First, Do No Harm: A Curricular Approach to Exceptions Introducing Refactoring to Promote Reliability

Duane Buck
Otterbein University
Westerville, Ohio
614-802-1775
dbuck@otterbein.edu

ABSTRACT
This paper advocates the adoption of deferred error coding within computer science curricula. It argues that it is both a sound development strategy and aligns well pedagogically. By deferring specific error handling, the student better appreciates its subtleties and its importance as an independent topic, and will tend to create more reliable applications. This paper includes other topics which may increase community awareness of the issues and enhance curricula: taxonomies of exceptions and exception handlers and the relationships between them, subtle pitfalls of exception handling, and factors influencing the selection of error reporting patterns. Much of the discussion is language independent, but specific attention is given to the Java checked exception controversy, which inspired the curriculum approach.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Error Handling and Recovery

Keywords
Java, Checked Exception, Refactoring.

1. INTRODUCTION
In the joint ACM/IEEE computer science curriculum [1], the 'Software Fundamentals/Cross-Layer Communications' area includes a conceptual discussion of 'reliability' and its relationship with several other areas. Inexplicably, it does not reference the 'Software Engineering' knowledge area that relies heavily upon it and for which reliability is a critical topic. This implies the need for increased community awareness of reliability in relation to software engineering. The need for awareness extends to industry, which will be evident below. This manuscript provides an overview of issues surrounding reliability. The broader goal here is to present a new curriculum approach which focuses better on reliability and highlights its importance. To facilitate further improvement of curricula, it also presents background material and important related topics which, in appropriate courses, may be a useful supplement to textbooks.

Section 1 presents the motivations. Section 2 discusses exceptions and introduces the Java checked exception controversy which inspired the new curriculum approach. Section 3 introduces taxonomies of exceptions and exception handlers, and relates the two. Section 4 discusses error reporting patterns in Java. It includes the description of a hybrid approach found in some newer Java library classes. Section 5 examines exception type inconsistencies in several Java library classes. Section 6 presents the "deferred error coding" curriculum approach where the application's direct-path is initially debugged before its error handling is refactored. Its implementation using Java is also addressed there, together with common exception handling pitfalls to be avoided. Section 7 provides a different point of view as it examines the design of error reporting within an API. Section 8 presents the conclusion where it is argued that the checked exception controversy may be resolved. It then presents a minimal curricular change that could by itself improve checked exception handling, based on "First, do no harm." Section 9 provides the references cited.

SPLASH '13, Oct 26-31 2013, Indianapolis, IN, USA

2. EXCEPTIONS
Reliability goes hand-in-hand with properly handing exceptions. The term *exception* is used broadly here to refer to a failed request. Although a failed request may sometimes be reported by the hardware or virtual machine (e.g., a null pointer error), an application programming interface (API) often needs to report exceptions to its client. A "direct-path" of an application is defined here as a thread of execution that results in providing one of the application's functions, in the absence of exceptions.

Exceptions fall into one of two broad categories, expected and unexpected. Expected exceptions represent circumstances that are unavoidable and should be anticipated, while unexpected exceptions typically indicate program bugs, which should not occur in a production application. Expected exceptions require specific alternative processing, which depends on their foreseeability and a cost/benefit analysis. An unexpected exception also requires alternative processing, but the response is limited: an appropriate shutdown that supports debugging.

In older languages, such as C, exceptions were indicated by API request return codes, and a great deal of the source code dealt with checking the codes, which was error prone and hampered reliability. There was a major advance when newer languages, such as C++, introduced modern exception handling. [4] With this facility, it was possible to code a direct-path without explicitly coding an action after each call to the API. For unexpected errors, no coding was required, period. This was possible because unexpected exceptions were addressed by a context appropriate default handler (sometimes called the uncaught exception handler).[1] Therefore, the programmer explicitly handled only expected exceptions, providing specific alternative processing.

