How Swift Developers Handle Errors

Nathan Cassee
Eindhoven University of Technology
Eindhoven, The Netherlands
n.w.cassee@student.tue.nl

Gustavo Pinto
Federal University of Pará
Belém, Brazil
gpinto@ufpa.br

Fernando Castor
Federal University of Pernambuco
Recife, Brazil
castor@cin.ufpe.br

Alexander Serebrenik
Eindhoven University of Technology
Eindhoven, The Netherlands
a.serebrenik@tue.nl

ABSTRACT

Swift is a new programming language developed by Apple as a replacement to Objective-C. It features a sophisticated error handling (EH) mechanism that provides the kind of separation of concerns afforded by exception handling mechanisms in other languages, while also including constructs to improve safety and maintainability. However, Swift also inherits a software development culture stemming from Objective-C being the de-facto standard programming language for Apple platforms for the last 15 years. It is, therefore, a priori unclear whether Swift developers embrace the novel EH mechanisms of the programming language or still rely on the old EH culture of Objective-C even working in Swift.

In this paper, we study to what extent developers adhere to good practices exemplified by EH guidelines and tutorials, and what are the common bad EH practices particularly relevant to Swift code. Furthermore, we investigate whether perception of these practices differs between novices and experienced Swift developers.

To answer these questions we employ a mixed-methods approach and combine 10 semi-structured interviews with Swift developers and quantitative analysis of 78,760 Swift 4 files extracted from 2,733 open-source GitHub repositories. Our findings indicate that there is ample opportunity to improve the way Swift developers use error handling mechanisms. For instance, some recommendations derived in this work are not well spread in the corpus of studied Swift projects. For example, generic catch handlers are common in Swift (even though it is not uncommon for them to share space with their counterparts: non empty catch handlers), custom, developer-defined error types are rare, and developers are mostly reactive when it comes to error handling, using Swift's constructs mostly to handle errors thrown by libraries, instead of throwing and handling application-specific errors.

KEYWORDS

Error handling, Swift, Language feature usage

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196428

ACM Reference Format:
Nathan Cassee, Gustavo Pinto, Fernando Castor, and Alexander Serebrenik. 2018. How Swift Developers Handle Errors. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196398.3196428

1 INTRODUCTION

Error handling (EH) mechanisms are common in modern programming languages. They provide a number of advantages over error codes employed in such languages as C [4], albeit also create complications [6, 25]. Different programming languages implement error handling mechanisms differently: e.g., error handling in C# is driven by maintenance and in Java—by reliability [6].

The Swift programming language, developed by Apple and intended as the replacement for Objective-C, introduces multiple changes in the ways errors are handled as opposed to Objective-C. Not surprisingly, the presence of different error handling constructs in Swift has been reported to hinder adoption of Swift by experienced Objective-C developers [24]. Moreover, as Swift has been evolving, the error handling mechanism has been evolving as well inducing further challenges.

To support Swift developers, both Apple itself and other companies and authors have published multiple guides related to Swift error handling. Many of these popular guides are published as blog posts [22] that include code examples developers learn from [29]. In this paper we study the way those guides affect the way Swift developers use error handling mechanisms. Specifically, we pose the following research questions:

RQ1. To what extent do developers follow the recommendations of popular error handling guides?
RQ2. To what extent do developers avoid the anti-patterns identified by popular error handling guides?
RQ3. How do the perceptions of experienced and novice developers differ about the usage of error handling?

Answers to RQ1 and RQ2 are important to shed some light on the current state of practice of developing error handling code in the Swift programming language. The latter question is motivated by the observation of Shah *et al.* that novice developers often choose to ignore or generalize error handling as they do not consider error handling to be of high priority [28]. This would suggest that novice developers are less likely to follow the recommendations and more likely to write Swift code with anti-patterns. Similarly, experts

might be more likely to follow Swift best practices, but due to its recent introduction (Swift was first released in June 2014 and its error handling mechanism in December 2015), it is still unclear whether experts indeed take advantage of them.

To provide answers to these questions, we apply a convergence mixed-methods approach [9, 23]: quantitative and qualitative research are performed in parallel with the expectation that their results will confirm one another. To conduct the quantitative study we compile a corpus of 2,733 Swift projects obtained from GitHub, the largest corpus used in an error handling study. To conduct the qualitative study, we have interviewed 10 Swift developers. We also manually inspected a sample of 789 commits made by experts and categorized those related to error handling.

Our study curates a list of four error handling recommendations and five anti-patterns. When studying Swift code with these patterns in mind, we observed that 50% of the projects did not employ a single error handling construct; merely 26% of the projects in the dataset employ a `try` variant (*i.e.,* `try`, `try?`, or `try!`). Similarly, only 39% of the projects employ at least one `catch` handler, that is, 1668 projects out of the 2733 either do not handle errors or do so without using `do-catch` blocks. Interestingly, the most often applied recommendation (85%) is related to the use of `try` statements within `do-catch` blocks; for the anti-patterns, the most common one is the use of empty, generic `catch` blocks, *i.e.,* , `catch` blocks that capture every possible error. Although we were able to confirm some of these findings in the qualitative analysis (*e.g.,* interviewees mentioned the use of `try` statements within catch handlers), some findings were not (*e.g.,* we did not find empty generic `catch` blocks in the commit analysis).

The main contributions of our work are as follows:

- A curated corpus of Swift projects for analysis, along with the methodology behind their selection;
- A tool for collecting a metrics related to the usage of Swift error handling constructs in these projects;
- An analysis of the results, considering implications for the practice of software development in Swift.

2 SWIFT ERROR HANDLING

Swift provides numerous ways of specifying and handling errors. Here we briefly discuss some of them. Further information can be found in developer guides discussed in Section 3.1.

