2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)

Exception Evolution in Long-lived Java Systems

Haidar Osman*, Andrei Chiş+, Claudio Corrodi*, Mohammad Ghafari*, and Oscar Nierstrasz*
*Software Composition Group, University of Bern, Switzerland
+Feenk GmbH, Switzerland

***Abstract*—Exception handling allows developers to deal with abnormal situations that disrupt the execution flow of a program. There are mainly three types of exceptions: standard exceptions provided by the programming language itself, custom exceptions defined by the project developers, and third-party exceptions defined in external libraries. We conjecture that there are multiple factors that affect the use of these exception types. We perform an empirical study on long-lived Java projects to investigate these factors. In particular, we analyze how developers rely on the different types of exceptions in throw statements and exception handlers. We confirm that the domain, the type, and the development phase of a project affect the exception handling patterns. We observe that applications have significantly more error handling code than libraries and they increasingly rely on custom exceptions. Also, projects that belong to different domains have different preferences of exception types. For instance, content management systems rely more on custom exceptions than standard exceptions whereas the opposite is true in parsing frameworks.**

***Keywords*-Exception handling; software evolution; empirical study**

I. Introduction

Robust software systems need to gracefully handle abnormal execution flows. Exception handling is a mechanism provided by many programming languages to support this goal. Given the widespread presence of exceptions in software projects [1], understanding how developers handle exceptions in practice can guide the design and improvement of current and future exception handling mechanisms. Previous studies approached this goal by analyzing corpora of software projects [1][2][3][4] and revealed that projects from different domains use exception handling differently, and that poor practices in writing exception handling code are widespread. For example, developers frequently write empty exception handlers [3], they catch and throw standard exceptions instead of specialized ones [3], or they just ignore exception handling until an error occurs [5]. Nevertheless, the aforementioned studies draw their findings only by looking at a particular version of a project, and it is unclear whether their findings hold throughout the lifetime of a project.

A recent pilot study on four Java projects revealed several phenomena in *the evolution of exception handling*, i.e., how developers handle exceptions throughout software development [6]. In that study we showed that the percentage of exception handling code remains constant during software development, and that there are several patterns in the evolution of exception handling. For instance, *(i)* the usage of exceptions defined by the Java language increases more than the usage of exceptions defined within software systems, and *(ii)* developers consistently encode the domain information into the standard Java exceptions as custom string error messages, instead of relying on custom exception classes. We conjecture that *the adoption of exception handling varies during the lifetime of software projects, and that there exist factors influencing it.*

In this paper we extend our previous study [6] by performing a broader empirical investigation to understand the evolution of exception handling in 90 Java projects, and to elicit factors that influence it. Each of these projects has been under development for more than five years. We extract, starting with the beginning of each project, 20 different commits at least 3 months apart, to cover a minimum of 5 years of active development. For each commit we identify all *(i)* exception handlers, *(ii)* defined exceptions and *(iii)* throw statements. We then classify exceptions into standard (defined by the Java language), custom (defined within the project), and third-party (neither custom nor standard) exceptions.

We formulate two main research questions to explore aspects that may affect the evolution of exception handling:

RQ1. How does the evolution of exception handling differ between libraries[1] *and applications?*
We approached this question by manually classifying the subject projects into these two categories and compared the results. We observe that exception usage does not change as frameworks evolve. Application developers, however, rely more on standard exceptions at the beginning of a project and, as the application evolves, increase the usage of custom and third-party exceptions. There also exist statistically significant differences between these two types of projects among several axes, with the number of exception handlers for `java.lang.Exception` persisting throughout development. We further notice a surprisingly stable percentage of exception handling code for both types of projects.

RQ2. Is there any difference in the evolution of exception handling between projects belonging to different domains?
We manually categorized 30 projects into 6 domains, namely compilers, content management systems, editors/viewers, web frameworks, testing frameworks, and parser libraries. We observed many differences in the evolution of exception handling between these domains, like the usage of `java.lang.Exception` and custom exceptions in catch blocks. Concretely, content management systems consistently have more exception handling code and throw more custom exceptions, as opposed to editors/viewers, which have less error-handling code and mainly use standard exceptions instead.

[1]In this paper we use "library" to refer to both libraries and frameworks.

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.21

Based on our findings, we acknowledge that the domain of a project has the biggest influence on the evolution of exception handling. We confirm previous findings regarding stable percentages of exception handling code, however, we only see a stable percentage across libraries and applications for thrown third-party exceptions and handlers for `java.lang.Exception`. Furthermore, applications use consistently more `java.lang.Exception` than libraries. Given that catching `java.lang.Exception` is considered an improper usage [7], we recommend quality assurance checks on applications to limit the spread of this behaviour.

During this empirical study, we also realized that to compare the usage of programming language features between different categories of software projects, one first needs to explore how the usage of a selected feature evolves over the lifetimes of the analyzed projects. For instance, by looking only at the 6th version of the analyzed projects in our study, we could conclude that there is a statistically significant difference between libraries and applications just regarding the usage of `java.lang.Exception` in exception handlers. However, for the 18th commit there are statistically significant differences in the number of caught third-party exceptions, custom and standard thrown exceptions.

The rest of the paper is organized as follows: Section II discusses the setup of our study. We then discuss the results in Section III and threats to validity in Section IV. We conclude the work and discuss future research directions in Section VI.

