Practical Exception Specifications

Donna Malayeri[1] and Jonathan Aldrich[1]

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 propose a novel and lightweight exception specification system that provides integrated support for specifying, understanding, and evolving exception policies. Our tool, implemented as an Eclipse plugin for Java, combines user annotations, program analysis, refactorings, and GUI views that display analysis results. Using our tool, we analyzed six programs and observed a 50 to 93% reduction in programmer-supplied 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. Moreover, this 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.

We believe that exception specifications are a useful tool for reasoning about exceptions (see, for example, [15, 13, 2, 7]). They serve to document and enforce a contract between abstraction boundaries, such as method calls. This facilitates modular software development, and can also provide information about exception flow in a scalable manner.

However, current exception specification systems are either impractical or flawed in one or more ways. In these solutions, specifications are either too low-level, too heavyweight, or do not provide adequate support for describing a high-level exception policy. We believe that a good exception specification system should be lightweight while sufficiently expressive, and should facilitate creating, understanding, and evolving specifications.

We have implemented a tool, ExnJava, which provides practical and integrated support for exception policies. It combines user annotations, program analysis, refactorings, and GUI views that display analysis results. ExnJava raises the level of abstraction of exception specifications, making them more expressive, more lightweight, and easier to modify.

Note that we focus on the problem of *specifying* various properties of exception behavior, rather than a proposal for a new exception handling mechanism. The problem of specifying such properties exists independently of the exception mechanism, though of course some details of our solution would not apply directly to languages whose exception handling mechanism is significantly different than that of Java. The goal of our work is to shed light on properties that are important for an exception specification scheme, regardless of the handler mechanism.

In the next section, we describe the essential properties of a practical exception specification system; in Sect. 3 we describe how previous solutions have failed to meet one or more of these criteria. We describe the details of our system and how it meets these criteria in Sect. 4.

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.

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.

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.

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.

As a motivating example, suppose some module provides support for managing user preferences. Suppose also that its implementation should hide how the preferences are actually stored (e.g., files, a database, etc). Accordingly, its exception policy is that exceptions pertaining to these implementation details (e.g., `FileNotFoundException`, `SQLException`) should *not* be thrown by any of its interface methods. Rather, perhaps such exceptions would be wrapped by a higher-level exception type, such as `PreferenceStoreException`. If a low-level exception is erroneously thrown by an interface method, clients cannot write a meaningful exception handler without knowing the modules's implementation details.

Specifications could also include high-level properties of handlers while remaining lightweight.[1] Note that these policies need not be exposed to clients, as they may express implementation details. Such policies (for a particular scope) could include the following: 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.[2]

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.

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.

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.

[1] Supporting these high-level properties is the subject of our future work.
[2] A common practice recommended by Bloch [2], among others.

3 Related Work

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

Specifications. 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.[3]

Though we believe it is useful to separate exceptions into the categories of checked and unchecked (see, for example, [12, 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 an 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 [5, 23, 9]. 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 [16, 11].

But even if programmers use checked exceptions as the language designers intended, exception declarations quickly become imprecise[4] as code evolves; statements that throw or handle exceptions will invariably be modified. In our study of several open-source Java programs (the subject programs are listed in Table 1), we found that between 16% and 81% of exception types within `throws` declarations were imprecise (with an average of 46%). Aside from illustrating the difficulty of maintaining `throws` declarations, this casts 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, 22, 2, 21].

Eclipse[5] 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

[3] 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.

[4] For a method m, the declaration `throws` E is imprecise if m does not throw the exception E, though it may throw subtypes of E. The case where m throws neither E, nor its subtypes is an interesting one, as this is perhaps less likely to be an intentional design decision. However, due to space limitations, here we do not distinguish between these kinds of imprecision (though our tool does provide this capability).

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

iteratively update declarations until a fixpoint was reached. Eclipse also includes an optional warning that will list methods whose `throws` declaration is imprecise, but this too applies to a single method at a time.

There are several proposals for specifying method post-conditions on exceptional exit [6, 13, 1], but these are even more heavyweight than Java `throws` declarations. These solutions do, however, provide powerful verification capabilities. Whether such benefits will outweigh the significant cost of annotating an entire program, however, is unclear.

Other Work. There are several languages and language extensions which ease the task of implementing a policy, but provide no way to *specify* the policy. These include languages with first-class exception handlers [4] and languages that allow applying handlers to some set of methods or classes [10, 14]. 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 [18] 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 [19, 20, 3], but they all perform a whole-program analysis, which does not scale [17]. For the task of understanding exception flow, Sinha et al. 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 Sect. 2. Our design raises the level of abstraction of exception specifications, while remaining lightweight.

