Practical Exception Specifications

Donna Malayeri and Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA 15213, USA,
{donna+, aldrich+}@cs.cmu.edu

Abstract. Exception specifications can aid in the tasks of writing correct exception handlers and understanding exceptional control flow, but current exception specification systems are impractical in a number of ways. In particular, they are too low-level, too heavyweight, and do not provide adequate support for describing exception policies.

We have identified the essential properties of a practical exception specification system and we present a methodology and tool that provides integrated support for specifying, understanding, and evolving exception policies. The annotations required of the programmer are lightweight and easier to maintain than those of current systems; in our studies we observed a 50% to 93% reduction in annotations. By leveraging these annotations, our system provides scalable support for understanding exception flow and for modifying exception annotations.

1 Introduction

Exceptions can be very useful for separating normal code from error handling code, but they introduce implicit control flow, complicating the task of understanding, maintaining, and debugging programs. Additionally, testing is not always effective for finding bugs in exception handing code, and these bugs can be particularly problematic (for example, a program that crashes without saving the user's data).

For programmers to write correct exception handlers, they need precise information about all exceptions that may be raised at a particular program location. Documentation is inadequate—it is error prone and difficult to maintain. On the other hand, precise information can be obtained through a whole-program analysis of exception flow (including analysis of all libraries used), but this is not a scalable solution, nor is it even applicable in situations where the whole program is not available. Moreover, relying on whole program exception analysis would complicate team development; if one programmer changes exception-related code, the control flow in apparently unrelated parts of the program may change in surprising ways.

1.1 A motivating example

We now present an example to motivate the need for practical exception specifications, and to illustrate the way a programmer would use our tool, ExnJava. Consider a module whose intended abstraction is that it present to its clients a high-level view for managing user preferences. The module is to hide implementation details of how the preferences are actually stored; clients should not depend on such details.

Fig. 1. Excerpts of Java code for managing user preferences. The classes are part of the package `util.userPrefs`.

```
1   public class UserPrefs {
2     public Object getValue (String key) {
3       ...
4       return PrefKeys.getValue(this.userID, key);
5     }
6     public void setValue (String key, Object value) {
7       ...
8       PrefKeys.setValue(this.userID, key, value);
9     }
10  }

11  class PrefKeys {
12    // class' methods are package-private
13    void Object getValue (String userID, String key) {
14      ...
15      return Serializer.readValue(userID, key);
16    }
17    static void setValue (String userID, String key, Object value) {
18      ...
19      Serializer.writeValue(userID, key, value);
20    }
21  }

22  class Serializer {
23    // class' methods are package-private
24    static Object readValue (String id, String key) {
25      ...
26      // old code: return registry.getValue(key);
27      return theDB.get(id, key);
28    }
29    static void writeValue (String id, String key, Object value) {
30      ...
31      // old code: registry.save(id);
32      theDB.set(id, key, value);
33    }
34  }
```

See Figure 1 for the corresponding code, written in Java. The classes `UserPrefs`, `PrefKeys` and `Serializer` reside in the `util.userPrefs` package, for which the programmer supplies the following package exception specification:

```
package util.userPrefs may only throw util.PrefException
```

This specifies the policy that `PrefException` is the only checked exception that can be thrown from the public methods of `util.userPrefs`. This is to prevent an interface method from throwing a low-level exception, in which case clients would not be able to write a meaningful exception handler without knowing the module's implementation details.

The class `UserPrefs` is the interface to the package; `PrefKeys` and `Serializer` are package-private. `PrefKeys` just passes through to `Serializer`, which previ-

ously stored preferences in the registry (lines 26 and 31), but now uses a database (lines 27 and 32). Note that preferences values are stored in the database as soon as `UserPrefs.setValue` is called.

Suppose that in the old version of the code, the calls to `registry`'s methods in lines 26 and 31 did not throw an exception. If the registry did not contain a value for a particular key, the `registry` object simply returned an appropriate default. Similarly, a failed save to the registry was simply logged by the `registry` class.

Now, however, `theDB.get` (line 27) and `theDB.set` (line 32) *can* throw an exception—a `SQLException` in this case. Since in Java, uncaught exceptions are automatically propagated up to the caller, all 6 methods now also throw this exception: `UserPrefs.getValue`, `UserPrefs.setValue`, `PrefKeys.getValue`, `PrefKeys.setValue`, `Serializer.readValue` and `Serializer.writeValue`.

Accordingly, the Java typechecker would require that all of these methods update their exception declarations. Though only 2 lines of code have been changed, programmers must now manually update the exception declarations of 6 methods—a threefold increase even in this simple example. This factor only increases in larger programs with more complex control flow. In Section 3, we will explore this problem in more detail, as well as look at the problems of other exception specification systems.

In contrast, in ExnJava, programmers need not add all of these `throws` declarations; `throws` annotations are only required on the public interface of a package. Since only the methods of `PrefKeys` are visible outside the package, these are the only methods that need to have a `throws` declaration; the `throws` declarations are automatically inferred for the other 4 methods. However, ExnJava will not accept the definition of `UserPrefs` as is—its package exception specification does not allow `SQLException` to be thrown from public methods. This error prompts the programmer to make a decision regarding how `SQLException` should be handled. The programmer realizes that line 27 in `Serializer.readValue` should be wrapped in a `try-catch` block:

```
catch (SQLException e) {
    return DefaultsManager.getDefault(key); }
```

For the exception originating from `Serializer.writeValue`, the correct behavior is that the exception be caught in `UserPrefs.getValue` and rethrown, so a handler is added around line 8:

```
catch (SQLException e) { throw new PrefException(e); }
```

1.2 Exception specifications

As shown in the example, exception specifications can be a useful tool for reasoning about exceptions (see, for example, [18, 15, 2, 7]). They serve to document and enforce a contract between abstraction boundaries, which facilitates modular software development, and can also provide information about exception flow in a scalable manner. In Section 2 we describe the essential properties that we believe a practical exception specification system must have: it should be lightweight while sufficiently expressive, and should facilitate creating, understanding, and evolving specifications.