This situation was ideal except for the problem that it was up to API designers to document expected exceptions, and up to the

[1] The use of a default handler is described by Longshaw and Woods [7, p. 40] as the "Big Outer Try Block Pattern."

programmer to read the documentation and implement the specific alternative processing required in the context of the application. Otherwise, unless the need to handle an exception was uncovered during debugging, a system could go into production missing a handler; if the exception occurred, it would be treated as a bug. The designers of Java attempted to further improve reliability by supporting a second type of exception for problems arising "outside of the immediate control of the program" (equivalent to expected exceptions, as defined here). When using checked exceptions, the lack of an explicit exception handler is statically detected by the Java compiler. Paradoxically, because the designers of Java did not trust that programmers would heed a warning regarding a missing handler, they created an environment that tended to result in applications having more serious flaws than those addressed by checked exceptions.[2] [1, 2, 10]

There has been a long standing controversy regarding Java checked exceptions [3, 6, and 11]. Two main issues are reported: (1) "catch or specify" is not always possible[3] and (2) for some applications "catch or specify" is unnecessary or even undesirable. These two issues are touched on only briefly below. However, this manuscript is the first to report a third issue, which is the controversy's unrecognized raison d'être: "catch or specify" creates a siren song inviting the practice of coding specific exception handlers simultaneously with coding a direct-path. In this case, the error handling code: (1) potentially masks bugs during debugging, (2) adds to the volume of code initially being debugged, (3) often requires a design scope larger than the method being coded, (4) competes for attention with coding the (usually more interesting) direct-path, (5) requires maintenance because the code-base may not yet have stabilized, (6) is developed without the insights gained from first-hand experience with the exceptions, and (7) requires a specialized skill set which the implementer of the direct-path may not possess. This manuscript is the first to identify these issues.[4] They point strongly to the practice being dysfunctional. The issue may have not been recognized because the natural tendency before Java was to develop the direct-path before the edge-cases. [11]

Because of the issues encountered when developing specific error handlers prior to initial debugging, as well as for pedagogical concerns, this manuscript proposes a two phase development strategy: *deferred error coding*.[5] This emerges organically in most languages, but requires an explicit coding technique when using Java. But, regardless of the language, the pedagogy is important.

3. EXCEPTION AND HANDLER TYPES

A taxonomy may be formed by noting that an exception is triggered by one of four situations: (1) an undetected program bug, (2) a system error (e.g., out-of-memory), (3) an environment fault (e.g., a network outage), or (4) an issue related to the application domain (e.g., incorrect input). The order here is from least likely to most likely to be "expected" in the sense that the potential error is understood well enough that specific alternative processing could be developed. The first two situations are more generally classified as unexpected, and the latter two expected.

It is beyond the scope of the present paper to discuss exception handling comprehensively. The paper by Chen et al. [2] has an excellent discussion of refactoring handlers, upon which this paper draws. The broad issues are discussed here. There are three major categories in the taxonomy of exception handlers:

(1) A *message/terminate handler* is provided by the uncaught exception handler, which assumes the exception is due to a bug. It may be customized for the execution environment as discussed earlier. Occasionally, a message/terminate handler is specifically coded for an unrecoverable situation not resulting from a program bug. In this case, the message would be tailored to communicate with the end-user.

(2) A *message/rollback handler* is used in the case where a single request could not be completed, but the system may be capable of completing other requests. The request is often an action requested from a user-interface. The handler informs the user, and then needs to transfer control back to the "event loop" so the user can make additional requests. The difficulty with this type of handler is that it must ensure that the partially completed execution of a request does not invalidate further execution of the application. Borrowing from database terminology, the transaction must be rolled back.

(3) A *retry/fallback handler* first tries to complete the function of the method invocation, which may involve attempting the same action again and/or executing an alternate implementation. Usually after some number of failed tries, it falls back to a message/rollback or message/terminate handler, depending on the context and the severity of the issue.

Table 1 shows for each exception type the types of handlers that may be employed. The shaded cells represent unusual handlers for that particular type of exception. Exceptions due to environmental faults are the only ones that do not have a usual handler type. For those, the choice of handler should be based on a cost and benefit analysis within the particular context. The retry/fallback provides the best user experience, but also has the highest development cost. If retry/fallback is not feasible or not cost justified, then a message/rollback handler should be considered if providing a subset of "commands" is possible, as this provides the next best user experience. Otherwise, the least desirable, but also least expensive, message/terminate handler is indicated.