From a usage perspective, error handling in Swift is akin to exception handling in Java and C#. Errors are values of types that conform to the `Error` protocol. As a protocol, errors can be implemented as `enum`erations, `struct`ures, or `class`es.The `throw` statement signals the occurrence of an error. A method that signals an error either `catch`es the error, or indicates in its signature that it `throws` an error. Listing 1 shows a common example of how to define errors using an `enum`. In Listing 1, `IOError` is an enumeration with two cases. One of these, `FileNotFound`, is particularly interesting because it carries contextual information that can be employed to capture errors in more specific situations. Since each case is a value that represents an error, it can be `thrown`, as in `readFromFile`. Methods invoking `readFromFile` must either also have a `throws` clause in their signatures or explicitly catch the error. Calls to `readFromFile` must be preceded by the `try` keyword to indicate that they may

Listing 1: Example of defining and manipulating errors

```
1  enum IOError : Error {
2    case FileNotFound(filePath: String)
3    case ConnectionTimedOut
4  }
5
6  func readFromFile(path: String) throws -> String {
7    if (!findFile(path)) {
8       throw IOError.FileNotFound(filePath: path)
9    } else {
10     ... // Start reading.
11   }
12 }
13
14 func processText(filePath: String) {
15   do {
16     let fileContents = try readFromFile(path: filePath)
17   } catch IOError.FileNotFound {
18     print("File not found: \(filePath).")
19   } catch { print(error) }
20 }
```

throw an error (line 16). Function `processText` presents an example of a method that invokes `readFromFile` and catches the `FileNotFound` error (and others).

To catch an error, it is necessary to place the code that may throw errors within a `do` block (line 15) having one or more associated `catch` clauses. For instance, the `catch` clause in line 17 catches `IOError.FileNotFound` errors whereas the one in line 19 catches any other error. We say that a `catch` block whose `catch` clause can capture any kind of error is a *generic catch block* (or a "*catch all*"). Generic `catch` blocks can be implemented by:

1. not specifying the type of the error (as in line 19 of Listing 1);
2. capturing errors of type `NSError`, the default type employed to report errors in Objective-C programs;
3. or capturing every error conforming to the `Error` protocol (this is the verbose version of item 1).

A `do-catch` block must either include at least one generic `catch` block, or the method in which the `do-catch` block is defined should be declared with `throws`. This is enforced by the Swift compiler. In line 19, `error` is a predefined variable that represents the caught error accessible within the block.

Swift has additional constructs for error handling [1]. By writing `try?` before an expression that may throw an error, that expression produces `nil`, instead of its expected result, if an error is thrown. Similarly, `try!` before such an expression converts an error into a runtime error. Programs cannot catch runtime errors and this will cause the application to crash. Finally, for high-order functions in Swift, one should check whether the received closure throws an error. For that, Swift has a special kind of `throws`: marking the signatures of higher-order function with `rethrows` signals that it only throws an error if the received closure `throws` one.

3 METHODOLOGY

In this section we start by identifying popular guides pertaining to Swift error handling (Section 3.1). These guides allow us to refine research questions **RQ1** and **RQ2** by identifying recommendations and anti-patterns. Then, we apply a convergence mixed-methods approach [9, 23] to estimate adherence to these recommendations and avoidance of the anti-patterns. Quantitative (repository mining) and qualitative (interviews) research are performed in parallel as discussed in Section 3.2 and Section 3.3. The interviews also allow

us to answer **RQ3**, *i.e.,* to obtain insights in differences between novices and experienced developers.

3.1 Swift Error Handling Guidelines

As we focus on the impact of guides on the developers' practice, we start by analyzing publicly available guides that discuss Swift error handling. To find which guides are available to Swift programmers we have searched "*Swift Error Handling*" on Google. We use Google rank as our primary popularity criterion since this is the way developers are likely to obtain knowledge about error handling. From the ten highest ranked results, six are guides that extensively discuss error handling in Swift. The remaining four link either to libraries or to questions on Q&A sites. The six guides are:

G-1) The Swift Programming Language, by *Apple Inc.*[1];
G-2) Error Handling in Swift, by *Abhimuralidharan*[2];
G-3) Magical Error Handling in Swift, by *Gemma Barlow*[3];
G-4) Intro to Error Handling in Swift, by *Bob Lee*[4];
G-5) Avoiding Swift Error & Swift Error Handling, by *Mindbrowser*[5],;
G-6) Error Handling in Swift, by *Andy Bargh* [6].

Using these guides we compiled a list of recommendations for handling errors in Swift by doing card sorting, grouping aspects of the error handling mechanism and analyzing what each guide says or shows about the particular aspect. Furthermore, we compiled a list of anti-patterns that indicate potentially wrong usage of the error handling mechanism. Throughout the paper we refer to recommendations by names defined in this section. As observed below the guides tend to agree on recommendations and anti-patterns: we can claim, therefore, that we have reached saturation on recommendations and anti-patterns presented in blog posts. Table 1 summarizes the recommendations and anti-patterns.

According to the guides, Swift projects should declare custom error types to signal exceptional conditions (CustomErrorTypes). G-5 shows an example using a `struct` as an error type; G-1 to G-4 and G-6 recommend using `enum`s as error types. In case a variable needs to be associated with an error condition, these variables can also be defined as part of the error type (cf. Line 2 of Listing 1).

When an exceptional condition occurs, an instance of the error type—possibly instantiated with one or more arguments (cf. Line 8 of Listing 1)—should be thrown (ThrowErrorValues). Due to the straightforward notion of throwing errors in Swift, the only requirement for throwing an error is that the `throw` statement should occur in the context of either a method that is declared with the `throws` keyword or in the context of `do-catch` statements. All guides mention and show examples of this pattern.