We have provided an integrated methodology for practical use of exception specifications and have designed a tool, described in Section 4 that leverages this. The tool combines user annotations, program analysis, refactorings, and GUI views that display analysis results. Our methodology and tool raise the level of abstraction of exception specifications, making them more expressive, more lightweight, and easier to modify. As we saw in the example above, this can aid programmers in writing better exception handlers.

Note that we focus on the problem of *specifying* various properties of exception behavior, rather than a proposal for a new exception handling mechanism. Additionally, though our work is performed in the context of Java, much of our basic design would be applicable to languages with similar exception handling mechanisms. There are two key properties of Java's exception handling mechanism: handlers are bound dynamically (the call stack is searched to find the handler, and cannot in general be statically determined), and exception propagation is performed automatically (if a method does not handle an exception, it is automatically propagated to its caller). The exception handling mechanisms of many popular languages have these properties, such as Ada, Eiffel, C++, C#, and ML.

The contributions of system are as follows:

- It performs a modular exception analysis and allows developers to easily browse the exception control flow within an application. This allows developers to easily answer questions about non-local exception behavior, and shows information about exceptions that are not provided by the host language.
- It allows developers to express and check exception specifications at the level of program modules. This raises the level of abstraction of exception specifications, helping developers to use exceptions consistently across many functions or methods in a module.
- It makes exceptions more lightweight by allowing developers to omit exception specifications from methods internal to a module, and providing refactoring tools that support transitive changes to the remaining specifications. These tools reduce the overhead of exception specification in general, and make evolving exception specifications as lightweight as the corresponding edits to throw and catch clauses in the code.

Additionally, in Section 5 we report both quantitative and qualitative data on the use of exception specifications in open source Java applications. These data motivate and validate the design of our tool.

2 Practical Exception Specifications

If an exception specification system is to be practical, we believe that it must posses several essential properties; we enumerate these here. We use the general term "exception policy" to refer to programmers' design intent regarding how exceptions should be used and handled. An exception policy specifies the types of exceptions that may be thrown from a particular scope and the properties that exception handlers must satisfy. We use the general term "module" to refer to a set of logically related compilation units

to which access control can be applied (i.e., the module can have a public interface, and private implementation details).

In our view, a good exception specification system, which may include both language features and tools, should be lightweight while sufficiently expressive, and should facilitate creating, understanding, and evolving specifications.

2.1 Specification Overhead

The specification system must be lightweight. Programmers are not fond of writing specifications, so the benefits must clearly outweigh the costs. Additionally, incremental effort should, in general, yield incremental results. If a specification system requires that an entire program be annotated before producing any benefit, it is unlikely to be adopted.

2.2 Expressiveness

The system should allow specifying exception policies at an appropriate level of abstraction. It should support the common policy of limiting the exception types that may be thrown from some scope. Such scopes need not be limited to a method or a class. Rather, they could consist of a set of methods, a set of classes, or a module. In our example in Section 1.1, we illustrated how a programmer might use such a policy.

Additionally, there should be a way to specify a policy independently of its implementation, though an implementation may perhaps be generated from a policy (e.g., code to log exceptions, or wrap some exception and rethrow). Solutions that make it easy to implement a policy are useful, but they do not obviate the need for one. Until it is possible to generate all desired implementations automatically—which may not ever be fully achievable—we believe that the distinction between specification and implementation is an important one.

2.3 Ease of Creating and Understanding Policies

The solution should provide tools that aid programmers in creating new exception policies and understanding existing policies. Without the aid of such tools, reasoning about exceptions is difficult due to their non-local nature. Such tools may, for example, include information on exception control flow.

2.4 Maintainability

The specification scheme should support evolving specifications as the code evolves, possibly through tool support. This differs from the property of being lightweight; a system may be lightweight but inflexible. The cost involved in changing specifications should generally be proportional to the magnitude of the code change.

In Java and in other commonly-used languages, exceptions automatically propagate up the call chain if there is no explicit handler. A specification system for these languages should take these semantics into account, so that small code changes do not require widespread specification changes.

3 Related Work

Previous solutions have failed to meet one or more of the criteria described above; we describe each of these here.

3.1 Java

One well-known exception specification scheme is that of Java, which requires that all methods declare the checked exceptions that they directly or indirectly throw.[1]

Though we believe it is useful to separate exceptions into the categories of checked and unchecked (see, for example, [14, 2]), the Java design has a number of problems that make it impractical. Java `throws` declarations are too low-level; they allow specifying only limited exception policies at the method level. This leads, in part, to high specification overhead. It is notoriously bothersome to write and maintain `throws` declarations. Simple code modifications—a method throwing a new exception type; moving a handler from one method to another—can result in programmers having to update the declarations of an entire call chain.

There is anecdotal evidence that this overhead leads to bad programming behaviors [4, 26, 10]. Programmers may avoid annotations by using the declaration `throws Exception` or by using unchecked exceptions inappropriately. Worse, programmers may write code to "swallow" exceptions (i.e., catch and do nothing) to be spared the nuisance of the declarations [19, 13].

Eclipse[2] provides a "Quick Fix" for updating a method's `throws` declaration if it throws an exception that is not in its declaration, but this can only be applied to a single method at a time. Consequently, programmers would have to iteratively update declarations until a fixpoint was reached. Eclipse also includes an optional warning that will list methods whose `throws` declarations are imprecise, but this too applies to a single method at a time.

Empirical results Based on our experience with Java programs, we hypothesized that even if programmers use checked exceptions as the language designers intended, exception declarations can easily become imprecise. To verify this, we analyzed several open-source Java programs. Descriptions of the programs analyzed, along with their code size, are displayed in Table 1.

