The Use of C++ Exception Handling Constructs: A Comprehensive Study

Rodrigo Bonifácio, Fausto Carvalho, Guilherme N. Ramos
Computer Science Department
University of Brasília
Brazil

Uirá Kulesza, Roberta Coelho
Informatics and Applied Mathematics Department
Federal University of Rio Grande do Norte
Brazil

Abstract—Exception handling (EH) is a well-known mechanism that aims at improving software reliability in a modular way—allowing a better separation between the code that deals with exceptional conditions and the code that deals with the normal control flow of a program. Although the exception handling mechanism was conceived almost 40 years ago, formulating a reasonable design of exception handling code is still considered a challenge, which might hinder its widespread use. This paper reports the results of an empirical study that use a mixed-method approach to investigate the adoption of the exception handling mechanism in C++. Firstly, we carried out a static analysis investigation to understand how developers employ the exception handling construct of C++, considering 65 open-source systems (which comprise 34 million lines of C++ code overall). Then, to better understand the findings from the static analysis phase, we conducted a survey involving 145 C++ developers who have contributed to the subject systems. Some of the findings consistently detected during this mixed-method study reveal that, for several projects, the use of exception handling constructs is scarce and developers favor the use of other strategies to deal with exceptional conditions. In addition, the survey respondents consider that incompatibility with existing C code and libraries, extra performance costs (in terms of response time and size of the compiled code), and lack of expertise to design an exception handling strategy are among the reasons for avoiding the use of exception handling constructs.

I. INTRODUCTION

The exception handling mechanism (EHM) [1] is a well-known technique for improving error recovery in a modular way by supporting a certain degree of separation between the expected control flow and exceptional conditions [1], [2], [3]. Although language mechanisms for EH were conceived almost 40 years ago [3], existing works report a small adoption of exception handling mechanisms in existing programs [4], [5], [6]. Some of these works draw their conclusions through the analysis of open-source systems written in Java and C#, while the EHM constructs were introduced in the first specification of the languages. Differently, the exception handling mechanism of C++ was designed between 1985 and 1989, being only introduced in its second release [7].

C++ is a programming language widely used in different domains, from desktop environments (e.g., Plasma Desktop from KDE) and end-user applications (office suites, Web browsers, e-mail clients) to server side containers (Web servers, CORBA servers, and so on), game development libraries (e.g., AlephOne), and graphical user interface toolkits. C++ is also frequently used for developing scientific applications, database, operating, and embedded systems—domains that have to accommodate conflicting requirements related to portability, response time, and error handling. In this paper we investigated the use of EHM in C++, in order to understand how developers harmonize the use of EHM for error recovery with the implementation of other concerns. Our study relies on a mixed-method approach: (i) a quantitative investigation that aims to gather information about the typical usage of EHM constructs in 65 subject systems and (ii) a qualitative assessment based on a survey regarding the understanding of 145 developers on the use of EHM in C++. To the best of our knowledge, there is no other comprehensive investigation on the use of exception handling in C++ and, although the literature discusses a number of reasons that might prevent developers of using EH constructs in C++ [8], this issue has not been empirically investigated yet.

The contribution of this paper is twofold. First, we report the typical and small use of exception handling constructs in C++ programs and discuss how the use of the return code strategy is the preferable approach for error recovery in C++, regardless of existing claims about the benefits of the EHM [3]. Second, we found that C++ developers avoid the use of EH constructs due to several motivations, including both technical (such as extra performance costs introduced by the use of exceptions) and non-technical reasons (such as the lack of experience to properly design an error recovering strategy using EH constructs). These findings have several implications, and understanding how C++ developers comprehend and use EH constructs might support future efforts on the implementation of EHM in the language. Moreover, this might help novice C++ developers during the design of the error recovering strategy for a new system and support documentation of the benefits and drawbacks of using EHM in C++.

Road map. The next section presents a general view of the C++ EHM semantics. Section III presents the research questions, the data collection procedures, and the criteria used to select the subject systems. Sections IV and V present and discuss the results of our assessment while Section VI discusses some limitations of our study. We relate our study with existing related works in Section VII and present some final remarks in Section VIII.
II. Exception Handling in C++

Although the original design of C++ considered exception handling, this mechanism was only introduced in the second version of the language. According to Stroustrup, “there was not enough time to carefully design the exception handling mechanism in the first version, and bad design decisions could lead to unnecessary complexity and runtime checks that could degrade the performance of C++ applications” [9].

