Trends on Empty Exception Handlers for Java Open Source Libraries

Ana Filipa Nogueira
CISUC, University of Coimbra
Portugal
afnog@dei.uc.pt

José C. B. Ribeiro
Polytechnic Institute of Leiria
Portugal
jose.ribeiro@ipleiria.pt

Mário A. Zenha-Rela
CISUC, University of Coimbra
Portugal
mzrela@dei.uc.pt

***Abstract*—Exception-handling structures provide a means to recover from unexpected or undesired flows that occur during software execution, allowing the developer to put the program in a valid state. Still, the application of proper exception-handling strategies is at the bottom of priorities for a great number of developers. Studies have already discussed this subject pinpointing that, frequently, the implementation of exception-handling mechanisms is enforced by compilers. As a consequence, several anti-patterns about Exception-handling are already identified in literature.**

In this study, we have picked several releases from different Java programs and we investigated one of the most well-known anti-patterns: the empty catch handlers. We have analysed how the empty handlers evolved through several releases of a software product. We have observed some common approaches in terms of empty catches' evolution. For instance, often an empty catch is transformed into a empty catch with a comment. Moreover, for the majority of the programs, the percentage of empty handlers has decreased when comparing the first and last releases. Future work includes the automation of the analysis allowing the inclusion of data collected from other software artefacts: test suites and data from issue tracking systems.

***Index Terms*—Exception-Handling, Open-Source, Software Evolution**

I. Introduction

Modern programming languages as Java or C# provide built-in exception-handling (EH) mechanisms allowing developers to handle exceptional occurrences and to customize the adequate recovery model. These mechanisms may include simple tasks, such as the acknowledgement of the exception and consequent program termination, or more complex recovery actions that will restore the execution's state into a valid one. Nevertheless, programmers and designers overlook the importance of proper exception-handling mechanisms.

Some reasons why developers tend to neglect EH have been examined in literature. For instance, Shah, Görg and Harrold [1] interviewed a few developers and noticed several common issues: i) EH is viewed as a debugging tool instead of a recovery action mechanism, ii) people tend to use the "ignore-for-now approach" because they believe it is not a high-priority task and iii) if the compiler does not enforce the usage of exceptions, people don't use them. On the other hand, literature also discusses the unsuitability and unnecessary complexity of current exception models [2], [3] and the impact of the specificities of the Object-Oriented paradigm [4].

Wirfs-Brock [5] highlights the importance of exception-handling on the overall quality of a software product implementation. The author supports that as soon as developers and designers start to agree and implement EH practices, the software will become "easy to comprehend, evolve and refactor". In other words, good EH practices contribute to more maintainable software. The author's contribution includes the description of several proven guidelines, techniques and patterns that can help improving the quality of EH structure. Additionally, the existence of anti-patterns is also acknowledged. A wiki discussing patterns and anti-patterns for Exceptions is available online [6]. No patterns are "oficially" established, but it is a useful tool to collect advice about good and bad practices. Anti-patterns are also discussed by McCune [7]; for instance, he recommends that exceptions shouldn't be suppressed or ignored.

Our main research theme is the *maintainability* of software, in particular Java software. We are interested on collecting information about the reasons that make a software product difficult to maintain. The existence and wide use of so many EH anti-patterns, make us believe that the impact of poor EH choices is bigger than one may think. The focus of this paper is on the anti-pattern: the empty catch handler. Following the discussion on the community [6], the majority of arguments are in favor of avoiding empty handlers pointing out that they are Code Smells. However, others claim that it is a valid solution if one really wants to ignore all failures.

This paper is organised as follows: Section II overviews some of the related work. Section III presents the set of research objectives and the methodological approach. Section IV presents and discusses the observations collected in this study. Finally, Section V concludes this paper and pinpoints the future work.

II. Related Work

In this section we present a few studies conducted on EH structures and their conclusions regarding this topic.

