First Do No Harm: Deferred Error Handling – A Curricular Approach to Exceptions

Duane Buck
Otterbein University
Westerville, Ohio
614-823-1793
dbuck@otterbein.edu

ABSTRACT
This paper advocates the adoption of deferred error handling within computer science curricula. It argues that it is both a sound development strategy and aligns well with pedagogically. By deferring error handling, the student better appreciates its subtleties and its importance as an independent topic. This paper also includes other topics which may enhance curricula: an analysis of error reporting patterns, a taxonomy of error handlers, and factors influencing the selection of error reporting patterns. Much of the discussion is language independent, but specific attention is given to deferred handling of Java checked exceptions.

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

General Terms
Algorithms, Reliability, and Languages.

Keywords
Java, Checked Exception, Refactoring.

1. INTRODUCTION
When executing an application, there is a path through the application's instructions as one service after another is requested of the underlying application programming interface (API) and fulfilled. Ultimately this results in the application providing one of its functions. This is referred to here as a "direct-path." However, some requests may not be satisfied, for various reasons, and require alternative processing, which is not considered part of the direct-path.

When using older languages, such as C, exceptional conditions were reported back to the caller as return codes. Each return code had to be meticulously checked by the programmer even if an error was unexpected (usually indicative of a bug). There was a major advance when newer languages, such as C++, introduced modern exception handling. This made it possible to code the direct-path without getting bogged down explicitly coding alternative actions for error conditions. For unexpected errors (usually program bugs), nothing needs to be coded. This is because unexpected problems lack specific solutions, enabling a default handler to take care of them. When debugging, the default handler usually supplies debugging information and terminates the activity in progress. Because this behavior may be unsuitable when an application is released, a custom default handler is usually installed. For instance, it might apologize to the user, log the error, and restart the system. Therefore, the programmer only has to explicitly catch the exceptions that are expected (usually indicative of invalid input data).

The exception mechanism improves reliability because it is no longer possible for a programmer to forget to a check return code. An ignored return code was very harmful to system reliability because in the event of an error because execution continued as if nothing was wrong. This led to both incorrect results and difficult debugging. In contrast, with modern exception handling, the programmer does not handle unexpected exceptions at all, and if they forget to handle an expected exception, the system behaves well, reporting it through the default handler as a bug, which in fact it is!

Error coding is best deferred until after a program has been initially debugged. This is because error coding (1) potentially masks bugs, (2) adds to the amount of code being debugged simultaneously, (3) often requires a scope larger than the method being coded, (4) requires a specialized skill set, and (5) competes for attention with coding the (usually more interesting) direct-path. Although all of these factors may not be appreciated, deferred error handling naturally occurs, if for no other reason than that it is more fun to get something basically working before worrying about the edge-cases.

This was a blissful state of affairs except for the problem that it was up to API designers to document expected exceptions, and up to the programmer to read the documentation. Otherwise, if an expected exception was not triggered during debugging, no handler would be included. In production, if the exception occurred, the default handler would be triggered reporting it as a bug. The designers of Java were concerned about this issue. They attempted to further improve reliability by supporting a second type of exception for problems arising "outside of the immediate control of the program" (expected exceptions). A checked exception *requires* explicit coding by the programmer. With this change, the lack of an exception handler for an expected exception is statically detected at compile time, rather than during debugging, a welcome improvement.

Unfortunately, the designers of Java did not recognize the importance of deferred error handling during development. They classified the lack of explicit checked exception handling a compilation error, rather than a warning, so that it could not be ignored by the programmer, even temporarily. The side effect of this decision is that nothing can be debugged until after some form of error handling has been coded. By forcing the programmer to code error handling before they can compile and

Conference'12, Month 1–2, 2012, City, State, Country.

start debugging, the programmer may be more likely to code a dysfunctional exception handler, creating a situation much worse than if the exception was not explicitly caught. [1, 2]

To minimize the coding of dysfunctional exception handlers, this paper proposes deferred exception handling, where the development of specific handlers is deferred until after initial debugging. As stated earlier, this naturally occurs in modern languages other than Java, and may be why the community has not fully realized its importance. In Java, explicit coding is required to defer writing application specific error handling. Fortunately, it is not difficult and worth the effort because of the benefits discussed above and also provides curricular advantages.

