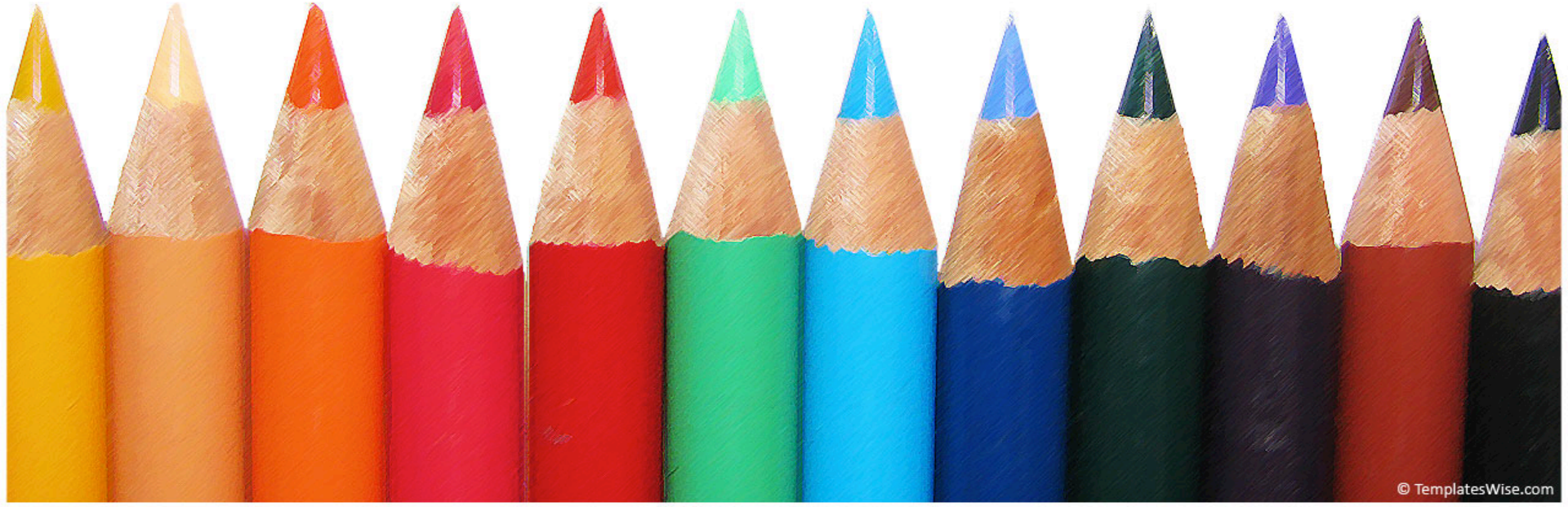
First, Do No Harm: Deferred Checked Exception Handling Promotes Reliability

Duane Buck, Ph.D.
Professor of Computer Science
Otterbein University
dbuck@otterbein.edu



Overview of Talk

- Advantages of exceptions over return codes
- Motivation for designating checked exceptions
- Why exception handling is best deferred
  - Why checked exceptions are controversial
  - Dealing with checked exceptions while debugging
- Refactoring exception handling
  - Taxonomies of exceptions and handlers
  - Selecting the appropriate type of handler
- Hybrid error reporting through validity queries

Return Codes vs. Exceptions

- Return codes have the issue that they must always be checked, even if no problem is expected.
- If used exclusively, this means the majority of the program code would be checking return codes.
- If any go unchecked, it can lead to debugging problems and aberrant behavior.
- Exceptions were motivated by the above problems experienced when using return codes.
  - Goodenough, J. B. Exception handling: issues and a proposed notation. Communications of the ACM, 1975.

