Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application

Joseph R. Kiniry

School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 8, Ireland
joseph.kiniry@ucd.ie

Abstract. Exceptions are frequently a controversial language feature with both language designers and programmers. Exceptions are controversial because they complicate language semantics—and thus program design, testing, and verification—and some programmers find them annoying or difficult to use properly. By examining two programming languages that have very different, even opposing, exception mechanisms, a set of exception principles is introduced that summarize the key semantic and social issues surrounding exceptions.

1 Introduction

The designers of future programming languages must decide whether to include exceptions in their new languages. If they decide exceptions are warranted, they must then consider what exceptions represent: (1) a structure for control flow, (2) a structure for handling abnormal, unpredictable situations, or (3) something in-between. Additionally, the syntax and meaning of exceptions must be considered.

The syntax of exception mechanisms is also important. Syntax impacts how program code looks and is comprehended, it influences the design and realization of algorithms, and it affects the manner in which programmers handle unusual cases and unexpected situations, and thus indirectly impacts software reliability. And, while the syntax of exception mechanisms is the aspect most programmers see, tool developers and language theoreticians must wrestle with exception semantics. In general, a small, elegant semantics is desired by all parties.

One way to consider how to design a feature like exceptions in future languages is to analyze their design in today's languages. While the analysis of exceptions in niche, historical, or research languages like Ada, Mesa, PL/I, and CLU reveals an "exceptional" gem or two[1], examining the contrary designer and user viewpoints that exist in two modern languages is more relevant to working programmers.

The programming languages Java and Eiffel offer two opposing viewpoints in the design and use of exceptions. A detailed analysis of exceptions in these two languages, as expressed through a series of *principles*: their language design, formal specification and validation, core library use, and non-technical "social" pressures, helps future language

[1] These three languages are frequently cited as the premier languages with innovative exception mechanisms.

C. Dony et al. (Eds.):Exception Handling, LNCS 4119, pp. 288–300, 2006.

creators design their own exception mechanisms. This analysis also informs developers, particularly those that only know one or two programming languages, of the sometimes radically different viewpoints that exist about exceptions.

While the discussions in this paper focus on two object-oriented languages, it is expected that the principles herein are not restricted to object-oriented languages. N.B. It is assumed that the reader is knowledgeable of the basic precepts of exceptions (e.g., exception nesting, handlers, etc.).

1.1 Terminology

The terminology used in this paper is the terminology used in the Java programming community.

A program is composed of a set of *threads* executing a sequence of *method calls* on a set of *objects* and *classes*. Objects are instances of classes, and classes are made up of *data fields* (or just *fields* for short) and *methods* in the Java vernacular. In the Eiffel vernacular, methods and fields are known generically as *routines*. The object calling a method is known as a *client*; the called object is known as the *supplier*.

A method's body specifies a *program behavior*. The execution behavior of a method is either *normal*, *abnormal*, or *divergent*. A method that terminates without an exception exhibits normal behavior; a method that terminates by a thrown exception exhibits abnormal behavior; and a method that never terminates exhibits divergent behavior.

In program code, a flow control structure is any program structure which diverts the execution of a program from the next statement. Conditional statements (e.g., an if-then-else block, a case statement, etc.) are the flow control structures typically associated with normal behavior. Exception-based program structures like try/catch blocks in Java are also flow control structures, and are typically related to abnormal behavior.

We characterize a system that behaves in an unexpected fashion as either *partial* or *total failures*. What "unexpected" means is contextual. Finding a file owned by the program disappear or not readable is an example of a typical unexpected *partial failure*, because the program can attempt to change the permissions of, or recreate, the file. An example *total failure* is discovering that a vital resource, say a physical device, is unavailable.

2 Language Design

Language design only partially influences the use of exceptions, and consequently, the manner in which one handles partial and total failures during system execution. The other major influence is examples of use, typically in core libraries and code examples in technical books, magazine articles, and online discussion forums, and in an organization's code standards. This latter "social" effect is clearly seen in the use of exceptions in Java and Eiffel and is discussed in Section 5.

Exceptions in Java are designed to be used as flow control structures. This is also true of exceptions in most other modern programming languages including Ada, C++, Modula-3, ML and OCaml, Python, and Ruby.

Eiffel exceptions, on the other hand, are designed to represent and handle abnormal, unpredictable, erroneous situations. The languages C#, Common Lisp, and Modula-2 use this interpretation for exceptions as well[2].

