A Catalogue of Java Exception Handling Bad Smells and Refactorings[1]

ROBERTA COELHO, Federal University of Rio Grande do Norte
JONATHAN ROCHA, Federal University of Rio Grande do Norte
HUGO MELO, Federal University of Rio Grande do Norte

Software is made by humans for human use and, for that reason, it is bound to fail. As language designers accepted failure as an inevitable factor, mechanisms had to be created to deal with it. Java was designed with an elaborate built-in exception handling mechanism which allowed programmers to anticipate failures and prepare the application to deal with them from a high level point of view. However, the exception handling code designed to make a system more robust often works the other way around and become a burden programmers have to cope with.

Some guidelines on how to better cope with the exception handling code have been proposed, papers have been written on this topic and tools have been built, nevertheless, such pieces of information are spread and structured in different ways. This paper aims to collect such guidelines on good and bad practices from different sources and compile it as a catalogue of *bad smells* and associated *refactorings* as a way to help developers to improve the exception handling code of Java programs.

Additional Key Words and Phrases: Exception Handling, Java, Patterns.

1. INTRODUCTION

Exception handling mechanisms are present in a variety of contemporary programming languages with the intention to offer a sophisticated way of improving robustness, nevertheless, when these mechanisms are not properly implemented they eventually snowball to become an intricate source of bugs (Barbosa et al. 2014) (Ebert et al. 2015) (Yuan et al. 2014) (Coelho et al. 2015)(Coelho et al. 2008), nourishing the one thing they were created to neutralize.

In order to help developers manage exception handling in a more effective way, many tools and guidelines were created, but those are spread across many sources in a scattered fashion. As identified by Wirfs-Brock 2016, there is a need for a pattern catalogue that structures these pieces of information on exception handling good and bad practices, especially when dealing with Java.

In this work we present a catalogue of exception handling bad smells and associated refactorings. We have compiled a set of practices taken from Bloch 2017, Wirfs-Brock 2016, Adamson 2015, Gosling et al. 2000 and also from static analysis tools, also known as "linters": Robusta (Robusta 2018), SpotBugs (SpotBugs 2018), SonarLint (SonarLint 2018) and PMD (PMD 2018). In this work we structured the good and bad practices collected from these sources in a refactoring catalogue format. Some of the bad smells and refactorings presented here represent known practices (e.g., avoid general handlers) others have not been widespread yet and were found in few sources (i.e., found by a single static analysis tool).

2. BASIC CONCEPTS

This section presents some basic concepts concerning the exception handling mechanism embedded in the Java language that support the understanding of the patterns presented next.

Exception Types. In Java, exceptions are represented according to a class hierarchy, on which every exception is an instance of the Throwable class, and can be of three kinds: checked exceptions (extends Exception), runtime exceptions (extends RuntimeException) and errors (extends Error) (Coelho et al. 2015). Checked exceptions received their name because they must be declared on the method's exception interface (i.e., the list of exceptions that a method might raise during its execution) and the compiler statically checks if appropriate handlers are provided within the system. Both runtime exceptions and errors

[1]Author's address:Universidade Federal do Rio Grande do Norte, Campus Universitário Lagoa Nova, CEP 59078-970, Caixa postal 1524, Natal/RN - Brazil

are also known as “unchecked exceptions”, as they do not need to be specified on the method exception interface and do not trigger any compile time checking. By convention, instances of Error represent unrecoverable conditions which usually result from failures detected by the Java Virtual Machine due to resource limitations, such as OutOfMemoryError. Normally these cannot be handled inside the application. Instances of RuntimeException are implicitly thrown by Java runtime environment when a program violates the semantic constraints of the Java programming language (e.g., out-of-bounds array index, divide-by-zero error, null pointer references). Some programming languages react to such errors by immediately terminating the program, while other languages, such as C++, let the program continue its execution in some situations such as the out-of-bounds array index. According to the Java Specification (Gosling et al. 2000) programs are not expected to handle such runtime exceptions signaled by the runtime environment.

User-defined exceptions can either be checked or unchecked, by extending either Exception or RuntimeException. There is a long-lasting debate about the pros and cons of both approaches (Jenkov 2014) (The Java tutorials 2017) (Phamn 2011), and even though the Java community hasn’t reached a consensus, some guidelines are presented in several of the patterns presented later.

Exception Propagation. In Java, once an exception is thrown, the runtime environment looks for the nearest enclosing exception handler (Java’s try-catch block), and unwinds the execution stack if necessary. This search for the handler on the invocation stack aims at increasing software reusability, since the invoker of an operation can handle the exception in a wider context (Miller and Tripathi 1997).
A common way of propagating exceptions in Java programs is through exception wrapping (also called chaining): one exception is caught and wrapped in another which is then thrown instead. Exception propagation can be easily noticed on exception stack traces messages, which are returned by the Java Virtual Machine whenever an exceptions propagates to the topmost layer of the system and is not handled properly. For simplicity, in this paper we will refer to “exception stack trace” as just “stack trace”.