Return Codes vs. Exceptions

- **Exceptions are always handled** if they occur, even if not explicitly.
  - A conscious choice must be made to ignore them.
- **Debugging is easier** with exceptions because of the above.
- **Exceptions reduce error handling** code volume.
  - A default handler may handle *all* irrecoverable exceptions.
  - Because they can do non-local transfer, they are great for overcoming the limitations of structured programming.
- **But, harder to use** than return codes.
  - Structured if/else coding is easier for many programmers.
  - For this reason, return codes are still common in APIs for recoverable errors.

Return Codes vs. Exceptions

- **Exceptions** have the advantage for unexpected API errors, because *the programmer does nothing* and enjoys the benefits.
  - Using exceptions for expected errors is sometimes necessary but means those exceptions will have to be explicitly handled, which is not popular with many application programmers.
- **Return codes** are still popular in APIs for errors that are expected because they *are easier to use* than exception handlers.
  - Using return codes for unexpected errors forces programmers to do unnecessary work that is error prone.

Motivation for Checked Exceptions

- Reporting errors with exceptions was nearly a blissful state of affairs.
- The only problem that it was up to API designers to document expected exceptions, and up to the programmer to read the documentation.
  - Otherwise, an expected exception would trigger the default handler, reporting it as a bug.
- For this reason, the designers of Java created the checked exception for problems arising “outside of the immediate control of the program” which therefore should be expected.

Motivation for Checked Exceptions

- Checked exceptions provide a direct line of communication between the API designer and the application programmer.
- If an exception is checked, it can happen even within a bug-free program, and therefore it should be dealt with in the context of the program, and not reported as a bug.

Why Exception Handling Should be Deferred

- Direct-path
  - There exists paths through the application's instructions as services are requested and satisfied by an API
  - A path that ultimately this results in the application providing one of its functions is referred to here as a "direct-path."
- Alternative processing
  - Some API requests may not be satisfied, for various reasons, and require processing which is not considered part of a direct-path.
- Because there are two types of coding, each one should be developed independently, for the several reasons to follow.

Two Types of Concern

- Direct-path
  - Application domain.
  - Concern is focused on the functionality being implemented.
- Alternative processing
  - May be either application or implementation domain.
  - Concern is focused on the user experience.

Two Scopes

- Direct-path
  - Method being coded.
- Alternative processing
  - Potentially the entire system.

Two Skill Sets

- Direct-path
  - Structured programming.
    - Sequential
    - Alteration
    - Iteration
- Alternative processing
  - Unstructured programming.
    - Control may “goto” many places.
    - All methods that may be on the stack have to be prepared to “rollback” things in progress.

Two Levels of Interest

- Direct-path
  - Has high interest as the method is developed.
  - Critical for the application to be of *any* value.
- Alternative processing
  - Lower level of interest as the method is first being developed.
  - Unnecessary for initial debugging.

Checked Exceptions Controversy

- Checked exceptions would have been widely accepted and included in other languages, except that they became controversial because of an unfortunate decision.
- The designers of Java thought checked exceptions were such a good idea that they would *force* programmers to take “advantage” of them.
- But, they did not realize the significant advantages of deferred error coding and refactoring during the development lifecycle.

Checked Exceptions Controversy
(continued)

- Many people don't like checked exceptions, but can't give convincing arguments why.
- This is because they were actually a good idea, so it's hard to argue against them.
- The fact is they *are* helpful *after* the direct-path has been rewritten and debugged.
- Before that time, enforcing the "catch or specify" *requirement* is likely not only to be a nuisance, but *harmful* to the development process and ultimately may negatively impact reliability.

Error Code Refactoring Methodology

- The goal of the *direct-path phase* is to implement the core capability.
  - The default exception handler is relied on to “report and abort” if any error arises.
- The goal of the *error-refactoring phase* is to remediate expected exceptions that may block the direct-path.

Error Code Refactoring Methodology

The *direct-path* Phase

- We must “catch or specify” each checked exception using the principle “First, do no harm.”
- The Approaches (each have issues):
  - Use the throws Exception clause on each method.
  - A method “body” is surrounded by a try/catch block that catches Exception.
  - Individual method invocations are surrounded by a try/catch block.