2.1 Exception Language Design in Java

Exceptions in Java are used to model many types of events; they are not just used for erroneous behavior. Exceptions sometimes indicate situations that should not be witnessed during a "typical" execution of a program. Most Java exceptions are meant to be dealt with at runtime—just because an exception is thrown does *not* mean that the program must exit.

Java exceptions are represented by classes that inherit from the abstract class `java.lang.Throwable`. They are generically called *throwables* because raising an exception is Java is accomplished with the `throw` keyword.

Each Java throwable is one of two (disjoint) kinds: *unchecked exceptions* or *checked exceptions*. The former inherit from either the class `java.lang.RuntimeException` or the class `java.lang.Error`, the latter inherit from `java.lang.Exception` [3, Section 11.2].

Some of the most commonly witnessed runtime exceptions are `NullPointerException` and `ClassCastException`. Two example errors are `AssertionError` and `ThreadDeath`. Examples of normal exceptions are `ClassNotFoundException`, `CloneNotSupportedException`, and `IOException`.

Checked Exceptions in Java. If a method can raise a checked exception, the checked exception type *must* be specified as part of the signature of a method. The `throws` keyword is used to designate such. A client of a method whose signature includes an exception E (i.e., the method signature includes "`throws E`") must either handle E with a `catch` expression or the client also must declare that it can throw E.

Thus, if a new checked exception is introduced or an existing exception is eliminated, all method signatures or method bodies involving these exceptions must change. Likewise, all methods that call these changed methods must be revised. This exception signature coupling leads to a fragile and annoying trickle-down effect that is frequently seen when programming large Java systems.

Checked exceptions are mainly used to characterize partial and total failures during method invocations, like a file not being readable or a buffer overflowing. Not all erroneous situations in Java are represented by exceptions though. Many methods return special values which indicate failure encoded as constant field of related classes. This lack of design uniformity leads to the introduction of the *Uniformity Principle*.

Principle 1 (Uniformity Principle). *Exceptions must have a uniform, consistent informal semantics for the developer.*

[2] Note that Modula-2 did not originally have exceptions; their addition caused a great deal of controversy through the early 1990s (i.e., compare [1] to [2]). See `http://cs.ru.ac.za/homes/cspt/sc22wg13.htm` for a historical discussion of such.

The use of exceptions in Java is contrary to the **Uniformity Principle**. While some attempt has obviously been made to use exceptions only for truly unexpected incidences, there are numerous examples of inconsistent use (e.g., `ArrayStoreException`, `FileNotFoundException`, and `NotSerializableException`). These inconsistencies are sometimes due to more serious language flaws (e.g., in Java's type system), but, for the most part, are simply inconsistencies in API design.

Unchecked Exceptions in Java. Unchecked exceptions are either runtime exceptions or errors.

Runtime exceptions can rarely (but potentially) be corrected at runtime, and thus are not errors. For example, `ArrayIndexOutOfBoundsException`, `ClassCastException`, and `NullPointerException` are common runtime exceptions of this kind.

Errors indicate serious problems that most applications cannot handle. Most errors indicate abnormal situations with either the operating environment or the program structure. Examples of errors are `AssertionError` (thrown when an assertion fails), `No-SuchMethodError` (thrown when a method that does not exist is called), `Stack-OverflowError`, and `OutOfMemoryError`. In general, if one of these errors is raised, the program exits.

Java 5. In Java 5 several new constructs were added to the Java language. Two of these constructs are parameterized classes and enumerations.

Programmers can use either of these language mechanisms to express richer exception semantics. For example, a enumeration can denote a precise set of distinct exceptions legal in a given context.

There is no evidence that the Java 1.5 team has considered either of these alternatives. There are no parameterized exception types in Java 1.5, no new exception types of note, and no use of enumerations and exceptions.

2.2 Exceptions in Eiffel

The fundamental exception principle in Eiffel is that *a routine*[3] *must either succeed or fail*: either it fulfills its contract[4] or it does not. It the latter case an exception is *always* raised [4,5]. Thus, exceptions are, by design, meant to be used in Eiffel exclusively to signal when a contract is broken.

Eiffel exceptions are not specified as part of the type signature of a routine, nor are they mentioned in routine contracts. In fact, there is no way to determine if a routine can raise an exception other than by an inspection of the routine's source code, and all the source code on which it depends.