3. METHODOLOGY

In order to collect the bad smells and refactorings that compose the present catalogue we performed a literature review as well as an investigation of existing static analysis tools. Exceptions bad smells, patterns and recommendations were then collected from these sources: PMD 2018, SonarLint 2018, SpotBugs 2018, Robusta 2018, Bloch 2017, Wirst-Brock 2006 and Adamson 2015. Overall we could collect 74 good and bad practices related to the development of exception handling code in Java (see APPENDIX A). Removing the duplicated practices (i.e., practices that were found on different sources which had the same purpose), we came up with 23 distinct exception handling bad smells and associated refactorings presented in Table 1.

Table 1 Bad Smells and Associated Refactorings.

Sm. ID	Bad Smell	Ref. ID	Refactoring
1	Handling coding errors	1	Let Programming Errors Flow
		2	Protect Entry Points
2	Unprotected main	2	Protect Entry Points
3	Redundant Exceptions	3	Reuse Standard Exceptions
4	Undocumented Runtime Exception	4	Document all exceptions thrown
5	Throwing Raw Exception	5	Provide context along with the exception
6	Exception Swallowing	6	Provide Appropriate Handling
		7	Let the Exception Flow

7	Unnecessary Exception Classes	8	Specific Exceptions for Specific Handling
8	Inflated Exception Interface	9	Exception Wrapping
9	Signaler Based Exception Naming	10	Problem Based Exception Naming
10	Throwing Generic Exception	11	Throwing Specific Exception
11	Handling Generic Exception	12	Use Specific Handlers
12	Destructive Wrapping/Logging	13	Cause-Preserving Wrapping/Logging
13	Throw from Within Finally	14	Use Try with Resources
		15	Add Try-catch Block Inside Finally Block
14	Relying on getCause()	16	Use alternative ways to access the Exception cause
		5	Provide context along with the exception
15	Mutable Exception	17	Exception classes should be immutable
16	Rely on "instanceof" in catch blocks	12	Use Specific Handlers
17	Exceptions Signaled from Entry Points	2	Protect Entry Points
18	Catches with the Same Handling Actions	18	Use a Multi-Catch Block
19	Nesting Try-catch Blocks More than Twice	19	Extract Method
20	Throwing null pointer exception	20	Use Specific Exception
		21	Return Default Object
		22	Include null checks
		23	Use Optional Object
21	Misnamed Class as Exception	24	Rename Class
22	Forgotten Exception	25	Throw Every Exception Created
23	Dead Catch Block	26	Remove Catch Block

3.1 Catalogue Structure

Each bad smell is presented within the sections: (i) the problem - reflecting upon why that is a bad smell or elucidating about it's context; (ii) a code example - showing code excerpts to illustrate the problematic scenario; (iii) the source - which shows from which paper, book, site or tool it was taken from. One or more refactorings can be associated to solve the bad smell, hence, each refactoring should contain: (i) a solution to the problems previously shown on the bad smell; and (ii) code excerpts to illustrate the solution.

4. EXCEPTION HANDLING BAD SMELLS

This section presents a list of exception handling bad smells and suggested refactorings. Although all the examples are written in Java, some of problems and solutions presented here are also applied to other languages that have embedded exception handling constructs.

Bad Smell 1: Handling Coding Errors

Problem	A set of JVM runtime exceptions such as NullPointerExceptions are usually caused by programming errors, and are often impossible to recover from, so it is not prudent to capture and handle them, because this will simply hide the true problem and hamper debug.

Code Example	void foo() {
	  ...
	  try {
	  someTrickyMethodWithProgrammingBugs();
	} catch (NullPointerException e) {
	  ...
	  }
	  ...
	}

Source	PMD Toward Exception-Handling Best Practices and Patterns, Rebecca J. Wirfs-Brock
Refactoring 1: Let Programming Errors Flow	
Solution	NullPointerException should not be captured and handled, only avoided by writing correct code.

Code Example	void foo() {
	  ...
	  someTrickyMethodWithProgrammingBugs();
	  ...
	}

Refactoring 2: Protect Entry Points	
Solution	Program entry points (such as service methods on Servlets, Android Activities life cycle methods, run methods on Threads and the main method) are typically the last place where an exception may be handled before the application boundary. In this sense, signaling exceptions from the entry-point cause the application crash. All exceptions should be handled on the entry points. The handling action can be for instance: logging the error and showing an error message to the user or requiring another input from the user.

Code Example	public void doGet(HttpServletRequest request, HttpServletResponse response)
	  throws IOException, ServletException {
	  String ip = request.getRemoteAddr();
	  try{
	  InetAddress addr = InetAddress.getByName(ip);
	  //...
	  }
	  catch(Exception e){
	  //...
	  }
	}

Bad Smell 2: Unprotected Main

Problem	Surrounding the main method with a giant try block followed by a catch block that captures the class Exception is a good practice that guarantees that the user will receive at least some message if a completely unexpected error occurs.