Cabral and Marques [8] performed a study on Java and .NET applications to find out how programmers use EH mechanisms; as part of it, they analysed the code inside catch blocks and finally blocks and categorised the actions observed. The authors found that *Empty handlers* is the prevailing

978-1-5090-5501-2/17/$31.00 © 2017 IEEE

type of handler in Server applications and the second most implemented in Libraries and Standalone applications.

Ebert, Castor and Serebrenik [9] analysed the perception of developers on EH and the reports submitted on a issue tracking system in order to define a classification for EH-bugs. In terms of *Empty catch block*, developers mentioned empty handlers as a cause of bugs they had solved in the past, but in the bugs' reports, the percentage of empty-related bugs is small. Interestingly, empty handlers are commonly used as part of bug-fixes, also including the fixes for EH-related bugs [9]. The authors pointed out the need of creating tools that would assist developers in the presence of exceptions (e.g., a recommendation system).

A set of scenarios where empty handlers are considered valid is discussed in [10]; the report claims that "when the global application state is correct, stopping the exception propagation is appropriate and this is what empty blocks do".

Common assumptions on studies about exception-handling include: EH structures require more attention from developers, EH code is hard to test and many EH-bugs could be prevented through proper testing. Our work intends to complement existing research by adding testability on the analysis of the evolution of EH structures. Hence, while analysing the evolution of a method we will consider test-related metrics (in addition to others, e.g., anti-patterns presence), in order to assess if the method is more or less maintainable from one release to another.

III. Study Methodology

This section describes the study methodology providing an overview of the questions that guided the research, the tools used and the case studies.

A. Research Questions

The main goal of this study was to characterize the evolution of empty catch handlers in a set of open source libraries, which are commonly used by the Java Community. In the scope of current research, an empty catch handler (EC) – which does not contain any executable statement – may fall in one of two categories: i) *empty handlers no comments (ECNC)* – the ones completely empty and ii) *empty handlers with comments (ECWC)* – the ones that only contain comments. These handlers obscure the existence of exceptional occurrences as they neither provide any kind of error notification nor recovery action. The characterization includes the observation of how handlers evolve from one release to another and which exception types are commonly defined in those EC.

- *RQ1: How do empty catch handlers evolve through software releases of a software product?*
- *RQ2: Which are the most common type of exceptions found in the type definition of all EC?*

Even though handlers enclosing logging/printing statements can be viewed as EC (as no recovery is done), its analysis will not be addressed here.

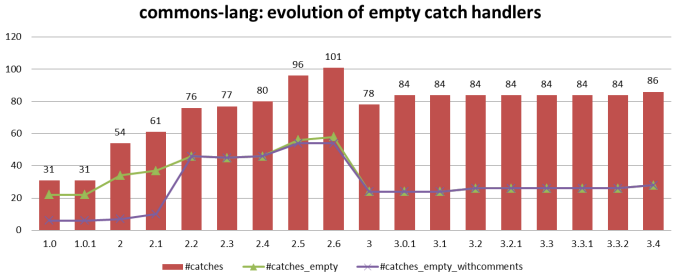

Fig. 1. Evolution of the values for metrics for project *commons-lang*: number of handlers (bars), number of empty handlers (line with triangles) and number of empty handlers with comments (line with crosses).

B. Collecting Exception-Handling Data

The collection of data was done in two steps: collection of data from source code and extraction of desired metrics. The data is available online at [11]. For the first step we used the tool srcML[1] to generate a set of XML files containing information about the source code. This allowed us to have a single file containing information about all the classes in a release; also, the XML content provides a more readable means to analyse the source code and extract the desired data. The second part of the process consisted on parsing each XML file (1 file per release) generating a set of measurement files that contains information about the methods in each release. This task was performed through a custom application. Java files from packages that refer to examples, samples or other non-core code (e.g., classes on sandbox packages) were not considered. Test code was included but it falls outside the scope of this paper; future work will address the test code for EH constructs.