Eiffel exceptions are represented by *INTEGER*[5] and *STRING* values—there are no exception classes[6]. Exceptions that are part of the language definition are represented

[3] An Eiffel *routine* is a method of a class.

[4] An Eiffel *routine contract* is a precondition/postcondition pair.

[5] Eiffel class names are always capitalized.

[6] The new ECMA standard for Eiffel introduces exception classes perhaps, in part, due to articles such as this one [6].

by *INTEGER* values, developer-defined exceptions by *STRING* values[7]. This limited and non-uniform representation of Eiffel exceptions inspires a new principle.

Principle 2 (Representation Principle). *Exceptions should have a uniform representation, and that representation should be amenable to refinement.*

Eiffel exceptions have two representations which causes design impedance when dealing with them. Additionally, because they are basic values and not objects, they have no inherent semantics beyond that which is expressed in a helper routine which necessarily cannot be foolproof because of the representation overloading in effect (e.g., one cannot differentiate two integers of the same value).

Contract Failure. Contracts are violated in several ways, all of which are considered *faults*, but only some faults are under programmer control.

Operating environment problems, such as running out of memory, are one situation in which exceptions are raised. In these cases a contract fails, but not necessarily because the client (the caller) or the supplier (the callee) did something wrong. Certainly, intentionally allocating too much memory, or otherwise using an extraordinary amount of system resources, is the fault of the program, but such situations are more malicious than typical.

Software infrastructure failures cause exceptions also. E.g., some operating system signals raise an exception. Failures in non-Eiffel libraries that are used by an Eiffel application cause these kinds of exceptions as well. For example, Eiffel programs that link with Microsoft Windows COM components witness an exception when a COM routine fails and Eiffel programs that use UNIX libraries see an exception raised when an external library fails and did not set the `errno` system variable. Additionally, a floating point exception is raised on some architectures when a division by zero is attempted.

But most exceptions used in Eiffel are not due to external factors, but instead are *assertion violations* or *developer exceptions*, both of which are used to indicate program errors.

If assertion checking is enabled during compilation, assertion violations cause an exception to be raised. These exceptions are classified according to the type of assertion that has been violated[8].

For example, the `check` instruction, which is equivalent to C or Java's `assert` statement, cause a `Check_instruction` exception to be raised. A `Loop_variant` exception is another assertion violation. This exception is raised when a loop variant does not monotonically decrease during loop execution.

Violating a contract, either by failing to fulfill a class invariant, a method precondition or postcondition, or a loop invariant, is the final kind of exception. Contract violations fall into two categories: those that are the fault of the client of a class, and those that are the fault of the supplier of a class. The classification of an exception is determined by the context of the failure during program execution.

[7] Earlier versions of the Eiffel language standard permitted developer-defined integer exception values, but this seems to no longer be the case. It is unclear when and why this change was made.

[8] JML uses the same assertion failure exception design [7].

If a contract is broken at the time a method is called, regardless of whether the caller is another object or the current object (in the case of a callback, or the use of the `retry` keyword, see below), then the blame lies with the caller.

Exactly one kind of exception, called `Void_call_target`, can be the blame of either the caller or the callee. If a method is invoked on an object reference with value `Void`, a `Void_call_target` is raised. If the caller set the value to `Void`, or did not check the reference prior to making the invocation attempt, then the blame lies with the caller. In situations where the reference was obtained via a routine call, either via a formal parameter or a return value, and the value is `Void`, the blame lies with the callee, as the specification of the routine is not strong enough to eliminate the possibility of the `Void` value[9].

The uniform design for signaling assertion failure with exceptions in Eiffel is contrary to that which exists in Java. Several languages exist for the formal specification of contract for Java code, and the Java Modeling Language (JML) is the de facto standard for such [8]. Unfortunately, because assertion violation semantics is so primitive in Java, there is no uniformity in exception specification across different assertion tools and specification languages. This muddle inspires the next principle.

Principle 3 (Language Specification Principle). *If exceptions are used to represent assertion failure, their design and semantics should be incorporated into the core language specification.*

Java programmers and JML specification authors have suffered because the creators of Java ignored this key point in language design, particularly because an assert statement was not introduced to Java until seven years into the evolution of the language.