II. Empirical Study Setup

This section describes the criteria used to categorize projects, the construction of the dataset, and the metrics on which we build the study.

A. Categorization criteria

We categorize projects based on their type and their domain. These are relevant criteria also used in other empirical studies looking into how developers use exception handling. For example, Cabral and Marques [3] explored differences between *servers*, *server apps*, *libraries* and *applications*. Others studies covered usage and problems related to exceptions in libraries [8][4].

Based on their types, we classify projects into two categories:

- *Applications:* Projects that can be run without further development, such as databases, desktop applications, and others. It is not the main goal of these projects to provide an API for other clients to reuse.
- *Libraries:* Projects meant to be reused in other code. In this category we also include frameworks. Libraries are usually called directly from another project though a provided API, while frameworks are extended and dictate more closely the shape of the project.

An important threat to validity related to this criterion is the process by which we classify projects as libraries or applications. Currently the line between a library and an application is unclear for many projects. For example, applications can provide some type of API for creating plugins, while libraries can come with default clients that make it possible to run them as standalone applications. In Section II-B, we detail the steps we took to limit this threat to validity and in Section II-C the classification of projects into domains.

B. Subjects

To answer *RQ1* we need a dataset consisting of a large number of long-lived and actively-developed Java projects. To construct it we start with downloading a body of projects from GitHub[2], as many open-source projects (both libraries and applications) are hosted there. Using the GitHub search facility, we query for repositories that are *(i)* written in Java, *(ii)* created before 1 January 2012, *(iii)* are starred more than 10 times and have been forked at least 10 times, *(iv)* are greater than 1MB in size, and *(v)* have at least one commit since 1 July 2016. This gives us a set of 1003 non-trivial projects with an active lifetime of at least five years.

We then filter these projects with criteria that are not supported by the GitHub search facilities. First, we record the latest commit and step back in the commit history (on the main branch) until we arrive at a commit made at least 3 months prior to the previously recorded one. We record the commit and iterate until we cover the whole commit history. This produces, for each project, a list of commits at least 3 months apart. In some cases, we obtain a low number of commits. For example, consider a project that has existed for five years, but was only active (*i.e.*, has commits in the repository) in the first two months and the most recent two months. With our approach, we only record two commits that are roughly 5 years apart. On the other hand, large numbers of recorded commits indicate that the projects are more consistently active or have a longer lifetime. We retain projects with at least 20 recorded commits, a requirement that holds for 415 repositories (20 consecutive commits correspond to 5 years of development).

As we want to use this dataset to answer *RQ1*, we proceed to classify the projects obtained at this point. The classification into libraries and applications is done by three researchers, so that each project is classified by two researchers according to the definitions from Section II-A. Each researcher classifies each project as one of *(i)* library/framework, *(ii)* application, or *(iii)* unclear. After the researchers independently classify all projects, they discuss those where there was a conflict in the individual classification. We keep the projects where there is no initial disagreement and those where there is an initial disagreement but all three researchers agree after the discussion on a single type. Most of the disagreements were caused by incomplete or misleading project descriptions. This results in 276 repositories.

Many of the classified repositories contain either small projects or projects that did not change much during their development. To avoid polluting our dataset with such projects, we select only those that *(i)* consist of at least 30 000 lines of code (measured using cloc[3]), and *(ii)* grew by a factor of at

[2]https://github.com

[3]https://github.com/AlDanial/cloc

least 1.5 between the first commit and the last commit. This leaves 90 repositories of actively developed Java projects, out of which 45 are libraries and 45 are applications. We refer to these projects as *Dataset A*. A current limitation in this filtering step is that it excludes projects that decreased in size. To finalize the dataset, we retain only the earliest 20 commits for each project, taking a snapshot for each selected commit. For 19 projects, this covers the first 5 to 6.5 years of development, and for one project it covers the first 9 years.

C. Domain-specific investigation

To answer *RQ2* we need a dataset with projects from particular domains. To build this dataset we start from *Dataset A* and categorize projects based on their domain. The categorization is done individually, by the same three researchers as in Section II-B, so that each project is classified by two researchers. The classification is open ended: the researchers have to individually indicate what domain they think the project belongs to. Nonetheless, when comparing the results, we observed a high level of disagreement in what should be considered a domain. Hence, we selected only the six domains for which we saw the highest level of agreement: compilers, CMSs, editors/viewers, web frameworks, testing frameworks, parsing libraries.

For each domain, we select 5 representative projects. We start by selecting the projects in their respective domains from *Dataset A*. However, in many domains, there are fewer than 5 projects. We performed a manual search on GitHub for more projects by lowering the minimum age requirement of a project to at least 2 years. This choice was necessary because in several domains, we were unable to find 5 projects that are at least 5 years old. To build the final dataset we extract the first 8 commits following the same approach as for *Dataset A*. This corresponds to two years of development. We refer to this dataset as *Dataset B*. We summarize the projects in Table I, stating category and domain, short description, repository URL, and an indication of whether the project was present in *Dataset A*. The results of analyzing these domain-specific projects are presented in Section III-B.

D. Definitions and Computed Metrics

In this study, we classify exceptions as follows:

- *Java Standard Exception* is any exception defined in the JRE, *i.e.*, all the classes that extend `java.lang.Throwable`.[4]
- *Custom Exception* is any class that is defined within the project code base and extends any of the Java standard exceptions, *i.e.*, an exception class that is defined in the project repository.
- *Third-Party Exception* is any exception that belongs to neither the standard exceptions nor the custom exceptions, *i.e.*, an exception class that is defined in a library used by the project.