Code Example	public static void main(String[] args) {
	  ...
	}

Source	Robusta
Refactoring 2: Protect Entry Points (already described on ***Bad Smell 1***)	

Bad smell 3 : Redundant Exceptions	
Problem	Redundant exception are custom exceptions that signal problems already covered by Java's standard exceptions. Having to understand and handle too many custom exceptions can hurt code readability and maintainability.
Code Example	public void method() throws InvalidArgumentException
Source	Effective Java: Programming Language Guide, Joshua Bloch
Refactoring 3: Reuse Standard Exceptions	
Solution	It is always best to work with what the Java community is already familiar with: Java's standard exceptions. Adopt the use of standard Java exceptions as a rule and only create new exceptions if they are truly needed.
Code Example	public void method() throws IllegalArgumentException

Bad Smell 4: Undocumented Runtime Exception	
Problem	Not documenting all exceptions that a public method can throw can cause many issues to clients, for they won't be prepared to handle these exception if they arise. Java has its own way to prevent this lack of documentation, which is forcing checked exceptions to be explicitly signaled in the method signature, but adopting the practice of using multiple checked exceptions is also a serious problem that only passes the responsibility of a code smell to the client.

Code Example	public void someMethod() {
	  ...
	  throw new SomeRuntimeException();
	  ...
	  throw new AnotherRuntimeException();
	  ...
	}

Source	Effective Java: Programming Language Guide, Joshua Bloch

Refactoring 4: Document all exceptions thrown	
Solution	All exceptions should be mentioned either in the method signature or the Javadoc. Although Java allows the developer to declare runtime exception on the method signature, this information is only used as documentation since the compiler does not check for exception handlers (as it does for checked exceptions).

Code Example	@exception SomeRuntimeException is thrown if error X occurs.
	@exception AnotherRuntimeException is thrown if error Y occurs.
	public void someMethod() {
	...
	throw new SomeRuntimeException();
	...
	throw new AnotherRuntimeException();
	...
	}
	
	or
	
	public void someMethod() throws SomeRuntimeException, AnotherRuntimeException {
	...
	throw new SomeRuntimeException();
	...
	throw new AnotherRuntimeException();
	...
	}

Bad Smell 5: Throwing Raw Exception	
Problem	A raw exception would be any exception that lacks useful information about its error context.

Code Example	public void someMethod() {
	...
	throw new SomeException();
	...
	}

Source	Effective Java: Programming Language Guide, Joshua Bloch, Toward Exception-Handling Best Practices and Patterns, Rebecca J. Wirfs-Brock
Refactoring 05: Provide Context Along With the Exception	
Solution	Sometimes the exception's name or message is enough to inform about the faulty context. In some scenarios, developers tend to recur to exception getCause() to get a more specific information - leading to a fragile code. It is more prudent to pass information that might help the failure to be handled properly as arguments to the exception constructor. Usually, relevant information can include variable values, iterator's positions, state flags and so on.
Code Example	public void someMethod() { ...

	throw new SomeException(currentState, relevantVariable, revelevantIterator);
	...
	}

Bad Smell 06: Exception Swallowing	
Problem	Ignoring an exception is like turning off a fire alarm and pretending nothing went wrong. But with an exception it is even worse, for the problem may be kept hidden inside the system and cause many other failures. Exception handling is a skill all Java programmers must develop in order to guarantee their systems will be able to recover gracefully from errors and expected user mistakes.

Code Example	try {
	...
	} catch (SomeException e) {
	}

Source	PMD SpotBugs Robusta Effective Java: Programming Language Guide, Joshua Bloch Exception-Handling Antipatterns Blog, Chris Adamson
Refactoring 6: Provide an Appropriate Handling	
Solution	The obvious solution for this problem is to properly handle the exception, or, if that is not possible, rethrow the exception or inform the user about the error.

Code Example	try {
	...
	} catch (SomeException e) {
	dealWithException();
	}
	
	or
	
	try {
	...
	} catch (SomeException e) {
	throw e;
	}
	
	or
	
	try {
	...
	} catch (SomeException e) {
	System.out.print(e.getMessage());
	}

Refactoring 7: Let the Exception Flow	

Solution	If no appropriate handling can be given to the exception, the method should let the exception flow until it reaches the entry points or places where an appropriate handling may be given.

Code Example	public void methodA( ) throws SomeException{
	
	methodB( ); // throws SomeException but I do not know how to handle it
	
	}

Bad Smell 7: Unnecessary Exception Classes	
Problem	Exceptions are classes like any other Java class, and for this reason they should comply with OO principles, meaning that each exception should have its own responsibility and not overlap others. The more distinct exceptions classes there are, the more complex exception handling will be and the burden when maintaining the code will expand accordingly.

Code Example	try {
	return parse(text);
	}
	catch(IllegalCharacterOnTextError | InvalidSyntaxError | EmptyTextError e) {
	// Handles all text related exceptions the same way
	...
	}

Source	Toward Exception-Handling Best Practices and Patterns, Rebecca J. Wirfs-Brock
Refactoring 8: Specific Exceptions for Specific Handling	
Solution	A new exception class should not be created if it's expected handling routine is a copy and paste from the routine that handles other existing exceptions. If the only difference between two or more exceptions classes is the message they display or the values they carry, they are essentially redundant.