The third point that all guides touch on is related to how to call methods declared with the `throws` keyword. Interestingly, no guide recommends using the `try!` operator for error handling. In fact, five guides (G-1 to G-4 and G-6) explicitly mention that `try!` should only be used in rare cases where it is certain that the called method never throws an error. As a consequence, we consider excessive usage of `try!` as an anti-pattern, ExcessiveTry!. When it comes to `try?`, the guides are not unanimous; the Swift programming guide by Apple (G-1) explains the following about `try?`: "*Using* `try?` *lets you write concise error handling code when you want to handle all errors in the same way.*". G-2 does not mention whether `try?` should be used or not—it only mentions its existence. G-4 and G-6 recommend against using `try?`, as `try?` generalizes and ignores errors. All guides recommend, and show examples of, the use of the `try` operator in a `do-catch` block to call methods that are declared with the `throws` keyword (TryWithinDoCatch).

Additionally, every guide explains how errors can be caught using `catch`. The Swift programming guide (G-1) explains how pattern matching can be used to match different error conditions. A small example is featured in every guide on how to catch error values with catch handlers, in five out of the six guides (G-1 to G-4 and G-6), these examples include case matching for `enum` error types. Thus, we consider CatchEnumCases to be the usage of catch handlers that use pattern matching for enum values. Two guides (G-1 and G-6) also provide examples of so called value binding to bind error parameters in the `catch` handler. Lines 17-18 of Listing 1 also provide an example of usage of this approach.

While there are differences in these guidelines, they all describe a common theme to Swift 4 error handling. According to the guidelines, when an exceptional condition occurs, developers should throw (ThrowErrorValues) an error of an application-specific error type (CustomErrorTypes). Methods that throw errors should be called with a `try` statement in a `do-catch` block (TryWithinDoCatch), with one or more catch handlers that pattern match different values of the error type (CatchEnumCases). Meanwhile, `try!` should only be used when it is certain that code will never throw an error during runtime (ExcessiveTry!). When it comes to `try?` there is less of a consensus; some guides (G-1, G-4 and G-6) discourage usage of `try?` while others only explain the semantics of `try?`. However, most guides (G-1, G-3, G-4 and G-6) explain that `try?` is most suited for specific cases when the type of error does not matter and there is no need to distinguish resulting errors. Which is why we consider excessive of `try?` to be an anti-pattern (ExcessiveTry?).

Even though `catch` handlers are an important part of the error handling mechanism of Swift, none of the guides provide information or examples on the contents of `catch` handlers. All six guides showcase toy examples of `catch` handlers where the latter just write information to the standard output. There are no examples of how best to notify the user or log the error.

Complementary to the guides found in the gray literature such as blogs, we investigated recent academic research to identify additional error handling usage patterns that might also be common in the Swift arena. Shah *et al.* [28] have found that novice developers often choose to ignore or generalize error handling as they do not consider error handling to be a high priority task. Signs that projects in Swift ignore or generalize error handling are a lack of any error handling primitives in a project (LackOfEH), and excessive usage of the `try?` operator, as the `try?` operator swallows all errors that occur (ExcessiveTry?). Furthermore Ebert *et al.* [10] found that error handling bugs in Java projects can be caused by `catch` handlers that swallow errors or empty `catch` blocks. Therefore we also consider empty `catch` handlers that catch specific errors

[1]https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ErrorHandling.html#//apple_ref/doc/uid/TP40014097-CH42-ID508
[2]https://medium.com/@abhimuralidharan/error-handling-in-swift-d0a618499910
[3]https://www.raywenderlich.com/130197/magical-error-handling-swift
[4]https://www.bobthedeveloper.io/blog/intro-to-error-handling-in-swift
[5]http://mindbowser.com/error-handling-in-swift/
[6]https://andybargh.com/error-handling-in-swift/

Table 1: EH recommendations and anti-patterns.

Recommendations
CustomErrorTypes: Use custom error types to signal errors.
ThrowErrorValues: In the presence of errors, throw an error type.
TryWithinDoCatch: Methods that throw errors should be called with a `try` statement in a `do-catch` block.
CatchEnumCases: One or more `catch` handlers with pattern case matching on different values of the error type.
Anti-Patterns
ExcessiveTry!: Excessive use of `try!`.
LackOfEH: Lack of error handling code in a project.
ExcessiveTry?: Excessive usage of `try?` to swallow errors.
EmptyCatch: Empty `catch` blocks that match specific error(s).
EmptyGenericCatch: The use of empty generic `catch` blocks.

(EmptyCatch) and empty generic handlers (EmptyGenericCatch) as anti-patterns. Since the generic and non-generic empty `catch` handlers can be used for different scenarios, we consider EmptyCatch and EmptyGenericCatch to be different anti-patterns: an empty `catch` handler that catches a specific error silences that error, while an empty generic `catch` handler silences all errors.

3.2 Mining Swift repositories

Next we discuss identifying representative Swift projects (Section 3.2.1) and error handling metrics we apply (Section 3.2.2).

3.2.1 *Finding Representative Swift Projects.*

Selecting a sample of active GitHub projects. To analyze how Swift developers employ the error handling mechanisms of Swift, we start by mining Swift Projects using GHTorrent, a third-party publicly accessible mirror of GitHub [13, 14]. As of November, 1st 2017, we found a total of 402,046 projects that were recorded as a Swift project by GitHub. When analyzing these projects, we found that some of them do not present recent activity. Therefore, we filter out projects with no commits after Jan 1st, 2016. With this additional constraint, we found a total of 150,675 Swift projects. The 150,675 recently active Swift projects however were still too many to analyze in depth, which would make a comprehensive source code analysis unfeasible. Therefore we selected a random sample of those projects. Since prevalence of the error handling constructs was not known *a priori* we had to assume the prevalence of 50%, maximizing the sample size for the given confidence level. For the confidence level of 99% and a confidence interval of 1.32, the sample size has been determined to be 9,000 projects[7].