[4]https://docs.oracle.com/javase/8/docs/api/overview-tree.html

We use this classification, as our previous work showed a difference in how they are used within software projects [6]. Furthermore, the creation of custom exceptions indicates an explicit activity taken by developers to either provide more domain specific information about errors, or handle errors in specific ways.

To perform our study we extract the following metrics from each snapshot:

- The number of custom exceptions.
- The percentage of exception handlers with a custom exception as a parameter.
- The percentage of exception handlers with a standard Java exception as a parameter.
- The percentage of exception handlers with a third-party (library) exception as a parameter.
- The percentage of throw statements with a custom exception as a parameter.
- The percentage of throw statements with a standard Java exception as a parameter.
- The percentage of throw statements with a third-party (library) exception as a parameter.
- The percentage of throw statements with a standard Java exception initialized using a string value.
- LOC_{catch}: The LOC of all catch blocks.
- LOC: The size of the project in LOC.

To compute LOC for a snapshot we run *cloc* with the parameter `--include-lang=Java` and retain the number of code lines. Regarding LOC_{catch}, we computed this metric by summing only the lines of code within catch block. Hence, an empty catch block counts as zero lines of code.

III. Results

In this section we discuss our analyses of the two research questions.

A. Applications vs. Libraries

The projects from *Dataset A* are all mature projects that increased their size by at least a factor of 1.5. Figure 1 summarizes the sizes of all projects, for each snapshot. We clearly observe that for this dataset, applications are larger than libraries, and also increase in size more during software development, especially towards the end of the analyzed period.

To understand how exception handling evolves in libraries and applications, we study the trends of each of the calculated exception metrics in each of the categories. We employ the LOESS smoother (*i.e.*, LOcally wEighted Scatterplot Smoothing) at a 95% confidence interval to plot the trends. LOESS is a regression analysis tool that fits a smooth line of time-series data, such as ours, to help researchers understand and analyze trends visually. LOESS is suitable to our dataset because it is tolerant to outliers and it is non-parametric, *i.e.*, it does not require any assumptions about the distribution of the data. When two trends interfere, then we cannot draw any conclusions about the difference between them. Only when there is a gap between the trends, can we state that the trends

Type	Domain	Name	Description	GitHub URL
Applications	Compilers	Yeti	Implementation of a ML like functional language	mth/yeti
		Rascal	Implementation of the Rascal meta-programming language	usethesource/rascal
		WebDSL	Compiler for a domain-specific language for web applications	webdsl/webdsl
		Graal	Compiler and evaluator	graalvm/graal-core
		Ceylon	Compiler and evaluator	ceylon/ceylon
	CMSs	uPortal	Portal framework university and research communities	Jasig/uPortal
		OpenCms	General-purpose CMS	alkacon/opencms-core
		Apache JSPWiki	Wiki engine	apache/jspwiki
		XWiki	Generic wiki platform	xwiki/xwiki-platform
		dotCMS	General-purpose CMS	dotCMS/core
	Editors/viewers	**jCAE**	3D modeling tool	jeromerobert/jCAE
		josm	OpenStreetMap editor	openstreetmap/josm
		lilith	Logging and access event viewer	huxi/lilith
		Weasis	Medical imaging viewer	nroduit/Weasis
		ISAcreator	Editor for ISA-Tab files for creating life science experiments	ISA-tools/ISAcreator
Libraries	Web	Spring Framework	General-purpose application framework	spring-projects/spring-framework
		Grails	Groovy-based web framework	grails/grails-core
		Apache Wicket	Web framework	apache/wicket
		Project Wonder	Collection of frameworks and libraries	wocommunity/wonder
		Vaadin	Web framework	vaadin/framework
	Testing	JUnit 4	Traditional testing framework	junit-team/junit4
		Truth	Framework for fluent testing and improved readability	google/truth
		Mockito	Mocking library	mockito/mockito
		TestNG	Testing framework	cbeust/testng
		jbehave	Testing framework focusing on accessibility and behavioural specifications	jbehave/jbehave-core
	Parsers	Apache Xerces2	XML parsing library	apache/xerces2-j
		jsoup	HTML scraping and parsing library	jhy/jsoup
		JavaParser	Java 8 parser and transformation library	javaparser/javaparser
		ANTLR 4	Parser generator library	antlr/antlr4
		Apache James Mime4J	E-Mail message and MIME format parser	apache/james-mime4j

TABLE I: Selected projects with their types and domains. Bold names indicate projects that were already present in *Dataset A*. GitHub URLs are prefixed with `https://github.com/` and were accessed prior to February 2017.

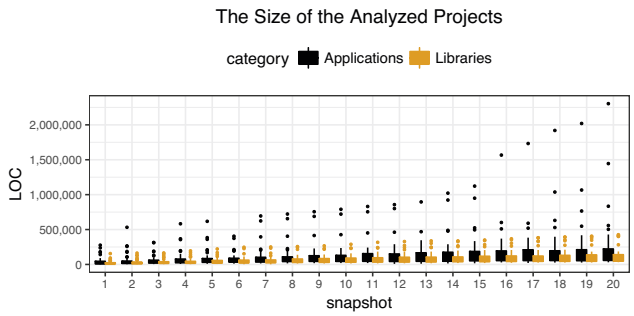


Fig. 1: Box-plot summary for the sizes of all projects from *Dataset A* per snapshot, categorized based on project type. Size is measured using the lines of code (LOC) metric.