Code Example	try {
	return parse(text);
	}
	catch(InvalidTextError e) {
	// Uses only one exception to signal that the text is not parsable
	...
	}

Bad Smell 8: Inflated Exception Interfaces	
Problem	When a given method throws several checked exceptions it leads the developer to use a generic handler, or swallow all exceptions.
Code Example	public void businessMethod A throws IllegalCharacterOnTextError, InvalidSyntaxError, EmptyTextError, IOException {

	... }
Source	Toward Exception-Handling Best Practices and Patterns, Rebecca J. Wirfs-Brock
Refactoring 9: Exception Wrapping	
Solution	Create a single exception that represent all conditions that can go wrong on the method and make this exception wrap the previously signaled ones.

Code Example	public void businessMethod A throws BusinessException {
	try{
	...
	} catch(IllegalCharacterOnTextError | InvalidSyntaxError | EmptyTextError e) {
	throw new BusinessException (e);
	...
	}

Bad Smell 9: Signaler Based Exception Naming	
Problem	Naming exceptions after what structure threw them is a thoughtless decision that ignores the layered architecture most well-built software follow and burdens the user of the method or class with the responsibility of reading more documentation to understand the actual failure.
Code Example	throw new DatabaseException();
Source	Toward Exception-Handling Best Practices and Patterns, Rebecca J. Wirfs-Brock
Refactoring 10: Problem Based Exception Naming	
Solution	Naming an exception after the error it represents rather than the class which raised it greatly helps the handling process, as the name itself will work as a description of the problem. Always keep in mind that the more descriptive and auto explanatory your code is, the easier will be to maintain, expand and understand it.
Code Example	throw new ConnectionFailedDBException();

Bad Smell 10: Throwing Generic Exception	
Problem	Declaring that a method throws a generic Exception creates a huge problem to the users of the method because they won't be able to tell what caused that failure and how to handle it effectively.

Code Example	public void foo() throws Exception {
	...
	}

Source	PMD
Refactoring 11: Throwing specific exception	
Solution	Declare non-generic checked exceptions in the method signature and make use of exception classes that inherit from RuntimeException.

Code Example	public class EmptyStackException extends RuntimeException {
	  public EmptyStackException() {
	  super();
	  }
	  public EmptyStackException(String s) {
	  super(s);
	  }
	  public EmptyStackException(String s, Throwable throwable) {
	  super(s, throwable);
	  }
	  public EmptyStackException(Throwable throwable) {
	  super(throwable);
	  }
	}

Bad Smell 11: Handling Generic Exception	
Problem	Catching the Throwable or Exception classes is a tempting move when programming under pressure, nevertheless, even though dealing with all checked exceptions at the same time is pretty convenient, it can lead to serious problems such as swallowing relevant exceptions and implementing inefficient handling mechanisms. In fact, having a generic catch as the standard strategy to avoid dealing with Java's obligations sabotages the whole purpose of the exception handling mechanism. Likewise, the Error class should not be caught either, for it indicates internal system problems that aren't in the responsibility of the software.

Code Example	catch (Throwable e) {
	  ...
	}

Source	PMD SonarLint Exception-Handling Antipatterns Blog, Chris Adamson
Refactoring 12: Use Specific Handlers	
Solution	Avoid implementing catch blocks that capture Throwable, Exception, and Error, except when the method signature carelessly throws these exception classes.

Code Example	catch (FirstException e) {
	  ...
	}
	catch (SecondException e) {
	  ...
	}

	catch (ThirdException e) {
	  ...
	}

Bad Smell 12: Destructive Wrapping/Logging	
Problem	When an exception is rethrown or logged, the original exception object might be discarded by an inattentive programmer, nevertheless this object might contain crucial information, like the stacktrace or context in which the failure happened.

Code Example	try { ... }
	catch (CauseException e) { LOGGER.info("context"); }
	
	try { ... }
	catch (CauseException e) { LOGGER.info(e.getMessage()); }
	
	try { ... }
	catch (CauseException e) {
	  throw new MyException("context");
	}

Source	PMD SonarLint Exception-Handling Antipatterns Blog, Chris Adamson
Refactoring 13: Cause-Preserving Wrapping/Logging	
Solution	When logging or rethrowing an exception always pass the original exception as a parameter to guarantee the context information won't be lost.

Code Example	try { ... }
	catch (CauseException e) { LOGGER.info("context", e); }
	
	try { ... }
	catch (CauseException e) { LOGGER.info(e); }
	
	try { ... }
	catch (CauseException e) {
	  throw new MyException("context", e);
	}

Bad Smell 13: Throw from Within Finally	
Problem	Throwing an exception from a finally block can create a huge problem by overwriting any previous exception that might have been propagated, swallowing that exception and hiding the real problem that caused a failure. If the exception in thrown directly from the finally block (and not from a method inside the finally that may sometimes throw an exception), programmers must be aware that finally blocks are always executed, meaning that this exception will always be thrown no matter what happened in the try block.