Identifying engineered projects in the sample As Kalliamvakou *et al.* [16] have shown, a large amount of software projects on GitHub are inactive and/or personal. To select engineered software projects as opposed to personal or naive ones, we applied Reaper, a classification framework introduced by Munaiah *et al.* [19]. Reaper extracts a set of metrics from a project based on, amongst other things, size of the project and history. Using Reaper, Munaiah *et al.* achieved a precision of 82% and a recall of 86% on their "utility" dataset, *i.e.,* the dataset based on the definition of engineered projects as projects that "have a fairly general-purpose utility to users other than the developers themselves" [19]. To apply Reaper to Swift projects, we had to extend Reaper to also be able to score Swift projects with respect to the six dimensions identified by Munaiah *et al.* : community, documentation, history, testing, licenses and continuous integration. These six dimensions closely match the original dimensions outlined by Munaiah *et al.* [19].

From the 9,000 Swift repositories, 2,801 projects have been classified as engineered software projects. It is important to note that these projects also include forks. However, we chose not to exclude forks as we consider the forks to be interactions of developers with Swift source code. In addition, as recent work suggests [34], forks not only present active development, but can also have different features, when compared to the original project.

Reaper dimensions in the engineered Swift projects We have observed that among 2,801 engineered projects 41% of the projects are configured to use *continuous integration* and that 37% of the Swift projects do not state any *license*.

Figure 1 shows distributions of the metrics representing four Reaper dimensions over 2,801 engineered projects. The *community* metric counts the smallest set of contributors whose total number of commits accounts for 80% or more of the total contributions. As we can see, the majority of studied projects have up to two core contributors. The *documentation* metric calculates the share of the commented lines of code among the non-blank lines of code. The *history* metric calculates the average number of commits per month. We removed the outliers from this figure to ease comprehension. On average, our studied projects perform 5.3 commits per month. Finally, the *testing* metric computes the ratio between the amount of testing code and the overall number of source lines of code. Although some projects have an extensive testing code base (three projects have a test ratio greater than 0.9), on average, the projects have a test ratio of 0.16. We did not analyze whether there were specific tests for error handling code.

3.2.2 *Metrics extraction.*

To analyze the source code of the 2,801 Swift projects, we cloned all of their git repositories. Two tools have been used to analyze these projects, cloc[8] has been used to count which programming languages are used in the projects, and how many lines of code each project has. Moreover, a custom abstract syntax tree analyzer, metric extractor, has been developed to extract the error handling information from the source code[9]. To extract syntactical information from the corpus metric extractor parses all Swift source code files with an open-source Swift 4 parser, SWIFT-AST[10]. For a given repository the metric extractor uses the abstract syntax tree returned by SWIFT-ASTto extract information related to how Swift developers (1) catch errors; (2) call methods that potentially throw exceptions; (3) throw and/or define error types.

The two leading dependency managers for Swift, CocoaPods and Carthage, both recommend checking the source code of dependencies into git[11][12]. As the location to which both CocoaPods and Carthage download dependencies is standard, we have removed all of these directories from the repositories, using the pattern

[7] https://www.surveysystem.com/sscalc.htm

[8] http://cloc.sourceforge.net/

[9] https://github.com/TheDutchDevil/parser_wrapper

[10] https://github.com/yanagiba/swift-ast

[11] https://guides.cocoapods.org/using/using-cocoapods

[12] https://github.com/Carthage/Carthage/blob/master/Documentation/Artifacts.md#carthagecheckouts

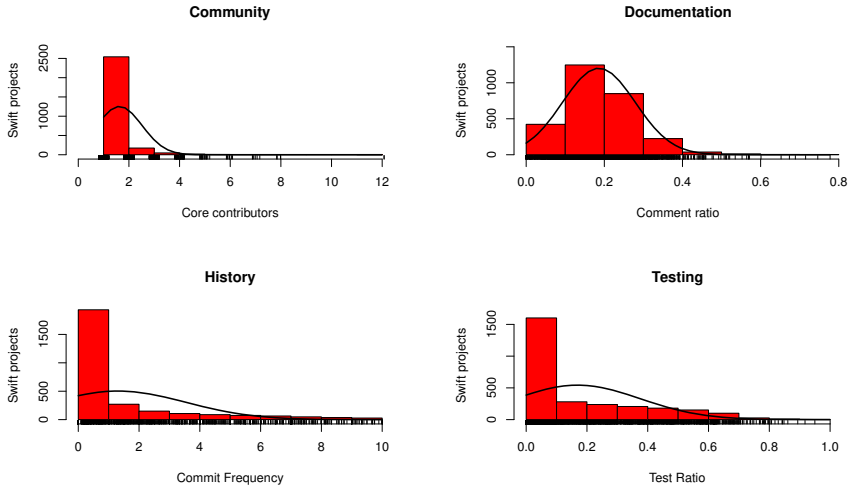

Figure 1: Characteristics of the studied project, using reaper metrics.

Table 2: Overview of the metrics extracted.

Category	Metrics
Calling EH code	Number of `try`, `try!`, `try?` and `do` statements
Defining Error types	Number of `enums`, `classes` and `structs` that inherit from `Error`
Throwing errors	Number of `throw` statements, number of methods declared with `throws` and `rethrows`
Catching errors	Number of `catch` clauses that catch an `enum`, catch all, and catch neither an `enum` nor all `Error` types.

`*/Carthage/Checkout` and `*/Pods`. After removing the dependencies from the Swift repositories there are a total of 149,154 code files as counted by cloc, and a total of 84,982 Swift files for which we have attempted to extract information related to error handling.