Specific handler types are indicated for bugs, system errors, and application errors. Bugs are corrected rather than specifically handled, and those that remain are usually handled by a message/abort handler, usually the uncaught exception handler.[6] System errors typically indicate that the system is unstable, and therefore they are also usually handled with a message/terminate handler, which may again be the uncaught exception handler. This is a reasonable choice because system errors are sometimes

---

[2] That checked exceptions protect the system from crashing due to an uncaught exception is a myth that has emerged to justify them. It has a basis in reality; early software environments did crash for that reason. Footnote 18 debunks this myth as regards Java.

[3] E.g., when using a framework without source code.

[4] The author delivered an invited presentation at JavaOne 2012, San Francisco, discussing this and other topics explored below.

[5] Haase's "Unhandled Exception" pattern [5, p. 105] is similar to the deferred error coding recommended here.

[6] When debugging, a default handler usually supplies debugging information and terminates the activity in progress. Because this behavior is inappropriate for an end user, a custom default handler is usually installed when running in production.

triggered by bugs. A message/rollback handler is usually appropriate for application domain exceptions.

Table 1: Possible handling of the four types of exceptions

	Type of Handler ►	message/ terminate	message/ rollback	retry/ fallback
Unexpected Exception Types	**program bug**	Report debugging information. Inform user.	Report debugging information. Inform user.	Report debugging information. Use alternate implementation.
	system error	Report debugging information, and inform IT and the user.	Continued execution is risky. Occasionally, system errors (e.g., out-of-memory), may be understood well enough to attempt one of these handlers.	
Expected Exception Types	**environment fault**	Report to IT. Inform user.	Report to IT. Inform user of the issue; continue w/o repair.	Attempt retries and/or alternate implementation; then terminate or rollback.
	application domain error	Inform user.	Inform user and allow them to try again.	N/A, same input will get same result.

4. JAVA EXPECTED ERROR REPORTING

4.1 Return Codes vs. Checked Exceptions

After he extensively reviewed the literature of error handling and recovery, Tellefsen [10, p. 50] concluded that "return codes are useful for returning error information, simply because they are easier to use, and they would probably be used even if they were disallowed by project guidelines." It is therefore not surprising that return codes remain in use today in the Java libraries to report expected errors, even though return codes are problematic.

A return code often takes the form of a single return value being multiplexed so it is either a result or a status indicator.[7] The Java library `Map` class' `get()` method is an example of this. It returns an object reference in the normal case, otherwise it returns `null`. An application programmer may find return codes beneficial because the `if/else` construct is familiar and easy to code. However, they need to be cognizant of the danger of not checking a return code and losing the source of an error. This is more likely to happen with a multiplexed return code, because it is tempting to code the function inside of another expression, assuming no error will occur. Fortunately, a null reference or negative index value frequently results in a quick exception.

An interesting juxtaposition occurs here. The Java library makes use of return codes for some expected errors even though failure to check return codes is a major issue affecting debugging and reliability. They do this seemingly because of the difficulty programmers have coding exception handlers. Meanwhile, Java forces coding explicit handling of checked exceptions before compilation and debugging can commence. For expediency, programmers tend to take shortcuts to a compilation: they ignore return codes and insert the minimal code required to ignore checked exceptions. The result is that execution continues for *both* error reporting mechanisms, complicating debugging, and potentially leading to unreliable applications.

4.2 Hybrid Error Reporting

For expected errors, when external events cannot asynchronously alter the validity of a request, an API may supply a method to check the validity of a request before it is made. This provides the best qualities of return codes and unchecked exceptions without their drawbacks. Some of the newer Java APIs (e.g., `Scanner`) use validity requests. An API documents an expected error, not with a checked exception, but with a method to precheck validity. The programmer uses the familiar `if/else` construct to code the alternative action, as with return codes. If the programmer fails to do the validity precheck, an exception signaling the expected situation will (hopefully) occur during testing. This gives a meaningful stack-trace pointing to the problem. For some APIs, the programmer may choose to catch the exception instead of using the query method, when that makes error coding easier.

5. JAVA EXCEPTION CLASSIFICATIONS

Although unanticipated issues arose with the forced early implementation of checked exception handlers, one might expect that the anticipated benefit, knowing that the correct exceptions have specific handlers, is enjoyed. However, the classifications of exceptions have proven idiosyncratic and guidelines have shifted.