In this discussion, we use the following definitions: a *throws conjunct* is an individual exception type listed in a `throws` declaration. For example, in the declaration "`throws` E_1, E_2", E_1 and E_2 are throws conjuncts. A throws conjunct E of a method m is *imprecise* if the analysis determines that m throws a proper subtype of E, but not

[1] In Java, the class `Exception` is the supertype of all exception types. One of its subtypes is `RuntimeException`, which represents *unchecked* exceptions. Exceptions that are a subtype of `Exception` but not a subtype of `RuntimeException` are *checked* exceptions; subtypes of `RuntimeException` are *unchecked exceptions*. A method must declare all checked exceptions that it throws (directly or transitively) in its `throws` declaration; unchecked exceptions may be omitted.

[2] Available at `www.eclipse.org`.

Table 1. Description of programs analyzed, along with lines of code (LOC denotes the non-comment, non-blank lines of code) and number of classes and methods. The program *Tapestry* refers to the Apache Jakarta Tapestry project.

Program	LOC	Classes	Methods	Description
LimeWire	61k	1291	8346	p2p filesharing client
Columba	40k	1054	5654	e-mail client
Tapestry	20k	515	3186	framework for developing web applications
JFtp	13k	104	1005	graphical network browser
Lucene	10k	178	1335	text search engine library
Metrics	7k	203	1378	Eclipse plugin, computes program metrics

Table 2. The number of throws conjuncts that were imprecise (proper subtype thrown), and superfluous (not thrown at all), the total number of throws conjuncts, and the percentage of throws conjuncts that were imprecise or superfluous. Within the *imprecise* and *superfluous conjuncts* columns, the total is displayed, as well as the number of instances where the imprecise or superfluous throws conjunct was the exception supertype `Exception`. The *subsume* sub-column within the *imprecise conjuncts* column indicates the number of exception types that were subsumed by the `Exception` declaration.

Program	Imprecise conjuncts			Superfluous conjuncts		Total throws conjuncts	Percent of Total
	Total	*e* = `Exception`		Total	*e* = `Exception`		
		occurrences	subsume				
LimeWire	**26**	2	4	**120**	0	917	**16%**
Columba	**275**	274	1130	**301**	272	826	**70%**
Tapestry	**16**	10	30	**93**	20	231	**47%**
JFtp	**5**	5	9	**17**	1	27	**81%**
Lucene	**7**	2	5	**209**	0	598	**37%**
Metrics	**1**	0	0	**17**	0	78	**23%**

E itself; E is *superfluous* if m does not throw E nor any of its subtypes. An *imprecise throws declaration* contains one or more imprecise or superfluous throws conjuncts; a *superfluous throws declaration* is comprised entirely of superfluous throws conjuncts.

Table 2 summarizes the number of imprecise and superfluous throws conjuncts, the total number of throws conjuncts, and the percentage of throws conjuncts that were either imprecise or superfluous. The table also lists the number of cases where the imprecise or superfluous conjunct was `Exception`. The declaration `throws Exception` is particularly problematic: aside from providing no information about exception flow, it precludes the method's client from writing anything other than a general exception handler.

In our subject programs, there were numerous imprecise and superfluous throws conjuncts; their totals range from 19% to 85% of the total throws conjuncts. Averaged over all the programs, one out of every 2 throws conjuncts is imprecise or superfluous. The number of occurrences of `throws Exception` was also quite high in Columba and Tapestry, relative to the exception declarations.

In the cases where the declaration `throws Exception` was imprecise, we computed the number of exception types that it subsumed, which is a measure of *how* imprecise the declaration is. For example, if method m declares that it throws `Exception`, but can actually throw `SMTPException`, `WrongPassphraseException`, and `ParserException` (all exception types defined by Columba), then in a sense, m's declaration is more imprecise than if it only threw `SMTPException`. The three excep-

tion types represent three opportunities for writing a handler that is specific to the error condition that was raised, but this is obscured by m's `throws` declaration.

Table 2 includes the total number of throws conjuncts that were subsumed by a more general type (e.g., for m, 3 conjuncts were subsumed). In Columba, the 274 instances of the imprecise declaration `throws Exception` subsumed 1130 exception types; an average of 4.1 per occurrence. We also observed several cases where 7 or 8 exception types, each of which appeared to represent semantically distinct error conditions, were subsumed by the type `Exception`. Thus, the imprecise `throws` declarations were large in both number and magnitude.

Aside from illustrating the difficulty of maintaining `throws` declarations, these results cast doubt on whether they are even a good tool for understanding exception flow and exception policies—though advocates often claim that this is one of their very benefits [8, 25, 2, 24]. Imprecise and superfluous `throws` declarations can obscure exception flow, leading to violations of intended exception behavior or `catch` blocks that are dead code.

3.2 Other Work

There are several proposals for specifying method post-conditions on exceptional exit [5, 15, 1], but these are even more heavyweight than Java `throws` declarations. These solutions do provide powerful verification capabilities, but it is unclear whether these benefits will outweigh the significant cost of annotating an entire program.

Some researchers have proposed languages and language extensions to facilitate the implementation of a policy, but these solutions provide no way to *specify* the policy. These include languages with first-class exception handlers [3] and languages that allow applying handlers to some set of methods or classes [12, 16]. However, unless new tools are created, these features will further complicate the task of reasoning about exceptions. It is also unclear how these schemes would work with programmer-supplied specifications; as far as we are aware, this problem has not been addressed.

Robillard and Murphy [21] provide a good methodology (though not a tool) for specifying exceptions at module boundaries; our tool builds on this work. A number of researchers have developed exception analysis tools [22, 23, 11, 6], and while most of these analyses are more sophisticated in terms of either precision or efficiency, they all perform a whole-program analysis which has inherent scalability problems [20]. Most of these analyses are superior to the one that we have implemented in our prototype tool, as the goal of our work is not to perform exception inference but rather to provide tools and a methodology for lightweight exception specifications.

For the task of understanding exception flow, Sinha et al. [23] propose a set of views that display the results of their exception analysis, but for these they provide only a high-level design.

4 Features of ExnJava

We have designed and implemented an exception specification system for Java 1.4 that satisfies the initial criteria outlined in Section 2. Our design raises the level of abstraction of exception specifications, while remaining lightweight.