Table 2 shows the metrics studied. These metrics are based on the identified guidelines (Section 3.1). In total, from the 2,801 Swift projects, we extract error handling metrics for 2,733 projects and 78,760 Swift 4 files. A more in depth description of the tools used to select the projects and parse the Swift source files is available online [13]. We refer to the collection of 2,733 projects as our *corpus*.

3.3 Interviews

We conducted semi-structured interviews with Swift developers to understand why and how do they use Swift's error handling constructs. Conducting interviews is commonly performed either as the main [28] or complementary [10, 24] method in software engineering research. We used a convenience sampling approach to recruit developers that use Swift. We started searching at LinkedIn for practitioners that have professional Swift experience. We also shared our invitation on two social networks: Twitter and Facebook. We also searched for conference speakers on Swift conferences. Finally, we contacted the students of the Apple Developer Academy, a 2-year educational program where undergraduate students receive

[13] https://gist.github.com/TheDutchDevil/31d2b54420ffab0d798a26c0b8fe2516

Table 3: Demographics of our interviewees.

ID	Location	Years of prof. exp.	Category
P1	Europe	10+ years	Expert
P2	Europe	6 years	Expert
P3	Asia	6 years	Expert
P4	South America	8 years	Expert
P5	North America	4 years	Novice
P6	South America	6 years	Novice
P7	South America	3 years	Novice
P8	South America	3 years	Novice
P9	South America	2 years	Novice
P10	South America	1 year	Novice

extensive education in app development. At the end, we interviewed 10 Swift developers (4 experienced Swift developers, and 6 novices).

Eight interviews were conducted via video calls (the remaining two ones were conducted via e-mail; the participants had very low time availability to be interviewed). The interviews were recorded, and audio was transcribed. On average, the interviews lasted about 40 minutes. At the beginning of the interview, we explained the purpose of the research, and sought permission to record the interview and to share the data anonymously. To ensure participant anonymity, they are identified as P1–P10. Table 3 shows some demographics of the participants. All the South-American interviewees are Brazilian and their interviews were conducted in Portuguese and later translated to English.

In the interviews we focused on four main topics: (1) the interviewees' perception of error handling in general, and in Swift in particular, (2) the reasons for the interviewee to use error handling in swift, (3) the reasons to avoid using EH, (4) the pitfalls and challenges they face when dealing with error situations. Each interview transcript, along with the associated recording, was analyzed by one of the authors following the card-sorting procedure [30]. The emerging topics are discussed in Section 4 along with quotes from the interviews. Among similar opinions, we chose to quote only the one we considered the most representative for each case.

Table 4: The division of Swift projects into the three categories. SLOC refers exclusively to Swift code.

Category	% projects	# projects	Mean SLOC	Median SLOC
noEH	50.71%	1386	1066.5	342.0
cEH	29.56%	808	1860.0	926.0
sEH	19.61%	536	6756.1	2475.0

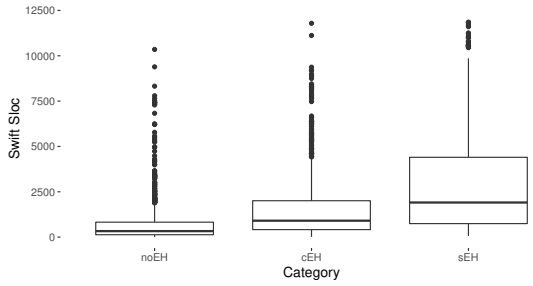

Figure 2: Projects that both consume and throw errors (sEH) tend to be larger (SLOC) than those that only consume errors of APIs (cEH) and the latter than those that do not use any error handling (noEH).

We complement the findings of the interview with a manual analysis of commits made by experts and novice Swift developers that touch error handling code. To identify experts, we followed a simplified version of the technique presented by Mo *et al.* [18]: for each project in the dataset, we selected the top-10 contributors (*i.e.,* the contributors that made the most commits in a given project). To avoid counting the same commit scattered in forked projects, for this analysis, we removed fork projects from the dataset. After the identification of the top-10 contributors, we analyzed other Swift projects that these users have contributed to. The selected experts have contributed significantly to at least four additional Swift repositories (4 is the third quartile of contributions to different repositories). Using this approach, we found six Swift experts. We then manually investigated the 789 commits they performed and categorized the 223 ones that touch error handling code.

4 RESULTS

This section presents the results obtained from both repository mining and interviews.

4.1 Error handling recommendations in practice

To analyze the error handling usage of the 2,733 projects in the corpus, the projects are split into 3 non-overlapping groups, namely:

(1) **noEH**: projects that do not use any of the error handling primitives available in Swift;
(2) **cEH**: projects that only use error handling primitives to consume APIs that throw errors, and;
(3) **sEH**: projects that use error handling to both consume error throwing APIs and use the error handling mechanism to throw errors.

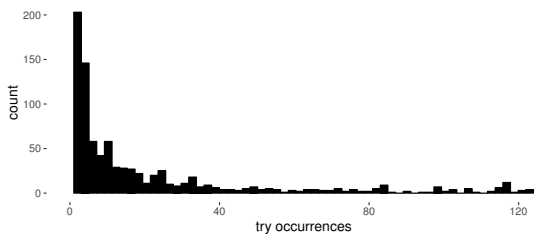

Figure 3: Distribution of `try` usage in projects with at least one `try` operator. For the sake of readability 76 outliers with more than 125 try statements are not included in this plot.