For this reason, the design of C++’s EHM had to consider backward compatibility with existing C code as well as with code written in the first version. Clearly, this constrained to some extent the design of the EHM in C++, which was founded on the ideas of fault-tolerant systems designers, instead of the ideas of language designers [9]. Nevertheless, the design of the EHM in C++ had a significant influence on that of Java and C# languages, despite substantial differences, such as:

- C++ allows a developer to throw any type as an exception. In Java and C#, only subclasses of Throwable (Java) and System.Exception (C#) can be thrown as an exception.
- Java supports checked and unchecked exceptions; whereas C++ and C# only support the latter. Checked exceptions provide means to enforce contracts between method calls: if a method m might throw a checked exception E, a method that calls m must either handle E or explicitly declare that it might throw an exception of type E.
- Besides C++’s try-catch block, Java and C# have the finally block, which executes in both normal and exceptional conditions.

It is also possible to declare the exceptions that a method might throw in C++, even though this practice is not encouraged anymore [10]. For example, consider the declaration of a Stack class in Figure 1. According to the original semantics of exception handling in C++, the method push might throw any type as an exception, the method pop might throw exceptions of type StackException, and the method size should not throw any exception [11].

The throw clause in the method signature of C++ classes defines a form of weak contracts, because no compiler error is reported when an exception is thrown within the implementation of the size method (see Figure 2). In addition, it is not possible to handle exceptions thrown by a call to the size method and, instead, the default terminate function is called. This semantics leads to the dead code related to the StackException handler in the main function of Figure 2.

```java
class StackException : public exception {
};
const int MAX_SIZE = 10;
class Stack {
private:
  int arr[MAX_SIZE];
  int top;
public:
  void push(int v);
  int pop() throw(StackException);
  int size() throw();
};
```

Fig. 1. A declaration of a stack class.

1Although it is possible to throw any type that can be copied in C++, it is recommended to throw only user-defined types [10].

Mainly for this reason, C++ developers are advised to avoid the specification of exceptions in method declarations. More recently, the C++11 standard deprecated exception specifications, recommending the use of noexcept as a replacement to the throw() specification, with similar semantics [10].

III. Study Settings

This study aims at understanding the usual practice of exception handling in C++. We organized this research in two phases: the first investigates the source code characteristics of 65 open-source projects using static analysis and the second consists of a survey to assess the understanding of C++ developers with respect to the use of EHM. This section first presents the research questions and data collection procedures of the first and second phases (Section III-A and Section III-B). Section III-C presents the selection criteria and an overview of the 65 open-source systems considered here.

A. First Phase: Source Code Analysis

To understand the use of EHM in C++ systems, we conducted a quantitative assessment for answering three research questions: (RQ1) To what extent C++ developers use the EHM?, (RQ2) Which measures developers typically take when an exception is caught in C++, and (RQ3) How does the EHM evolve along different releases?

Concerning RQ1, our goal is to understand how widespread is the use of the EHM through the case studies. We investigate the use of the exception handling constructs (try-catch blocks, throw statements, and method declarations that might specify which exceptions could be thrown) in the subject systems, comparing our observations to existing reports. To
answer RQ2, we classify exception handlers in five categories: handlers that throw a new exception (the body of the catch block throws an exception), handlers that rethrow the caught exception (often performing some recovery statements before the throw statement), handlers that return an expression, handlers that perform general recovery strategies (and that are not classified in the other categories), and empty handlers that do not execute any statement. Therefore, RQ2 allows us to carry out a descriptive analysis to understand the usual exception handling practices in C++, without going into the details of the individual handlers’ statements. Regarding RQ3, some authors have argued that the EHM should be considered an architectural concern [12], so that it would be worth to introduce the EHM at an earlier stage of software development. We investigate RQ3 by analyzing different versions of the subject systems, trying to figure out whether the EHM is present independently of the version of the systems. To investigate RQ3, we only considered the 15 subject systems which make most use of the exception handling mechanism.

Data collection procedure. To investigate the research questions of the first phase, we developed a static analysis tool for collecting information about exception handling usage. This tool builds on the Eclipse CDT API and consists of (a) five visitors that extend abstract classes of Eclipse CDT and gather exception handling data by traversing the abstract-syntax of the source code, (b) two pretty printers for exporting the results of the analysis, and (c) several domain objects that represent the collected data. Regarding the computation of lines of code for each project, we used the SLOCCount tool [13], since it has been used in a number of previous works. The supplementary material for this work, such as list of selected projects, the static analysis tool, survey questions, and the response data of this study, is available in a web site³.