2.3 Comparing Eiffel Exceptions to Unchecked Exceptions in Java

Given the above analysis, Eiffel exceptions and Java unchecked exceptions are exclusively focused on unexpected, erroneous behavior that an application should not try to handle. Thus, one might expect every Eiffel exception to map to a single Java unchecked exception. This is not the case.

Some of Eiffel's built-in exception types are equivalent to standard *checked* exceptions in Java. For example, Eiffel's `Io_exception`, `Runtime_io_exception`, and `Retrieve_exception` are similar to `IOException` and some of its children.

A number of *unchecked* exceptions are equivalent to standard Eiffel exceptions. For example, `Void_call_target` is equivalent to `NullPointerException`, and `Floating_point_exception` is equivalent to `ArithmeticException`.

Finally, some errors are equivalent to the remaining Eiffel exceptions: `AssertionError` is equivalent to the set of specification-centric Eiffel exceptions (`Check_instruction`, `Class_invariant`, `Loop_invariant`, `Loop_variant`, `Postcondition`, and `Precondition`), and `No_more_memory` subsumes `OutOfMemoryError` and `StackOverflowError`.

Missing Mappings. Several exceptions that exist in each language have no peer in the other language.

[9] The new ECMA standard for Eiffel introduces non-void types to deal with these issues.

`Rescue_exception` has no mapping, as Java does not perform any special handling of exceptions thrown in a `finally` clause. An extended discussion on this point is below in Section 2.3.

An equivalent for `Signal_exception` is not part of the core Java language as Java's definition focuses on multi-platform development and not all platforms have signals[10]. The original Eiffel language specification states that such system-specific exceptions should be contained in system-specific classes, but no compilers implement this suggestion [10].

An error like `Void_assigned_to_expanded` is not possible in Java as Java has no expanded types and the type system prohibits assignment of `null` to built-in types like `int` and `boolean`[11].

The Eiffel literature claims that Eiffel has no casting (cf., [11, page 194]), thus there is no equivalent to Java's `ClassCastException`. This claim is a bit disingenuous because Eiffel's assignment attempt operator '`?=`' is simply a built-in conditional downcast in the form of an operator[12].

`Routine_failure` is a generic exception in Eiffel that indicates a routine has failed for some reason. The reason is sometimes recorded (as a `STRING`) in the `meaning` associated with the exception, but this is not mandatory. This is also true of Java exceptions, each of which has an optional message associated with it obtainable via the `Throwable.getMessage()` method. Unfortunately, there is absolutely no uniformity to the use of these representations in either language, which motivates the introduction of the following principle.

Principle 4 (Meaning Principle). *When defining a new type of exception, the availability of a human **and** unambiguous machine comprehensible representations (e.g., a string value and a predicate) is mandatory.*

For the most part, exception design in both Java and Eiffel fail to fulfill the *Meaning Principle*.

None of the various Java exceptions involving out-of-bounds array access and strings exist in Eiffel because the contracts of accessor routines for these types prohibit such. Cloning-related exceptions do not exist because all objects can be cloned in Eiffel. In fact, in general numerous exception types simply do not exist in Eiffel because routine contracts prohibit the situations that must be manually dealt with in Java catch blocks. This evidence inspires a principle on contracts.

Principle 5 (Contract Principle). *Integrated contracts significantly decrease the number and complexity of exceptions.*

The *Contract Principle* is critical with regards to the development of complex modern applications and components, particularly with respect to component and system testing, verification, and evolution.

[10] One can catch and handle signals in Java, but internal classes like `sun.misc.Signal` and `sun.misc.SignalHandler`, or a package like that seen in [9], are needed.

[11] And, in fact, this error cannot occur with the introduction of autoboxing in Java 5.

[12] This is not the only "pragmatic circumvention" in Eiffel. Other examples include the dual semantics of routine calls (with and without an explicit "`Current`") and the semantics of the `equal` and `clone` routines of ANY.

This principle is supported by the quantitative analysis of Section 4.

The Eiffel language standard does not have several features of Java: reflection, introspection, concurrency, and sandboxing. While these features contribute significantly to the complexity of Java's exception class hierarchy it is expected that the continued application of the **Contract Principle** will see Eiffel's exception hierarchy change little with the adoption of such features[13].

Controlling Exceptions in Eiffel. Exceptions are primarily controlled in Eiffel using *rescue clauses* and the `retry` instruction. Exceptions are also indirectly controlled by the choice made in *compilation mode* during application development.