C. Case Studies

A set of 33 projects from the Apache Software Foundation[2] was analysed, counting a total of 285 releases. Table I displays the number of releases per project. We have collected several metrics for characterizing the EH structures on each project, however the focus of this paper is on the evolution of empty *catch handlers*. Due space constraints, it is not possible to present all measurements for all projects, thus we only present an example for project *commons-lang*; data for remaining projects is available at [11]. Figure 1 shows the evolution of the values for metrics: number of catch handlers, the number of empty handler and the number of empty handlers with comments. In the first releases (from 1.0 to 2.1), it is possible to observe a difference between the two lines EC and ECWC: there is a discrepancy showing that several handlers are completely empty. This may be a reflection on poor planing of EH mechanisms evidenced in first releases. Between releases 2.2 and 2.4 all EC are ECWC, which reflects a tendency to eliminate completely empty handlers. In releases 2.5 and 2.6 we can observe a slight separation of the lines,

[1]http://www.srcml.org/
[2]https://www.apache.org/

TABLE I
NUMBER OF RELEASES

Project	# releases	Project	# releases	Project	# releases
bcel	1	commons-dbutils	7	commons-jxpath	4
bsf	2	commons-digester	17	commons-lang	18
commons-beanutils	18	commons-discovery	4	commons-launcher	2
commons-chain	3	commons-el	1	commons-logging	10
commons-cli	5	commons-email	8	commons-net	16
commons-codec	10	commons-exec	5	commons-pool	21
commons-collections	12	commons-fileupload	10	commons-primitives	1
commons-compress	13	commons-httpclient	19	commons-proxy	1
commons-configuration	15	commons-io	13	commons-scxml	4
commons-daemon	16	commons-jcs	2	commons-validator	11
commons-dbcp	11	commons-jxel	3	commons-vfs	2

which is levelled again in release 3.0. Since release 3.0, all empty handlers contain a comment, showing an attempt to provide some insight about the exceptions being caught.

IV. RESULTS AND DISCUSSION

In this section, we address the research questions by discussing the collected data.

A. Research Questions

1) RQ1: How do empty catch handlers evolve through software releases of a software product?: To answer this research question we have picked empty catch handlers detected in each release of each software product and analysed how they evolved in the following releases. The extraction of the methods with EC that would be analysed was done in a automated manner, but the visualization of its evolution through the several releases was validated manually – which may represent a thread to validity. The main modifications observed are discussed next. Figure 2 shows the differences between the first and last releases: it shows how %EC and %ECWC are distributed in the several projects. It is possible to highlight two main observations:

- *the number of ECs decreased*: in the first release, 75% of the projects had a percentage of EC greater than 29.36% (the worst case was 70.97% of EC), whereas for last releases we can observe a decrease on the percentage of EC: 75% of projects have 17.06% of EC and the worst case is a project with 41.61%. In first releases: 25% of the projects had between 0% and 3.70% of EC, while in last releases: 25% of the projects contained 0% to 4.06% EC.
- *the number of ECWC followed EC's trend*: last releases' values are in-line with the numbers for EC, meaning that the majority of empty handlers are ECWC.

The next paragraphs describe a few types of modifications observed in our analysis.

An empty handler ECNC is converted to an ECWC. We have observed a great number of empty catches changed from being completely empty to containing comments. Typically the comment was something like "//ignore" or/and "// NOPMD". The majority of the new comments observed was not as meaningful as they could be, i.e., they did not provide a reason why the exception was being swallowed or ignored. The usage of comment "// NOPMD" was an unexpected observation; this comment works as a suppression rule defined by the code analysis tool PMD[3]. When this comment is applied, the line in question is ignored, thus the tool will not report a problem in that line. Besides, we have observed some comments that reflect uncertainty of the developers. Listing 1 exemplifies a comment that is neither helpful in terms of the method's maintainability/understandability nor pinpoints the actions that should be done in future. This example was found in project *common-beanutils*[4] from releases 1.8.0-BETA to 1.9.2.

