HAL
open science

Challenging Analytical Knowledge On Exception-Handling: An Empirical Study of 32 Java Software Packages

Martin Monperrus, Maxence Germain de Montauzan, Benoit Cornu, Raphael Marvie, Romain Rouvoy

To cite this version:

Martin Monperrus, Maxence Germain de Montauzan, Benoit Cornu, Raphael Marvie, Romain Rouvoy. Challenging Analytical Knowledge On Exception-Handling: An Empirical Study of 32 Java Software Packages. [Technical Report] hal-01093908, Laboratoire d'Informatique Fondamentale de Lille. 2014. hal-01093908

HAL Id: hal-01093908

https://inria.hal.science/hal-01093908v1

Submitted on 19 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Challenging Analytical Knowledge On Exception-Handling: An Empirical Study of of 32 Java Software Packages

Martin Monperrus Maxence Germain de Montauzan Benoit Cornu
Raphael Marvie Romain Rouvoy

Technical Report, Inria, 2014

Abstract

In this paper, we aim at contributing to the body of knowledge on exception-handling. We take neither an analytical approach ("we think exception handling is good because X and Y") nor an empirical approach ("most developers do Z and T"). Our method is to compare analytical knowledge against empirical one. We first review the literature to find analytical knowledge on exception handling, we then set up a dataset of 32 Java software applications and an experimental protocol to statically characterize and measure the exception handling design. We eventually compare our measures against the claims on exception handling that authors have made over time. Our results show that some analytical principles for exception design do not support the empirical validation: 1) practitioners violate the principle and 2) upon analysis, there are indeed very good use cases going against this principle. This is in particular the case for "Empty Catch Blocks are Bad" and "Do not Catch Generic Exceptions".

1 Introduction

C code handles errors using return codes. Java, as well as other most modern programming languages has a dedicated construct for errors: exceptions. The core idea behind exception dates back to the 70ies [10] and its implementation in Java with `try/catch/finally` is very close to the one of Modula designed in the 80ies [19]. However, there is still no established knowledge on what a good or a bad design of exception handling is.

We can of course find authoritative experts who wrote a book chapter or an article on this topic [2, 21]. However, many points on this topic are still debated [16] and you can find on open-source forums lively debates (if not flame wars) about exceptions.

In this paper, we aim at contributing to the body of knowledge on exception-handling, by identifying the reasons behind known principles of exception handling design. We take neither an analytical approach ("*we think exception handling is good because X and Y*") nor an empirical approach ("*most developers do Z and T*"). Our idea is rather to compare analytical knowledge against empirical one. We hope that from this confrontation will emerge principles founded on analytical arguments and validated by empirical practices.

We first review the literature to find analytical knowledge on exception handling (Section 2), we then set up a dataset of 32 Java software applications and an experimental protocol to statically characterize and measure the exception handling design (Section 3). We eventually compare our measures against the claims on exception handling that authors have made over time.

To sum up, our contributions are:

- a survey of analytical knowledge on exception handling,
- a set of empirical facts on the exception handling design of 32 Java libraries,
- the validation or falsification of 6 important claims about good exception handling.

2 Analytical Knowledge About Exception Handling

What is *analytical knowledge* about exception-handling? It is knowledge that is derived from what one thinks about exceptions from an analytical point of view. Conversely, *empirical knowledge* is knowledge that is derived from empirical observations on how practitioners use error and exception handling.

In this section, we review what has been said by authoritative references about exception handling design. We consider as *"authoritative"* the knowledge described in books, articles published

```
try {
  // code
} catch (IOException ex) {
  throw new ServerConnectionProblemException(
      ex);
}
```

Listing 1: Example of Exception Wrapping/Recast in Java

in respectful venues and Internet posts authored by senior respected engineers from known IT companies. Thanks to the Internet, many developers assert analytical knowledge on exception handling based on their own experience[1], we do not consider this non-authoritative knowledge.

We propose to classify the analytical knowledge about exception usages into six categories: the design of *catch sites* (Section 2.1), the design of *throw sites* (Section 2.2), the design of *exception classes* (Section 2.3), the design of *try blocks* (Section 2.4), the design of *finally blocks* (Section 2.4). They are ordered by frequency of use.

2.1 Catching Exceptions

When working with a programming language that offers exceptions, the first thing one learns about the exception system is how to catch an exception. Indeed, many exceptions occurred at development time, due to the program under construction being temporarily incorrect. Those exceptions may be thrown by the runtime environment (*e.g.*, a `NullPointerException` in Java) or by libraries (*e.g.*, an `IllegalArgumentException`).

We have found many pieces of analytical knowledge about catching exceptions. First, it is recommended that, there should be no empty catch blocks (*"Don't Catch and Ignore"* [14], *"Don't ignore exceptions"* [2]).

Second, the scope of the caught exception is important, the scope is statically specified by the exception type ("`catch (IOException e)`"). Many believe [14, 6] that one should not catch the most generic exception object ("`Exception`" or even "`Throwable`" in Java): *"Do not catch Exception"* [14].

Third, there is a known practice consisting of translating an exception type into another one, that is more appropriate, as shown in Listing 1. This is known as "exception translation" [2], "exception wrapping" [6] or "exception recast" [21]. The goal of exception wrapping is to provide a clean exceptional behavior, both in terms of exception types and in terms of system layers. All the authors agree on this point. Bloch states that this as *"higher layers should catch lower-level exceptions and, in their place, throw exceptions that can be explained in terms of the higher-level abstraction"* [2]. Cwalina recommends *"wrapping specific exceptions thrown from a lower layer in a more appropriate exception, if the lower layer exception does not make sense in the context of the higher-layer operation"* [6]. Wirfs-Brock goes along the same line and advises to *"Recast lower-level exceptions to higher-level ones whenever you raise an abstraction level"* [21].