The primary focus of this paper is on how deferred error coding is useful in the computer science curriculum, independent of its general applicability. However, it also presents several other topics that may suitable for inclusion in the undergraduate curriculum. Section 2 introduces patterns available to API designers for informing applications when requests cannot be satisfied. Section 3 discusses in detail those patterns that deal with expected situations. Section 4 examines exception handling in Java when exceptions are poorly classified, and includes examples from standard Java classes. In Section 5 the "deferred error coding" curriculum approach is presented where the application's direct-path is completed before refactoring its error handling. Its implementation using Java is also addressed. Section 6 discusses refactoring error handling. Section 7 provides a different point of view as it examines the design of an API's error reporting. After the conclusion is presented in Section 8, Section 9 provides CS1 and CS2 curriculum units.

2. EXCEPTIONAL SITUATIONS

2.1 Unexpected Exceptional Situations

An unexpected exceptional situation is triggered by one of two conditions. It may be (1) an as yet undetected program bug, or (2) a system error (e.g., out-of-memory). That is, one does not expect (plan for) a program bug or a failure unrelated to either the application or its input. Limited remediation is possible, usually involving reporting the problem and terminating the activity. There are two ways for an API to signal an unexpected error. These are return codes (also called status codes) and exceptions.

Return codes are problematic in unexpected situations. The programmer is often most interested in getting the central task done and forgets to go back and check the return codes while refining the program. If something goes wrong, the program continues running and the origin of the error is lost. Another difficulty is that in a completed application a large amount of the program involves checking return codes superfluous to its normal functioning. The checks take time to write and debug, and obscure the important steps when reading the program. The modern exception mechanism was invented for these reasons. [3] In modern languages, (unchecked) exceptions should always be used for reporting unexpected situations.

2.2 Expected Exceptional Situations

An expected exceptional situation, while triggered by factors outside the immediate control of the program, nonetheless can be anticipated, like invalid user input. An API designer can choose one of three ways to interact with the application programmer: (1) a return code from the request, (2) exceptions defined as part of the request interface, and (3) validity checking prior to the request. The factors involved in selecting the error reporting mechanism for a particular API will be discussed in Section 7.

3. REPORTING EXPECTED ERRORS

3.1 Return Codes

Using return codes is problematic for expected situations because, as with unexpected errors, if the programmer fails to check them execution continues and the origin of the error is lost. But, even with this problem, modern APIs sometimes use return codes. After he extensively reviewed the literature of error handling and recovery, Tellefsen [7, 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." A return code often takes the form of a single return value being multiplexed so it is either a result or a status indicator.[1] The Java library `Map` class `get()` method is an example of this. It returns an object reference in the normal case, or else it returns `null`. An application programmer may find return codes beneficial, because the `if/else` construct is familiar and easy to code. However, he or she needs 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 another expression, assuming no error will occur.

3.2 Conventional (Unchecked) Exceptions

Once exception handling mechanisms were available, return codes were no longer required. Now, one can simply assume that a requested function will be carried and code the direct-path. The program will work in the direct-path but not be able to recover from any type of error (including invalid user input). If a requested function cannot be carried out, the default handler reports the error (usually including a stack-trace) and terminates the activity.

Through testing (or preferably reading the documentation) expected exceptions can be "caught" by the suspended method on the stack that can resolve the situation. The programmer only has to worry about the expected exceptions and put code into the location capable of handling each situation.[2] If the programmer fails to recognize an expected exception and code the handler, it does not create debugging nightmares. However, programmers must to deal with the more difficult exception handling mechanism, compared to the familiar `if/else` construct.

3.3 Java Checked Exceptions

The addition of language exception handling mechanisms was a welcomed advancement. The designers of Java hoped to further improve reliability by having the API formally classify exceptions as either *unchecked* or *checked*.

Unchecked exceptions are defined to represent application program bugs or system errors from which the client cannot reasonably recover. In terms of this paper, because a client is not expected to make any specific provision for them, unchecked exceptions appear to be *unexpected*. On the other hand, checked exceptions are defined to represent situations that arise outside of

---

[1] Alternatively, an API can provide a separate method to access the return code (e.g., `Scanner`, see Footnote 8 below).