Code Example	finally {
	...
	methodThatMightThrowException();
	
	}
	
	or
	
	finally {
	...
	buffer.close(); // it throws IOException
	}

Source	PMD Exception-Handling Antipatterns Blog, Chris Adamson
Refactoring 14: Use Try with Resources	
Solution	If the method that may throw an exception from within the finally block implements the AutoCloseable interface, make use of the try-with-resources.

Code Example	try (BufferedReader buffer =
	new BufferedReader(new FileReader(path))) {
	
	return buffer.readLine();
	
	}

Refactoring 15: Add Try-catch Block Inside Finally Block	
Solution	If the exception is signaled by a resource closure, and the resource does not implement the AutoCloseable interface, or the exception is signaled within finally due to a different reason add a try/catch block inside the finally block. This might be ungraceful, but if the method must be called inside the finally block, it is better than applying a deeper refactoring.

Code Example	finally {
	…
	try {
	methodThatMightThrowException();
	} catch (anException e) {
	…
	}
	}

Bad Smell 14: Relying on getCause()

Problem	Relying on the exception getCause() method makes exception handling fragile, because if the exception is encapsulated again (during a maintenance task, for instance) the previously implemented handling code may fail, since the code will handle a wrapped exception instead of the original cause.

Code Example	catch(DirectException e){
	  if(e.getCause() instanceof CauseException){
	  ...
	  }
	}

Source	Exception-Handling Antipatterns Blog, Chris Adamson
Refactoring 16: Use alternative ways to access the Exception cause	
Solution	The best solution to this problem is not to rely on get cause. Instead the exception to be handled should provide additional context information so that the exception could be handled without relying on its cause. This solution is described on ***Bad Smell 05.*** However, if the getCause() of an exception must be checked in order to give it proper handling, then all causes must be decapsulated until the root cause is reached. Another option would be to verify if one of the causes of the exception has a given type using ExceptionUtils from 'Apache Commons Lang'.

Code Example	catch(DirectException e){
	  if(ExceptionUtils.getRootCause(e) instanceof CauseException){
	  ...
	  }
	}
	
	or
	
	catch(DirectException e) {
	  if(ExceptionUtils.hasCause(e, CauseException.class)){
	  ...
	  }
	}

Refactoring: Provide context along with the exception (described on ***Bad Smell 05***)	

Bad smell 15: Mutable Exception	
Problem	Exceptions must be a snapshot of the system by the time the error occurred. Mutable exceptions may corrupt this snapshot, rendering it useless to debugging and handling the error.

Code Example	public void throwsMutableException() {
	  // ...
	  throw New InvalidOperationException(value1, value2);
	}
	public void muteException() {

	try {
	throwsMutableException();
	}
	catch (InvalidOperationException e) {
	e.setValue1(anotherValue);
	throw e;
	}
	}

Source	SonarLint
Refactoring 17: Exception classes should be immutable	
Solution	All exceptions should be immutable. To guarantee immutability, all attributes should be declared as final. If an exception has too many parameters, these parameters should be encapsulated in an object.

Code Example	public class InvalidOperationException extends ...{
	final Value value1, value2;
	public InvalidOperationException (Value v1, Value v2)
	{...}
	}

Bad Smell 16: Rely on "instanceof" in Catch Blocks	
Problem	The use of "instanceof" is usually a lazy workaround to avoid catching specific exception classes individually with several catch blocks. This bad practice may cause problems if an unexpected instance of an exception is propagated through that catch block, which won't be prepared to handle it properly. It also hurts readability and code maintainability, for new exception may arise as the system evolves, but the exception handling architecture will always be dependent on this rudimentary hard coded checking.

Code Example	try {
	...
	} catch (SomeBaseException e) {
	if(e instanceof MyException) { ... }
	if(e instanceof OtherException{ ... }
	}

Source	Sonarlint
Refactoring 12: Use Specific Handlers	
Solution	Handle exception individually or group them accordingly when the same recovery routine is to be run for more than one exception. If there is a standard handling mechanism, do it as a generic catch after all specific catch blocks.
Code Example	catch (MyException e) { … } catch (OtherException e) { … }

	catch (SomeBaseException e) {
	… //default handler
	}

Bad smell 17: Exceptions Signaled from Entry Points	
Problem	Program entry points (such as service methods on Servlets, Android Activities life cycle methods, run methods on Threads and the main method) are typically the last place where an exception may be handled before the application boundary. In this sense, signaling exceptions from the entry-point cause the application crash.

Code Example	public void doGet(HttpServletRequest request, HttpServletResponse response)
	throws IOException, ServletException
	{
	String ip = request.getRemoteAddr();
	InetAddress a = InetAddress.getByName(ip); //throws UnknownHostException
	//...
	}

Source	Sonarlint
Refactoring 2: Protect Entry Points (already described on ***Bad Smell 1***)	

Bad Smell 18: Catches With the Same Handling Action	
Problem	Having multiple catches that implement the same handling actions tend to overextend the code and hinder maintainability.