Table 4 shows how many projects fall into each group. Most of the projects do not use any of the error handling primitives of Swift to consume or signal errors; less than a third just consume and do not signal any errors. Ultimately, one in five projects uses the error handling mechanism of Swift to actively signal errors. Figure 2 shows the distribution of the source lines of code (SLOC) of the Swift projects in each category. Statistical comparison using the $\widetilde{\mathbf{T}}$-procedure [17] shows that sEH projects are larger than cEH, and cEH are larger than noEH . We apply $\widetilde{\mathbf{T}}$ as it is robust against unequal population variances, respects transitivity, does not suffer from well-known problems of two-steps approaches [11] (such as ANOVA followed by pairwise *t*-tests or Kruskal-Wallis followed by pairwise Mann-Whitney tests), and has been successfully applied in empirical software engineering [31–33].

Out of 1,344 projects that utilize Swift error handling (cEH and sEH together), only 725 use any `try` variant (*i.e.,* `try`, `try?`, `try!`) more than five times (cf. Figure 3). Therefore, even in the projects that consume errors overall consuming errors is not common. One example of a project heavily using `try` statements is Casperhr/engine with 6,041 Swift Sloc and 595 `try` instances.

TryWithinDoCatch recommends using the `try` statement in a `do-catch` block. However, out of the 1,344 projects that employ error handling, there are 190 projects that do not use `try` to call methods that potentially throw errors. Among the 1,154 projects that use the `try` operator, half have less than 7 try statements (cf. the distribution of `try` statements in Figure 3). Still according to TryWithinDoCatch a `do-catch` block should be used for error handling. However, among the aforementioned 1,344 projects, we found that 272 projects do not contain any `do` statements. More interestingly, these 272 projects call code that signals errors, with `try?`, `try!`, or by declaring methods that have the potential of throwing errors. For instance, out of the 272 projects that do not contain any do statements, there are 19 projects that do not throw any errors, but declare methods that can throw errors. Manual inspection on these projects reveals that 17 of these 19 projects use the `try` operator to call methods that might throw errors. These 17 projects hence propagate any resulting errors up the call stack, without intercepting them, *i.e.,* not using any aspects of the error handling system of Swift to catch or report on errors, more importantly, not following recommendations CustomErrorTypes and ThrowErrorValues.

Out of the 1,344 projects that utilize Swift error handling, 1,065 contain at least one `catch` handler. The recommendations suggest

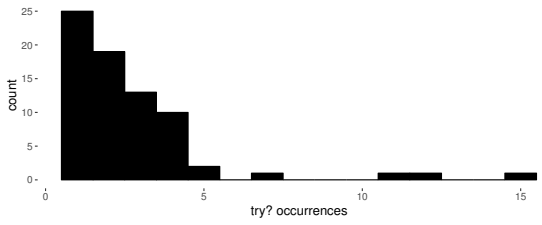

Figure 4: Distribution of `try?` occurrences in projects that only use the try? operator for error handling

using case matching on the elements of an `enum` to catch specific errors (CatchEnumCases). However, out of the 1,065 projects, only 124 projects use one or more `catch` handlers with `enum` case matching. This finding indicates that even though projects are aware of and adhere to some recommendation (*e.g.,* TryWithinDoCatch), they do not necessarily follow all of them (*e.g.,* CatchEnumCases). The guidelines for error handling in Swift mention that custom `enum` values should be thrown when exceptional conditions occur (ThrowErrorValues). Nonetheless, only 536 Swift projects follow ThrowErrorValues and actually throw errors. The median of errors thrown per KSloc for these projects is 2.88, i.e., on average, approximately one line of code throws an error out of every 350, indicating that if a project throws errors, it does so regularly. However, not all projects that throw an error also declare methods that throw errors. In fact, 27 projects have one or more `throw` statements but do not declare any methods that throw errors. Therefore, `throw` statements in these projects occur in the context of a `do-catch` statement and errors are never propagated up the call stack.

While almost all guides provide an example of the declaration of an `enum` type that inherits from the `Error` protocol, and all guides recommend the declaration of types that inherit from `Error` (CustomErrorTypes), 182 projects out of the 536 projects in sEH do not declare any error types. Among the remaining 354 that do declare at least one error type, 210 do not declare any `enum` type that inherits from `Error`.

4.2 Error handling anti-patterns in practice

The guidelines for Swift developers regarding the usage of `try`, `try?` and `try!` specify that `try!` statements should not be used extensively (ExcessiveTry!). In order to compare the usage of the different kinds of `try` statement, we investigated their occurrences in the 725 projects with more than 5 `try`, `try?`, or `try!` statements. We found that 10.76% of these projects use `try!` in at least half of the all cases to call code that signals errors, a clear anti-pattern.

Another anti-pattern for handling errors in Swift is ignoring errors. One way of ignoring errors is by calling a method that throws errors with the `try?` operator. Verifying whether the call succeeded or failed can be done by checking whether the returned value is `nil`. However, if `try?` is used, all information regarding the error (*e.g.,* the error message) is lost. This makes it nearly impossible to handle different errors based on their type, value, or parameters (ExcessiveTry?). Out of the 1,344 projects that use error handling, there are 569 that use the `try?` operator at least once, and a total of 73 projects that only use the `try?` operator.

However, Figure 4 presents a different perspective. It shows that most of these projects do not use the `try?` statement excessively. In fact the majority of these projects rarely uses the `try?` statement, as opposed to using `try?` to ignore many errors. The projects that only use `try?` for error handling rarely use other features of the Swift error handling mechanism. This finding suggests that these projects do not follow recommendations CustomErrorTypes and ThrowErrorValues. Only 6 of the projects actually declare error types, declare methods that throw errors, or actually throw errors.