[2] In some cases, `finally` blocks will also be required lower in the call hierarchy (to release resources, for example).

the immediate control of the program, and from which a client can reasonably be expected to recover: Checked exceptions appear to be *expected*. The intent of the language designers was to relieve application programmers of the burden of depending on the documentation and debugging to inform them of expected exceptional situations. When a method invocation might trigger a checked exception, the compiler statically checks that an alternative action has been specified. As discussed in Section 1, requiring handlers before debugging has proven problematic.

3.4 Validity Query Methods

For expected errors, when an API supplies methods to check the validity of requests before they are made, it may provide the best qualities of return codes and unchecked exceptions without their drawbacks. The programmer uses the familiar `if/else` construct to code the alternative action, as with return codes. If the programmer forgets to do the validity check, an exception signaling the expected situation will (hopefully) occur during testing. This gives a meaningful stack-trace pointing to the problem. Also, the programmer can choose to catch the exception instead of using the query method if that makes the coding easier.

4. JAVA EXCEPTION CLASSIFICATIONS

In addition to the problems discussed in Section 1, another problem with classifying exceptions is that the classifications are idiosyncratic because the guidelines require interpretation. Some exceptions, which for all practical purposes are expected, are confusingly classified as unchecked, and vice-versa. For example, the familiar method `int Integer.parseInt (String s)` is a library function that 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.[3] An apparent misclassification of this kind, where an expected exception is classified as unchecked, does not cause a major problem. It results in the bug being found at run-time versus at compile-time, as would be true in a language without checked exceptions.

The opposite problem is more troublesome in the Java libraries. Quasi-system errors, like network outages and database problems, from which there is no reasonable expectation of recovery for most applications, are classified as checked. Also, expected and unexpected situations are sometimes merged into one checked exception class. This is the case with `IOException` which is thrown by the read() method of several IO classes. If an expected error occurs (like losing a connection to a network resource, which is outside the control of the programmer), this exception is properly thrown. However, it is also thrown if a program bug resulted in the object being closed. This is inexplicable because clearly it should be reported by a `RuntimeException` indicating a program bug. (An obvious candidate would be `InvalidStateException`.) Because there are different reasons that the exception might have been thrown, it is unclear how to remedy the situation, and the programmer may continue execution after a program bug is detected, thus causing the same problems as forgetting to check return codes.

Table 1: Handling of the four classifications of exceptions

	Unexpected	Expected
Unchecked	Use the default handler (may be set by application).	Insert a `try/catch` for the exception at the point where a corrective action is possible.
Checked	Use default handler by wrapping the exception inside an unchecked exception and throwing it.	Insert a `try/catch` as above and also list the exception in `throws` clauses of any methods deeper in the stack.

In the experience of the author, third party APIs sometimes classify all of their exceptions as checked, even if they are due to a programming bug, even though this goes against the published guidelines. This may be because their designers, for esthetic concerns, want to have all of their exceptions directly under a common ancestor. But, there is at least one additional reason. When the matter was brought to the attention of the developer of such an API, he said that it was the only way to make sure the exceptions were caught. This developer was a computer science PhD. student with many years of Java experience working in a research team. He continued to hold his view after it was challenged. This experience may be indicative of how poorly the community understands the purpose of checked exceptions, which inhibits their accuracy as a classifier.

Table 1 identifies how to handle exceptions classified under the four combinations of checked versus unchecked and expected versus unexpected. The upper left and lower right quadrants correspond to correctly classified exceptions; this is what the designers of the Java language were seeking to have happen. The upper right and lower left quadrants (in grey) specify the actions a programmer should code for exceptions that are misclassified. Note: "Deferred error coding" as defined below treats all errors as unexpected until refactoring.

5. CURRICULUM IMPLICATIONS

Error coding is an important but complex topic that deserves attention in the curriculum. When an API does not use checked exceptions, there tends to be a natural division of coding into two phases. Expected exceptions are often addressed after the program is basically working. During testing, if a handler is missing for an expected exception, the exception will (hopefully) be triggered and result in a stack-trace and termination of the activity, helping locate the bug.