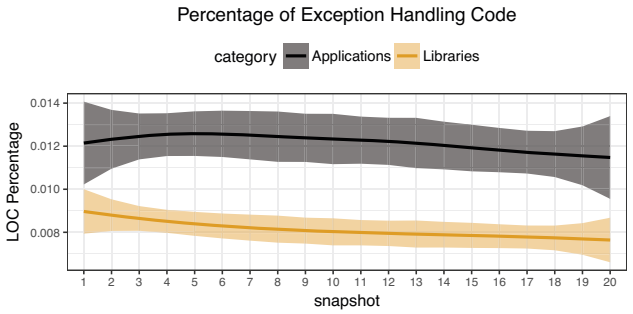


Fig. 2: Trends of the exception handling code showing a significant difference between libraries and applications

are statistically on different levels with a confidence interval of 95%.

To answer *RQ1* we first investigate the evolution of error handling code. For this, we study the evolution of LOC_{catch} relative to the overall size of the studied projects ($LOC_{catch} \div LOC$). We do not consider the `finally` block to be error handling code because the code it contains is executed whether there is an exception or not and has nothing to do to with the exceptions thrown in the `try` block. Figure 2 shows that there exists a statistically significant difference between libraries and applications throughout development. The percentage of exception handling code has only slight variations (1.15%–1.25% for applications, 0.75%–0.9% for libraries), indicating that the amount of error-handling code increases at a similar rate as the number of lines of code of a project. Next, we study the evolution of exception handlers and throw statements. We use, as a proxy for exception usage, the percentages of

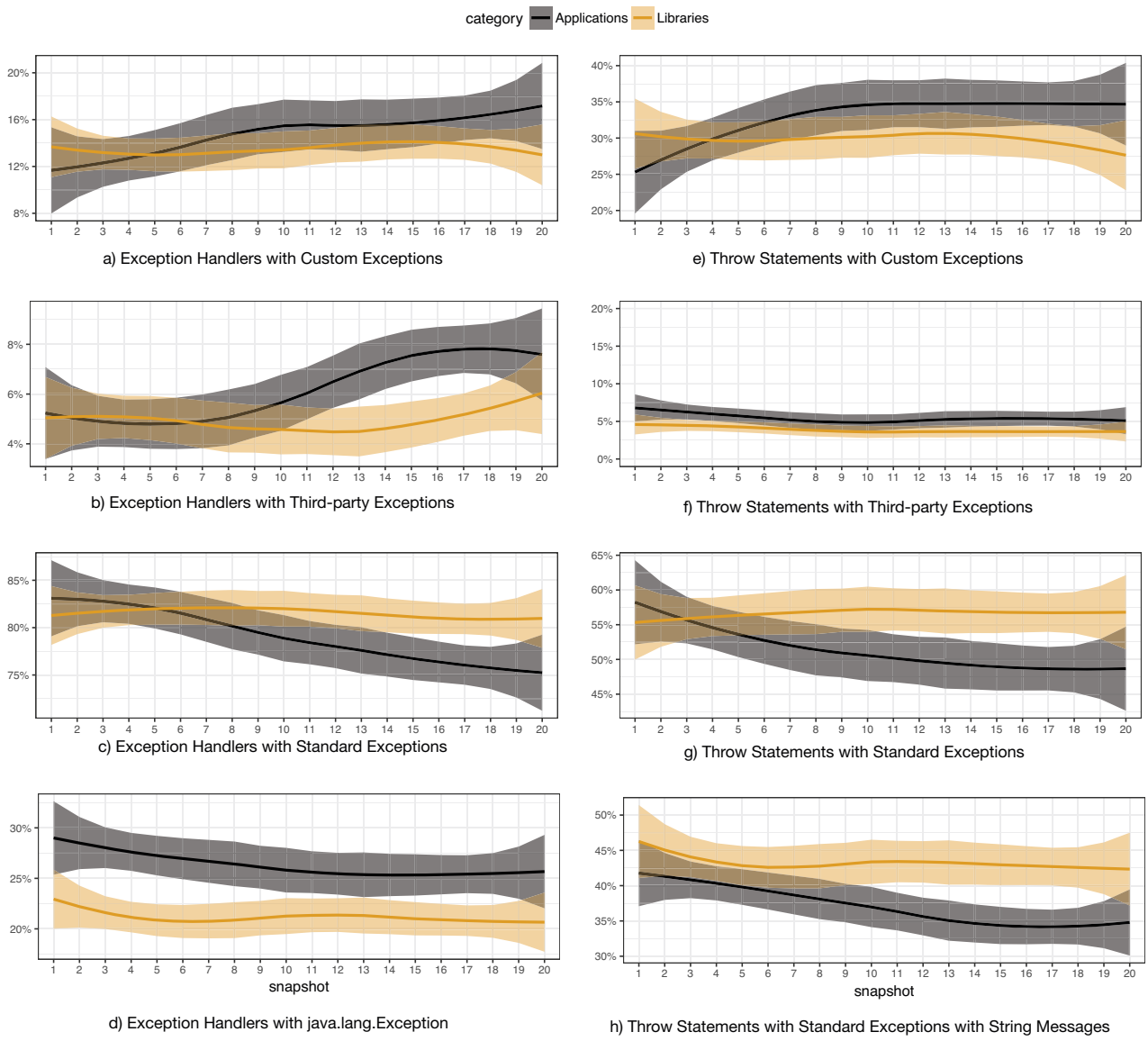


Fig. 3: Trends of exception usage in libraries and applications.

exception handlers and throw statements with different types of exceptions. Figure 3 shows the overall analysis. As a general observation, we note that, for both applications and libraries, developers rely the most on standard exceptions: between 72% and 88% of exceptions used in catch blocks and throw statements are standard exceptions throughout development. Custom exceptions come next, and third-party exceptions are used the least (under 9%).