Code Example	catch (IOException e) {
	doCleanup();
	logger.log(e);
	}
	catch (SQLException e) { // Noncompliant
	doCleanup();
	logger.log(e);
	}
	catch (TimeoutException e) { // Compliant; block contents are different
	doCleanup();
	throw e;
	}

Source	Sonarlint
Refactoring 18: Use a Multi-catch Block	
Solution	Since Java 7 it is possible to combine multiple exceptions classes in a same catch block, but programmers have not fully adopted this convenient tool, probably out of unfamiliarity or simply because sometimes coping and pasting a code block seems easier than agglutinating exceptions in a single line.

	If multiple exceptions happen to be handled the same way, combine them in one single catch block. If these exceptions happen to always be handled the same way, it would be prudent to check if both exception are truly needed. If they are not, refactor the code.

Code Example	catch (IOException|SQLException e) {
	  doCleanup();
	  logger.log(e);
	} catch (TimeoutException e) {
	  doCleanup();
	  throw e;
	}

Bad Smell 19: Nesting Try-catch Blocks More than Twice	
Problem	Nested try-catches hamper code readability and maintainability. It is acceptable to have it nested once, but be careful when the nesting goes deeper for it can lead to loss of context information from the original exception, which might be overwritten by a new one.

Code Example	public void method1() {
	  try {
	  ...
	  try {
	  ...
	  try {
	  ...
	  } catch (YetAnotherException e){
	  ...
	  }
	  } catch (AnotherException e){
	  ...
	  }
	  } catch (AnException e) {
	  ...
	  }
	}

Source	Sonarlint Robusta
Refactoring 19: Extract Method	
Solution	Refactoring the code and extracting the inner try blocks are the best way to fix this code smell and prevent confusing debugging sessions.

Code Example	public void method1() {
	  try {
	  ...
	  method2();
	
	  } catch (AnException e) {
	  ...

	}
	}
	
	public void method2() {
	try {
	...
	} catch (AnotherException e){
	...
	}
	}

Bad smell 20: Throwing null pointer exception	
Problem	A NullPointerException warns about a programming mistake and, therefore, should never be thrown by your own code. This bad practice may hamper readability and end up forcing other programmers to debug looking for a problem that never truly existed.

Code Example	void foo(Integer value) {
	if(value == null){
	throw new NullPointerException("...");
	}
	//...
	}

Source	PMD
Refactoring 20: Use a Specific Exception	
Solution	If a generic unchecked exception is needed in your code, choose one of the many others Java offers, such as the IllegalArgumentException.

Code Example	String foo(Integer value) {
	if(value == null){
	throw new IllegalArgumentException("...");
	}
	//...
	}

Refactoring 20: Return Default Object	
Solution	Instead of returning null your code could return a default object.

Code Example	Object foo(Integer value) {
	...
	if(result == null){
	return new DefaultValue();
	}
	return result;
	}

Refactoring 22: Include Null Checks	

Solution	Null checks should be included every time you receive data that crosses the method boundary.

Code Example	String void foo(Integer value) {
	  if(value == null){
	  //perform the proposed computation
	  }
	  //...
	}

Refactoring 23: Use Optional Objects	
Solution	Since Java 8 there is an option to make use of an Optional object to encapsulate other objects that may be nullable. By using this tool the method which uses the said object is forced to check whether the nullable object is null or not. By choosing to return an Optional object the programmer will be directly avoiding the Java Exception handling mechanism, so this decision must be dealt with care.

Code Example	Optional foo(Integer value) {
	  if(value == null){
	  return optionalObject;
	  }
	  //...
	}

Bad Smell 21: Class Misnamed as Exception	
Problem	Naming non-exception classes with a string that ends with "Exception" can cause readability problems, what can hurt software maintainability.

Code Example	public class HandlesException {
	  ...
	}

Source	Spotbugs
Refactoring 24: Rename Class	
Solution	Refactoring the code and renaming classes appropriately is the only option.

Code Example	public class ErrorHandler {
	  ...
	}

Bad Smell 22: Forgotten Exception	
Problem	There is no good reason to create an object and not make use of it, similarly, there is no reason to create an exception and not throw it.

Code Example	if (x < 0) {
	  //exception is created but never thrown
	  new IllegalArgumentException("x must be nonnegative");
	}

Source	SpotBugs
Refactoring 25: Throw Every Exception Created	
Solution	Create exceptions as close as possible to the line of code in which the exception is actually thrown.

Code Example	if (x < 0) {
	  throw new IllegalArgumentException("x must be nonnegative");
	}

Bad Smell 23: Dead Catch Block	
Problem	In Java, the compiler checks whether or not a given catch clause is reachable by at least one checked exception. Dead Catch Block is any catch that is unreachable by any *checked exception*. A dead catch block is one that tries to catch exceptions that are not thrown by any line of code in its associated try block.