TABLE II
SUMMARY RANGE

	First		Last	
	%EC	**%ECWC**	**%EC**	**%ECWC**
Minimum	0.00	0.00	0.00	0.00
25th Percentile	3.70	0	4.06	1.41
Median	12.30	6.45	9.52	8.00
75th Percentile	29.36	15.79	18.18	16.18
Maximum	70.97	58.62	41.67	39.51

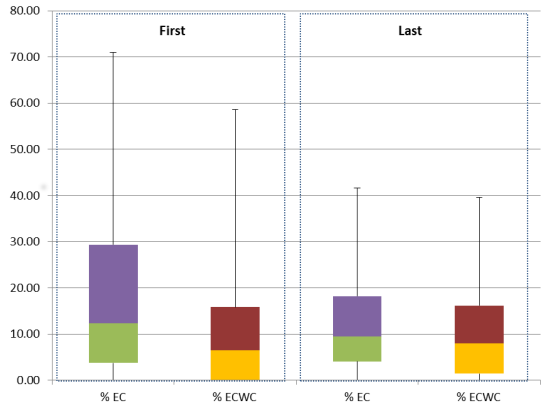

Fig. 2. Percentage of all empty handlers (%EC) and empty handlers with comments (%ECWC) for the first and last releases.

[3]http://pmd.sourceforge.net/snapshot/usage/suppressing.html
[4]http://commons.apache.org/proper/commons-beanutils/

Listing 1. Handler with comment that reveals uncertainty

```
catch (SecurityException se) {
    /* Swallow SecurityException * TODO: Why? */
}
```

An EC is converted to an non-empty handler implementing a recovery action. Many of the EC analysed were modified to provide specific recovery actions, including: i) call to a utility method that handles the exception, ii) reset of a variable, iii) log of the exception caught, iv) re-throw of an exception, v) termination of the flow or vi) a call to the *printStackTrace* method. The first four aforementioned actions can be considered as improvements on the understandability of the handlers, as concrete actions are performed when an exceptional scenario occurs. The remaining two actions are on the opposite side as they implement well-known anti-patterns [7] [6]: the usage of the *return* instruction makes the EH structure work as a decision structure, and the call of *printStackTrace* will print the stack several times making it difficult to read the logs.

An ECWC for a super-type converted to *n* ECWC specialized handlers. A good practice observed was the specialization of exceptions, i.e., instead of catching the super-type *Exception* (Listing 2) programmers evolved it and defined four specific handlers (Listing 3). Even though each handler has the same recovery action (no real action), this improves the maintainability of the code as one is aware of the different issues that may occur. Its noticeable that in addition to the original comment, the warning suppressor NOPMD comment was included. This was an example observed in project *commons-discovery*[5].

Listing 2. Example of handler for super-type in release 3.2.2 (before)

```
catch (Exception ex) { // ignore }
```

Listing 3. Example of handlers in release 4.0-alpha1 (after refactoring)

```
} catch (final RuntimeException e) {
    // NOPMD // ignore
} catch (final NoSuchMethodException e) {
    // NOPMD // ignore
} catch (final IllegalAccessException e) {
    // NOPMD // ignore
} catch (final InvocationTargetException e) {
    // NOPMD // ignore
}
```

2) RQ2: Which are the most common type of exceptions found in the type definition of all EC?: In a first analysis we have picked all empty handlers for all releases of the 33 projects analysed and accounted the percentage of handlers per exception type (Figure 3). The #1 position goes to type *Exception* which is a super-type for EH classes. This means that at least 22% of the analysed handlers are swallowing checked and unchecked exceptions, and may not provide specific recovery actions as the structure handles all *Exception*'s sub-classes in the same way. Nevertheless, this may be an informed decision and for some cases, the catch may be in accordance with the exception raised in the code, i.e., the handler is too broad because the exception thrown is also too broad. However, it may pose some difficulties in maintaining the software if a new exception type enters the scope. The types in top 10 include exceptions caught while: manipulating input/output resources – *IOException* (13%); parsing data – *NumberFormatException* (13%) and *ParseException* (4%); using *reflection* – *NoSuchMethodException* (7%) and *IllegalAccessException* (3%); working with SQL – *SQLException* (6%); handling runtime issues – *SecurityException* (6%) and working with Threads – *InterruptedException* (5%).