In attempting to improve the Java language, its designers inadvertently created a troublesome issue in the curriculum. The problem with Java checked exceptions is that they force the student to attempt error coding at inopportune times. For instance, the `try/catch` block must be addressed in early assignments because of the use of the Java libraries. A more important concern is that students are forced to divide their concentration between the direct-path, which is their central concern, and error handling. The result can be that the student learns expedient but dysfunctional error coding. Examples of dysfunctional error coding are: (1) ignoring an exception, (2) noting but otherwise ignoring an exception, (3) fixing-up an exception that is actually indicative of a bug, and (4) using the throws clause for an exception that has subtypes.

The solution is to teach "deferred error coding" which includes two phases: coding and debugging the direct-path in preparation for refactoring the error handling. To prepare for refactoring, when a student encounters a method invocation that may throw a

checked exception, they are taught to insert the invocation into this standard boilerplate template:[4]

```
try {
   theMethod();
} catch (TheCheckedException ex)
   {throw new RuntimeException(ex);}
```

Then, the student can more easily code the direct-path, which may be all that is required in early assignments. If anything goes wrong, the default handler[5] will be invoked and the student will come to understand the circumstances leading to the exception.

Advanced assignments will require students to refactor the error coding. This may involve both studying the API and testing to determine which checked exceptions are actually expected in the context of the application. For those, the student will code an alternative action. The student might have to include multiple lines of code in a `try/catch` block, sometimes need to use a `throws` clause to send the exception to the next level, and perhaps have the need for a `try` with a `finally` clause to release resources. Refactoring of exception handling is explored further in the next section.

There are several advantages to using the deferred error coding approach. The student programmer is taught an expedient coding approach that is not dysfunctional when first starting out. Early on the student will see in which contexts things can go wrong and trigger exceptions. Later the student will learn how to refactor his or her application to create a robust solution. In courses that explore API design other topics discussed here may be addressed.

Deferred error coding and refactoring is also defined for the other API error reporting patterns: conventional exceptions, validity query methods, and return codes. Like deferred coding 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. The CS1 and CS2 curriculum unit outlines in Section 9 briefly present those topics in addition to deferred handling of checked exceptions.

6. REFACTORING ERROR HANDLING

In the previous section, dysfunctional coding was avoided by transforming each checked exception into a `RuntimeException`. That is an alternative to fully addressing error handling while coding the direct-path of an application. When refactoring, one needs to examine each location where a `RuntimeException` is thrown and ask how the code should be altered. One of two things should be done. This depends on whether the situation is expected or unexpected in the context of the application.

First, even for a well classified checked exception, the exception may not be expected in the context of given application. Also, one must be wary of misclassification pitfalls when using APIs. If a method throws a vague, generic checked exception (perhaps `Exception` or a direct descendant like `IOException`) be suspicious that it is really an unexpected exception. Another way to determine if the exception is expected is to ask if there is anything that could be done within the application to resolve the problem. If not, it is probably best treated as unexpected. Finally, if the exception occurs during testing (unless it is due to a program bug), it should be treated as expected.

If the checked exception is expected, a fix-up should be coded during refactoring. Either the body of the catch should be replaced or the `try/catch` block should be eliminated and the exception caught elsewhere, as discussed in the previous section.

If the exception is unexpected, the `try/catch` code should be left in place, but slightly altered to indicate that its refactoring has been completed. To document that the error is unexpected and requires no further refactoring, a different exception should be thrown. For that purpose, the application should declare another exception type that extends `RuntimeException`. A good name for this class is `UnexpectedException`.

The error code refactoring phase requires much more than learning the syntax of the try/catch statement. It is beyond the scope of the present paper to discuss exception refactoring comprehensively. The paper by Chen et al. [1] has an excellent discussion of refactoring exception handling. Only the broad issues are discussed here. There are three major categories in the taxonomy of handlers: (1) "message/terminate," (2) "message/rollback", and (3) "retry/fallback.