Our current simplifying assumption is to take a Java package as a unit of modularity. Thus, methods with public or protected visibility are the package's interface methods; private and package-private[3] methods are internal methods.

Our system, ExnJava, is implemented as an Eclipse plugin. It contains one language change: the Java rules for method `throws` declarations are relaxed such that only package *interface methods* require a `throws` declaration. ExnJava also includes package-level exception specifications, checked on every compilation. This is implemented as an extra-linguistic feature. Additionally, there is a Thrown Exceptions view to facilitate creating and understanding exception policies. Three refactorings help programmers evolve specifications: Propagate Throws Declarations, Convert to Checked Exception, and Fix Imprecise Declarations. In the subsections below, we describe each of these features.

4.1 Exception inference

The main goal of ExnJava's exception inference is to determine the checked exceptions that each method throws, and use these to ensure that methods adhere to package exception specfications and to infer the `throws` declarations for private and package-private methods. Our analysis does compute the unchecked (i.e. "runtime") exceptions that are explicitly thrown by the program, but this analysis is unsound and intended merely to provide additional information. The reason for this design choice is that the goal of our work is to allow programmers to specify and check exception *policies*, and unchecked exceptions are not suitable for this task.

Analysis modes ExnJava can be run in two modes: *per-package mode* or *whole-program mode*. Programmers can choose the former for a more scalable analysis, or the latter for more detailed information. Both analyses provide complete information about the checked exceptions that each method throws, but the whole-program analysis provides information about exception control flow between packages. The same views and refactorings are available in either mode (with the exception of the Convert to Checked Exception refactoring).

Per-package analysis This is the main mode of operation for ExnJava. This analysis is quadratic in the size of the largest package and scales well to large programs.

Whole-program analysis This analysis is useful in cases where a programmer requires a detailed understanding of exception flow, and for the Convert to Checked Exception refactoring. However, its worst-case complexity is quadratic in the size of the program and therefore may not be efficient enough for frequent interactive use.

Analysis algorithm Since ExnJava does not require throws declarations on all methods, there is an inference algorithm to determine the checked exceptions thrown by

[3] Also known as "default" or "friendly" access.

internal methods. ExnJava analyzes each package as a whole and performs a conservative intra-procedural dataflow analysis, similar to that of previous systems [22]. The analysis determines the checked exceptions thrown by each internal method (through either a `throws` statement or a call to an interface method or a library method) and iterates until a fixpoint is reached. For calls to non-final (i.e., virtual) methods, the union of the exceptions thrown by all known overriding methods are considered.

4.2 Specifying Exception Policies

In ExnJava, programmers specify exception policy at the package level. We believe this is a more appropriate level of abstraction than the low-level declarations of previous solutions, such as method-level declarations in Java. A package has two kinds of exception policy: one applies to each individual interface method, the other to the package as a whole.

Interface Method Policies. The exception policy of interface methods is specified using Java `throws` declarations. In contrast to Java however, the declarations for internal methods need not be specified—they are inferred by ExnJava.[4] Consequently, this design raises the level of abstraction of `throws` declarations.

To determine the checked exceptions thrown by internal methods, ExnJava performs an intra-package dataflow analysis. Within a package, the implementation of our analysis is similar to the whole-program analyses of previous systems [22, 23]. However, our analysis is scalable, as it depends on the size of each package rather than the size of the entire program. The results of this analysis, as well as additional information about exception control flow, are displayed in the Thrown Exceptions view, described below in Section 4.3.

There are several advantages to this scheme. First, annotations are more lightweight. As we describe in Section 5.1, in our subject programs we found that inference reduces the number of required declarations by a range of 50% to 93%. Also, inference gives programmers more precise information. Rather than examine Java `throws` declarations, programmers use the Thrown Exceptions view to determine the checked exceptions thrown by internal methods. And, in contrast to a pure exception inference tool, programmers can enforce exception policies by specifying `throws` declarations on interface methods.

Package Exception Policies. Our design of package exception policies extends the work of Robillard and Murphy [21]. Their work, in turn, builds on work by Litke [17], who provides recommendations for designing systems with good exception structure (in the context of Ada). Litke's recommended software engineering practice is that exceptions that can occur at module boundaries to be precisely and completely specified.[5] Litke argues that using modules reduces complexity, making it easier for programmers to reason about program behavior and to write and modify error handlers. He also

[4] It may sometimes be useful to include `throws` declarations on internal methods; this is supported.

[5] Litke uses the term "compartment", but this is equivalent to our definition of "module."

recommends the use of automated methods for checking conformance of the program against the module specifications.[6]

For simplicity, our approach considers each Java package as a module; the package's public and protected methods are the module boundaries.[7] For each package, its exception specification (which corresponds to Litke's module specification) consists of a list of entries that are either exception class names or regular expressions (e.g., `java.lang.*`). An exception type E can be thrown from a package interface method if either E or a supertype of E is listed in the specification, or if E's fully-qualified name matches a regular expression in the specification.

Unchecked exceptions can be included in the specification; however, to ensure conformance, the entire program, including any libraries that are used, would have to be analyzed. Therefore, the analysis can find some violations of such specifications, but cannot assure complete conformance with the specified policy.

Package exception specifications thus ensure that the exception policy of each interface method (the types of exceptions that they throw) also conforms to the general exception policy of the package. Recall the example of Section 2.2 where the storage details of the user preferences package were to be hidden from clients. This package's specification would include perhaps `PreferenceStoreException` but would *not* include `FileNotFoundException`. If such an exception were thrown within the package, the exception could be wrapped as a higher-level exception type (such as `PreferenceStoreException`). This idiom, called exception translation, is recommended by Bloch [2] (among others) and is common in Java programs.

Figure 2 is a representative excerpt of our package exception specification of the Columba program; we defined this specification while performing the case study described below (Section 5.2). The entry for the `org.columba.mail.plugin` package is particularly interesting, as it suggests that some low-level exceptions are inappropriately thrown, while they should probably be wrapped by a more abstract exception type. We will revisit this issue in Section 5.2.