B. Second Phase: Survey on the Perception of EHM

To understand the knowledge and perception of C++ developers regarding the use of EHMs, we conducted a survey to investigate two research questions: (RQ4) Which strategies C++ developers use to deal with unexpected conditions? and (RQ5) Which are the reasons that motivate C++ developers to avoid the use of EHM? We investigate (RQ4) to discuss what is the preferred approaches for recovering from unexpected conditions in C++. In (RQ5), we investigate the perception of C++ developers regarding the use of EHM and the reasons that might lead C++ developers to avoid the use of EH constructs.

SurveyMonkey. To carry out the investigation of the second phase, we designed an online survey using the services available at SurveyMonkey. The survey comprises 9 questions, where the first three (SQ1, SQ2, SQ3) focus on the personal data of the respondents (the developer’s experience with C++ and the projects she has contributed to), the next four questions (SQ4 – SQ7) elaborate on the research questions (RQ4) and (RQ5), and the remaining two questions (SQ8, SQ9) evaluate quality attributes of the survey, such as easiness to follow and whether the respondents consider that there are missing questions.

We first invited a small number of researchers and C++ developers to answer and provide feedback about the survey. The results of this trial led us to perform small changes on the organization of the survey.

Sample selection and survey procedure. Our general population includes C++ developers with different degrees of experience with the language. To contrast the survey findings with the results of the first phase, we decided to work with the communities of C++ developers that have contributed to the subject systems (Section III-C). Although this might narrow the generalization of our findings, we claim that this working population [14] satisfies the goals of this study. We recruit the respondents by posting messages on the mailing lists of the subject systems. The survey was conducted during the weeks from August 12, 2014 to August 30, 2014. A total of 145 respondents answered our survey, though 112 responses (77%) were sent in the first three days. In addition, 84 respondents have more than 10 years of experience working with C++ and the responses came from developers that have contributed to 40 out the 65 open-source projects we considered here.

C. Subject Systems

We selected 65 open-source systems from different domains to get a comprehensive overview about the use of the EHM in C++. Those systems were selected from different open-source hosting sites—such as Apache Foundation, GNU, SourceForge, GitHub, KDE, Google Code, Google Summer of Code, Microsoft Open Source Directory, and NASA Open Source Initiative. We considered the following criteria for inclusion:

- Availability of at least 5 public stable releases.
- In-depth use of C++ as programming language. Several projects combine different programming languages, but we only considered projects whose implementation contains at least 25% of C++ code.
- Active contribution from the community (the GitHub repository of MongoDB reported that 35 contributors have pushed 404 commits between December 15, 2013, and January 15, 2014).
- Relevance for its target domain. For instance, Apache Open Office has reached 75 million downloads in 18 months (from May 2012 to October 2013). Boost C++ Libraries achieved more than 200,000 downloads in the week of January 9-16, 2014.

Almost all projects have been listed as one of the most popular projects from SourceForge or member of the GitHub trends, at the time in which they were analyzed (December 2013 and January 2014). Projects not hosted in neither SourceForge or GitHub were selected using an arrangement of the above criteria. For example, projects from GNU and NASA Open Source Initiative do not present information about popularity, in terms of number of downloads. Nevertheless, the

---

³For a list of works citing SLOCCount, visit http://goo.gl/2mFkk2
³http://rbonifacio.net/papers/scam2015/
selected projects from these foundations meet at least the first two criteria.

Furthermore, Apache Xalan, a stable library for parsing XML files, did not meet the active contribution criterion, because its last public release was made available in 2001—even though the last commit in its source code repository dates back to September 2013. GCC is another exception to the above criteria, because its code base comprises 14.53% of C++ code, which represents 900 KLOC.

Similar to other studies that evaluate the use of exception handling mechanisms [4], [15], we did not consider information available on code standards or guidelines as inclusion or exclusion criterion. However, after the selection of the subject systems, we realized that 8 of them present explicit guidelines for avoiding the use of exceptions, whereas the 57 remaining do not present explicit recommendations about the use of exception handling mechanisms. We decided to not exclude any selected subject system based on this criterion; and the reasons for this decision are two fold. First, based on descriptive statistics, we found that the recommendation for avoiding the use of exceptions does not change the main findings of this paper (we discuss this issue on Section VI). Second, we claim that it is also important to consider projects that recommend developers to avoid the use of EHMs, mainly because a fraction of these projects does still use exception handling to some degree, they are likely to represent a subset of C++ systems, and we should try to understand the rationale for such recommendations.