Exceptions often initiate a debugging session. For the sake of debugging, there are two important guidelines: one should *"specify the inner exception when wrapping exceptions"* [6] as shown Listing 1; also, the stack trace of the original exception—of primary importance for debugging—should be never be destroyed (what McCune calls "destructive wrapping" [14]).

2.2 Throwing Exceptions

The second basic activity when working with a language with exception is to throw exception objects (instances of exception classes), usually when error conditions are met.

First, let us recall the traditional way of handling errors in exception-less programming language, such as standard C. It consists of returning error codes or passing mutable error variables to store them. This has several drawback and exceptions have been invented to cope with them (see [6] for a comprehensive discussion). However, due to habits or education, many developers keep using error-codes even with languages with exceptions. Consequently, it has to be said to *"report execution failures by throwing exceptions"* [6] and to *"use exceptions rather than return codes"* [13]. However, exceptions can be very handy to have an agile control flow and some use them outside exceptional cases. Some authors disagree and consider that exceptions should be used *"only to signal emergencies"* [21].

Second, there exist guidelines on the exception types to be thrown. Standard development libraries already define a lot of exception classes. It has been recommended to *"favor the use of standard exceptions"* [2, 6]. For instance, Java's `IllegalArgumentException` is appropriate to signal that a parameter is incorrect. Also, it has been advocated not to throw completely generic excep-

[1] See for instance `http://c2.com/cgi/wiki?ExceptionPatterns` (last visited: July 2, 2013)

tions [14].

Another important point in the literature is about how to create a new exception. Cwalina et al. recommends *"to consider using exception builder methods"* [6]. Wirfs-Brock warns from repeatedly re-throwing the same exception [21]. Both guidelines aim at facilitating debugging and evolution.

Finally, the guidelines on throwing exceptions can also be very technical as illustrated by the following recommendations of Cwalina et al.'s book [6]: 1. *"Avoid explicitly throwing exceptions from finally blocks"*; 2. *"Do not have public members that can either throw or not based on some option"*; 3. *"Do not have public members that return exceptions as the return value or an out parameter"*.

2.3 Designing Exception Classes

Once a programmer is acquainted with throwing and catching exceptions, she may design her own exception classes. Let us now review the main guidelines on designing exception classes.

First, Java, .Net and other languages provide two kinds of exceptions: *checked* and *unchecked* exceptions. The checked exceptions are subject to compile-time verification. Unchecked are similar to those exceptions found in dynamic languages. We detail their semantics later (in Section 3.7). There is no consensus on *when* and *how* to choose between checked or unchecked exceptions. For example, some technology gurus, such as Robert C. Martin ("Uncle Bob"), recommend to *"use unchecked exceptions"* [13].

Second, thrown exceptions are meant to signal an error and to trigger some kind or recovery (or to log for offline analysis). Both use cases require detailed information. As such, it is recommended to add state to exception classes (*e.g.*, using class fields) [13, 21], which is summarized as *"provide context with exceptions"* [13].

Finally, exceptions in object-oriented languages are fully fledged classes and can form *exception hierarchies*. As Jeff Atwood puts it [1], *"designing exception hierarchies is tricky"*. Wirfs-Brock proposes to mitigate the problem by avoiding *"declaring lots of exception classes"* [21].

2.4 Other Guidelines

The last guidelines to be discussed relate to the design of `try` and `finally` blocks.

Designing Finally Blocks Cwalina et al. value `finally` blocks that can be found in some languages, such as Java and .Net. They recommend them especially for cleanup code [6]. However, they assert that one should not explicitly throw exceptions from `finally` blocks [6]. When an exception is thrown in a `finally` block, the initial exception and its stack trace is lost forever and this greatly hinders debugging. This also holds for "return" statements in finally blocks.

Designing Try Blocks There are very few discussions on the how to design `try` blocks. The only notable guideline is by Google engineers, who recommend to *"minimize the amount of code in a try block"* [17].

To sum up, much has been analytically said about exception design. Nevertheless, what does the reality of programs and programmers look like?

2.5 Relation between Analytical and Empirical Knowledge

Analytical and empirical knowledge on exception-handling are not completely independent. Analytical knowledge about exception handling has an impact on practice.

First, it gives the foundations of the exception-handling constructs and semantics of a programming language. Given their experience and analysis, the designers of a programming language make design decisions on how exceptions are specified and handled (statically and dynamically). This has an impact on practice: due to the choices of the language designers, some design may be impossible to implement given the constructs of the language and the associated compilation errors.

Second, thinkers derive guidelines from this analytical knowledge about exception handling. Within a given programming language, those guidelines aim at helping practitioners to design and implement "good" exception handling ("good" w.r.t. performance, understandability, reuse, maintenance, etc.). The readers of those guidelines may follow them, which is an impact from the analysis to the practice. There may be a relation between the programming language design decisions and guidelines: some guidelines may be workarounds for pitfalls of the exception system.

```
<try>try <block>{
    <expr_stmt><expr><call>
        <name><name>s</name>.<name>update</name></name>
        <argument_list>()</argument_list>
    </call></expr>;</expr_stmt>
}</block><catch>catch (
    <param><decl>
        <type><name>GeneralSecurityException</name></type>
        <name>e</name>
    </decl></param>)<block>{
        <throw>throw <expr>new <call><name>DNSSECException</name><argument_list>(
            <argument><expr><call>
                <name><name>e</name>.<name>toString</name></name>
                <argument_list>()</argument_list>
            </call></expr></argument>)
        </argument_list></call></expr>;</throw>
    }</block></catch>
</try>
```

Listing 3: The XML tree extracted from Listing 2 using srcML

```
try {
    s.update(); }
catch (GeneralSecurityException e) {
    throw new DNSSECException(e.toString()); }
```

Listing 2: An excerpt of DNSJava that throws an exception

3 Empirical Findings on Exception Handling

We now set up an experiment to observe, characterize, and measure the exception handling design in Java libraries. Our protocol consists in statically studying the source code of Java software.