[5]https://commons.apache.org/proper/commons-discovery

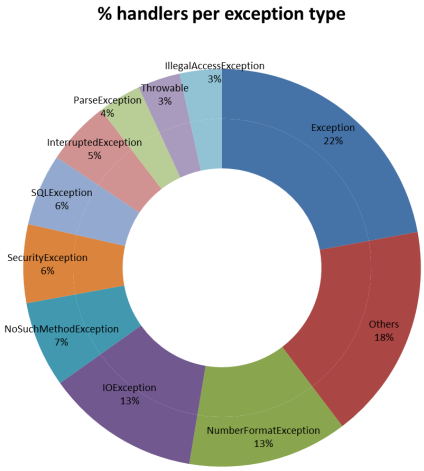

Fig. 3. Percentage of EC per exception type.

Also, a "bad practice" accounts 3%: EC for type *Throwable*. This does not help any recovery action as the problem is unknown: the root of the problem is lost. Moreover, it makes the code more difficult to read and to comprehend. Finally, the group of exception types that by themselves do not feature the top 10 (18%) groups a set of 36 distinct types. The majority of exception types (99.57%) are *pre-defined* (from Java Runtime Environment), while 0.43% are *user-defined* types (3rd party libraries or classes defined in the project). 64.24% of types are checked, while 10.12% are unchecked. Finally, 25.46% refer to super-types (*Exception* and *Throwable*) and 0.18% of types extend from *Error*.

When analysing ECWC handlers, some comments may not be as useful as they could be, but at least we can speculate about conscious decision of not implementing any kind of recovery or logging action. In turn, ECNC handlers may be more difficult to understand and maintain as there is no pinpointing direction about a possible recovery action. Data about exception types defined in ECNC handlers are not presented here, but similarly, the top 10 exception types for ECNC are in-line with values presented in 3. The *Others* set of exception types is the larger slice, it accounts a total of 19% of the ECNC. The distinct type that counts the higher percentage is the *NumberFormatException* (15%). Typically, developers tend to initialize a variable with a default value and if the parse action fails an *NumberFormatException* is raised and default value is used. Nevertheless, to proper diagnose the failure while calling the parse method, at least the log of the exception raised would be considered useful. *Exception* and *IOException*

follow with 14% each, meaning that issues with I/O resources are completely swallowed and undisclosed as well as any "Exception" instance. The remaining percentages are distributed as follows: 8% for *NoSuchMethodException*, 7% for *SecurityException*, 6% for *SQLException* and *InterrupedException*, 4% for *ParseException* and *IllegalAccessException*, and finally, 3% for super-type *Throwable*. Even though there are common exception types in the EC, we need to highlight that those depend on the type of project under analysis. Nevertheless, a great number of exception types are (non project specific) super-types, which is a well known anti-pattern ([6], [7]).

B. Threats to Validity

Internal threats to validity include the usage of an external tool to generate a XML representation of the source code, as well as the creation of a custom application to parse the XML content for each release. Both of these tools may present a threat to validity as they may enclose their own issues. Nevertheless, more tests will be done in order to track possible issues. The manual analysis of handlers' evolution may be considered prone-to-error, and that is why future developments will incorporate its automation. An *external* threat to validity is the difficulty to generalize the observations to other environments. We have observed commonalities among the several projects in terms of: decrease of the total number of EH, levelling of EHWC with EC (majority of EC are EHWC) and similarities of exception types on handlers definition. Still, these observations cannot be generalized when characterizing the evolution of Java open source libraries. They can, however, pinpoint directions and good/bad practices applicable on handlers' evolution.

V. Conclusions and Future Work

In this study we analysed the evolution of one well-known anti-pattern: empty handlers. Globally, it was possible to observe a positive trend on the evolution of empty handlers. A considerable number of empty handlers have been re-factored to an improved version, at least according to our vision. Even so, there are several handlers with no recovery or logging action that keep hiding exceptional occurrences. Depending on the code implementation, some can argue that empty handlers can be acceptable, if the code related is able to continue its proper execution when the exception occurs or the exception can be ignored. Additionally, many of those handlers are unable to distinguish between different problems as they catch broad exception types.