Code Example	try {
	  methodWithoutCheckedException();
	} catch (IOException e) {
	  ...
	}

Source	Spotbugs
Refactoring 26: Remove Catch Block	
Solution	Simply don't use a try/catch in that case.
Code Example	methodWithoutCheckedException();

5. RELATED WORKS

5.1 Guidelines to Exception Handling Design and Implementation

The work closest to ours is the work of Haase 2002. Haase proposes a pattern language composed by eleven Java patterns to support the high level design of the exception handling behaviour of a system. This pattern language aims at bringing the exception handling concerns to the early phases of software development and proposes a set of patterns to be applied when designing the exception handling behaviour of a system. Although there can be found some similarities between few patterns of the the catalogue presented here and the pattern language proposed by Haase, each patterns set focus on a different level of abstraction. While the pattern language (comprising 11 patterns) proposed by Haase tackles on early phases of software development such as architecture and high-level design phases, this catalogue (comprised by 40 patterns)

aims at helping developers during the low-level design and implementation of the exception handling code. Hence, our work complements the work of Haase.

Chen et al. 2009 present six bad smells on the exception handling code and propose a set of refactorings to make the exception handling code more reliable. All bad smells pointed in this work can be automatically detected by Robusta tool and were included in the present catalogue.

5.2 Bug Classification

This catalogue presents a set of problems on the exception handling code that can lead to a failure. Some of the problems presented here are not a bug in itself (i.e., nested try blocks) but can lead to one. Hence, the catalogues aims at preventing bugs on the exception handling code by point to possible bugs and bug hazards in the code. Other works (Barbosa et al. 2014) (Ebert et al. 2015) (Yuan et al. 2014) (Coelho et al. 2015) performed investigations on the other way around: they inspected crash reports and connected some real failures with bugs and bug hazards in the exception handling code. These works motivated the need for the present catalogue as they linked real failures to a set of problems on the exception handling code discussed in this catalogue.

6. CONCLUDING REMARKS

In this work we have presented a catalogue on exception handling patterns to support Java development. This catalogue was created by compiling, explaining and expanding a set of practices taken from (Bloch 2017), (Wirfs-Brock 2016), (Adamson 2015), (Gosling et al. 2000) and also from static analysis tools, also known as "linters": Robusta (Robusta 2018), SpotBugs (SpotBugs 2018), SonarLint (SonarLint 2018) and PMD (PMD 2018).

This work is an afford in the direction of structuring the knowledge on Java exception handling as a set of patterns. The aim of this catalogue is not to be complete, other patterns and good design and implementation solutions may exist on other sources that were not included in this catalogue. However, through this catalogue we aim at supporting the development of more robust Java systems, helping developers to detect and prevent bugs when developing the exception handling code. Although most of the patterns presented here focuses on Java exception handling constructs some of the ideas presented here may also be useful when developing in other languages (e.g., C#, C++) which contains similar exception handling built in constructs.

APPENDIX A

The Table 2 below present the set of practices found in each information source used in this study.

Table 2: Exception handling practices described or detected by each source.

Practice source	ID	Practice	**Similar**
Book: Effective Java	1	Use exceptions only for exceptional conditions	17, 52
	2	Use checked exceptions for recoverable conditions and runtime exceptions for programming errors	-
	3	Avoid unnecessary use of checked exceptions	21, 34
	4	Favor the use of standard exceptions	-
	5	Throw exceptions appropriate to the abstraction	13
	6	Document all exceptions thrown by each method	-
	7	Include failure-capture information in detail messages	-

	8	Strive for failure atomicity	-
	9	Don’t ignore exceptions	25, 45, 49, 61
Paper: Toward Exception-Handling Best Practices and Patterns	10	Don’t try to handle coding errors	53
	11	Avoid declaring lots of exception classes.	-
	12	Name an exception after what went wrong, not who raised it.	-
	13	Recast lower-level exceptions to higher-level ones whenever you raise an abstraction level.	5
	14	Provide context along with an exception.	-
	15	Handle exceptions as close to the problem as you can	-
	16	Assign exception-handling responsibilities to objects that can make decisions	-
	17	Use exceptions only to signal emergencies.	1, 52
	18	Don’t repeatedly rethrow the same exception.	55
Exception-Handling Antipatterns Blog	19	Log and Throw	33, 44
	20	Throwing Exception	51
	21	Throwing the Kitchen Sink	3, 34
	22	Catching Exception.	37, 50, 59
	23	Destructive Wrapping	36, 60
	24	Log and Return Null	(?)
	25	Catch and Ignore	9, 45, 49, 61
	26	Throw from Within Finally	32, 46, 57
	27	Multi-Line Log Messages	-
	28	Unsupported Operation Returning Null	-
	29	Ignoring InterruptedException	-
	30	Avoid Relying on getCause()	-
SONARLINT	31	Jump statements should not occur in "finally" blocks	-
	32	Exceptions should not be thrown in finally blocks	26, 46, 57
	33	Using printStackTrace .	19, 44
	34	Public methods should throw at most one checked exception	3, 21
	35	Exception classes should be immutable	(?)
	36	Exception handlers should preserve the original exceptions	23, 60
	37	Throwable and Error should not be caught	22, 50, 59
	38	Exception types should not be tested using "instanceof" in catch blocks	-