We select 6 design principle for exception handling, devise measures for them and assess whether those principles hold in practice. This process enables us to validate or invalidate analytical knowledge. Furthermore, our experiments also confirm some existing findings through replication.

3.1 Challenged Analytical Principles

Based on our analysis of analytical knowledge (see 2), we select 6 analytical principles that are prominent in the literature:

1. *"Empty catch blocks are bad"*,
2. *"Reuse Standard Exceptions"*,
3. *"Define Exceptions With State"*,
4. *"Use Checked Exceptions"*,
5. *"Consider Exception Builder Methods"*,
6. *"Do not Catch Generic Exceptions"*.

We choose those principles because they are either emphasized by different authors (*e.g.*, *"Empty catch blocks are bad"*) or they resonate with our experience and discussions with our fellow programmers.

3.2 Experimental Protocol

We now present the process we follow to statically analyze the exception handling design of Java libraries. First, the *abstract syntax tree* (AST) of a corpus of Java programs are extracted and persisted as XML trees. Second, those trees are queried and analyzed using the XQuery language [3].

3.2.1 Dataset

We build a dataset of Java program as follows. The inclusion criteria is to maximize the diversity of application domains and development processes. It goes from parts of the standard Java library (java-regexp, java-io) to database applications (h2) and desktop applications (columba). Some programs come from well-known and established development organizations (Apache Commons Collections, Apache Lucence) and other are more specific (FraSCAti [20] is a middleware platform developed in our software engineering research group for several years). There is a total of 32 Java libraries in our dataset. These are argouml, avrora, batik, carol, columba, commonscollections, DNSJava, fop, foray, FraSCAti, h2, itext, java-regexp, java-util, jboss, jedit, jface, jhotdraw, jUnit, jython, log4j, lucene, org-eclipse-jdt-core, org-eclipse-ui-workbench, pmd, rhino, Scarab, solr, Struts, sunflow, tomcat, xalan. This dataset is available as auxiliary material.

3.2.2 Implementation

Transformation of Java Source Code into XML We extract the ASTs of java libraries using srcML [5]. An entire program consisting of dozens

of Java files is transformed into a singe XML file. In our dataset, those XML files contain between 105,000 and 4,826,000 nodes and weight between 1,172 and 52,893 kilobytes. Listing 3 shows an excerpt of Java source code and the corresponding XML representation. The snippet contains a `try` and a `catch` block, both are converted into the appropriate XML tags.

Analysis of XML Abstract Syntax Trees To explore the exception handling design of real Java libraries, we query the XML abstract syntax trees using the XQuery language. This language, combined with XPath selectors, enables to quickly obtain interesting metrics. For instance, the number of `catch` blocks containing a `throw` statement is expressed as `count(//catch//throw)`. Listing 2 would match this selector.

3.3 Overview of the Java Exception System

Since we analyze Java libraries, this section briefly introduces the Java exception handling system. In Java, an exception is *"thrown"* (or *"raised"*, we equate those terms) when a problem arises. For example, a `NullPointerException` indicates that an access is performed to a variable that does not contain any reference to a valid object, or `IllegalArgumentException` is thrown when a method parameter is invalid.

Some exceptions are thrown by the underlying *Java Runtime Environment* (JRE) (*e.g.*, `NullPointerException`, `ArrayIndexOutOfBoundsException`). The others exceptions are also explicitly thrown from the application code, by using the keyword `throw`. Note that developers can also manually throw JRE exceptions (*e.g.*, `throw new NullPointerException()`.

There are two categories of exceptions in Java: *checked* and *unchecked*. Checked exceptions require to be declared in the associated method signature if they can be thrown, but are not caught within the body of this method. The idea of checked exception is to force the developers of client code to handle foreseeable errors.

Catching an exception is achieved by the pair of keywords `try` and `catch`. The block associated to the `try` (a try block) describes a sequence of instructions that can potentially throw an exception, while the block associated to the `catch` refers to the instructions to be executed whenever a specific exception is thrown within the try block. By

```
int port = 8080; // default value
try {
    port = loadPortFromConfigFile();
} catch (Exception ex) {
  // keeps the default value
}
openSocket(port);
```

Listing 4: Empty catch blocks to support setting default values

```
Strategy o = createDefaultStrategy();

try {
   o = createBetterStrategy();
} catch (Exception ex) {
 // keeps default strategy
}

try {
    o = createBestStrategy();
} catch (Exception ex) {
 // keeps default or better strategy
}
```

Listing 5: Empty catch blocks for implementing a strategy design pattern

using several `catch` clauses, the developer can distinguish the treatment to be applied upon error depending on the type of exception thrown by the system.

In addition, the keyword `finally` can be appended at the end of a try block (whether there is no, one or multiple catch blocks) to describe a body of statements that are always executed (whether an exception is thrown or not).

3.3.1 Presentation Template

For all the challenged design principles, we use the same template for presentation: definition, analytical knowledge, previous empirical knowledge, our empirical results, discussion.

The *definition* explains the content and rational of this principle. The *analytical knowledge* summarizes the arguments presented in Section 2. The *previous empirical knowledge* gives empirical figures about this claim (*e.g.*, abundance), which we found in previous papers. Then, we present out *empirical results* measured on the dataset presented in 3.2.1. Finally, a *discussion* synthesizes the pros and cons of the principle and whether it has supported the confrontation with reality.

```
// in ./org/eclipse/jface/viewers/deferred/
    BackgroundContentProvider.java
while (true) {
  try {
    // this is the main work
    doSort(sortingProgressMonitor);
  } catch (Exception ex) {
    // ignore
  }
}
```

Listing 6: Excerpt of an empty catch block for resilience in Eclipse JFace

3.4 Challenging "Empty Catch Blocks are Bad"

Definition An empty catch, as its name suggests, contains no code. Upon execution, an empty catch stops the propagation of an exception and leaves the program state as is.