A routine may end with a rescue clause. A *rescue clause* of a routine is a block of code that will execute if any exception is raised during the execution of the routine.

The rescue clause does not discriminate between different types of exceptions. In this respect, it is functionally equivalent to the surrounding every Java method body with a `try/catch` block where the catch expression's type `java.lang.Throwable`. The rescue clause is *not* equivalent to Java's `finally` construct. The code enclosed in a finally block is *always* executed when a method completes, whether it completes normally or abnormally, while a rescue clause only executes when a routine fails.

The `retry` instruction in Eiffel causes a routine to restart its execution. This instruction may only be used within a rescue clause. If a rescue clause does not contain a retry instruction, then the routine fails and the current exception is raised in the immediate caller. The details of `finally` and `rescue` are discussed in the sequel.

Exceptions are manipulated in Eiffel using the EXCEPTIONS class. Using this class one can find out information about the latest raised exception (much like `errno` in C), handle certain kinds of exceptions in a special way, raise special developer-defined exceptions, and prescribe that certain exceptions must be ignored at run-time. The EXCEPTIONS class is part of the Eiffel Kernel Library, thus is available in all Eiffel compilers.

3 Exceptional Specifications and Validation

The key difference between the use of exceptions in specifications in the two languages in that exceptions are *part* of a method contract in Java and are *not* part of a routine contract in Eiffel. Thus, a fundamental notion of "Design by Contract," that of exceptions exclusively indicating contract failure, has a different interpretation in Java.

3.1 Contracts with Exceptions in Java

As mentioned previously, the Java Modeling Language is used to write formal specifications of Java components [12]. The discussion in this section is based upon experience in participating in the development and application of a denotational semantics for Java and JML and the design, specification, and verification of several Java systems [13,14,15].

[13] This claim is supported by the recent addition of reflection and concurrency in commercial and experimental Eiffel compilers.

The semantics of Java, and thus JML, are significantly complicated by the possibility of abrupt method termination. Verification proofs must cover three cases in Java: normal termination, abrupt termination, and divergent behavior, sometimes tripling proof size.

The default specification for a failure is simply *true*, which means that the routine guarantees nothing in particular when a failure takes place. Usually something stronger is specified and, in fact, exceptional cases are often the first part of a formal specification written.

This information, what is true of the system when an exception is thrown, helps the caller deal with the exceptional cases rather than just halting. In fact, the specification of a postcondition for abrupt termination is *mandatory* for reasoning about systems during abrupt termination. Without such assertions, class invariants can become significantly more complex because, for example, specification variables are needed to represent failure states for all of the routines of a class.

3.2 Specifications of Eiffel Exceptions

In Eiffel, the semantics of *exceptional-correct* routines is rolled into the definition of *class correctness* [11, Chapter 15 and Section 9.16].

The definition [11, Section 15.10] of *exception-correct* is:

> A routine r of a class C is exception-correct if and only if, for every branch b of its rescue block:
>
> 1. If b ends with a `Retry`: `{true} b {INV_C and pre_r}`
> 2. If b does not end in a `Retry`: `{true} b {INV_C}`
>
> where `INV_C` is the class invariant of C; `pre_r` is the precondition of routine r.

This semantics is problematic in practice because it means that an Eiffel routine must always have a rescue block that "puts everything right" (fulfills the normal precondition). But how does the routine know what went wrong and how to change the current state to fulfill the postcondition[14]?

Some programmers weaken the postcondition of retryable routines because one can barely state anything is true if the routine can either fail or succeed. Another solution is to write complex postconditions using a set of disjuncts with error-flag guarded expressions[15]. For example,

```
   method_call_failed implies (F || G || H)
|| not method_call_failed implies (I || J || K)
```

This kind of specification is evident in the very few places where exceptions are handled in Eiffel code, and we speculate this is true because of the inherent complexity in such specifications.

Specifications in JML that use keywords like `exsures` and `exceptional_behavior` that are simply shorthand for these more complex expressions. E.g., an

[14] The new ECMA-367 standard no longer forces the restoration of the precondition.

[15] It should be noted that the new ECMA Eiffel standard changes this definition and no longer forces a restoration of the precondition.

`exsures` assertion specifies exactly what is true when a particular exception is thrown. Eiffel will benefit from such assertion expressions as well.