While Swift error handling guidelines recommend the usage of `catch` handlers (CatchEnumCases), empty `catch` handlers are potentially dangerous (EmptyCatch). Out of the 267 projects that have at least one `catch` handler that is not a catch all *i.e* a specific catch handlers, 108 projects have at least one empty specific catch handler. These empty `catch` handlers effectively swallow the error(s) that match the pattern of the `catch` clause. Among these 108 projects with empty `catch` handlers, 43 only have empty handlers. It is important to note that these 43 projects are not perfect examples of how not to use error handling; most of them employ some of the recommendations, including 18 projects that throw errors (ThrowErrorValues) and 11 projects that declare error types (CustomErrorTypes). Moreover, all of these 43 projects use `try` in some form (TryWithinDoCatch). While an empty `catch` handler is a clear anti-pattern, it appears that in several projects these are used in combination with other recommended error handling characteristics as can be observed in Table 5.

In addition to empty `catch` handlers, empty generic `catch` handlers are also an anti-pattern (EmptyGenericCatch), since empty generic `catch` handlers silently swallow all errors. This kind of error handler can be rewritten by means of `try?`, which better captures the intent of the code. Out of the 1,344 projects that use error handling, 1,025 projects use at least one generic `catch` handler. Out of these 1,025 projects 230 projects have at least one empty generic `catch` handler and in 26 projects all generic `catch` handlers are empty. However, from the 230 projects with at least one empty generic `catch` handler, 153 projects use no specific `catch` handlers, and therefore only use generic `catch` handlers. From these 153 projects, 20 projects have only empty `catch` handlers, indicating that these 20 projects swallow all errors that occur.

4.3 Novices and experts

Previous studies have investigated how developers perceive the importance, challenges, and implementation techniques of error/exception handling code [2, 10, 27, 28]. The uniqueness of the Swift error handling constructs fostered the need to update this perception. Generally speaking, our interviewees were aware of the existence of additional error handing features in Swift. Moreover, all of them have recently employed or refactored some code related to error-handling. However, some technical background is still lacking, in particular, on the novice side.

This claim can be exemplified in difference situations. For instance, when we asked for scenarios in which our participants avoid using EH constructs, P10 mentioned that "*I believe I did not get to the top of the learning curve in relation to error handling and many times when I write this type of code I get many compilation errors*". This interviewee believes that one of the main reasons that makes error

Table 5: Sub-division of catch handlers usage over different projects. Indentation in the lines of the table is used to indicate to which group a line belongs.

Projects	Category
1344	**sEH and cEH**
267	Projects with more than one specific catch handler
108	Projects with at least one empty specific catch handler
43	Projects with only empty specific catch handlers
43	Conforms to TryWithinDoCatch
18	Conforms to ThrowErrorValues
11	Conforms to CustomErrorTypes
1025	Projects with more than one catch all handler
230	Projects with more than one empty catch all
26	Only empty catch alls
153	Only catch alls
20	Only empty catch alls

handling in Swift difficult is that it is not intuitive: "*I do not think it is intuitive. For me a programming language is intuitive when I can start writing the code and 'hit the syntax' as a logical consequence of the rules and examples I saw earlier. In the case of Swift error handling, I always end up confusing some keywords or not knowing if I should put* `?` *or* `!`*. It is also common to get compilation errors whose recommended fix does not solve the problem.*" The problem reported by this interviewee can be considered as a mix of the anti-patterns related to the use of `try!` (ExcessiveTry!) and `try?` (LackOfEH).

As a comparison, expert developers avoid EH constructs in a more conscious way, as P2 mentioned: "*Its good to handle the errors in the code, but it is not a compulsion to handle each and every error with the custom error handler.*". This statement goes along the lines of ExcessiveTry!. However, as the interview went further, it became clear that excessive use of error handling in general, not necessarily only the `try!` statement, can be harmful: "*As long as the errors do not affect the user experience and performance of an app, the user will not want to be informed about them*". The respondent complemented this answer with examples of particular cases where error handling can be avoided: "*a) Error occurred while sending analytic events to the remote server; b) App failed to connect to the third party service like ad services etc; c) Failure in image cache on disk.*"

Similarly, when we asked what are the reasons to ignore error handling, P2 suggested that "*When we parse the information received from the server, there may be error occurred in JSON parsing. We can handle the parse error by try?. Indicates, ignore error and allow the variable to assign with nil, whenever function throws them.*" By ignoring errors, this interviewee introduces an anti-pattern (ExcessiveTry?). Novice developers often mentioned not ignoring EH. Only one novice interviewee, P10, went in the opposite direction, stating that "*when I am working on a temporary solution, I leave some variables hard-coded or use force-unwrapping*".

Moreover, regarding the importance of error handling code, both experts and novice developers concur on the benefits brought by error handling mechanisms. One novice programmer highlighted the particular need for error handling. "*One of my projects made many calls to the server and due to the lack of error handling, the application ceased responding when there was a problem in those calls.*" Likewise, our respondents also mentioned scenarios where error handling is not particularly relevant. P5 develops an iOS framework and, in extreme cases, if developers that use the framework violate some hard constrains that they are not supposed to, internally, the framework uses a `try!` statement to raise a runtime error. Note that, differently from ExcessiveTry!, which highlights the excessive use of `try!`, this respondent consciously uses this statement in extreme scenarios. As one interviewee suggests "*if nothing else can be done, no error needs to be caught*".

When we asked what could be improved in the EH mechanisms, two experts provided suggestions. One expert (P4) mentioned a limitation particularly relevant to the product the interviewee works on. This expert develops an SDK organized in different modules. Each module is managed by a single thread. In the Android version of the SDK, when an error occurs, the interviewee needs to know which thread was running the module, so then the interviewee can finish the execution of this single module, while keeping the other ones running. The interviewee highlighted that "*as far as I know, in Swift there is no way to catch the thread id that had an error*". In the Android version of the SDK, the interviewee does that using the `Thread.UncaughtExceptionHandler` class. One novice programmer mentioned that "*At least compiler errors could be better*".