Analytical Knowledge Bloch says *"don't ignore exception"* (in the sense of not silencing exceptions with empty catch blocks) [2, p. 258] and Tim McCune goes along the same line [15]. Bloch's arguments are sharp: *"The advice in this item applies equally to checked and unchecked exceptions. Whether an exception represents a predictable exceptional condition or a programming error, ignoring it with an empty catch block will result in a program that continues silently in the face of error. "* However, Bloch may admit some empty catch blocks when they *"contain a comment explaining why it is appropriate to ignore the exception."*.

Previous Empirical Knowledge Reimer [18] observed that between 2% and 26% of catch blocks are empty in the JDK and in 6 closed-source applications. Fu and Ryder [7] confirmed that between 5% and 40% of catch blocks are empty handlers (out of 2099 catch blocks from 5 applications).

Our Empirical Results In our dataset, all (32/32) libraries contain empty catch blocks, up to hundreds of times. For instance, Eclipse's JDT-core has 423/1430 (30%) of them and Tomcat 309/2348 (13%). This further confirms the aforementioned empirical results [18, 7]. We have refined the measurement to count empty documented catch blocks (this measure has never been computed before). For instance, iText contains 118 empty catch blocks, 43 of them being documented. Some libraries document catch blocks very systematically, up to 100% (423/423 for JDT-core).

Discussion According to Bloch's statement, empty catch blocks should be rare. In practice, they often occur up to 1 catch block out of 3 (for Eclipse JDT). This empirical reality suggests that there exist good reasons to write empty catch blocks. To further understand the meaning and the relevance of empty catch blocks, we have randomly browsed dozens of empty catch blocks. We now present the result of this grounded approach and discuss usages of empty catch blocks that we consider as proper design: falling back to default values (see Listing 4), trying different strategies (see Listing 5), and resilience (see Listing 6).

Falling back to a default value means that if the computation fails, there is a meaningful best-effort value to continue the execution. There are many variants of this, such as assigning the default value inside the catch block or returning a default value when in a method, Listing 4 presents such a catch block.

Trying different strategies consists in variations of the chain of responsibility design pattern where different alternatives are available for the same task. For instance, Listing 5 shows several statements that are chained in order to try several strategies. The empty catch block that stops the exception of a strategy enables one to keep the previously successfully computed object.

By resilience, we mean that an exception may crash the current action being performed, but not the whole application. For instance, a request to server may fail, but should not crash the application. An application of such a resilience scenario is shown in Listing 6. Similarly, in a loop over elements, an exception may occur for one element but should not crash the rest of the computation being performed (not presented for sake of space). In both cases, an empty catch block is relevant.

Empty catch blocks can be used to transform exception-oriented error-handling into return-oriented error-handling as shown in Listing 7.

Finally, there exists *hidden* empty catch blocks—*i.e.*, programming designs or idioms that are in essence equivalent to empty catch blocks. For instance, using a new thread to perform an action is equivalent to an empty catch: when an exception occurs, the newly created thread crashes, but not the whole application. Hence, applications with many short-lived threads, such as GUIs with thread-based event handling, have a kind of built-in resilience due to those many *"implicit empty catch blocks"*.

```
// in org/mozilla/javascript/Kit.java
try {
  return loader.loadClass(className);
} catch (ClassNotFoundException ex) {
} catch (SecurityException ex) {
} catch (LinkageError ex) {
} catch (IllegalArgumentException e) {
  // Thrown if className has incorrect
      characters
}
return null;
```

Listing 7: Empty catch blocks in Rhino can convert exception-oriented error-handling in return-oriented error-handling

> Empty catch blocks are much used in practice. When the global application state is correct, stopping the exception propagation is appropriate and this is what empty catch blocks do.

Bloch's sharp reject of empty catch block was not completely right with regards to empirical evidence: we would reformulate *"don't ignore exceptions"* as *"consider using empty catch blocks when the global application state is correct"*. To conclude, the empirical evidence has seriously challenged the analytical belief that *"empty catch blocks are bad"*. We have presented a list of scenarios in which they are indeed relevant and meaningful.

3.5 Challenging "Reuse Standard Exceptions"

Definition A standard exception is an exception whose definition is provided by a third-party software library, including the standard runtime environment, such as the JDK in Java. Examples of standard exceptions include `IllegalArgumentException` and `IllegalStateException`. Such exceptions are usually caught and handled by the applications building on these libraries.

However, *"reuse standard exceptions"* actually focuses on throwing exceptions and not catching them. Indeed, standard exceptions can also be thrown by application code whenever their semantics matches the context of execution. For instance, in Java, when an incorrect parameter is passed as an argument of a method call, it is meaningful to throw an *IllegalArgumentException*, as shown in Listing 8, which reports such an example from our dataset.

Analytical Knowledge Cwalina and Abrams [6] promote reusing standard exceptions. In particular, they advise to *"consider throwing existing exceptions residing in the System namespaces instead of creating custom exception types (esp for usage errors)"*. This means adopting, whenever it is considered as appropriated, the standard exceptions that are provided by the underlying system (*e.g.*, the .Net or Java runtime environments). According to them, developers should *"create and throw custom exceptions if [they] have an error condition that can be programmatically handled in a different way than any other existing exception. Otherwise, [they should] throw one of the existing exceptions."* This means that before creating her own exception, the developer should look at whether some the exception abstractions provided by the libraries at hand can be meaningfully applied. In any case, they advise to *"not create and throw new exceptions just to have ´your team's´ exception."* All these rules are acknowledged by Bloch in [2], who recommends to *"favor the use of standard exceptions."* Indirectly, by claiming to "avoid declaring lots of exception classes", Wirfs-Brock [21] advises the developers to carefully design custom exceptions, an objective that can be achieved by reusing standard exceptions.

Previous Empirical Knowledge We do not have identified any empirical studies on throwing standard exceptions.