Given that as projects mature, developers become more knowledgeable about the domain and the project, we expect to see a decrease in the use of standard exceptions conveying little domain-specific information. We observe a stable evolution the percentage of caught and thrown standard exceptions in libraries (Figure 3.c and Figure 3.g). However, we observe that as applications evolve, they rely less on standard exceptions in both catch blocks and throw statements. We also note that the percentage of caught third-party exceptions (Figure 3.b) increases during development in the case of applications and differs significantly from libraries through snapshots 13 to 19. This can be attributed to the fact that application dependencies on third-party software increase during development, resulting in an increasing third-party exception usage.

We see no statistically significant difference in the use of custom exceptions between the two types of projects (Figure 3.a and Figure 3.e). Libraries also have stable percentages when it comes to the use of custom exceptions during development. However, the trend in applications is to move more towards custom exceptions, especially in the first year of development.

We also see no change or difference in the percentages of throw statements with third-party exceptions (Figure 3.f). This indicates that even if application developers rely more on libraries during software development, they only catch third-party exceptions and rarely throw them.

Regarding the use of `java.lang.Exception`, we observe that a significant number of all exception handlers directly use this exception (Figure 3.d). We also notice a statistically significant difference during development: applications use `java.lang.Exception` more in exception handlers than libraries. The corresponding percentages show only minor variations.

Custom exceptions offer one way to encode domain specific information. An alternative is to rely on standard exceptions and initialize them with custom error messages. To analyze this behaviour we use as a proxy the number of throw statements where a string value if used in the initialization of the exception. We observe that during the second half of the analyzed period, libraries use such exceptions significantly more often than applications (Figure 3.h). This difference appears as applications reduce their reliance on this kind of exception.

To summarize, when looking only at libraries we observe surprisingly stable percentages for the three types of analyzed exceptions in exception handlers and throw statements. Exception usage in applications, however, changes during development and there exist aspects related to exception usage that are significantly different between libraries and applications.

Observation An interesting observation made during this analysis is that on average, the amount of error handling code across the 20th snapshot of all projects is 1% (libraries 0.8%; applications 1.2% — Figure 2). Cabral and Marques [3], however, report in analyzing exception handling code in 16 Java projects that on average, the amount of error handling code in Java projects is 5%. To explain this difference we ran our analysis on a project used in the aforementioned study. The system in question is Apache Tomcat 6.0.[5] While the previous study does not indicate the location of the project or the exact version of Apache Tomcat, we find it safe to assume, based on the date of the study, that version 6.0 is the most likely candidate. We observe that 4% of the Apache Tomcat 6.0 code, together with other versions of Apache Tomcat around the date of the previous study, is dedicated to exception handling. To account for the 1% difference, we notice that the previous study also considers code from *finally* blocks to be exception handling code, while we do not. After adapting our analysis to also count code from finally blocks, we obtain for Apache Tomcat a 5% proportion of error handling code. Hence, both studies report the same amount of error handling code. However, in the current study the population of analyzed systems is larger, with Apache Tomcat being an outlier. We can conclude, that the amount of error handling code in Java projects is lower than currently believed.

[5]https://github.com/apache/tomcat60

Fig. 4: The size of the projects from *Dataset B*.

B. A Domain-specific Investigation

In this section, we present the results of analyzing *Dataset B* consisting of 30 projects, each with 8 snapshots. The goal of this analysis is to determine if the domain of an application influences the evolution of exception handling during development. Table I presents the analyzed projects and their domains.

Since we have only 5 projects per domain, analyzing trends as in the previous section cannot provide much insight. We therefore apply the Mann-Whitney-Wilcoxon test on each snapshot to see whether there is a difference between the domains. Mann-Whitney-Wilcoxon is a non-parametric test applied to multiple populations to decide whether the population distributions are identical or not, without assuming them to follow a normal distribution. In other words, we test, for each snapshot, whether the projects that belong to different domains are actually different populations, from the statistical point of view. Regarding the size of the dataset, most projects are below 100 000 lines of code in all observed snapshots, as illustrated in Figure 4. The main reason for this is that we analyze these projects during their initial development time.

We perform the same analysis as for *Dataset A*. Figure 6 gives an overview. As a general observation, we find results in agreement with the ones from Section III-A for both domains. First, we do not see any different patterns of exception handling evolution for domains representing libraries nor any statistically differences between the analyzed domains (with the exception of third-party exceptions). Second, for domains representing applications, we observe differences both throughout development, and only during certain development phases. Also, only for CMSs do we observe a significantly higher percentage of exception handling code than for compilers and editors.