A message/terminate handler is usually provided by the default uncaught exception handler, which assumes the exception was due to a bug and tailors the message to the programmer by providing debugging information. (When deployed a custom default handler is installed.) 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.

A message/rollback handler is used in the case where a single request could not be competed, 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 must ensure that partially completed execution of the request does not invalidate further execution of the application. Borrowing from database terminology, the transaction must be rolled back, as if it never occurred.

The 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 message/rollback or message/terminate, depending on the context and the severity of the issue.

The choice of handler should be based on the cost and benefit analysis within a particular context. If feasible, retry/fallback provides the best user experience, but also the highest development cost. If retry/fallback is not feasible or not cost

---

[4] The default code template for checked exceptions in Eclipse is similar, except that the handler notes and then ignores the exception. A simple solution would be to create an Eclipse template based on the boilerplate template shown here.

[5] Although unnecessary during development, when an application is deployed, the developer should write a custom `UncaughtExceptionHandler` with the end-user in mind, in case an unexpected exception occurs. It should be set as the default handler using the `Thread` class method `setUncaughtExceptionHandler()`. The use of a default handler is described by Longshaw and Woods [5, p. 14] as the "Big Outer Try Block Pattern."

justified, then message/rollback should be considered, as it provides the next best user experience. If it is not feasible or not cost justified, then the least desirable, but also least expensive, message/terminate is the only other choice

7. API DESIGN IMPLICATIONS

The underlying problem to be solved in an API is how to give feedback to the application regarding an action it 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 alternative coding at compile time through checked exceptions may appear beneficial, as has been discussed extensively, throwing checked exceptions tends to encourage dysfunctional error coding and should be used with caution. Additional concerns have also been raised. Robillard and Murphy [6, p. 2] discuss how coding using the checked exception mechanism tends to lead to "complex and spaghetti like exception structures." 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." [4, p. 94] He goes on to describe causes of reduced flexibility and discusses design patterns that address those problems in large systems. His "Unhandled Exception" pattern is the most important in the context of the present paper.

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 [8], and according to Chen et al. [1, 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 a separate query method that an application can use to evaluate a request's validity. If the request is valid, the application can make the request and the proper outcome is guaranteed.[6] If the programmer fails to make the check, and an invalid request is made, a runtime exception will be thrown, triggering the default handler.[7] 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 used query methods.[8] 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 use 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 manner.[9] As the student gains more insight, he or she will enter the second phase and refactor the handlers.

The Appendix presents curriculum units on deferred error coding and refactoring. Independent of those units, a significant benefit will follow from discussing with students the dangers involved in handling checked exceptions. The Hippocratic Oath includes "First, do no harm," which is good advice in this context. The student should be instructed to code using the standard boilerplate template presented in Section 5 whenever they encounter a checked exception that they either: (1) think will not occur, or (2) are unsure of how to handle. This will take little class time and will significantly reduce the level of dysfunctional error coding.

9. APPENDIX – CURRICULUM UNITS

9.1 CS1 – Deferred Error Coding

I. The ideas behind deferred error coding:
   A. We want to cause any error to throw an uncaught exception so that the default handler will be invoked if a problem takes us off the direct-path.
   B. This lets us actually use the program as long as there are no problems encountered (invalid input, for example).
   C. It also lets us get familiar with the potential problems because the default handler prints a stack-trace and terminates the activity.
   D. Deferred error coding also sets the stage for the subsequent step of refactoring the error handling so that expected situations, such as invalid user input, are smoothly handled without invoking the default exception handler.

II. Deferred coding of method invocation handlers:
   A. If the method's API documentation mentions that an **unchecked exception** might be thrown:
      1. The only thing you should do is add a comment,
      2. It should document the potential expected exception
   B. If the method's API documentation says it throws a **checked exception**:
      1. You should code the method invocation using the following template:

```
try {theMethod();}
  catch (TheCheckedException ex){
   throw new RuntimeException(ex);}
```

      2. No comment is needed, as this is self-documenting.
   C. When the API supplies a **validity query method** that

---

[6] Query methods are not applicable when external events can alter the validity of the request asynchronously.

[7] The programmer might catch the exception instead of using a query method if that simplified his or her application.