IV. RESULTS OF THE QUANTITATIVE ASSESSMENT

To answer our first research question (*to what extent C++ developers use the EHM?*), we use a slight adaptation of the Dedication metric, which is a measurement of tangling used in the modularity assessment of crosscutting concerns [16], [17]. Dedication measures the ratio between the lines of code of a component \( T \) that are related to a concern \( S \) and the total lines of code of the component \( T \). Here, in order to answer RQ1, we are interested in the exception handler concern (consisting of try–catch blocks) and the recovery concern (consisting only of catch blocks). Due to the C++ preprocessor, we decided to not consider lines of code in our evaluation, and instead we use the number of visited statements as an alternative. Therefore, for a given translation unit, we traverse the statements that matches a preprocessor configuration using the visitor API of Eclipse CDT. Hereafter, we use the term statements to refer to the visited statements of a system. In this way, we compute two metrics for a system:

- **Exception handler dedication (\( EH_{\text{ded}} \))** is given by the ratio between the number of exception handler statements (statements within try–catch blocks) and the total number of statements.
- **Recovery dedication (\( REC_{\text{ded}} \))** that is the ratio between the number of recovery statements (statements within catch blocks) and the total number of statements.

We found a small dedication regarding the use of exception handling constructs in the subject systems. Figure 3 presents two boxplots summarizing our findings. The first shows that the central tendency of \( EH_{\text{ded}} \) is 0.40% (with a standard deviation of 3.21%). In absolute terms, this means that, from the 18,684,811 visited statements, 383,318 are surrounded by try-catch blocks. Regarding the recovery concern, the boxplot of Figure 3(b) shows that a small fraction of C++ systems is devoted to recovering from exceptional conditions using catch blocks—that is, far less than one percent of the statements handle exceptions.

In details, the median value of \( REC_{\text{ded}} \) is equal to 0.03%, with a corresponding standard deviation of 0.47%. It is worth to compare these numbers with existing reports that investigate a similar issue in other programming languages. For example, Cabral and Marques analyzed 32 open-source Java and .NET systems. They examined 3,410,294 lines of code, from which 137,720 (4.03%) were dedicated to exception handling [4]. The authors stated that these numbers are “much less than...
what would be expected”. Our findings reveal that developers of C++ systems use an order of magnitude less exception handling constructs. Furthermore, there is a strong correlation (0.98) between the \( EH_{ded} \) and \( REC_{ded} \) metrics and a moderate correlation between these metrics and the number of try–catch blocks and the number of handlers of a system. More surprisingly, there is a weak correlation between our dedication metrics and the size of a system (SLOC and number of statements), though C++ designers associate the use of EHM to system complexity [10]. Table I summarizes these correlations. We used the Spearman’s method [18] because the relationships are monotonic.

| TABLE I |
| SPEARMAN CORRELATION MATRIX CONSIDERING THE CONCERN DEDICATION METRICS |
| SLOC & Statements & TryCatchBlocks & Handlers |
| --- | --- | --- | --- |
| \( EH_{ded} \) | 0.191 | 0.107 | 0.898 |
| \( REC_{ded} \) | 0.169 | 0.140 | 0.882 |

Regarding our second research question (which procedures developers often take when an exception is caught in C++?), we carried out an exploratory data analysis with respect to the number of statements of each handler and the frequency that developers use statements for throwing a new exception, rethrowing an exception (with the effect of propagating the caught exception) or returning an expression within a catch block. It is important to note that C++ presents a construct for rethrowing a caught exception: a throw statement without arguments. Here we consider that a catch statement rethrows an exception when such a construct is used, which is a simplification with respect to Fu and Ryder’s approach [19].

We found that most of the recovery strategies of the exception handlers comprise a single statement. Accordingly, Figure 4 presents a density plot of the number of statements within catch blocks. Most of the catch blocks comprise less than 5 statements, with median value of 1 and standard deviation of 2.28. This result might suggest that (a) recovering from exceptional situations (such as performing some cleanup operation) is a trivial task that requires either a simple statement or a function call that can be dedicated to exception handling; (b) there is a common approach for error recovering, and thus it is worth to modularize it as a function; or (c) the error recovering is not a critical concern of the subject systems. Note that, from 36,646 catch blocks in all systems, 6127 handlers are empty (16.71%), which might lead to fail-silent faults [20], [21], [22].