4.3 Understanding Exception Policies

The Thrown Exceptions view (Figure 3) displays the details of exception control flow, to help programmers understand the implemented exception policies. Without the information provided by this view, we believe that it would be difficult to correctly create and modify exception policies. We believe that the general difficulty of programming with exceptions is partly due to lack of information on a program's exceptional control flow.

The Thrown Exceptions view displays information computed by either a whole-project analysis or a per-package exception analysis (as described above); the former will provide more information, but the latter is more scalable. The view has two modes: method level and package level. The method level view, inspired by the work of Sinha

[6] As far as we are aware, Litke did not publish any work regarding such a tool, though one paper [17] mentioned that an implementation was in progress.

[7] It would also be possible to devise a module specification language, where a module could consist of a set of packages or classes; we will consider this in future work.

Fig. 2. Excerpt of our package exception specification for the Columba application. `IOException` refers to `java.io.IOException`. The other unqualified exception names refer to application-defined checked exceptions.

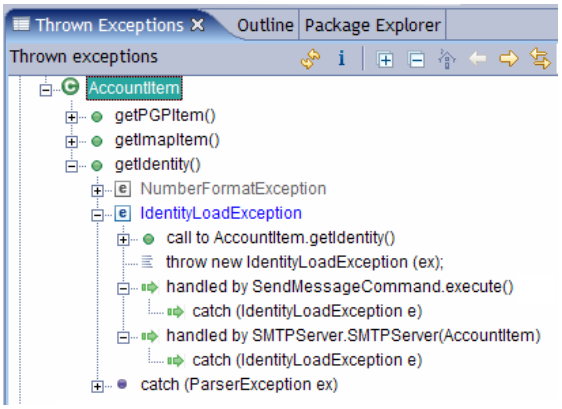
```
org.columba.mail.pop3    IOException, CommandCancelledException, ParserException,
                         POP3Exception

org.columba.mail.smtp    IOException, CommandCancelledException, SMTPException

org.columba.mail.pgp     IOException, WrongPassphraseException, JSCFException

org.columba.mail.plugin  java.lang.ClassNotFoundException,
                         java.lang.IllegalAccessException,
                         java.lang.InstantiationException,
                         java.lang.NoSuchMethodException,
                         java.lang.reflect.InvocationTargetException,
                         java.net.MalformedURLException,
                         PluginHandlerNotFoundException
```

Fig. 3. The Thrown Exceptions view in method level mode.



et al [23], displays a tree view of the project's methods, grouped by package and class. For each method, the checked and unchecked exceptions[8] thrown by the method are listed, as well as the lines of code that cause the exception to be thrown. Using this view, the programmer can jump to method definitions that throw exceptions, and can also quickly jump to the ultimate sources of a particular exception (i.e., the original throw statements or library method calls that caused an exception to flow up to this part of the code.) Additionally, for each exception that a method throws, the view displays all catch blocks that may handle that exception. (This is limited, of course, to catch blocks in code available to the analysis.)

[8] Information on unchecked exceptions will not be complete, due to the fact that a whole-program analysis (including all libraries used) would be required. However, even partial information on unchecked exceptions can be useful.

The package level view displays, for each package, the checked exceptions that are thrown by its interface methods. For each exception type, the methods that throw the exception are listed, as well as the detailed exception information described above. The package view can be useful for creating a package's exception policy. (In fact, we used it to define Columba's exception policy in Figure 2.) The view can also help identify possible errors in the exception policy. For example, if a particular exception type is only thrown by one or two methods, it is possible that the exception should have been handled internally or wrapped as a different exception type.

4.4 Evolving Exception Policies

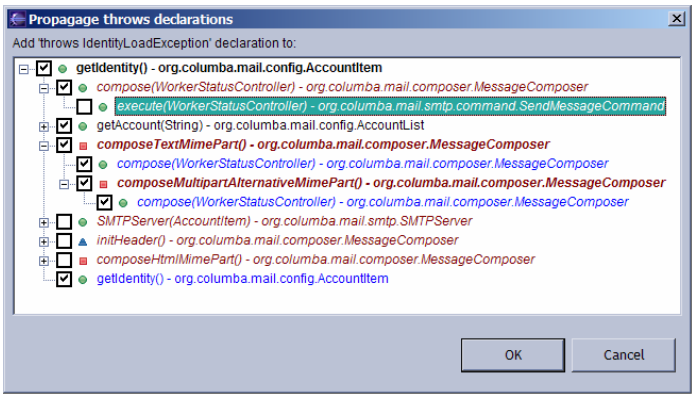
Our system raises the unit of abstraction to which an exception specification applies; this alone makes it easier to evolve specifications. If the set of exceptions thrown by an internal method changes, no `throws` declarations need to be updated, unless one or more interface methods throw new exceptions. This often occurs when an exception handler is moved from one method to another in the same package. Though this is a conceptually simple modification, a number of internal methods may now throw a different set of exceptions. In standard Java, the `throws` declaration of each of these methods would have to be manually updated.

Propagating Declarations. Still, if a code change causes an *interface* method to throw new exceptions, the same "ripple effect" of Java `throws` declarations may result—requiring changes to the declarations of the transitive closure of method callers. To avoid this problem, ExnJava provides a Propagate Throws Declarations refactoring (accessible as an Eclipse "Quick Fix") that will propagate declarations up the call graph (see Figure 4). The goal of this refactoring is to help programmers find the correct location for new exception handlers, rather than tempting them to carelessly propagate declarations to every method that requires them. To this end, the refactoring displays a checkbox tree view of the call graph (which includes only methods whose declarations need to be changed), which is initially collapsed to show only the original method whose declaration needs to be updated. The programmer then expands this one level to display the method's immediate callers (and callers of the methods that it overrides), and so on for each level in the tree. Checking a particular method in the tree will add the declaration to both that method and all the overridden superclass methods (so as not to violate substitutability).