[8] 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 was added relieve the programmer from having to deal with `IOException`. The programmer should check that the method returns `null` before executing a user action. Otherwise, an `IOException` is treated as end-of-file, and unintended action may result. Many users may unknowing use applications have this bug. `Scanner`'s complex solution is a good case study of problems encountered when using checked exceptions.

[9] This is called Goal Level G1, also known as failing-fast, which is the first step in the refactoring methodology presented by Chen, et al. [1]

corresponds with the request:
   1. You should insert a comment adjacent to the request,
   2. Describe the validity query method.

D. If a request has a **return code** (also called status code):
   1. Note that in some cases, you must follow a request with the request's status query function to get the code.
   2. The returned quantity may be:
      a) A pure status code, often an `int`, having several possible values, one representing success and the other values indicating various types of failures.
      b) A single returned quantity that is multiplexed, so that a null value (or a negative value for an integer result) indicates a failure, and other values are the result of the request.
   3. In either case, code using the following template:

```
Object result = x.request();
if (result == null) throw
   new RuntimeException(
     "x.request() returned null");
```

      a) The result variable should be of the type returned.
      b) Code the correct test to detect the request's failure.
      c) Pass a meaningful message to the constructor.
   4. No comment is needed, as this is self-documenting.

III. Executing a program coded in this style:

A. You will be able to use it for its purpose,

B. However it will crash on any error, e.g., invalid input.

9.1 CS2 – Refactoring Error Coding

I. The ideas behind refactoring error coding:

A. We will examine each of the locations in the code that can trigger an uncaught exception.

B. *Only* if we determine that the potential problem is *expected* will a fix-up will be coded.

C. For problems that we do *not* expect, the error handling will not be changed. Those will still use the default handler.

II. To prepare for refactoring:

A. Run the program many times and try to make it crash in every possible way.

B. For each crash, note the circumstance, the exception that was thrown, and the line number.

C. If there are some errors you can't trigger, it may be that those are unexpected in your application.

III. Refactoring error coding:

A. Search through the source code for locations that have been flagged:
   1. For unchecked exceptions and validity query methods, there will be comments.
   2. For checked exceptions and return codes there will be code that throws `RuntimeException`.

B. For each flagged location, determine whether or not the potential problem is *expected:*
   1. If you were able to trigger the error, the error is probably expected.
   2. If you were not able to trigger the error:
      a) If there is an obvious corrective action you could take, the error is expected.
      b) If, logically, the error could not possibly occur, based on your program, it is unexpected.

C. If the problem cannot be determined to be either *expected* or *unexpected*, do nothing to the code.

D. If the problem is determined to be *unexpected*, we will just flag the code so we don't have to revisit the question:
   1. For unchecked exceptions and validity query methods, update the comment documenting potential error.
   2. For checked exceptions and return codes change from throwing `RuntimeException` to instead throw `UnexpectedException`, which should extend `RuntimeException` in your application.

E. If the problem is *expected* you should code a fix-up. This may be quite involved; these instructions are only general:
   1. For unchecked exceptions you need to code a `try/catch` block to accomplish the fix-up.
   2. For checked exceptions you need to replace the body of the `catch` clause or eliminate that `try/catch` and code one elsewhere to accomplish the fix-up.
   3. For validity query methods you need to add an `if/else` construct testing the validity query method to accomplish the fix-up.
   4. For return codes replace the body of the `if` statement to accomplish the fix-up.

IV. After refactoring the error coding:

A. Your application should recover from invalid conditions.

B. With more experience using the program, you might encounter errors which you have not refactored:
   1. You can revisit the refactoring process at any time because the errors that have not been refactored are still identified either by a comment or a thrown `RuntimeException`.
   2. It is important to *not* refactor errors not understood:
      a) The default exception handler will report them
      b) Attempting a fix-up will likely compound the error

C. When an application is deployed, it should have a custom default handler to log any unexpected error encountered and inform the end-user of an issue.

10. REFERENCES

[1] 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

[2] Goetz, B. "Java theory and practice: The exceptions debate" Available via search at http://www.ibm.com, May 2004.

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

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

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

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

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

[8] 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.