When looking at the usage of custom exceptions within applications we observe statistically significant differences between domains in the second half of the analyzed period. Compilers and CMSs catch more custom exceptions than editors. Regarding throw statements, all three domains are different, with CMSs throwing the highest number of custom ex-

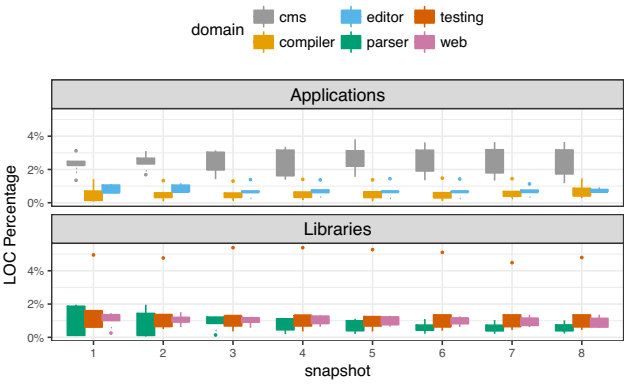


Fig. 5: Exception handling code percentages *Dataset B*.

ceptions. To complement this, we observe that in the first half of the development period there also exist significant differences in the use of `java.lang.Exception` between all three domains, with CMSs catching `java.lang.Exception` the most and compilers the least.

Analyzing handlers of standard exceptions reveals that, in the second half of the analyzed period, editors catch more exceptions of these types than the other two domains. For throw statements with standard exceptions there is a difference throughout development, with editors throwing standard exceptions the most. If we now compare CMSs to editors during the second half of the analyzed period, we can conclude that CMSs catch and throw more custom exceptions, with editors catching and throwing more standard exceptions.

The percentage of throw statements with third-party exception shows that in both categories, there is a separation between the percentages of each domain. Interestingly, the selected parsers, editors, and compilers make very little use of third-party exceptions. The remaining domains do use considerably more third-party exceptions, but with a higher variance.

Another important observation regarding the evolution of exception usage within each domain is that projects belonging to the same domain tend to converge in their usage patterns as projects evolve. This can be implied from the often decreasing sizes of the boxes in Figure 6. For instance, the percentages of throw statements with custom exceptions display a high variance in the first analyzed snapshots. This variance decreases as the projects evolve, which indicates that some domains have common usage patterns. The same can be observed about catching third-party exceptions in web frameworks, throwing custom exceptions in compilers and testing frameworks, and throwing standard exceptions in parsers and testing frameworks.

To summarize, when looking at applications belonging to different domains, we see a difference in the usage of custom, standard, and library exceptions. For some domains we see more inclination towards standard exceptions, while others rely more on custom exceptions. These observations indicate that more in-depth studies are needed to better understand particular patterns of exception usage within domains during longer periods of development. These could directly benefit developers working on applications belonging to those domains, as they could better prioritize their efforts for improving exception handling.

IV. Threats to Validity

Our analysis tools are written in Java, making use of JavaParser[6] to obtain and count relevant code. We use cloc 1.64[7] to count the overall number of lines of code in the projects. We opted to use cloc as it is a well-known tool and used in other work for similar purposes. To increase our confidence in the metrics extraction, we did a second implementation in Pharo[8] and SmaCC[9]. We applied both metric calculation implementations on several projects and they produced identical results.

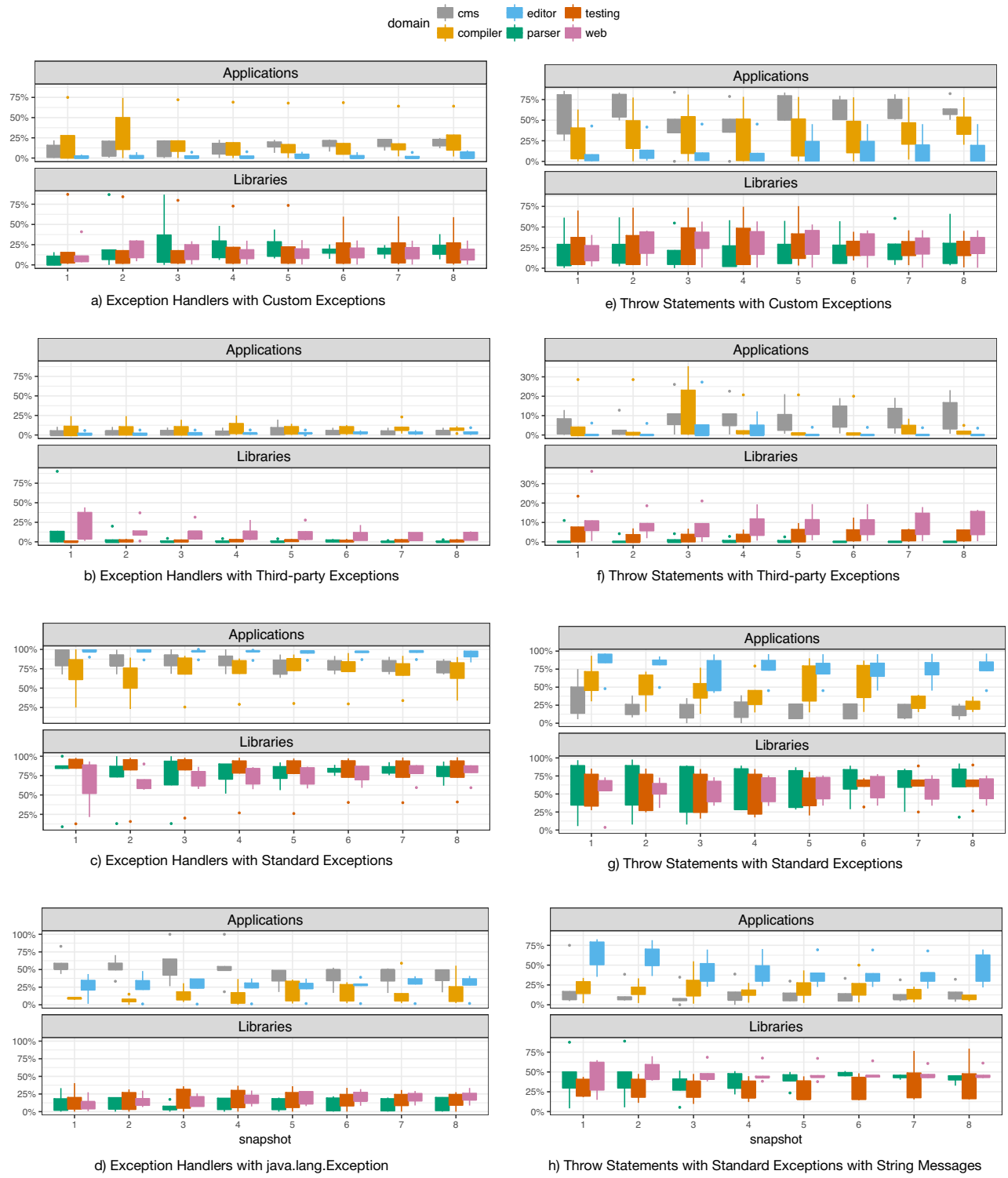
In this study, we analyzed a set of projects obtained from GitHub. While we were able to capture a wide range of open-source projects that differ in size, age, and activity, we are aware that the selection of projects may directly influence the results. While the first dataset was semi-automatically selected, where the manual filtering only entailed the categorization and removal of a project if we did not reach an agreement, many of the projects in the second dataset were selected entirely manually. This is due to the fact that few projects in our first dataset match the domains we analyzed and thus we needed additional projects to have five entries in each domain. We further identify as an external threat to our study the generalizability of our results given that all projects from the two datasets are open-source [9]. Due to this aspect we do not know if the same observations hold for closed source projects.