In the Java library, some exceptions, which for all practical purposes are expected, are confusingly classified as unchecked, and vice-versa. For example, consider the familiar library function `int Integer.parseInt (String s)` which converts the input `s` to an `int`. The origin of the input is almost certainly from outside of the program (probably an end-user), so it would be expected to occasionally be incorrect. However, `parseInt()` throws an unchecked exception when given invalid input.[8] A misclassification of this kind, where an expected exception is classified as unchecked, defeats the purpose for which checked exceptions were designed.

The opposite problem is also troublesome because checked exceptions may unnecessarily complicate the use of an interface. The guideline published in the quasi-official Java Tutorial regarding the use of checked exceptions has shifted from virtually mandating their use when not reporting bugs, [3] to using them when a client can "reasonably be expected to recover," which better focuses on their purpose. [6] This was apparently a workaround addressing the "unnecessary complication" argument.

What is worse than the preceding issues is that expected and unexpected situations are sometimes merged into a single checked exception class when reported. This is the case with `java.io.IOException` which is thrown by the `read()` methods of several IO classes. If an expected error occurs (like losing the connection to a network resource, which is outside the control of the programmer), a checked exception is properly thrown. However, `IOException` is also thrown if the IO object is closed before the `read()` method is invoked. The latter is inexplicable because clearly such a situation should be reported

[7] Alternatively, an API may provide a separate method to access the return code (e.g., `Scanner`; see Footnote 17 below).

[8] If the `Integer` class API provided a function to check a `String` for valid integer syntax, it would be correct to consider passing an invalid string to `parseInt()` unexpected. This would follow the hybrid error reporting pattern given above.

by a subtype of `RuntimeException` indicating a program bug (the canonical choice would be `IllegalStateException` in the `java.lang` package). The unfortunate result is that parsing the exception's message is the only alternative available to determine the cause of that exception.[9]

Some Java library APIs (e.g., `java.sql`) and third party APIs classify all of their declared exceptions as checked, even when they are due to programming bugs, although this goes against the published guideline. This misclassification may be due to esthetic concerns of the API's designers, who want to have all of their exceptions extend a single API defined supertype, which makes it impossible for some of the exceptions to be checked and some unchecked.[10] A similar esthetic may also be behind the unusual choice of using `IOException` to report a bug.

6. CURRICULUM IMPLICATIONS

Error coding is an important but complex topic that deserves attention in the curriculum. Having the students first learn direct-path implementation without the complexity of error coding is important to avoid cognitive overload, in addition to its other advantages. This raises the question of when refactoring should be studied. Examining textbooks reveals little or no early coverage when using other languages. It may be advantageous to move the topic to a more advanced course, perhaps as late as software engineering, or divide the topics across multiple courses.

When introducing deferred error coding, the motivations presented in Section 2 may be a useful topic. When teaching refactoring, the taxonomies and their relationships examined in Table 1 would be a resource, as would be the dysfunctional examples and their alternatives in Subsection 6.2. During later courses, the API factors to be discussed in Section 7 may also be a topic of interest. The remainder of this section examines the two phases of the deferred error coding pedagogy advocated here.

6.1 Direct-path With Fail-fast Handlers

As previously discussed, when using languages other than Java, deferred error coding is organic, and curricula have implicitly embraced it (at least for exceptions). When using Java in a curriculum, a technique for implementing deferred error coding must be taught. Although the technique recommended here adds more verbiage than desirable, it imposes the least cognitive load among the available choices. The student is instructed to insert the following "boilerplate" template around any method invocation that throws a checked exception:

```
try {
   aMethodThrowingACheckedException();
} catch (ACheckedException ex)
   {throw new RuntimeException(ex);}
```

This handler will trigger the uncaught exception handler which should be a context appropriate message/terminate handler.[11] A program with this boilerplate handler has a valid form of error handling, although it might provide a suboptimal user experience.

The deferral of specific error coding until refactoring also applies to the other API error reporting patterns: conventional exceptions, hybrid reporting using validity query methods, and return codes. For return codes, the fail-fast behavior required for deferred error coding is provided by inserting code to throw a runtime exception in the event of an error.[12] Deferred error coding for these patterns has the same advantages as it has for checked exceptions: it allows the direct-path to be coded expediently, yields good debugging information, and provides a foundation for the refactoring that follows.[13]

There are several advantages to using the deferred specific error coding approach. Then starting out, the student is taught an expedient approach that is not dysfunctional. Early on the student will see in which contexts things can go wrong and trigger exceptions. The student also comes to understand that a program without specific error handling, for at least domain level exceptions (user errors), is not "finished."[14] Later the student will learn how to refactor applications to create robust solutions.