Error Code Refactoring Methodology

The *direct-path* Phase
Using “throws Exception”

- Disadvantage:
  - The invoker must also “catch or specify” Exception, so you can’t use this technique without potentially impacting other methods.
- Advantages:
  - Very clean looking code.
  - No additional work per method invocation.

Error Code Refactoring Methodology

The *direct-path* Phase
Whole Method try/catch Exception

- Disadvantage:
  - When refactoring, all checked exceptions are exposed simultaneously when the block is removed.
- Advantages:
  - No additional work per method invocation.
  - Does not impact any other method.

Error Code Refactoring Methodology

The *direct-path* Phase
Method Invocation try/catch

- Disadvantage:
  - Ugly looking code.
  - Must add a try/catch block for each method invoked.
- Advantages:
  - Can be done at any time during the development process without impacting other methods.
  - You know what methods are being thrown during the direct-path phase so you can think about it.
  - You can refactor and handle one exception at a time.

Error Code Refactoring Methodology:

The *direct-path* Phase Recommendation for Method Invocation

```
try {
   aMethodThrowingCheckedException();
    } catch (CheckedException ex)
        {throw new
             RuntimeException(ex);}
```

- It acts most like the compiler only gave a warning.
- You can see what is ahead of you as you look at the direct-phase code.
- You can delete the surrounding block and refactor the handling of the one checked exception.

Error Code Refactoring Methodology

The *handler-refactoring* Phase

- For each method, for each checked exception, we must design an appropriate handler.
- We must also design handlers for each misclassified RuntimeException that we encountered during debugging.
- We implement and test one handler at a time.
  - Our experience debugging may help with triggering the exceptions.

Taxonomies of Exceptions and Handlers

- There are four types of exceptions.
- There are three types of handlers.
- This yields twelve combinations.
  - Some combinations do not make sense.

Taxonomy of Exceptions

- **Program bug**
  - Not expected once debugged
  - Example: Use of null reference.
- S**ystem error**
  - Not expected because it's irrecoverable
  - Example: out-of-memory error.
- **Environment fault**
  - Expected because it is not controllable.
  - Example: a network outage.
- **Application domain error**
  - Expected because it is not controllable.
  - Example: incorrect user input.

Taxonomy of Handlers

- "**message/terminate**"
  - The default hander is an example.
- "**message/rollback**"
  - The domain function requested is not completed, but the system continues to process additional user requests.
- "**retry/fallback**"
  - The application retries the failed action attempting to complete the request.
  - After a limited number of failed attempts, it falls back to one of the above.

Exception/Handler Combinations

	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.	N/A, continued execution is risky.	N/A, continued execution is risky.
Expected Exception Types	**environment fault**	Report to IT. Inform user.	Report to IT. Inform user of the issue; rollback and 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.

Hybrid error reporting

- Clearly, the optimal case would be if *all* exceptions were *unexpected*.
- It is possible to make more exceptions *unexpected*!
  - If the API can provide a pretest to determine the validity of a request before it is made, then *that error becomes unexpected*.

Hybrid Error Reporting

- Pretests have become popular in newer additions to the Java API, such as `Scanner`.
- Pretest methods allow the programmer to decide if they want to handle the possible error with an if/else *or* with an exception handler, by not testing!
- A pretest method "provides cover" for the API designer to throw a runtime exception for an otherwise expected situation, which otherwise should be a checked exception under the guideline!

Conclusion

- Exception handlers are best coded independently from the direct-path.
- This supports “First, do no harm” so that the system may be reliably debugged.
- As a second phase, the exception handlers are developed.
- In practice, the above naturally happens, except in Java because of the “catch or specify” *requirement*.
- A workaround was presented for Java.

First, Do No Harm: Deferred Checked Exception Handling Promotes Reliability

Duane Buck, Ph.D.
Professor of Computer Science
Otterbein University
dbuck@otterbein.edu