The extraction of data and measurements was done through software, but the analysis of the handlers in the several releases was done manually. This proved to be a time-consuming task that needs to be enhanced through automation. What's more, the scope of our research is expected to include data from test suites, but without automation this is impractical. Hence, following experiments are expected to include data from test suites. To complement the data, we intended to analyse the issue tracking systems in order to understand the impact of bugs on the evolution of the EH structures and the importance of unit-tests on the process of discovering those bugs. Our research intends to provide easy and informed access to how exception-handling structures are evolving. Currently, we are collecting information about the types of EH issues reported in issue tracking system, and we are also taking into account the tests reported/implemented in the scope of each issue. The main goal is to start a knowledge base that can be used by developers in order to access the quality of the current release and comparing it with previous releases to ensure that quality of EH is improving (and consequently the overall quality). Therefore, it is expected to complement existing working by considering: testability of EH structures, EH anti-patterns, bugs/issues reported and practices that originate to EH-related bugs. The motivation for this research includes:

- to provide the developer with the perception on how the exceptional cases are being handled when a re-factoring is made on a method. For some of the methods that were re-factored, it was easy to some extent to check new methods called and check if the exception handling structure was the same. For others was not as simple, a new automatic approach would be interesting to evaluate the quality of the "refactorings".
- through the tests, provide a better scope of the exceptions raised while invoking the code, and point out alternatives to catching super-types: *Exception* and *Throwable*.

This could help answering the following questions: How many handlers evolved from empty to better recovery actions? How many handlers worsen their maintainability by implementing some anti-patterns? How complex are current handlers?

References

[1] H. Shah, C. Görg, and M. J. Harrold, "Why do developers neglect exception handling?" in *Proc. of the 4th Int. Workshop on Exception Handling*, ser. WEH '08. New York, NY, USA: ACM, 2008, pp. 62–68.

[2] B. Cabral and P. Marques, "A case for automatic exception handling," in *Proc. of the 2008 23rd IEEE/ACM Int. Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 403–406.

[3] A. F. Garcia, C. M. F. Rubira, A. Romanovsky, and J. Xu, "A comparative study of exception handling mechanisms for building dependable object-oriented software," 2001.

[4] R. Miller and A. Tripathi, "Issues with exception handling in object-oriented systems," in *In Object-Oriented Programming, 11th European Conference (ECOOP*. Springer-Verlag, 1997, pp. 85–103.

[5] R. Wirfs-Brock, "Toward exception-handling best practices and patterns," *Software, IEEE*, vol. 23, no. 5, pp. 11–13, Sept 2006.

[6] C2.com, "Exception patterns," 2015. [Online]. Available: http://c2.com/cgi/wiki?ExceptionPatterns

[7] T. McCune, "Exception-handling antipatterns," 2006. [Online]. Available: https://community.oracle.com/docs/DOC-983543

[8] B. Cabral and P. Marques, "Exception handling: A field study in java and .net," in *Proc. of the 21st European Conference on Object-Oriented Programming*, ser. ECOOP'07, Berlin, Heidelberg, 2007, pp. 151–175.

[9] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *Journal of Systems and Software*, vol. 106, pp. 82–101, 2015.

[10] M. Monperrus, M. Germain De Montauzan, B. Cornu, R. Marvie, and R. Rouvoy, "Challenging Analytical Knowledge On Exception-Handling: An Empirical Study of 32 Java Software Packages," Laboratoire d'Informatique Fondamentale de Lille, Technical Report, 2014. [Online]. Available: https://hal.inria.fr/hal-01093908

[11] A. F. Nogueira, "Dataset - era@saner2017 - v1.1," Jan. 2017. [Online]. Available: http://dx.doi.org/10.5281/zenodo.234334