6.2 Refactoring Error Handling

By refactoring error handling, students are *not* forced to divide their attention between the direct-path, which is their central concern initially, and error handling. Advanced assignments will require students to refactor the error coding. This may involve both studying the API and testing to determine which exceptions are recoverable in the context of the application. For those, the student will code a specific alternative action. Each location where boilerplate code throws a `RuntimeException` needs to be studied to determine if it should to be refactored into a more specific handler. As discussed earlier, testing is also required because some methods throw misclassified unchecked exceptions that should be caught and vice-versa. Once the type of handler is selected, the student may need to include multiple lines of code in a `try/catch` block, possibly need to use a `throws` clause to send an exception to the invoking method, and might have the need for `try` blocks with a `finally` clauses to release resources[15] when a non-terminating handler is invoked.

It is beyond the scope of this manuscript to cover implementation of error handling in detail. Instead, some practices whose dysfunctionality may not be apparent will be enumerated. The examples are drawn from textbooks, the standard Java library, and an Eclipse code template, and probably arose under the influence of checked exceptions. Each example is immediately followed by an alternative that addresses the problem cited. Unfortunately the authors, who will remain anonymous, appear to be oblivious to the issues raised.

---

[9] If a checked exception is irrecoverable, it should be wrapped in a `RuntimeException` and rethrown. See Subsection *5.2.3*.

[10] Anecdotally, some believe the myth described in Footnote 2.

[11] In Java, a custom default handler must extend the base class `UncaughtExceptionHandler` and be set as the default handler by invoking the `Thread` class' method `setUncaughtExceptionHandler()`.

[12] A returned error status becomes a `RuntimeException`:

```
<result> = aRequestWithReturnCode();
if (<result-indicates-failure>)
    throw new RuntimeException(<message>);
```

[13] When *all* errors result in a runtime exception (known as failing-fast), a code base has reached the first goal (G1) of the error handling refactoring methodology presented by Chen, et al. [2]

[14] A mathematics colleague points out that for his purposes it is finished because an exception handler might mask an error and cause erroneous output, which is far more troublesome than rerunning. This is probably true of many one-off programs.

[15] Every method with a `throws` clause is a candidate for this.

6.2.1 Ignored Checked Exception

The following ignores a checked exception that is unexpected:

```
try {
   Thread.getCurrentThread().sleep(10);
} catch (InterruptedException e) {}
```

In this case, ignoring the exception seems innocuous. Unless the application uses cooperating threads, and invokes the current thread's `interrupt()` method, theoretically the exception will not occur; however, if it does, it would be indicative of a bug. However, this code ignores it and proceeds. Although ignoring an *expected* exception might be a reasonable fix-up, one should never ignore an exception that is *unexpected*; Müller and Simmons [8, Subsections 2.1 and 4.2] provide an extended discussion.

Alternative: To handle an unexpected checked exception, simply retain the boilerplate code discussed above. To document refactoring is completed, `RuntimeException` may be replaced by an application subtype, e.g., `UnexpectedException`:

```
try {
   Thread.getCurrentThread().sleep(10);
} catch (InterruptedException e)
   {throw new UnexpectedException (e);}
```

If the exception occurs, it will be properly reported as a bug.

6.2.2 Noting and Ignoring a Checked Exception

Examine the following code that is creating an `InputStream`:

```
try {
   is=new FileInputStream(f);
} catch (FileNotFoundException e)
   {e.printStackTrace();}
```

Here the code reports the error, but does so in a way that is insensitive to the user currently executing the program (perhaps it is the end-user). Printing to the console is also problematic because applications are typically deployed without console windows. The bigger problem here is that the program keeps running and its results are unpredictable. Unfortunately, this is similar to Eclipse's default code template that assists coding the invocation of methods that throw checked exceptions.

Alternative: The standard boilerplate code is far superior to the above. However, because this error is expected (the existence of the file is not under the control of the programmer), a message/rollback handler is probably indicated when refactoring.