Regarding the actions that developers usually take to handle exceptions, we found that 2282 handlers throw the caught exception (6.22%), 1440 handlers throw a different exception (3.92%), and 4031 handlers return an expression (10.99%). That is, most of the handlers (the remaining 80%) do not propagate exceptions and neither return from the function call—they must only perform some cleanup procedure. Handlers that rethrow the caught exception or return an expression often comprise more statements than catch blocks that throw a different exception.

In the first two cases, catch blocks usually present additional actions before rethrowing or returning an expression. Differently, when a new exception is thrown from a catch block, the throw statement is often the sole statement within the block. Table II presents some statistics about the number of statements for these cases. Additionally, returning an error code from a catch block is another recurrent strategy. Nevertheless, in some cases it is a solution that resembles the old style of handling exceptional conditions in C. Moreover, from 4031 catch blocks with return statements, 495 (12.33%) return nothing—they are just “return” statements that might also lead to fail-silent faults.

We found that most systems throw and catch a small number of distinct exceptions. Figure 5 shows the ten most frequent types of exceptions thrown within the subject systems. Almost 20% raise Runtime, IncorrectValue, and IllegalArgumentException exceptions and 30% of all throw statements raise one of the types presented in the figure. Throwing quite general exceptions is a prevalent behavior, but it often does not represent the specific cause of a problem. Therefore, it seems to be a common practice to raise general, non-domain specific exceptions in C++ projects.

Remarkably, there is a weak correlation\(^5\) (0.31) between the number of distinct types thrown as exceptions and the size of a system (in terms of lines of code). For example, we found 7 different types of exceptions being thrown within kOffice (976,012 lines of code and 1406 throw statements) whereas 195 different types of exceptions are thrown within OpenOffice (2,963,117 lines of code and 4101 throw statements).

| TABLE II |
| DESCRIPTIVE STATISTICS OF THE NUMBER OF STATEMENTS WHEN CONSIDERING CATCH BLOCKS |
| --- | --- | --- | --- | --- |
| Category | Min | Median | Mean | Max | sd |
| Rethrow | 1 | 2 | 3.06 | 43 | 3.14 |
| Throw | 1 | 2 | 2.76 | 80 | 4.60 |
| Return | 1 | 2 | 2.31 | 26 | 2.59 |

\(^5\)The correlation’s analysis only consider projects that throw exceptions.
Types raised as exceptions
- RuntimeException
- IllegalArgumentException
- IncorrectValueException
- Exception
- IndexOutOfBoundsException
- NotImplementedException
- CIMException
- CommandException
- IOException
- DisposedException

Fig. 5. Most frequent types thrown as exceptions.

Nevertheless, there is a strong correlation (0.839) between the number of distinct types thrown as exceptions and the number of try-catch blocks as well as a strong correlation between this measurement and the number of throw statements. This might suggest that, the greater the concern with exception handling (try-catch blocks), the greater the number of distinct types thrown as exceptions by a system. We found a moderate to high correlation between such number of distinct types thrown as exceptions and the concern dedication metrics (see Table III). Furthermore, the number of distinct types thrown by a system presents a high dispersion, as illustrated in Figure 6(a), which also reveals several outliers, a central tendency of 13 and a median absolute deviation (MAD) of 17.79—where MAD is a measure of statistical variation more resilient to outliers [18].

![Boxplot with the distribution of the number of distinct exceptions](image)

**TABLE III**

<table>
<thead>
<tr>
<th>SLOC</th>
<th>TS</th>
<th>TryCatchBlocks</th>
<th>ET</th>
<th>EC</th>
<th>TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.361</td>
<td>0.881</td>
<td>0.839</td>
<td>0.688</td>
<td>0.680</td>
<td></td>
</tr>
<tr>
<td>0.250</td>
<td>0.833</td>
<td>0.916</td>
<td>0.765</td>
<td>0.763</td>
<td></td>
</tr>
<tr>
<td>0.353</td>
<td>1.000</td>
<td>0.877</td>
<td>0.711</td>
<td>0.684</td>
<td></td>
</tr>
</tbody>
</table>

In addition, sixteen projects do not throw any exception at all—among them are Google Chromium Web browser, Otter cross platform game engine, Keepass password manager, FileZilla portable FTP client, the HotSpot Java Virtual Machine, and seven projects from the KDE community. We realized that the KDE community discourages the use of the C++ exception handling mechanism. We sent messages to the developers’ groups of Google Chromium and KDE community. Developers explained that they build Chromium without exception support to avoid incompatibility with existing Google code that is not exception safe and to avoid performance costs related to exception handling⁶. Differently, developers from KDE community relate the EHM to modularity issues, as stressed in an answer to our post.⁷

Don’t use C++ exceptions if you can avoid it, and if you must use them, keep them tightly controlled in an isolated part of the application. If you use exceptions in one place the entire call stack needs to be exception safe . . .