This semantics significantly complicates contracts and weakens their application. Neither case is surprising: either (in case 1) a rescue clause must fulfill the invariant and the precondition of the retried routine or, (in case 2) a retry does not happen so the routine has to leave the object in a legitimate state by fulfilling its invariant. What is surprising is that *nothing* is know about when or why the exception happened in the first place, since both preconditions are as weak as possible, and nothing *new* can be specified about the state of the objects when a failure takes place, since the postcondition is exactly the invariant.

JML, on the other hand, provides the ability to state a stronger postcondition in these exceptional cases, and this information is essential to verifying programs with exceptions. These observations provide evidence for a principle about exceptional postconditions.

Principle 6 (Abrupt Termination Principle). *The specification of object state when an assertion is raised, either via an exceptional postcondition or an exception predicate, is mandatory if programs are to be formally verified.*

The Java Modeling Language fulfills this principle admirably, while Eiffel fails in this regard.

4 Qualitative and Quantitative Comparisons

In the end, it is unclear how important exceptions are in the Eiffel world. This might be due to exception's perceived second-class nature in the Eiffel universe of "correct" software, as evidenced by their rare use (see below).

If exceptions in Eiffel are equivalent to unchecked exceptions in Java, and if library programmers for the two languages equally careful and capable of handling unexpected circumstances, then an analysis of exception usage in the two core code bases should yield comparable results.

The data in Table 1 is the result of such an analysis. The specific large Eiffel systems chosen for this analysis are four of the largest, highest-quality Open Source Eiffel systems available today.

In the case of the Gobo and SmartEiffel systems, all code, library and applications, was analyzed for this data. In Java 1.4.1, all source under the top-level package `java` was examined. The number of declared exceptions is determined by counting and classifying all calls to *EXCEPTIONS.raise* and *EXCEPTIONS.die*, in the case of Eiffel, and counting all descendants of `java.lang.Throwable`, in the case of Java. The number of raised exceptions is determined by a count of the number of calls to *EXCEPTIONS.raise* and *EXCEPTIONS.die*, in the case of Eiffel, and the number of `throw` statements, in the case of Java. The data on stack traces is determined by counting and analyzing all calls to routines `exception_name`, `tag_name`, `meaning`, and `developer_exception_name` of class `EXCEPTIONS`. All numbers are approximate and measured using the `wc` command.

Table 1. Use of Exceptions in Eiffel and Java

Library	Gobo 3.1	ePosix 1.0.0	ISE Eiffel 5.3	SmartEiffel 1.0	JDK 1.4.1
Number of direct/indirect mentions of EXCEPTIONS, or unchecked exceptions	18	3	17	0	525/15,000
Number of unchecked/checked exceptions declared	3/-	6/-	5/-	0/-	50/150
Number of raised unchecked/ checked exceptions	66/-	87/-	13/-	0/-	3,000/2,650
Number of `rescue` or `finally` clauses	6	10	29	0	50
Number of `retry` commands	81	3	15	0	N/A
Number of times a stack trace is (a) checked or manipulated, or (b) printed or ignored	0/0	0/0	0/0	0/0	8/79
Total lines of code and documentation	250,000	25,000	372,000	115,000	421,000

To summarize the result of this analysis: in Java an unchecked exception is thrown for approximately every 140 lines of code, where in Eiffel one is used for every (approximately) 4,600 lines of code. This represents a difference of over thirty times in frequency. The above statistics clearly show that either or both (a) exceptions in Eiffel, either through technical issues or social pressure, have a second-class (or perhaps even ignored) status, or (b) the (built-in) existence of reasonable specification technologies inherently leads to fewer assertions being thrown. Given the preponderance of quality Eiffel software available, the latter point holds much more weight. This fact is especially highlighted in the complete lack of exception use and support in the GNU SmartEiffel system.

This data should be carefully considered by the language standardization committees for Eiffel and Java. It also provides evidence for potential avenues for language refinement, particularly with regards to the specification of abnormal behavior.

5 Exception Equivalency

Both languages have exceptions mechanisms that can be treated as equivalent. For example, a hierarchy is representable by integer or string values in a number of ways (e.g., De Bruijn indices or simple lexical encodings), so one can define an artificial type hierarchy for Eiffel exceptions if necessary.

Likewise, the minimal exception interface of Eiffel, embodied in the *EXCEPTIONS* class, is possible in Java. In fact, some Java developers advocate avoiding checked exceptions entirely, instead inheriting exclusively from `RuntimeException` [16]. Programming in this fashion pushes Java toward a more dynamic style, akin to programming in Objective-C.