	39	Exceptions should not be thrown from servlet methods	-
	40	Try-with-resources should be used	43, 65, 66, 71
	41	Catches should be combined	-
	42	Try-catch blocks should not be nested	47
ROBUSTA	43	Careless Clean-up	40, 65, 66, 71
	44	Dummy Handler	19, 33
	45	Empty catch block	9, 25, 49, 61
	46	Exception thrown in finally block	26, 32, 57
	47	Nested try statement	42
	48	Unprotected main	-
PMD	49	Catch vazio	9, 25, 45, 61
	50	AvoidCatchingThrowable	22, 37, 59
	51	SignatureDeclareThrowsException	20
	52	ExceptionAsFlowControl	1, 17
	53	AvoidCatchingNPE	10
	54	AvoidThrowingNullPointerException	-
	55	Rethrow the same exception	18
	56	DoNotExtendJavaLangError	-
	57	Throw Exception on Finally	26, 32, 46
	58	Throw new Instance of the same exception	-
	59	AvoidCatchingGenericException	22, 37, 50
	60	AvoidLosingExceptionInformation	23, 36
SPOTBUGS	61	DE: Method might ignore exception	9, 25, 45, 49
	62	IMSE: Dubious catching of IllegalMonitorStateException	-
	63	It: Iterator next() method can't throw NoSuchElementException	-
	64	Nm: Class is not derived from an Exception, even though it is named as such	-
	65	ODR: Method may fail to close database resource on exception	40, 43, 66, 71
	66	OS: Method may fail to close stream on exception	40, 43, 65, 71
	67	RV: Method ignores exceptional return value	-
	68	NP: Null pointer dereference in method on exception path	-
	69	NP: Value is null and guaranteed to be dereferenced on exception path	-
	70	NP: Possible null pointer dereference in method on exception path	-
	71	OBL: Method may fail to clean up stream or resource on checked exception	40, 43, 65, 66
	72	RV: Exception created and dropped rather than thrown	-

	73	UL: Method does not release lock on all exception paths	-
	74	REC: Exception is caught when Exception is not thrown	-

REFERENCES

Barbosa, E. A., Garcia, A., & Barbosa, S. D. J. (2014, September). *Categorizing faults in exception handling: A study of open source projects*. In Software Engineering (SBES), 2014 Brazilian Symposium on (pp. 11-20). IEEE.

Bloch, J. (2017). *Effective java*. Addison-Wesley Professional.

Chen, C. T., Cheng, Y. C., Hsieh, C. Y., & Wu, I. L. (2009). *Exception handling refactorings: Directed by goals and driven by bug fixing*. Journal of Systems and Software, 82(2), 333-345.

Chris Adamson. (2015). *Exception-Handling Antipatterns Blog*. Retrieved May, 2018 from https://community.oracle.com/docs/DOC-983543.

Coelho, R., Almeida, L., Gousios, G., & van Deursen, A. (2015, May). *Unveiling exception handling bug hazards in Android based on GitHub and Google code issues*. In Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on (pp. 134-145). IEEE, 2015.

Coelho, R., Rashid, A., von Staa, A., Noble, J., Kulesza, U., & Lucena, C. (2008, October). A catalogue of bug patterns for exception handling in aspect-oriented programs. In Proceedings of the 15th Conference on Pattern Languages of Programs (p. 23). ACM, 2008.

Ebert, F., Castor, F., & Serebrenik, A. (2015). *An exploratory study on exception handling bugs in Java programs*. Journal of Systems and Software, 106, 82-101.

Gosling, J., Joy, B., & Steele, G. (2000). *The Java language specification*. Addison-Wesley Professional.

Haase, A. (2002, July). *Java Idioms-Exception Handling*. In EuroPLoP (pp. 41-70).

Jenkov, J. (2014). *Checked or unchecked exceptions?* Retrieved May, 2018 from http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html.

Miller, R., & Tripathi, A. (1997, June). *Issues with exception handling in object-oriented systems*. In European Conference on Object-Oriented Programming (pp. 85-103). Springer, Berlin, Heidelberg.

Phamn, Thang. (2011). *Java: checked vs unchecked exception explanation*. Retrieved May, 2018 from http://stackoverflow.com/questions/6115896/java-checked-vs-unchecked-exception-explanation.

PMD. (2018). Retrieved May, 2018 from https://pmd.github.io/.

Robusta. (2018). Retrieved May, 2018 from https://marketplace.eclipse.org/content/robusta-eclipse-plugin.

SonarLint. (2018). Retrieved May, 2018 from https://www.sonarlint.org/.

SpotBugs. (2018). Retrieved May, 2018 from https://spotbugs.github.io/.

The Java tutorials (2017). *Unchecked exceptions: The controversy*. Retrieved May, 2018 from http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html.

Wirfs-Brock, R. J. (2006). *Toward exception-handling best practices and patterns*. IEEE software, 23(5), 11-13.

Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., ... & Stumm, M. (2014, October). *Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems*. In OSDI (pp. 249-265).