The refactoring also incorporates the package exception specification; if updating the `throws` declaration of a particular method would violate the package specification, the method is displayed in a different color, with a tooltip describing the reason for the inconsistency. The declaration for the method can still be changed, but ExnJava will display an error until the package specification is modified.

Unchecked Exceptions. Sometimes, unchecked exceptions are used where checked exceptions are more appropriate. In fact, some programmers prefer to use unchecked exceptions during the prototyping phase, and then switch to checked exceptions later.

Fig. 4. The dialog for propagating throws declarations. Methods that are typeset in italics are those for which the package specification does not allow throwing this particular exception type.



ExnJava includes a Convert to Checked Exception refactoring which changes an exception's supertype to `Exception` and updates all `throws` declarations in the program accordingly.

Imprecise Exceptions. As previously noted, `throws` declarations can become unintentionally imprecise as code evolves: they may include exception types that are never thrown or types that are too general. (Of course, sometimes imprecise declarations are an intentional design choice, to provide for future code changes. Our tool allows programmers to retain such declarations.)

When a `catch` block is moved from one package to another, for example, a number of interface methods may include an exception type that they will consequently never throw. New callers of these methods will then have to include handlers for these exceptions—which would be dead code—or must themselves add superfluous exceptions to their `throws` declarations. Such problems do occur in actual code; for example, Robillard and Murphy found a number of unreachable catch blocks in their analysis of several Java programs [22].

To solve this problem, ExnJava includes an Fix Imprecise Declarations refactoring, which can be run on a package or set of packages. The refactoring first lists the exception types which appear in imprecise declarations; the programmer chooses an exception type from this list. The exception type is chosen first so that the view can show the propagation of this exception declaration. For this exception, the view displays all methods where that type appears in an imprecise declaration. The view displays a call graph tree (similar to that of the Propagate Throws Declarations refactoring) showing the propagation of imprecise declarations. This allows the programmer to determine the effect of fixing (or not fixing) a particular imprecise declaration. Initially all methods are checked, indicating that their declarations will be updated; the programmer can choose to not change the declarations for particular methods by unchecking them. (We chose

this design as we hypothesize that most imprecise declarations are out-of-date rather than intentional design choices.) The view ensures that a consistent set of methods is chosen; if a method is unchecked, all of its transitive callers will also be unchecked.

Our tool could be extended to include a "Fix Imprecise" refactoring at the package specification level, to inform the programmer of specifications that may no longer be valid. Such a tool would display each package whose specification lists one or more exceptions that are not actually thrown in the implementation.

5 Evaluation

We evaluated ExnJava with quantitative analyses and with case studies. Exception inference was evaluated for 1) its potential annotation savings and 2) its impact on reducing the incidence of imprecise and superfluous `throws` declarations. We also analyzed the annotation overhead of package exception specifications. Finally, we conducted case studies to determine how ExnJava could be used to improve a program's exception structure and ease program understanding and maintenance.

5.1 Quantitative Results

Package-private inference. ExnJava's checked exception inference is most useful for programs with well-encapsulated packages with as few public members as possible. We hypothesized that the visibility of classes and methods are often not restrictive enough; that is, many classes and methods are public when in fact they should be package-private or private. To this end, we have developed an Eclipse plugin that changes the visibility modifiers on classes and methods to *package-private* wherever possible (i.e., when they are not accessed outside of their defining package).

Table 3. Percentage of methods that were private or package-private before and after visibility modifier refactoring. The results show that a significant percentage of methods have weaker visibility modifiers than are necessary.

Program	Methods	Package-private and Private Methods	
		before refactoring	after refactoring
LimeWire	8346	38%	67%
Columba	5654	10%	57%
Tapestry	3186	14%	75%
JFtp	1005	14%	58%
Lucene	1335	46%	73%
Metrics	1378	27%	72%

We found that the percentage of package-private methods increases dramatically when this refactoring is applied; results are displayed in Table 3. Grothoff et al. found similar results in their work on confined types [9]. Note that in computing this data, we considered the *actual* visibility of methods, not merely the method's access modifier (e.g., a method with `public` visibility in a `private` class was counted as a private method). Before refactoring, an average of 25% of methods were private or

package-private, as compared to an average of 67% after refactoring. However, for some of these programs (for example, Tapestry and Lucene), the refactoring may have been too aggressive: it is likely that some of the classes and methods that were changed to package-private were intended to be used by library clients. On the other hand, classes in the programs LimeWire, Columba, JFtp, and Metrics were not intended to be used directly by clients (aside from some plugin capabilities in Columba and Metrics that were excluded from the refactoring), so the refactoring for these was likely accurate.

Effectiveness of inference. In the discussion that follows, *inferable checked exceptions* denote checked exceptions that could be inferred by our analysis, and therefore omitted by the programmer (that is, exceptions thrown by private and package-private methods). Table 4 compares the number of inferable checked exceptions before and after the package-private refactoring was performed, as computed by a whole-program analysis. As expected, the percentage of inferable exceptions was closely related to the percentage of private and package-private methods in the program. In these programs, 50% to 93% of thrown checked exceptions could be omitted with ExnJava.[9]

Table 4. Total number of exceptions thrown, and percentage of inferable checked exceptions before and after package-private refactoring. Each exception type thrown by a method was counted separately. Exceptions thrown were computed using the whole-program exception analysis.

Program	Checked	Percent Inferable Exceptions	
	Exns Thrown	before refactoring	after refactoring
LimeWire	966	45%	**72%**
Columba	1510	44%	**50%**
Tapestry	146	12%	**75%**
JFtp	14	44%	**93%**
Lucene	492	56%	**81%**
Metrics	54	23%	**72%**

We also found that many occurrences of imprecise and superfluous throws conjuncts are on private and package-private methods; see Table 5. The data suggests that 53% to 78% of these could be eliminated without any additional tool support. That is, by simply removing all `throws` declarations from private and package-private declarations, ExnJava will eliminate more than half of the imprecise declarations through its exception inference. Of course, there are still a number of public and protected methods with imprecise or superfluous declarations, for which ExnJava does not perform inference. As it would be very error-prone and tedious to correct these by hand, we believe that tool support such as our Fix Imprecise Declarations refactoring would be beneficial.