Incompatibility with non-exception-safe code written in C is also reported as a reason for not using exceptions in C++.  

⁶http://goo.gl/d1i8qj  
⁷http://goo.gl/CUZAFE
Nevertheless, we found a weak correlation between our exception handling metrics and the ratio between non-cpp and cpp code of the subject systems. For example, Firebird is a multi-language software mostly written in C code (50.6%), compared to C++ (45.2% of the code). Nevertheless, it dedicates 4.13% of the statements within try-catch blocks. Although the analyzed KDE applications are mostly written in C++, they present a low degree of interest in handling unexpected situations using C++’s EHMs.

Similarly, we also investigate the types used as arguments to catch blocks in all subject systems. From 36,646 observations, 4424 blocks (12.07%) catch all exceptions (a catch(...) { //...} block). Additionally, the three most frequent types caught as exceptions (Exception, uno::Exception, and std::bad_alloc) correspond to 30.2% of the handlers within all systems. It is important to note that several projects declare their own Exception base class, from which other exceptions descend. The boxplot of Figure 6(b) summarizes the number of distinct types used as arguments to catch blocks by each project and the second row of Table III correlates this number to other measures used in this work. A high concentration of handlers in terms of generic exceptions suggest a certain degree of homogeneity of the exception handling concern and, thus, the use of aspect-oriented constructs to modularize the exception handling concern might be recommended [23].

To answer our third research question (how does the EHM evolve along different releases?), we randomly selected and analyzed at least four releases of 15 subject systems, to understand whether the EHM is a concern that emerged in the later stages of a software life cycle. If this is the case, we should observe some differences in the metrics EH_ded and REC_ded between the earlier and releases and the most recent.

We found that there is no relation between the usage of EHMs constructs and the releases of a system. Figure 7 presents the scatterplots of the EH_ded and REC_ded metrics, grouped by project and release (versions of the systems labeled as v1, v2, v3, v4, and v5). Most of the systems present a certain degree of constancy for both metrics through their different releases. Differently, four projects (eMule, RethinkDB, Gnash, and Boost) present a tendency for increasing both metrics. For example, the exception handling dedication (EH_ded) metric of RethinkDB jumps from 0.60% in the first version to 3.02% in the last version that we analyzed. Two projects (OpenCV and Apache Xalan for C++) present a decreasing tendency for the EH_ded metric. In the case of OpenCV, which used to dedicate 3.96% for exception handling (statements within try-catch blocks) in the first analyzed version whereas in the last version this number decreased to 2.21%. Other systems present an increasing tendency followed by a decreasing tendency (MongoDB) or vice-versa (such as MyServer and Firebird).

Based on this analysis, we cannot infer any pattern of how the exception handling mechanism evolves during the software life-cycle. In particular, we reject the hypothesis that the EHM is a late concern that presents an increasing importance along the evolution of a software life-cycle.

V. RESULTS OF THE QUALITATIVE ASSESSMENT

To assess C++ developers’ understanding of the use of EHMs, we conducted a qualitative assessment based on a survey (Section III presents more on its design). We received answers from 145 developers, though from this set we discarded 28 observations that did not provide complete answers.

We organized the survey in three major parts, the survey questions (SQ) are available in the supplementary material website. In the first part, we collected personal information
about the respondents (SQ1, SQ2, and SQ3). In the second part, we elaborated three closed-ended questions—to collect the overall perception of developers about the use of EHMs in C++ (SQ4, SQ5, and SQ6)—and one open-ended question to provide further details on the reasons that might lead a developer to avoid the use of EHMs (SQ7). In the third part, we provide two questions (SQ8 and SQ9) so that the respondents could evaluate the questionnaire. Analyzing the responses for question SQ8 revealed that almost 80% of the respondents agree or totally agree that the questionnaire was easy to understand (only one respondent totally disagreed). In addition, fourteen respondents provided additional pairs of questions and answers that could be used to complement their responses. A summary for the main findings of the qualitative research is presented next.