6.2.3 Fix-up of an Exception Triggered by a Bug

This code reading `InputStream is` has subtle issue:

```
try {
   b=is.read();
} catch (IOException e) {b=0;}
```

The error handler performs a simple fix-up to an environment fault, using a default value and continuing. But, as has been discussed, the issue is that unexpected and expected situations have been merged into one exception class by the API designer. It might signal an IO error, which this handler addresses, but another possible cause of the exception is that the `InputStream` has been closed, a program bug. In that case, the application will continue execution and make it difficult to locate the error. This mixing is common in some class libraries (e.g., `java.sql`).

Alternative: The boilerplate handler should be augmented to examine the exception, verify it was not caused by a program bug, and if so, execute specific handling. Otherwise, it should throw a `RuntimeException` to report the bug.

6.2.4 Supertype Exception in `throws` Clause

The elided method given below throws to the invoking method a checked exception which is a supertype of other exceptions:

```
void fun1() throws IOException
    {…
    b=is.read();
     …}
```

This code is dysfunctional because the `throws` clause throws all subtypes of `IOException`. As a result, a programmer opening a file within the same method will not be required by the compiler to handle the `FileNotFoundException`, for instance.

Alternative: A supertype checked exception should be caught locally. If it cannot be dealt with locally, it should be wrapped in an application defined exception and thrown:

```
void fun1() throws IOExceptionWrapper
  {…
   try {
      b=is.read();
    } catch (IOException ex)
      {throw new
          IOExceptionWrapper (ex);}
   …}
```

The above `throws` clause names the class wrapping the supertype exception, so the invoking method will have to "Catch or Specify" `IOExceptionWrapper` in this case. Now, when a programmer opens a file within the method, they will be required to catch or throw `FileNotFoundException`.

Fortunately, supertype exceptions are infrequently thrown in library APIs. The bigger lesson is that generally, only specific subtypes should be caught, or appear in a throws clause; if a supertype is specified, subtypes will not require specific handling.

7. API DESIGN IMPLICATIONS

The underlying problem to be solved in an API is how to give feedback to the application regarding an action the application requests or plans to request. As discussed earlier, some form of return code may always be with us even though using return codes is problematic. The use of return codes is justified for especially common situations, like a failure searching for a substring.

Although enforcing specific error coding at compile time through checked exceptions may appear beneficial, as has been discussed extensively, doing so tends to encourage dysfunctional error coding and should be used with caution. Additional concerns have also been raised. Robillard and Murphy [9, p. 2] discuss how coding using the checked exception mechanism tends to lead to "complex and spaghetti like exception structures" (e.g., the tunneling scenario to be discussed in Section 8). Another concern with checked exceptions is noted by Haase at the end of his summary: "The benefits of checked exceptions can be summarized by saying that their use provides documentation and ensures that exceptions are handled. There is however a downside to this, namely that checked exceptions reduce flexibility." [5, p. 94] For example, when an application being modified needs to invoke a method that throws a checked exception, it is not easily accomplished. Either the call hierarchy must be modified up to the point the exception is handled, or it must be tunneled.

The above concerns about checked exceptions are serious and recognized by the wider software community. The designers of the post Java language C# chose not to include checked

exceptions [11], and according to Chen et al. [2, p. 335] "unchecked exceptions are preferred in several well-known open source projects written in Java, including the Eclipse SWT project and the Spring Framework."

A better alternative to both checked exceptions and return codes may be to provide hybrid error reporting as discussed in Subsection 4.2. Using a separate query method, an application can evaluate a request's validity. If the request is valid, the application can make the request and the proper outcome is guaranteed. If the programmer fails to make the check, and an invalid request is made, a runtime exception will be thrown, reporting the program bug.[16] Using a validity query for each type of request, programmers employ the `if/else` construct, with which they have vast experience. This makes the query style easier to code and read for many application programmers. The designers of the `Scanner` class provided hybrid error reporting with query methods.[17] Because that class is a recent addition to the Java library, its designers may have called upon experience to point to that solution.

8. CONCLUSION

Java checked exceptions, although in theory beneficial for reporting expected exceptions, have created a problem in the curriculum. They distract the student from the central function of their project, and force them to reason about constructs they may not yet understand. The recommendation made here is to have students follow the two phases of "deferred error coding." The first phase implements the direct-path and keeps the code base behaving in a predictable, fail-fast, manner. As the student gains more insight, he or she will then enter the second phase, refactoring the error handling.