We computed the average number of exceptions thrown by packages in our subject programs, and our results indicate that package exceptions specifications have a very low annotation overhead: in most applications, packages generally throw a small number of exception types—fewer than 2 exceptions per package, on average.

[9] This data essentially assumes that all `throws` declarations are precise. However, we derived very similar results when considering the actual, imprecise `throws` declarations.

Table 5. The number of imprecise or superfluous throws conjuncts and the number of superfluous `throws` declarations; the percentage of each these that are on private or package-private methods (and that could therefore be inferred). Data was gathered after package-private refactoring was performed.

Program	Imprecise/ superfluous conjuncts	Private or pkg-private	Superfluous "throws" decl	Private or pkg-private
LimeWire	146	53%	108	42%
Columba	576	42%	291	40%
Tapestry	109	68%	72	69%
JFtp	22	59%	16	44%
Lucene	216	75%	209	75%
Metrics	18	78%	13	69%

5.2 Case studies

We used the programs Columba and LimeWire as subjects of our case studies, as they are the largest two programs and contain the most uses of exceptions. The two programs were quite different in their exception structure: Columba had very poor exception structure and exception handling. Though it had no application-defined unchecked exceptions, the declaration `throws Exception` was ubiquitous. In contrast, LimeWire had a well-designed exception structure. Despite the fact that it was the largest program we analyzed and contained the most uses of exceptions, it had the fewest percentage of imprecise and superfluous exceptions (see Table 2).

Columba. As described in Section 3.1, Columba had many imprecise `throws` declarations, and these subsumed an average of 4.1 exception types that were actually thrown. Thus, in this program, `throws` declarations are useless for inferring a package exception policy; considering `throws` alone, of the 61 packages that threw exceptions, 38 of them contained `Exception`, or 62%.

We therefore examined the package exception specification for Columba as inferred by the whole-program analysis. We found that two packages, `org.columba.mail.plugin` and `org.columba.core.plugin`, appeared to be throwing exceptions that were inappropriate to the abstraction (see Figure 2). These packages were involved with loading user-defined plugins and were each throwing 5 low-level exceptions related to class loading. Inspecting the `throw` point for each of these exception revealed that they all originated from the `org.columba.core.loader` package, which actually performed the class loading operation. Further, there were no specific handlers for the 5 low-level exceptions in clients of the plugin packages, while there were handlers for an existing exception `PluginLoadingFailedException`. In the plugin packages, we found a few cases where the low-level exceptions were translated to this type. This all suggests that the intended policy was that the class loading-related exceptions be wrapped as the more meaningful `PluginLoadingFailedException`, but this policy was inconsistently applied.

Adding just two handlers to perform this exception translation had the effect that only the `core.loader` package threw the low-level exceptions and the `plugin` packages threw only plugin-related exceptions. The total number of exceptions thrown by

each package was reduced from 201 to 193 (at the very start of our refactoring, this total was 229); the resulting exception structure is simpler and better modularized. This could probably be improved further, but would require an understanding the design intent of the various packages.

We were also surprised to discover that our inference determined that the supertype `Exception` was thrown by 7 different packages. Similar to a throws declaration of `throws Exception`, a package specification that includes `Exception` provides no information and essentially circumvents the package specification checking. The source of the problem was two abstract methods relating to the operation of encoding a MIME attachment in an e-mail message. We will discuss one of these methods, `renderMimePart`, though the discussion applies to the other method as well (in fact, the same catch block handled exceptions from both methods).

The abstract method `renderMimePart` was presumably given the declaration `throws Exception` to provide implementers with maximum flexibility. However, this flexibility comes at a high cost: none of its clients would be able to write specific exception handlers, unless they knew that some particular mime encoder object was used. Most of the implementations of this method did not throw any exceptions, but the implementation in `MultipartEncryptedRenderer` could throw a checked exception, `EncryptionException`. This exception was a direct subtype of `Exception`.

Using our exception view (Figure 3) which shows the handlers that may handle a particular exception, we determined that there was only one catch block c that could catch exceptions from this source, and that it did not catch any instances of `EncryptionException` originating from other throw points. The handler c was contained within code that was responsible for sending an e-mail message. If an `EncryptionException` was thrown, the message composer window would be displayed along with an error message. Note that the action that was taken was related to the fact that an exception occurred during MIME rendering, not that there was a problem during encryption. We therefore created a new exception type `MimePartRenderException` and wrapped the instance of `EncryptionException`.

Note that it would not be easy for a developer to produce a correct implementation for the original handler c. He could not simply look at `throws` declarations to determine that `EncryptionException` could be thrown from the operation of MIME rendering. There were many methods (including several that were called indirectly) between the catch and the corresponding `throw` points, and all were annotated with `throws Exception`. We speculate that this exception handler was added after testing uncovered this execution path.

We updated the `throws` declarations in the methods that we changed and after full exception inference the imprecision of `throws` declarations were even more pronounced. There were an additional 81 imprecise or superfluous throws conjuncts—now 80% of all throws conjuncts were imprecise or superfluous (up from 70%). Somewhat unsurprisingly, there were 168 catch handlers that caught the type `Exception`, comprising 31% of all catch handlers. Manual inspection of a subset of these handlers revealed that most did little more than log a message to the console. We suspect that more specific error handlers would make the program more robust, but it is obviously difficult to write such handlers if 80% of throws conjuncts do not provide accurate information.

LimeWire. As mentioned previously, LimeWire had a very good exception design and contained much fewer imprecise exceptions declarations than Columba. There were only two application-defined unchecked exceptions, one of which represented an assertion failure. We converted the other one to a checked exception using our refactoring, though this required few new annotations (the exception was always being caught by the immediate callers).