Finally, on average, the exception type of 10% of the extracted throw statements could not be determined, due to the inherent limitations of static analysis.

V. Related Work

Asaduzzaman *et al.* [2] leveraged the Boa infrastructure[10] to analyze how developers use exceptions in a dataset of about 267 000 projects. They identify bad exception usage patterns and a set of *exception chaining* patterns (*i.e.*, (re-)throwing of exceptions in `catch` blocks). Their main conclusion is that the bad usage patterns are common in code written by beginners and experts alike. The study differs from ours in the size of the dataset and the fact that Asaduzzaman *et al.* neither distinguish between different kinds of projects nor do they take the history of the projects into account. On the other hand, we do not report results based on programmer experience. We reach the same conclusion that certain patterns, such as catching generic types instead of specialized ones, are common in the investigated projects.

[6]https://github.com/javaparser/javaparser
[7]http://cloc.sourceforge.net/
[8]http://pharo.org
[9]http://www.smalltalkhub.com/#!/~JohnBrant/SmaCC
[10]http://boa.cs.iastate.edu



a) Exception Handlers with Custom Exceptions

e) Throw Statements with Custom Exceptions

b) Exception Handlers with Third-party Exceptions

f) Throw Statements with Third-party Exceptions

c) Exception Handlers with Standard Exceptions

g) Throw Statements with Standard Exceptions

d) Exception Handlers with java.lang.Exception

h) Throw Statements with Standard Exceptions with String Messages

Fig. 6: Boxplots of the different exception handling usage in different domains.

In a study on exception handling in 32 Java and .NET applications, Cabral *et al.* [3] investigated whether there is a difference in how exceptions are used between Java applications with checked exceptions and .NET applications where exceptions are unchecked. In their analysis, the authors categorized the different projects by domains, which is where we became inspired to perform a similar categorization in our own study. The main conclusion is that in both kinds of projects, exceptions are in fact used similarly. Like others, they observe that generic catch statements are common and many catch bodies are empty or contain code that essentially ignores the caught exception. The authors believe that the exception mechanisms in Java and .NET are flawed, as they are mostly poorly used or ignored. Note that in their analysis, Cabral *et al.* generally report higher amounts of exception handling code than we do. This is due to the facts that *(i)* they count `finally` blocks as exception handling code and *(ii)* our datasets do not—with the exception of Apache Tomcat—overlap. When we include the `finally` blocks in our calculations and compare the obtained numbers for a similar version of Apache Tomcat to the numbers presented by Cabral *et al.*, we obtain similarly high percentages of exception handling code and are thus confident in our calculations. Section III-A discusses this aspect in detail.

Nakshatri *et al.* [1] did a similar study focused on Java applications and exception usage patterns. With large datasets from GitHub and SourceForge, the authors determine usage patterns and observe that many best practices from Bloch's *"Effective Java"* [10] are ignored. For example, they find that catching generic exception classes is a common occurrence in the analyzed code. Furthermore, they conclude that programmers use exception mechanisms poorly and often ignore thrown exceptions. The authors assert that there is a significant gap between best practices and actual exception usage.