A. On the Error Recovery Strategies Typically Used in C++

Most of the respondents agreed that C++ developers often avoid using EHMs. Analyzing the answers for SQ4 (see Figure 8), we could observe that 76% of the respondents (regardless of their experience) agree with the sentence “C++ developers often avoid using exception handling constructs”. The fifth question (SQ5) complements this question by asking for the respondents to vote among several error recovery strategies typically used in C++. To investigate this question, we use a variant [24] of the paired comparisons method of Bradley and Terry [25], available in the prefmod R package [26]. Based on a ranking question, this method estimates the preference probability of each option—where the options for SQ4 are related to the error recovery strategies in C++. Table IV, presents the result of this investigation, showing that the return code alternative is preferred over the use of exception handling mechanisms. These results corroborate our quantitative findings discussed in Section IV.

![Fig. 8. Answers to the question Do you agree that C++ developers often avoid using exception handling constructs? This barplot summarizes the answers grouping them by the respondent experience.](image)

B. On the Reasons for Avoiding EHMs in C++

The respondents considered that, among the available options, incompatibility with C code and existing C libraries and extra performance costs (in terms of response time and size of the compiled code) are the main reasons leading C++ developers to avoid using EH constructs. Here, our analysis considered the closed question SQ6, which asks for the respondents to vote among five possible reasons drawn from C++ mailing lists. To analyze this, we use the same paired comparisons method discussed in the analysis of SQ5, and present the results in Table V.

![Table IV](image)

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Voting Preferences</th>
<th>Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Code</td>
<td>66 30 7 6 0</td>
<td>0.44</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>42 30 21 14 4</td>
<td>0.29</td>
</tr>
<tr>
<td>C++ Macro for Error</td>
<td>4 27 44 16 1</td>
<td>0.16</td>
</tr>
<tr>
<td>Handling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global State Variable</td>
<td>2 19 21 45 0</td>
<td>0.11</td>
</tr>
</tbody>
</table>

![Table V](image)

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Voting Preferences</th>
<th>Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incompatibility existing code and libraries</td>
<td>45 32 14 15 1</td>
<td>0.29</td>
</tr>
<tr>
<td>Extra performance costs</td>
<td>26 22 27 22 13</td>
<td>0.21</td>
</tr>
<tr>
<td>Portability</td>
<td>8 21 33 24 14</td>
<td>0.17</td>
</tr>
<tr>
<td>Design of the EHM in C++</td>
<td>17 16 15 19 28</td>
<td>0.16</td>
</tr>
<tr>
<td>EHMs are not necessary</td>
<td>17 18 16 21 31</td>
<td>0.16</td>
</tr>
</tbody>
</table>

In addition, we carried out a qualitative text analysis [27] for interpreting SQ7, which is an open-ended question where the respondents could fill in detailed answers regarding their perceptions on reasons for avoiding the use of EHMs in C++. We first read the 42 answers to this question, so that we could grasp the essential terms reported by the respondents as the reasons that might lead developers to not use EH constructs. Some respondents considered options already given in SQ6. Then, following an iterative approach, we coded the answers using RQDA—a R package for qualitative data analysis. In this case, we assigned the 14 recurrent terms in the answers to text fragments, and the strength of each term, according to the frequency they appear in the answers of SQ7, is illustrated in Figure 9.

![Fig. 9.](image)

In absolute values, 12 answers relate to the tag Educational Issue. Respondents emphasize that most C++ developers learn to deal with exceptional conditions using examples available in systems and libraries written in C, which does not support exception handling constructs. This is clear when the respondents explain that C++ developers read “…bad examples from
old code” and that “…people do not understand enough about exception handling”.

Among other reasons mentioned for avoiding exception handling constructs, 10 developers revealed that EHMs impaired modular reasoning. For example, one of the respondents explains that “…the changes in program flow caused by exception handling constructs are very counterintuitive” while another discusses that “…exception handling constructs lead to extra code paths that are often difficult to analyze and test”.

In addition, 8 respondents consider it hard to design exception handling code in C++. For example, one of the answers claims that “…error and exception handling (code) is hard”. Based on this, we cannot conclude whether it is harder to implement EH code in C++ or if the concern (error handler) is hard to design. We also found some relation between hard to design and the lack of experience and educational issues in some answers. For example, one answer states that some C++ developers “understand the syntax but have no idea how to design flexible/extensible exceptions aware classes or when (to) use exceptions”.