There were no obvious cases where a package in LimeWire was throwing inappropriate exception types. Many packages threw `IOException`, which made this task more difficult. LimeWire had 12 application-defined subtypes of `IOException`, so it is likely that some packages should have been throwing one of these subtypes. It is also unclear whether it was appropriate for all of these exceptions to be subtypes of `IOException`, since some of them were generated by actual I/O operations, while others were generated in response to a higher-level network event. However, as in Columba, considering actual `throws` declarations made it difficult to understand the exception behavior of each package; according to the declarations, packages threw a total of 50 exceptions, as compared to 80 as computed by inference. The discrepancy was caused by imprecise exception declarations.

5.3 Performance

Analysis times for our prototype whole-program and per-package analysis are displayed in Table 6, as measured on a 3.2 GHz Pentium 4. We measured only the time for the exception analysis itself, after the AST was loaded and dependencies were computed. The average time to analyze each package in the per-package analysis is reasonable, and optimizations would likely improve this quite a bit.

We were limited by the high memory consumption of the program analysis infrastructure that we used, and were unable to analyze programs that were much larger than LimeWire, though this is not a problem inherent in our analysis. We are currently working to address this issue.

Table 6. Analysis times for whole-program analysis, total per-package analysis (entire program), and average time to analyze one package in the per-package analysis.

Program	LOC	Whole program	Total per-package	Average per-package
LimeWire	61k	180 s	126 s	2210 ms
Columba	40k	68 s	51 s	411 ms
Tapestry	20k	29 s	17 s	304 ms
JFtp	13k	10.4 s	10 s	1250 ms
Lucene	10k	9.6 s	9.5 s	950 ms
Metrics	7k	10 s	9 s	563 ms

6 Summary and Future Work

We have described a novel exception specification methodology which combines inferred and programmer-specified annotations and illustrated how this scheme can be

used to display a variety of analysis results to the user. The basic design of this scheme is applicable to any language with dynamically bound exception handlers, and where exception propagation is performed automatically. We have also provided quantitative evidence that our design addresses many of the problems of a commonly used exception specification system—that of Java.

Future work includes support for Java 1.5 generics, which will require changes to both our underlying algorithm and the implementation. We would also like to support richer exception specifications. The expressiveness of exception specifications would be greatly increased if they would be applicable to *modules* rather than Java packages. A module would consist of a set of Java classes, and there would be support for hierarchical modules with support for controlling visibility. We are currently designing such a module system, and our initial experiments indicate that performing exception inference over a module rather than a package can reduce the number of programmer-supplied annotations by up to an order of magnitude.

We are also considering support for a lightweight notation for specifying the high-level properties of handlers in the module exception specification (such properties need not be exposed to clients, as they may express implementation details). Some examples of policies we would like to support include: "handlers for exceptions of type E should be non-empty"; "thrown exceptions of type E should be logged"; "exceptions of type E should always be wrapped as type F before they escape the interface of this scope."

7 Acknowledgments

We would like to thank David Garlan and George Fairbanks for their comments on an earlier version of this paper, and Bill Scherlis for his suggestions and discussions.

This work was supported in part by NASA cooperative agreements NCC-2-1298 and NNA05CS30A, NSF grant CCR-0204047, and the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems."

References

[1] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system. In *Cassis International Workshop, Ed. Marieke Huisman*, 2004.

[2] Joshua Bloch. *Effective Java*. Addison-Wesley Professional, 2001.

[3] Christophe Dony. A fully object-oriented exception handling system: rationale and Smalltalk implementation. In *Advances in exception handling techniques*, pages 18–38, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[4] Bruce Eckel. *Thinking in Java, 3rd edition*. Prentice-Hall PTR, December 2002.

[5] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of PLDI 2002*, 2002.

[6] Chen Fu, Ana Milanova, Barbara Ryder, and David Wonnacott. Robustness testing of Java server applications. In *IEEE Transactions on Software Engineering*, pages 292–312, April 2005.

[7] Alessandro F. Garcia, Cecília M. F. Rubira, Alexander B. Romanovsky, and Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[9] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, pages 241–255. ACM Press, 2001.

[10] Anson Horton. Why doesn't C# have exception specifications? Available at http://msdn.microsoft.com/vcsharp/ team/language/ask/exceptionspecs.

[11] Jangwoo Jo, Byeong-Mo Chang, Kwangkeun Yi, and Kwang-Moo Choe. An uncaught exception analysis for Java. *Journal of Systems and Software*, 2004.

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[13] Joseph R. Kiniry. Exceptions in Java and Eiffel: Two extremes in exception design and application. In *Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems*, 2003.

[14] Jorgen Lindskov Knudsen. Fault tolerance and exception handling in BETA. In *Advances in exception handling techniques*, pages 1–17, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[15] K. Rustan M. Leino and Wolfram Schulte. Exception safety for C#. In *SEFM*, pages 218–227. IEEE Computer Society, 2004.

[16] Martin Lippert and Cristina Videira Lopes. A study on exception detecton and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 418–427. ACM Press, 2000.

[17] John 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 Press, 1990.

[18] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In *ECOOP*, pages 85–103, 1997.

[19] Darell Reimer and Harini Srinivasan. Analyzing exception usage in large Java applications. In *Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems*, 2003.

[20] Martin P. Robillard, May 2005. Personal communication.

[21] Martin P. Robillard and Gail C. Murphy. Designing robust Java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '00)*, pages 2–10. ACM Press, 2000.

[22] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, 2003.

[23] Saurabh Sinha, Alessandro Orso, and Mary Jean Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 336–345. IEEE Computer Society, 2004.

[24] Bill Venners. *Interface Design: Best Practices in Object-Oriented API Design in Java*. Available at http://www.artima.com/interfacedesign, 2001.

[25] Bill Venners. Failure and exceptions: a conversation with James Gosling, Part II. Available at http://www.artima.com/intv/solid.html, September 2003.

[26] Bill Venners and Bruce Eckel. The trouble with checked exceptions: A conversation with Anders Hejlsberg, Part II.
Available at http://www.artima.com/intv/handcuffs.html, August 2003.