Our Empirical Results In our dataset, all (32/32) applications throw standard exceptions within their code and for 22 of them (73%) the most thrown exception is a standard exception. For instance, in iText, the most commonly thrown exception is an `IllegalArgumentException` (168/881 thrown exceptions, 19%). More specifically, one can observe that standard exceptions like `IllegalArgumentException`, `IllegalStateException`, and `IOException` are all thrown by the libraries of our dataset.

Discussion Among the standard exceptions that are reused, our experience with browsing exception-handling code reveals specific cases. First, the standard exceptions `Exception` and `Throwable` are used to report error messages to the end-user (see Listing 9). Second, the standard exception `RuntimeException` is mostly used to convert a checked exception into a unchecked

```
// in java/org/ow2/carol/jndi/ns/AbsRegistry.
    java
public void setPort(int p) {
  if (p <= 0) {
    throw new IllegalArgumentException("The
        number for the port is incorrect. It
        must be a value > 0. Value was '" + p
        + "'");
  }
  this.port = p;
}
```

Listing 8: Reuse of the standard exception `IllegalArgumentException` to capture non-authorized parameters.

```
// in src/java/org/apache/fop/render/rtf/
    rtflib/tools/BuilderContext.java
public void replaceContainer(RtfContainer oldC
    , RtfContainer newC) throws Exception {
  // treating the Stack as a Vector allows
      such manipulations (yes, I hear you
      screaming ;-)
  final int index = containers.indexOf(oldC);
  if (index < 0) {
    throw new Exception("container to replace
        not found:" + oldC);
  }
  containers.setElementAt(newC, index);
}
```

Listing 9: `Exception` used to report an error.

one, we will come back on this point in 3.7.

> All in all, the design principle *"Use Checked Exceptions"* is much applied in our dataset. There is a clear match between analytical and empirical knowledge, which validates the principle.

3.6 Challenging "Define Exceptions With State"

Definition An exception with state is an exception that contains additional data on the failure cause or the failure context. This additional data is usually encoded as fields in the exception object.

```
// in src/org/argouml/util/MyTokenizer.java
public void putToken(String s) {
  if (s == null) {
    throw new NullPointerException("Cannot put
        a null token");
  }
  putToken = s;
}
```

Listing 10: `NullPointerException` used to capture an illegal argument.

For example, a `IllegalParameterException` can be defined with a field that contains the name of the incorrect argument.

Analytical Knowledge Martin says *"provide context with exceptions"* [13, p. 107]. Wirfs-Brock goes along the same line: *"provide context along with an exception"* [21]. Martin says that additional data should always be present: *"Each exception that you throw should provide enough context to determine the source and location of an error. In Java, one can get a stack trace from any exception; however, a stack trace can not tell you the intent of the operation that failed."*

Bloch is more specific about the nature of the additional information: *"Include failure-capture information in detail messages"* [2, p. 254]. In particular: *"the detail message of an exception should contain the values of all parameters and fields that "contributed to the exception.""*

As we can see, all authors agree that the stack trace is insufficient and that additional data is required.

Previous Empirical Knowledge We do not have found anything on this topic.

Our Empirical Results In our experiment, we consider that an exception has a state if and only if it contains at least one additional significant field[2]. In our dataset, there are 389 domain specific exceptions. Among them, we found 112 exceptions with additional fields. Also, we notice that in practice, many exception definitions duplicate already existing information. For example, 44 exceptions have the cause exception as a field (already present in `Throwable`), 2 exceptions duplicate the message (idem) and 1 explicitly defines its stack trace (ibidem). All this data being already present in a basic exception, these exceptions are not considered as exception with state.

On those 112 exceptions with state, 101 exceptions contain additional data directly related to the context where the failure occurs. For instance, Listing 11 reports an excerpt of a stateful exception: the immutable exception includes data about when a signature expired (field `when`) and when this signature has been called (field `now`).

Also, there are 11 exceptions that contain information on the type of error encoded as an error

[2]The Java specific field `serialVersionUID` is not considered as an additional field

```java
class SignatureExpiredException extends DNSSECException {
  /* When the signature expired */
  private Date when;
  /* When the verification was attempted */
  private Date now;

  SignatureExpiredException(Date when, Date now) {
    super("signature expired");
    this.when = when;
    this.now = now;
  }
}
```

Listing 11: Domain exception with state from DNSJava

code value. Listing 12 shows a example of such an exception.

```java
class ClassFormatException extends Exception {
  public static final int ErrBadMagic = 1;
  public static final int ErrBadMinorVersion = 2;
  public static final int ErrBadMajorVersion = 3;
  ...
  public static final int ErrInvalidMethodSignature = 28;

  private int errorCode;
  public ClassFormatException(int code) {
    this.errorCode = code;
  }

  /**
   * @return int
   */
  public int getErrorCode() {
    return this.errorCode;
  }
}
```

Listing 12: Domain exception with error-code from jdt-core

Discussion We observe that less than 30% domain-specific exceptions (112/389) define some state. This shows that the design principle *"define exceptions with state"* does not hold systematically.

Figure 1 shows the number of exceptions with state and the number of stateless exceptions for each project. Only 1 project (java-regexp) contains 100% of stateful exceptions (in total 1 domain-specific exception) where 8 projects do not contain any exception with state. The distribution of the exceptions with state does not seem to be correlated with the project size. For example, fop and java-util have the same number of domain-specific exceptions (20 exceptions defined) but fop has only 4 exceptions with state (20%) where java-util has 12 (60%). The distribution does not seem either linked with the reputation of the organization behind each library. Indeed, the well-known and used tomcat has 15% of exception with state, the famous Eclipse's jdt-core has 22% while the much less known libraries and Avrora have respectively 57% and 68% of stateful exceptions.

These figures show that Bloch, Martin, and Wirfs-Brock's analytical design principles do not really find their ways in real projects.

3.7 Challenging "Use Checked Exceptions"

Definition The Java language defines two kinds of exceptions: checked and runtime exceptions (also called unchecked exceptions) [11, Chapter 11]. The checked exceptions are subject to compile-time verification. The verification ensures that raised checked exceptions are either handled or explicitly declared in the enclosing method's signature.