VI. Threats to Validity

We found two main threats to the validity of our work. First, with respect to the external validity, one might suggest that we cannot generalize our findings because our study is restricted to the selected, open-source systems. Regarding this issue, we argue that we considered systems from different domains—from end-user applications (office suites and Web browsers) to database and development libraries. They also present different characteristics in terms of size and lifetime (some projects started 20 years ago, whereas others have had only a few years of development). Finally, although we have only analyzed open-source systems, some of these either have emerged as non-open source efforts from private companies or are guided by individual corporations. Thus, it is possible that our findings could be generalized to other scenarios.

Considering internal and construction validities, static analysis of C++ code is a difficult task [28], mainly because, using the C++ preprocessor, there might exist different configurations of the source code. Considering all fragments is not a feasible approach, because some fragment’s combinations might result in invalid C++ code. This means that we only evaluated a subset of possible configurations and another subset could lead to results different from those that we report in this paper. For this reason, we decided to concentrate our analysis using only the statements visited by our static analysis tool, instead of all LOC. In a future work, we will investigate the impact of the preprocessor in our static analysis tool.

A subset of the selected projects present explicit recommendations for not using exceptions. However, as explained in Section III, we did not consider code guidelines as an inclusion or exclusion criteria. Nevertheless, we repeated our analysis without considering all projects that either recommend avoiding exceptions or do not present any use of EHMs, which leads to a list of 39 projects. Besides the expected variation in the median, mean, and standard-deviation values, we claim that the main finding present in this paper is still valid: C++ developers tend to use other approaches (in particular returning code) for dealing with unexpected conditions, instead of using EHMs. We also believe that removing C++ systems that do not use EHMs from our analysis narrows the potential to generalize our findings, in particular because some of the subject systems not considered in this final analysis have been used to demonstrate major use of C++ [29].

VII. Related Work

Although the use of exception handling mechanisms was proposed almost 40 years ago [3], there exist a few empirical assessments of their use, particularly in production systems. For instance, Robillard and Murphy [12] explain that the lack of a careful design of the EHM in Java projects makes it difficult to reason about exception propagation and increases the costs for evolving exception interfaces (particularly when using checked exceptions).

Recent works try to understand how developers understand EHMs. For instance, Ebert and Castor Filho carried out a survey whose results suggest an infrequent documentation and testing of exception handling code [5], even though 40% of the respondents considered the quality of EH code either good or very good. Shah et al. interviewed novice and experienced developers to understand the viewpoint of these groups towards exception handling [6]. Their results suggest that experienced developers consider exception handling as a crucial aspect in software development, whereas newcomers do not consider it a priority concern.

Several empirical studies have been conducted to evaluate the benefits of modularizing the exception handling concern using aspect-oriented programming (AOP). For instance, Lippert and Lopes argue that AOP might significantly reduce the amount of code for detecting and handling exceptions [23]. Differently, Castor Filho et al. argue that several conditions (such as homogeneity, context independence, and simplicity) must be satisfied in order to get the benefits of modularizing the exception handling concern using AOP [5]. Our investigation was not conducted to evaluate these properties explicitly, but some of our findings indicate that the use of EH code in C++ programs is not as frequent as in other languages (like Java). Moreover, we also observed that most of the handlers are generic (handling exception super-types).
and rarely implement specific handling actions, indicating that C++ implementations could benefit from the use of AOP.

Closely related to our investigation is the work of Cabral and Marques [4], which investigates the use of EH in open-source projects written in Java and .NET. Our goals are similar, but we restricted our study to C++ and here we correlate our observations among different properties of the system (number of statements, number of try-catch blocks) and complement the results of a survey with C++ developers which were involved in the implementation of the subject systems—the survey helped us to understand the reasons for the reduced use of EHM in C++. To the best of our knowledge, this was the only survey conducted so far aiming at insights on the characteristics of the exception handling code of C++ programs.

VIII. Final Remarks

Criticism of exception handling mechanism (EHM) usage is nothing new. In 1979, Liskov and Snyder argued that EHM had been ignored in programming languages. Although several attempts have been made to solve this limitation (nowadays, mainstream programming languages support some sort of EHM), the use of exception handling constructs is still often neglected. In this paper, we presented the results of an empirical study of this issue, focusing our assessment on the use of exception handling in C++.

The results provide evidences that developers favor the use of different techniques to deal with exceptional conditions over using exception handling constructs in C++, even in well-known, non-trivial systems like Google Chromium browser or MongoDB document-oriented database. In this research, some C++ developers reported that exception handling in C++ leads to performance costs, portability challenges (some compilers for specific platforms do not support exceptions), and incompatibility issues with existing C code. Further investigation to understand the negative impact of EHM on these concerns is still necessary.

REFERENCES