Grechanik *et al.* [11] used 2 080 open-source projects from Sourceforge for their analysis. In their study, the authors try to answer a set of questions related to various Java language features. Particularly relevant to this study is the section on exceptions, where the authors present histograms of exception usage code. In contrast to our study, they neither investigate various kinds of exceptions nor report the exception-handling code percentages.

Other studies investigate exception handling from different points of view (such as identifying good and bad practices or inspecting thrown exceptions by analyzing stack traces), or use similar methodologies where they use large bodies of source code as subjects.

Sena *et al.* [4] studied exception handling in Java libraries. They selected 656 projects and identify bad practices and patterns when using exceptions, such as ignoring caught exceptions. They also report that there are high percentages of undocumented exceptions in Java libraries that can lead to bugs in unaware clients. An additional manual study with 7 libraries revealed that 134 out of 647 bug reports are related to the inappropriate use and documentation of exceptions. This finding reveals the need for better exception handling rules and recommendation mechanisms [12][13][14].

In a study involving 4 900 Android stack traces from crashes, Kechagia and Spinellis [8] evaluated whether API documentation can be a factor in the crashes. By determining whether methods involved in the stack traces are public and whether they have undocumented exceptions (both checked and unchecked), the authors conclude that 19% of the stack traces actually do and thus state that these could potentially have been avoided with proper documentation.

Shah *et al.* [5] performed a study where eight participants were asked a set of questions about how they use exceptions. In the second part, the authors evaluated a tool for working with exceptions. For us, the first part is relevant. There, the authors find that the participants tend to use exceptions as a debugging mechanism. After initially ignoring exceptions, once they happen at runtime and crash the program, the participants try to locate the source of the problem. These findings are similar to others and again indicate a misuse of Java exception mechanisms.

VI. Conclusions and Future Work

Exceptions are the error handling mechanism provided by the Java programming language. Developers are supposed to use exceptions to recover from expected abnormal executions. It is reported in the literature that exceptions are used differently in different projects and they are widely misused for error recovery. However, it is unclear how exception usage evolves as a software project matures and its developers become more experienced in its domain and different execution scenarios.

In this paper we investigate how different types of exceptions are used in libraries as opposed to applications throughout the evolution of projects. We observe that applications employ more error handling code than libraries throughout the evolution of the projects. We observe that both applications and libraries rely mostly on standard Java exceptions. However, while the percentages of the different exception handling types in libraries remain constant, we notice an increasing trend towards custom exception usage in applications.

We also explore the effect of the domain of a project on the exception usage. We indeed observe that projects that belong to different domains rely on different types of exceptions. For instance, content management systems rely more on custom exceptions than standard exceptions while web frameworks exhibit the opposite pattern.

In the future, we plan to investigate further in two directions. The first direction is related to the exception handling code itself. We conjecture that different types of exceptions are handled differently in practice. It could be, for example, that custom exceptions are handled with dedicated recovery code, but standard exceptions are just logged. The second direction is to track each individual exception handler throughout the evolution to see whether developers change the types of caught exceptions and how. This way we can further investigate the reasons behind the change of exception usage trends.

ACKNOWLEDGMENT

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, 01.01.2016 to 30.12.2018). We also acknowledge the financial support of the Swiss Object-Oriented Systems and Environments (CHOOSE)[11] for the presentation of this research.

REFERENCES

[1] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in Java projects: An empirical study," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 500–503. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903499

[2] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How developers use exception handling in Java?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 516–519. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903500

[3] B. Cabral and P. Marques, "Exception handling: A field study in Java and .NET," in *Proceedings of European Conference on Object-Oriented Programming (ECOOP'07)*, ser. LNCS, vol. 4609. Springer Verlag, 2007, pp. 151–175.

[4] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio, "Understanding the exception handling strategies of Java libraries: An empirical study," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 212–222. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901757

[5] H. Shah, C. Görg, and M. J. Harrold, "Why do developers neglect exception handling?" in *Proceedings of the 4th international workshop on Exception handling*. ACM, 2008, pp. 62–68.

[11] http://www.choose.s-i.ch

[6] H. Osman, A. Chiş, J. Schaerer, M. Ghafari, and O. Nierstrasz, "On the evolution of exception usage in Java projects," in *Proceedings of the 24rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Feb. 2017, p. to appear. [Online]. Available: http://scg.unibe.ch/archive/papers/Osma17b-exception-usage.pdf

[7] Oracle, "Three rules for effective exception handling blog," https://community.oracle.com/docs/DOC-983219, archived at http://www.webcitation.org/6pLmGLfev. [Online]. Available: https://community.oracle.com/docs/DOC-983219

[8] M. Kechagia and D. Spinellis, "Undocumented and unchecked: Exceptions that spell trouble," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 312–315. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597089

[9] H. K. Wright, M. Kim, and D. E. Perry, "Validity concerns in software engineering research," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 411–414.

[10] J. Bloch, *Effective Java (2nd Edition) (The Java Series)*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[11] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "An empirical investigation into a large-scale Java open source code repository," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 11:1–11:10. [Online]. Available: http://doi.acm.org/10.1145/1852786.1852801

[12] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 496–506.

[13] M. M. Rahman and C. K. Roy, "On the use of context in recommending exception handling code examples." in *SCAM*, 2014, pp. 285–294.

[14] E. A. Barbosa, A. Garcia, and M. Mezini, "Heuristic strategies for recommendation of exception handling code," in *Software Engineering (SBES), 2012 26th Brazilian Symposium on*. IEEE, 2012, pp. 171–180.