Analytical Knowledge The authors of the Java language specification [11] say that *"Most user-defined exceptions should be checked exceptions. "* because *"compile-time checking [..] aids programming in the large"*.

Bloch [2] recommends to *"use checked exceptions for recoverable conditions and runtime exceptions for programming errors"* (item 40) and to *"avoid unnecessary use of checked exceptions"* (item 41). Both recommendations are related to a known debate on first, whether Java checked exceptions are good or not [16], and what should be the design principles to choose between one form and the other [9].

We can list many other documents, either in favor or against checked exceptions. Indeed, as the Java language designers themselves put it, checked exceptions are sometimes considered as *"an irritation to programmers"* [11].

For unrecoverable checked exceptions, the programmers have 2 choices: either they write a fake try-catch block to simulate a recovery block or they declare the checked exception in the enclosing method signature. The former is a known practice that consists of catching checked exceptions and wrapping them in runtime exceptions, as shown in Listing 13. Let us call this implementation pattern the *"checked-runtime wrapping pattern"*.

The other solution, explicitly declaring the exception that can be thrown is called the *"exception declaration pattern"*. When developers declare

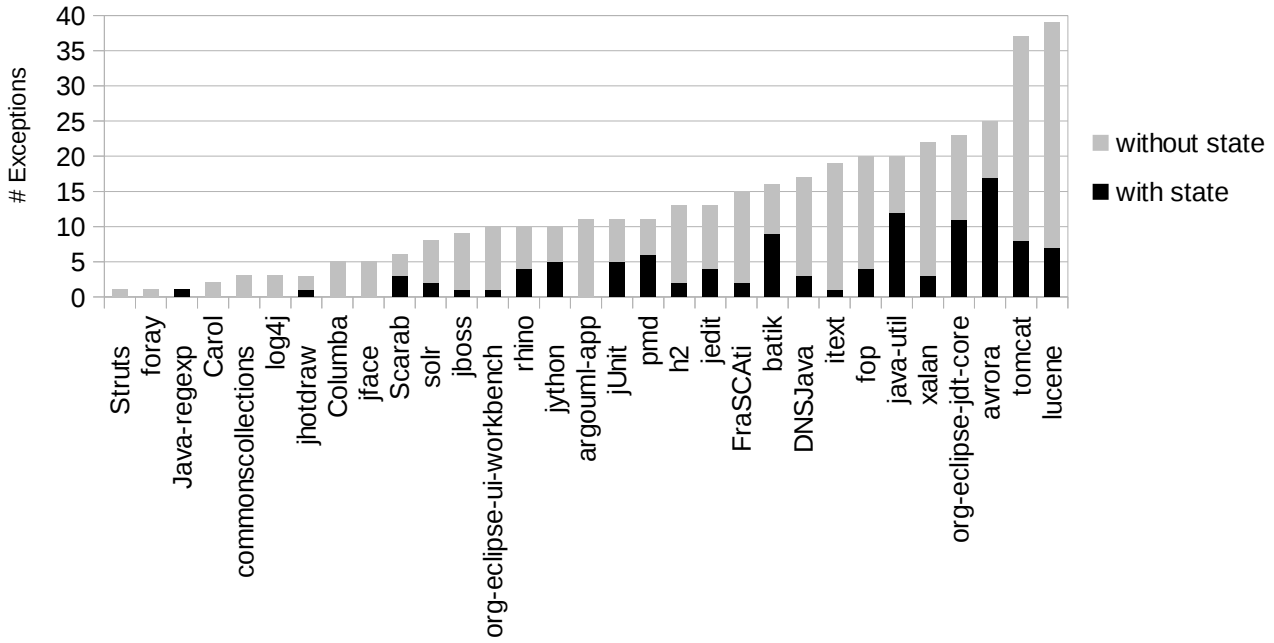

Figure 1: Number of exceptions with state and the number of stateless exceptions for each project

```
try {...}                                          1
catch (IOException e) { //IOException is           2
    checked
  throw new RuntimeException(e);                   3
}                                                  4
```

Listing 13: The Classical Workaround For Checked Exceptions

`throws Exception` in method signatures instead of declaring a specific type of checked exception, it means that they do not think that the fine grain exception declaration of Java would make sense.

Both patterns are somehow empirical recipes to disable the exception checking mechanism of the Java compiler.

Previous Empirical Knowledge Kiniry [12] analyzed the JDK 1.4.1 and asserted that there are 50 defined runtime exceptions and 150 checked exceptions. He also shows that runtime exceptions are more often thrown: there are $3,000$ times where a runtime exception is thrown versus $2,650$ where a checked exception is thrown.

Our Empirical Results We have measured the usage of checked and runtime exceptions.

First, we have looked at whether programmers of software libraries from our dataset prefer to use checked or runtime exceptions. In total, 15/32 defines more checked exceptions than runtime exceptions and 12/32 defines more runtime exceptions than checked exceptions (the remaining declare as many checked as runtime). On the extremes, Eclipse's jdt-core defines 19 runtime exceptions and 3 checked exceptions, and Rhino defines 9 runtime exceptions out of 10 domain exceptions.

Second, we have measured the number of instances of the checked-runtime wrapping pattern. The most basic version of this pattern consists of catching the checked exception and throwing a RuntimeException: 26/32 projects use at least once this pattern. The maximum number of instances is in Apache Lucene: the code base throws 263 new runtime exceptions.

Third, we have measured the number of instances of the exception declaration pattern for `Exception`, that is the number of times `throws Exception` is declared in the method signature. 27/32 projects use at least once the exception declaration pattern. The maximum number of instances is in Scarab where 525 methods declare `throws Exception` in their signature.

Finally, we consider the concrete example of `IOException` which is the most common checked exception of the JDK. It happens when an input/ouput operation fails, for instance, when one opens a file that does not exist. All projects do catch `IOException` (up to 456 times in Lucene). All projects declare `IOException` to be throwable in at least one method. The maximum value is for Lucene: 4765 methods declare throwing `IOException`.