In developing our system, we found that Java classes and packages are not always appropriate units of abstraction; accordingly, we have designed a simple module system to be overlaid on standard Java code. A *module* consists of a set of Java classes or interfaces; each Java class or interface belongs to exactly one module (a default module is included for convenience). We add the following accessibility rule: methods may be accessed outside their module if and only if they are visible by standard Java accessibility rules *and* they are marked as module-public. Such methods are *interface methods*, as they effectively comprise the interface of the module; all other methods are *internal methods*.

We are in the process of implementing support for modules. Consequently, our current simplifying assumption is to equate modules with packages; each package is a separate module. Thus, methods with public or protected visibility are interface methods; private and package-private[6] methods are internal meth-

[6] Also known as “default” or “friendly” access.

ods. To emphasize our design goals, in the discussion below we use the term "module" rather than "package."

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 module *interface methods* require a `throws` declaration. ExnJava also includes module-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, and our initial empirical results.

4.1 Specifying Exception Policies

In ExnJava, programmers specify exception policy at the module 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 module has two kinds of exception policy: one applies to each individual interface method, the other to the module 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.[7] Consequently, this design raises the level of abstraction of `throws` declarations.

To determine the checked exceptions thrown by internal methods, ExnJava performs an intra-module dataflow analysis. Within a module, the implementation of our analysis is similar to the whole-program analyses of previous systems [19, 20]. However, our analysis is scalable, as it depends on the size of each module 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 Sect. 4.2.

There are several advantages to this scheme. First, annotations are more lightweight. As we describe in Sect. 4.4, 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.

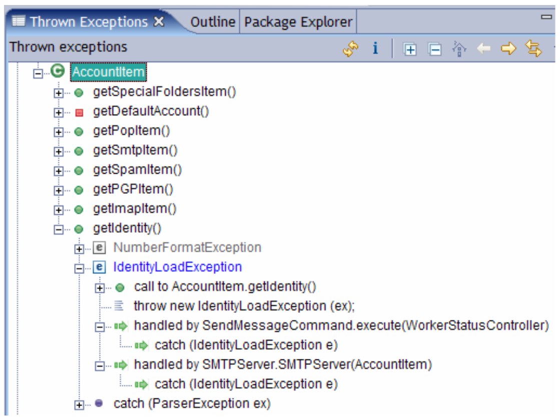
[7] It may sometimes be useful to include `throws` declarations on internal methods; this is supported.

Module Policies. It is also useful to specify and enforce a policy that applies to all of the interface methods of a module. In ExnJava, for each module, programmers specify the set of exception types that may be thrown by its interface methods. This ensures that exceptions that are logically internal to a module are not leaked to its clients.

Module 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 module. Recall the example of Sect. 2 where the storage details of the user preferences module were to be hidden from clients. For such a module, its specification would include perhaps `PreferenceStoreException` but would *not* include `FileNotFoundException`.

4.2 Understanding Exception Policies

The Thrown Exceptions view (Fig. 1) 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. Based on some anecdotal evidence, as well as our own programming experience, we believe that the general difficulty of programming with exceptions is partly due to lack of information on a program's exceptional control flow.



```
AccountItem
    getSpecialFoldersItem()
    getDefaultAccount()
    getPopItem()
    getSmtpItem()
    getSpamItem()
    getPGPItem()
    getImapItem()
    getIdentity()
        NumberFormatException
        IdentityLoadException
            call to AccountItem.getIdentity()
            throw new IdentityLoadException (ex);
            handled by SendMessageCommand.execute(WorkerStatusController)
                catch (IdentityLoadException e)
            handled by SMTPServer.SMTPServer(AccountItem)
                catch (IdentityLoadException e)
        catch (ParserException ex)
```

Fig. 1. The Thrown Exceptions view in method level mode.

The Thrown Exceptions view displays information computed by either a whole-project analysis or a per-module exception analysis; the former will provide more information, but the latter is more scalable. The view has two modes: method level and module level. The method level view, inspired by the work of Sinha et al [20], 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.)

The module level view displays, for each module, 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 module view can be useful for creating a module's exception policy and can also be used to discover 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.

