but rather defined two non-overlapping abstract exceptions.

Use of exception values to distinguish between failure types. In our refined approach, instead of using exception values to distinguish between different types of failures, we used subtypes to express the more specific information. Subtypes are easier to track and distinguish than values, making the exception structure easier to reason about. Exception values were used only to carry supplemental information. As an example, in all the case studies, when raising an `AlgorithmicException`, we nested the "original" exception object within the `AlgorithmicException`, to retain information about the particular source of the problem.

Confusion over the use of system-defined exception. This case is avoided simply by refraining from using pre-defined exceptions.

Unbounded unchecked exceptions. Exception guards automatically bound all exceptions at the compartment boundaries.

6.2 Costs and Tradeoffs

Applying this technology to three different programs allows us to roughly evaluate the effort involved in setting up compartments. The basic design problem for software compartmenting is to establish a semantically meaningful set of abstract exceptions. As is the case when designing "normal" module interfaces, the task of designing exception interfaces is difficult to perform and its costs are difficult to assess.

In our experience, once exception interfaces are chosen for the compartments, the effort required to adapt a program to the compartment interfaces primarily comprises two activities:

1. setting up exception guards at compartment boundaries to prevent unanticipated exceptions from escaping compartments, and
2. tracking down meaningful exception sources to map them into the exception interfaces.

The cost of the first activity is roughly proportional to the number of exception guards needed. The number of guards is determined by the granularity of compartments and the size of their functional interface. The cost of the second activity is mostly related to the complexity of the original exception structure. It is difficult to generalize this factor. Tracking down meaningful error sources was easy in Bobby, where only a few exceptions were used. In the case of Jex, this cost was slightly higher, since Jex has more functionality, resulting in the need for a greater number of abstract exceptions. In the case of JTar, re-mapping the error sources to abstract exceptions was particularly arduous, mostly because of the ad hoc error handling scheme used.

7. RELATED WORK

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 [3]. In a later paper, Christian also proposed "a programming language suitable for writing well-structured robust programs" [4, 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 [9] 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 [5].

Tools and modeling techniques integrating exceptions have also been suggested. An example of a tool is OODREX [1], which helps take exceptions into account when designing C++ classes. An example of a notation that integrates exceptions is the Unified Modeling Language (UML) [2]. Additionally, an extension to UML to model exceptions as pre- and post-condition constraints using the Object Constraint Language [14] has been proposed recently by Soundarajan and Fridella [12].

The work described in this paper differs from these efforts as it focuses on reasons why exception structure degrades, and investigates a disciplined design technique to alleviate the problems causing degradation.

More closely related to the problems discussed in this paper is an analysis by Miller and Tripathi [10] of why it is difficult to design exceptions in object-oriented systems. However, their analysis focuses 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. Finally, Lippert and Lopes have investigated how designing exception handling code can be simplified using aspect-oriented programming in AspectJ$^{\text{TM}}$ [7]. Their focus was primarily on reducing redundant information in exception handlers and in enabling different configurations of exception structures rather than in the design of an exception strategy for a system. It is possible that AspectJ provides another way to implement exception guards.

8. SUMMARY

To help developers during design, principles, methods, and notations have been elaborated. For the most part, these design approaches have focused on the normal operations of a system. Despite the presence of mechanisms, such as exception handling, for expressing and separating what a program should do in an unusual situation, there has been less guidance available to help a developer structure the "exceptional" portions of a system.

In our experience, this lack of information about how to design and implement with exceptions leads to complex and spaghetti-like exception structures. To gain insight into why this complexity arises, we reflected upon our experiences trying to build a "good" exception structure into a program analysis tool we were implementing in Java. Based on the causes of complexity which arose in that system and analyses of other Java programs, we believe there are two main factors which contribute significantly to the difficulty of designing exception structures: the global flow of exceptions, and the emergence of unanticipated exceptions. To help control these factors, we refined an existing software compartmenting technique for exception design. We report on our experiences applying it to three different Java programs. In each case, the refined compartmenting approach helped by providing a basis on which to make decisions during exception design.

This paper thus makes two contributions. First, it identifies some reasons why and how exception structure becomes complex. Second, it describes a straightforward set of design guidelines we have used to help simplify exception structure. To date, these guidelines have been applied to existing systems for which some exception handling structure already existed. The next steps are to try applying these guidelines to the development of new systems or new parts of systems, and to track the effect of the guidelines over the longer evolution of the systems.

ACKNOWLEDGMENTS

We would like to thank IBM for providing us with the source code of Bobby. We are also grateful to A. Lai and anonymous reviewers for useful comments on an earlier version of the paper. This work was funded by a NSERC graduate fellowship and research grant.

9. REFERENCES

[1] C. Bamford and B. Dollery. OODREX: An object-oriented design tool for reuse with exceptions. In *Proc. of the International Conference on Object-Oriented Information Systems*, pages 248–251. Springer-Verlag, 1995.

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

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

[4] F. Christian. Correct and robust programs. *IEEE Trans. on Soft. Eng.*, 10(2):163–174, March 1984.

[5] R. de Lemos and A. Romanovsky. Exception handling in a cooperative object-oriented approach. In *Proc. of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 3–13, May 1999.

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

[7] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proc. of ICSE'2000*, pages 418–427. ACM, June 2000.

[8] B. H. Liskov and A. Snyder. Exception handling in CLU. *IEEE Trans. on Soft. Eng.*, 5(6):546–558, Nov. 1979.

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

[10] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In *Proc. of ECOOP'97*, LNCS 1241, pages 85–103. Springer-Verlag, June 1997.

[11] M. P. Robillard and G. C. Murphy. Analyzing exception flow in Java programs. In *ESEC/FSE'99*, LNCS 1687, pages 322–337. Springer-Verlag, Sep. 1999.

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

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

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