Discussion Checked exceptions are used in practice: programmers indeed define and throw checked exceptions. However, there are also two major pieces of evidence that checked exceptions are questionable.

First some projects deliberately only use runtime exceptions. They do not seem to be written by novice uneducated developers. In our dataset, there are 6 software projects hosted by the Apache foundation that prefer runtime exceptions. The Apache foundation is known to gather very good developers and to foster best-practices. Furthermore, Eclipse jdt-core also much favors runtime exceptions. This is very interesting: the developers of a Java compiler, who are aware of all subtleties of the language, who have handled thousands of compiler bug reports, prefer to only use runtime exceptions.

> Our results challenge the statement that *"compile-time checking [..] aids programming in the large"*: many respectful developers do not think so.

Second, we have shown many traces of workarounds to trick the static exception checking mechanism (in the form of the checked-runtime wrapping pattern), both in terms of number of projects using the pattern and total number of occurrences. This means something with respect to library design. When one defines checked exceptions within one's own project, one is the only impacted by this design choice. When one declare checked exceptions in a library interface, all clients are impacted. Our empirical study gives numerical values of this impact: there are hundreds of times where developers catch a potential checked exception declared by a library and wrap them as a runtime. For instance, there is a total of 1034 catch elements of `IOException` which re-throw a fresh runtime exception (in our dataset). If `IOException` had been a runtime exception, many workarounds and similarly irritating code would have been avoided.

> There is a difference between inner-application checked exceptions and library checked exceptions. The latter can induce workarounds in thousands of methods of client code.

```
// in DNSJava
throw parseException(fullName, "Name too long"
    );

// in Jython
throw makeException(t);

// in Foray
throw unexpectedValue(value, fobj);
throw this.unexpectedRetrieval();

// in H2
throw convertException(e);
throw throwUnsupportedExceptionForType("+");
throw getUnsupportedException();
```

Listing 14: Examples of the Usage of the Exception Builder Pattern

3.8 Challenging "Consider Exception Builder Methods"

Definition An exception builder method is a helper method to obtain exception objects. It is a variant of the builder pattern [8]. Instead of using a `throw new A(...)` the developer invokes an helper in order to instantiate the exception to be thrown, *e.g.* `throw helper.raiseException(...)`.

Analytical Knowledge Kwalina recommends *"using exception builder methods."* The main reason is to avoid code bloat for the configuration of the exception objects.

Our results In our dataset, 19/32 libraries contain usages of the *exception builder pattern*. Still, some projects do not use this pattern at all, this is the case for Apache Commons collections and java-utils. Then, the frequency of usage ranges from exceptional (less than 1%) in projects such as Eclipse jdt-core, Jboss, or Tomcat and raises up to 60.5% in Java-regexp, 81.7% in Jython, and 81.8% in H2 code-bases. Jython and H2 are particularly interesting as they are following the design principle very systematically.

Discussion We have identified 3 forms of using the *exception builder method* pattern.

First, the object where the exception occurs embeds the builder method. This follows the *single responsibility principle* in the scope of the object. A method centralizes the exception building mechanism for a given class. Examples from 4 different projects are shown in Listing 14.

Second, a utility class provides some static methods for exception building. This centralized

```
// Jython
throw Py.TypeError("__dict__ must be set to a Dictionary "+newDict.getClass().getName());
```

Listing 15: Examples of the Usage of the Exception Builder Pattern as Static Method

```
// Jython (info is the injected object)
throw info.unexpectedCall(args.length, true);

// DNS Java (st is the injected object)
throw st.exception("unexpected tokens at end of record");
```

Listing 16: Modularizing exception building using dependency injection

class is accessible from anywhere in the project. As shown in Listing 16, Jython uses this pattern for managing the building of Python exceptions.

Third, an injected dependency provides some exception related methods. This elegant design allows the replacement of the exception building mechanism by another one when necessary. In addition, a second benefit of this form is to permit the use of exception mocks for testing purpose.

To our knowledge, we are the first to provide an empirical characterization of the usage of exception builders. We note that projects that use *exception builder method* pattern do not limit its implementation to a particular form, but mixes them. H2 uses (1) and (2), DNSJava uses (1) and (3), Jython uses all three forms.

> Not all projects use a form of exception builder, but when it is done, it gives birth to very advanced design.

3.9 Challenging "Do not Catch Generic Exceptions"

Definition In most mainstream object-oriented programming languages, the exceptions use the same type system as classes. Consequently, one can define a hierarchy of exceptions. This has very important consequence, when a try-catch block declares to catch an exception type t, it actually catches exception instances of all subclasses of t. In other terms, when one writes `catch(Exception e)`, all exceptions are caught, when one writes `catch(FileNotFoundException e)`, only specific exceptions are caught (In Java, the most generic type of exception is `Throwable` but this is implementation-specific. In this paper, when we use `Exception`, we refer to the conceptual most generic kind of exception).

Analytical Knowledge McCune asserts *"Don't catch Exception"*. Cwalina qualifies this assertion and adds a distinction between framework code and application code: *"Do not swallow errors by catching non-specific exceptions in framework code"*, *"Avoid swallowing errors by catching non-specific exceptions in application code"*. Here the key difference lies in the "do not" versus "avoid". A search on the Internet indicates that *"Don't catch Exception"* is a kind of urban legend, as witnessed by the wiki page of Ward Cunningham's website[3].

Previous Empirical Knowledge Cabral and Marques [4] give the percentage of generic catch blocks (`catch(Exception e)`) for 4 kinds of systems (libraries, server applications, server and stand alone applications) in 2 different programming languages (.Net and Java). According to their experiments, in .Net, the generic exception type is the most common caught exception for all 4 kinds of systems. It even accounts for more than 50% of all catch blocks in standalone applications. In their dataset of Java applications, the generic exception types (`Exception` and `Throwable`) are ranked #2 in the top caught exception list.