The issues raised regarding checked exceptions are almost exclusively the result of the classification of failure to "catch or specify" as an error. If the compiler instead generated a warning, an application could be debugged without explicit error handling, and during refactoring the warnings could be used to locate checked exceptions not handled. Issuing warnings also solves the other major issue: the need to throw checked exceptions across foreign software boundaries. Rather than wrapping a checked exception inside a runtime exception and tunneling it across the boundary, one would *expect* a warning. A warning would also be issued for the method handling the exception while invoking the foreign software. This would verify that the exception is caught. Ultimately, one might add annotations to suppress the warnings, similarly to when generics are not statically verifiable. Also, because checked exceptions will no longer impose an unnecessary burden, the guideline for an exception being checked can return to its being "outside of the immediate control of the program." This is simpler than the work-around guideline, which is that the exception be checked if the client may "reasonably be expected to recover," because the latter requires an API designer to make assumptions. Technically, it is apparent that failure to "catch or specify," could easily be designated a warning,[18] which would resolve the major issues that have arisen in the controversy.

Even if the central recommendation of this manuscript – deferral of specific error coding until later in the curriculum – is not adopted, a significant benefit will follow from discussing with students the dangers involved when handling checked exceptions. The Hippocratic Oath includes "First, do no harm," which is good advice in this context. The student should be instructed to first code using the standard boilerplate template presented in Subsection 6.1 when they encounter a checked exception which they: (1) think will not occur, or (2) are unsure how to handle. This will take little class time and will reduce the level of dysfunctional error handling during debugging, which may help provide the additional insight the student needs handle the error.

[16] The programmer might catch the exception instead of using a query method if it simplified the implementation.

[17] It is interesting that `Scanner` has a little known "return code retrieval" method, `ioException()`, which returns the `IOException` last thrown by the `Scanner`'s underlying `Readable`, or `null` if no such exception exists. It apparently was added to relieve the client from having to deal with the troublesome `IOException`. The programmer should check that the method returns `null` before executing a user action. Otherwise, because an `IOException` is treated as end-of-file, an unintended action may result. Many users may unknowingly use applications that have this bug. `Scanner`'s handling of the checked exception provides a good case study of how checked exceptions, paradoxically, may negatively impact reliability.

[18] To confirm this, note that when generics are used, in some cases the compiler is unable to detect the failure to "catch or specify" a checked exception. Therefore, unchecked exceptions must already have runtime support in Java, even if they are not explicitly caught. Also see Footnote 2 for a myth "explaining" why checked exceptions must be explicitly handled.

9. REFERENCES

[1] ACM/IEEE-CS Joint Task Force on Computing Curricula. Computer Science Curricula 2013, Ironman Draft (v 1.0).

[2] Chen, C., Cheng, Y. C., Hsieh, C., and Wu, I. Exception handling refactorings: Directed by goals and driven by bug fixing. *Journal of Systems and Software* 82:333–345, 2009

[3] Goetz, B. "Java theory and practice: The exceptions debate" http://www.ibm.com/developerworks/java/library/j-jtp05254, May 2004.

[4] Goodenough, J. B. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[5] Haase, A. Java idioms: exception handling. In *Proc. of the EuroPLoP'2002*.

[6] The Java Tutorial. "Unchecked Exceptions — The Controversy." http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html, accessed September 2012.

[7] Longshaw, A. and Woods, E. Patterns for the generation, handling and management of errors. In *Proc. of the EuroPLoP'2004*.

[8] Müller, A. and Simmons, G. Exception Handling: Common Problems and Best Practice with Java 1.4. In *Proc. of NetObject Days '02,* 2002.

[9] Robillard, M. P. and Murphy, C. Designing robust Java programs with exceptions. *ACM SIGSOFT Software Engineering Notes*, 25(6): 2-10, November 2000.

[10] Tellefsen, C. *An Examination of Issues with Exception Handling Mechanisms*. Master's thesis, Norwegian University of Science and Technology, 2007.

[11] Venners, B. with Eckel, B. "The Trouble with Checked Exceptions" (A Conversation with Anders Hejlsberg, Part II) http://www.artima.com/intv/handcuffs.html, 2003.