We can find no evidence of the converse, that of Eiffel programmers using exceptions as flow control mechanisms. While Eiffel exceptions can be used in such a way, programmers simply do not use them in this way.

As any Java programmer knows, the volume of `try/catch` code in a typical Java application is sometimes larger than the comparable code necessary for explicit formal parameter and return value checking in other languages that do not have checked exceptions.

In fact, the general consensus among in-the-trenches Java programmers is that dealing with checked exceptions is nearly as unpleasant a task as writing documentation. Thus, many programmers report that they "resent" checked exceptions. This leads to an abundance of checked-but-ignored exceptions, as evidenced by the next to the last line of the table of the previous section.

Additionally, the presence of checked exceptions percolates through the system, as discussed in Section 2.1. As discussed by the designers of C# [17]:

> Examination of small programs leads to the conclusion that requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result – decreased productivity and little or no increase in code quality.

This attitude guides the design of error handling in the .NET framework as well [18, see Section "Error Raising and Handling Guidelines"].

These issues lead to our last, and perhaps crucial principle.

Principle 7 (Checked Exception Principle). *Checked exceptions generally increase system fragility (because of signature refactoring), increase code size (due to explicit, localized, mandatory handling), and cause programmer angst (as evidenced by the number of empty or spiteful* `catch` *blocks in public Java code), so their inclusion in a language should be considered carefully.*

In the end, so long as an exception mechanism has a simple semantics, is consistently used, and provides a tool which programmers can understand, depend upon, and not resent, then they can be included in future languages.

Acknowledgments. This work was supported by the Netherlands Organization for Scientific Research (NWO). Thanks to the anonymous reviewers and Alexander Kogtenkov for their comments.

References

1. Sutcliffe, R.J., ed.: Modula-2 (Base Language). Number 10514-1:1996 in ISO/IEC Modula-2 Standardization. ISO/IEC (1999)
2. Wirth, N.: Programming in Modula-2. Springer–Verlag (1982)
3. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. third edn. Addison-Wesley Publishing Company (2005)
4. Meyer, B.: Object-Oriented Software Construction. second edn. Prentice-Hall, Inc. (1988)
5. Meyer, B.: Disciplined exceptions. Technical Report TR-EI-13/EX, Interactive Software Engineering (1988)

6. Bezault, E., Howard, M., Kogtenkov, A., Meyer, B., Stapf, E.: Eiffel analysis, design and programming language. Technical Report ECMA-367, ECMA International (2005)
7. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Kiniry, J.: JML Reference Manual. Department of Computer Science, Iowa State University, 226 Atanasoff Hall. Draft revision 1.94 edn. (2004)
8. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design. In: Behavioral Specifications of Business and Systems. Kluwer Academic Publishing (1999) 175–188 Available via http://www.cs.iastate.edu/∼leavens/JML.html.
9. Hester, K.: What is JavaSignals? (1999) See `http://www.geeksville.com/kevinh/projects/javasignals/`.
10. Meyer, B.: Eiffel the Language. third edn. Prentice-Hall, Inc. (2002)
11. Meyer, B.: Eiffel: The Language. Prentice-Hall, Inc. (1992)
12. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (2005)
13. Jacobs, B., Poll, E.: A logic for the Java Modeling Language JML. In: Proceedings of Fundamental Approaches to Software Engineering (FASE). Volume 2029 of Lecture Notes in Computer Science., Springer–Verlag (2001) 284–299
14. Kiniry, J.R., Cok, D.R.: ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In: Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004. Volume 3362 of Lecture Notes in Computer Science., Springer–Verlag (2005)
15. Jacobs, B., Poll, E.: Java program verification at Nijmegen: Developments and perspective. In: International Symposium on Software Security (ISSS'2003). Volume 3233 of Lecture Notes in Computer Science., Springer–Verlag (2004) 134–153
16. Eckel, B.: Does Java need checked exceptions? (2003) See `http://www.mindview.net/Etc/Discussions/CheckedExceptions`, particularly the ensuing feedback on this issue.
17. Posted by Eric Gunnerson; original author unknown.: Why doesn't C# require exception specifications? (2000)
18. Microsoft Corporation: .NET framework general reference (2003) Documentation version 1.1.0.