Our Empirical Results In our dataset all 32 libraries catch at least once `Exception`. In addition, 25/32 catch at least once `Throwable`. They are no lonely catch blocks written by novice developers and uncaught by a code review or the continuous checking system: In Tomcat, they are 475 catch blocks (out of 2,348, 20%) catching `Exception`. Tomcat is not an exception, there are 13 libraries for which we found more than 100 generic catch blocks.

Discussion Many developers write generic catch blocks. Qualifying this claim by our dataset, a stronger argument is that many respectful developers of high-quality open-source projects write generic catch blocks. The unqualified *"Don't catch exceptions"* is very much challenged. If one trusts so many expert developers, one accepts that catching generic exceptions is sometimes a good design principle. The question that arises is: when is it good to catch the most generic type of exceptions? Over the course of our experiments, we analyzed

[3] `http://c2.com/cgi/wiki?DontCatchExceptions`

many such generic catch blocks. We discovered two main reasons to have catch generic exceptions.

First, the resilience exception handling pattern involves catching the most generic exception type. We have presented and discussed this pattern above in Section 3.4. This pattern indeed involves a caught exception type that is generic.

Second, catching `Exception` is useful for ensuring a strong and interesting exception contract. An exception contract is contract on a code elements related to exceptions. For instance, in Java and related languages, checked exceptions relates to a contract on methods: *"no checked exceptions will ever be raised by calling this method"*.

We discovered that catching `Exception` is necessary to provide the following exception contract: *"if an exception is thrown, it will be of type X"*. We call this contract the *"all-in-one exception contract"* (for referring to the idea that all exception types are translated in one single exception type). This contract can be implemented as follows: 1) the complete method body is wrapped in a try-catch, 2) the catch block catches the most generic exception type, 3) the catch block throws an exception of type `X` (and optionally wraps the caught exception inside the thrown exception).

For instance, Listing 17 shows a method of the Apache Tomcat server which implements this contract. By construction, this method guarantees that if an normal exception is thrown[4], it will be of type `JasperException`.

Note that this contract is different from the checked exception contract. Adding "`throws JasperException`" says that "there exist situations where such a JasperException can be thrown". On the contrary, a well-implemented all-in-one exception contract ensures that only `JasperException` can be thrown.

To sum up, we empirically found two cases in which catching the most generic exception type makes sense and contradicts the bold statement that one should not catch `Exception`. To our knowledge, we are the first to unveil this part of exception handling design.

```
// in Tomcat's JspRuntimeLibrary.java
Object convert(String propertyName, String s,
    Class t, Class propertyEditorClass) throws
     JasperException {
  try {
    // 39 lines of code
  } catch (Exception ex) {
    throw new JasperException(ex);
  }
}
```

Listing 17: A Real World Example of the *All-in-one Exception Contract*. This method guarantees that if an exception is thrown, it will be of type `JasperException`

4 Conclusion

In this paper, we have studied the exception handling design of 32 Java libraries. The outcome of this study is twofold.

First, some analytical principles for exception design do not support the empirical validation: 1) practitioners violate the principle and 2) upon analysis, there are indeed very good use cases going against this principle. This is in particular the case for *"Empty Catch Blocks are Bad"* and *"Do not Catch Generic Exceptions"*.

Second, our study has reveals specific exception design patterns, such as resilience and the all-in-one exception contract. They all contribute to the body of knowledge on exception handling.

[4] We explained above that we discard the language-specific and conceptually irregular Java "Error"

References

[1] J. Atwood. Creating more exceptional exceptions. `http://www.codinghorror.com/blog/2004/10/creating-more-exceptional-exceptions.html`, 2004.

[2] J. Bloch. *Effective Java*. Addison-Wesley, 2001.

[3] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. Xquery 1.0: An xml query language. W3c working draft 04 april 2005, W3C, 2005. `http://www.w3.org/TR/xquery/`.

[4] B. Cabral and P. Marques. Exception handling: A field study in java and. net. In *ECOOP 2007–Object-Oriented Programming*, pages 151–175. Springer, 2007.

[5] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An XML-Based Lightweight C++ Fact Extractor. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 134–143, 2003.

[6] K. Cwalina and B. Abrams. *Framework Design Guidelines: Conventions, Idioms, and*

Patterns for Reuseable .NET Libraries. Microsoft .NET development series. Addison-Wesley, 2008.

[7] C. Fu and B. G. Ryder. Exception-chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In *Proceedings of the International Conference on Software Engineering*, 2007.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[9] B. Goetz. Java theory and practice: The exceptions debate. `http://www.ibm.com/developerworks/java/library/j-jtp05254/index.html`, 2004.

[10] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[12] J. R. Kiniry. Exceptions in java and eiffel: Two extremes in exception design and application. In *Advanced Topics in Exception Handling Techniques*, pages 288–300. Springer, 2006.

[13] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[14] T. McCune. Exception handling antipatterns. Online: http://today. java. net/pub/a/today/2006/04/06/exception-handling-antipatterns. html, accessed, 2006.

[15] T. McCune. Exception-handling antipatterns. `http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html`, 2006.

[16] Oracle. Unchecked exceptions: The controversy. `http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html`.

[17] A. Patel, A. Picard, E. Jhong, J. Hylton, M. Smart, and M. Shields. Google python style guidelines. `http://google-styleguide.googlecode.com/svn/trunk/pyguide.html`, Rev 2.54.

[18] D. Reimer and H. Srinivasan. Analyzing Exception Usage in Large Java Applications. In *Exception Handling in Object Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms*, pages 10–19, July 2003.

[19] P. Rovner. Extending modula-2 to build large, integrated systems. *Software, IEEE*, 3(6):46–57, 1986.

[20] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 42(5):559–583, May 2012.

[21] R. H. Wirfs-Brock. Toward Exception-Handling Best Practices and Patterns. *IEEE software*, 23(5):11–13, Sept. 2006.