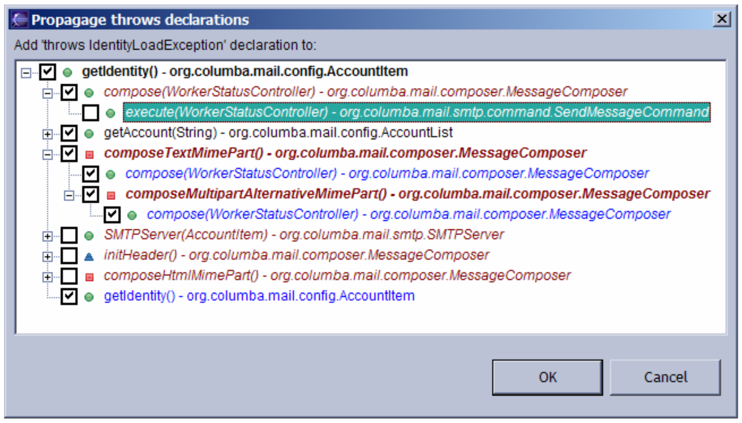
4.3 Evolving Exception Policies

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 module. 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 Fig. 2). 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

[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 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).



Add 'throws IdentityLoadException' declaration to:

- ☑ **getIdentity() - org.columba.mail.config.AccountItem**
  - ☑ *compose(WorkerStatusController) - org.columba.mail.composer.MessageComposer*
    - ☐ *execute(WorkerStatusController) - org.columba.mail.smtp.command.SendMessageCommand*
  - ☑ getAccount(String) - org.columba.mail.config.AccountList
  - ☑ ***composeTextMimePart() - org.columba.mail.composer.MessageComposer***
    - ☑ *compose(WorkerStatusController) - org.columba.mail.composer.MessageComposer*
    - ☑ **composeMultipartAlternativeMimePart() - org.columba.mail.composer.MessageComposer**
      - ☑ *compose(WorkerStatusController) - org.columba.mail.composer.MessageComposer*
  - ☐ *SMTPServer(AccountItem) - org.columba.mail.smtp.SMTPServer*
  - ☐ *initHeader() - org.columba.mail.composer.MessageComposer*
  - ☐ composeHtmlMimePart() - org.columba.mail.composer.MessageComposer
  - ☑ getIdentity() - org.columba.mail.config.AccountItem

Fig. 2. The dialog for propagating throws declarations. Methods that are typeset in italics are those for which the module specification does not allow throwing this particular exception type.

The refactoring also incorporates the module exception specification; if updating the `throws` declaration of a particular method would violate the module 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. 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. (We realize, of course, that 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 module 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 [19].

To solve this problem, ExnJava includes an Fix Imprecise Declarations refactoring, which can be run on a module or set of modules. 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 module specification level, to inform the programmer of specifications that may no longer be valid. Such a tool would display each module whose specification lists one or more exceptions that are not actually thrown in the implementation.

4.4 Empirical Results

We analyzed six open-source programs to determine the feasibility of interface method and module specifications; results are in Table 1. Even with the rough estimation of each package as its own module, we found that when the declarations of internal methods are inferred there are considerable annotation savings.

We first refactored the visibility modifiers of methods, making them as restrictive as possible. This was to simulate a good module design that hides as many implementation details as possible. (Of course, it is likely that some of the methods that were not currently used outside their package were intended to be accessible for future use, but we hope that this inference provides a reasonable estimate.) We found that after refactoring, the `throws` inference results in a 50% to 93% reduction in declarations. Also, since many imprecise declarations appeared on internal methods, inference reduces imprecision by 42% to 78%. (That is, internal methods contained 42% to 78% of all imprecise declarations.) Before refactoring, 12% to 56% of declarations were inferable.

Using modules would very likely increase these annotation savings, since we expect that most modules will consist of several packages. In such a case, there likely would be more internal methods (and therefore more inferred declarations).

We also computed the average number of exceptions thrown by the interface methods of packages in our subject programs. In most applications, packages generally throw few distinct exception types—fewer than 2 exceptions per package, on average. This strongly suggests that module exception specifications have a low annotation overhead.

Table 1. The subject programs studied, number of lines of code, percentage of declarations that could be inferred (i.e., appeared on internal methods) before and after refactoring to reduce visibility of methods, percent reduction in imprecise exceptions after refactoring (i.e., percentage of imprecise exceptions that appear on internal methods), and average number of exceptions types thrown by the interface methods of packages.

	LOC	Inferable decls		Imprecise reduction	Exceptions thrown per package
		Before refactoring	After refactoring		
LimeWire	61k	45%	72%	53%	2.1
Columba	40k	44%	50%	42%	1.3
Tapestry	20k	12%	75%	68%	0.45
JFtp	13k	44%	93%	59%	0.88
Lucene	10k	56%	81%	75%	1.9
Metrics	7k	23%	72%	78%	0.5

5 Acknowledgments

We would like to thank David Garlan and James Hendricks 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] Byeong-Mo Chang, Jang-Wu Jo, Kwangkeun Yi, and Kwang-Moo Choe. Interprocedural exception analysis for Java. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC ’01)*, pages 620–625. ACM Press, 2001.

[4] 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.

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

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.