In addition to the interviews, we have also analyzed the error handling code in 789 commits authored by experienced contributors to projects in the corpus. Among the commits, 223 modify lines of code that are related to error handling; 177 out of these either add, remove, or update code that uses error handling constructs. However, 50 commits modify lines of code related to error handling but do not modify error handling code itself. *e.g.,* 34 commits renamed variables used within catch handlers, and 8 commits modified whitespace. More interestingly, the remaining 173 commits actually modify error handling code. Among these commits, 14 modify error handling to account for changes in external code *e.g.,* a method changing its signature to signal that it now throws errors.

The remaining 159 changesets show that the experienced Swift developers in general follow the recommendations as laid out by the guidelines. Especially CustomErrorTypes (declare `enum` error types), ThrowErrorValues, and TryWithinDoCatch (call methods that throw errors with `try`) are adhered to by the experienced developers in the analyzed repositories. All of the 16 projects for which we analyzed the commits of experienced developers are libraries. Therefore, instances of CatchEnumCases (write `catch` handlers that match enum case patterns) are relatively rare, as most projects cascade errors up the call stack.

Anti-patterns were rarely found in the changesets of experienced developers. There are several instances of generic `catch` handlers. However, these handlers are never empty, and therefore do not match EmptyGenericCatch. Additionally, the usage of `try!` and `try?` operators is sparse and appears to be aimed at specific cases. Listing 2 is an excerpt from a changeset[14] by kylef to kylef/Stencil where line 1 is replaced by line 2. In this excerpt, old Objective-C style error handling (passing an error parameter) is replaced by a more appropriate Swift error handling primitive. Line 2 shows usage of the `try!` operator as recommend by Apple, to call code

[14] https://github.com/kylef/Stencil/commit/dcf2611ac24829ffe80cf46894ebc98fdda62e0c#diff-87b77d847adec9ff08f07ca050ac9519

Listing 2: Snippet of a changeset made by a Swift expert to the kylef/Stencil project. It shows the usage of `try`!

```
1 -  let regex = NSRegularExpression(pattern: "
       (\\{\\{.*?\\}\\}|\\{%.*?%\\}|\\{#.*?#\\})", options: nil,
       error: nil)!
2  +  let regex = try! NSRegularExpression(pattern: "
       (\\{\\{.*?\\}\\}|\\{%.*?%\\}|\\{#.*?#\\})", options: [])
```

that is guaranteed not to fail, as the regular expression pattern has probably been tested during development and is valid.

5 DISCUSSION

In this study we found that roughly half of the 2,733 GitHub projects analyzed do not use the EH mechanism provided by the Swift programming language. The definitive explanation regarding the low adoption of error handling mechanisms is still lacking. However, results of the interviews show evidence that both novice and experienced Swift developers report problems with the error handling mechanism. While novices programmers find it counter-intuitive or confusing, experienced developers describe practical limitations of either the error handling mechanism or the Swift compiler.

These problems related to Swift tooling are not new. An earlier study by Rebouças *et al.* [24] found that, for developers that used Swift 2, problems with the Swift compiler used to be common. Additionally, although there are extensive guidelines on the subject, Rebouças *et al.* found that developers still rely on StackOverflow to ask questions about the error handling mechanisms of Swift 2.

Moreover, the error handling guides used in this study to derive the recommendations and anti-patterns do not extensively discuss advanced use cases of error handling. For instance, `catch` handlers are discussed only with contrived examples (*e.g.*, to call a print function). However, none of the guides discuss how `catch` handlers could also be used to deal with, for instance, unreliable networks or how to properly present an error to the application user. This lack of in-depth explanation might suggest that there is a disconnect in the information available online for Swift developers when it comes to error handling. This knowledge gap explains why novice developers consider the error handling mechanism to be counter-intuitive and why we find that half of the Swift projects do not use any error handling. Moreover, this lack of explanation might reflect inherent limitations of the error handling mechanism, as more experienced users cannot express the error handling logic that they are used to in other languages.

Analysis of the Swift projects that use error handling shows that several of the recommendations are adhered by Swift developers. Especially when it comes to calling code that throws errors using `try` (TryWithinDoCatch); almost all projects that use error handling adhere to this guideline. While not all projects that use error handling also throw errors, 19.61% of the projects follow ThrowErrorValues and throw errors. Since this is a common exception handling practice, we hypothesize that developers that employ these recommendations might be motivated by the old culture of other programming languages. Ultimately, Swift developers are not likely to follow recommendations regarding the declaration of custom error types (CustomErrorTypes) and the recommendation regarding catch handlers (CatchEnumCases). Consequently, most anti-patterns found center around swallowing or ignoring errors. Either through `try?` (ExcessiveTry?) or through empty generic `catch` handlers (EmptyGenericCatch).

In a similar vein, out of the 1065 that have at least one `catch` handler, 798 exhibit generic `catch` handlers exclusively. The use of generic `catch` is not among the identified anti-patterns. In fact, Swift requires a generic `catch` whenever a method may throw errors but does not include `throws` in its signature. Nonetheless, recommendation CatchEnumCases suggests that `catch` handlers should be specific, capturing errors by pattern matching on the error values. Previous work [8, 21, 25] has shown that generic `catch` handlers are a potential source of errors, due to the possibility of capturing errors accidentally, by means of type subsumption.

Low adoption of these recommendations and the occurrence of the anti-patterns could be attributed to the fact that this area is less discussed by the guides on Swift error handling. Therefore, improvements in how these constructs are explained to Swift developers might increase adoption of these recommendations and boost the overall code quality of Swift code. These findings come as an opportunity for tool builders that can create or recommend refactoring tools that might ease the adoption of Swift error handling.

6 LIMITATIONS AND THREATS TO VALIDITY

Fist, mining GitHub as done in this study presents a threat as a large part of the projects on Github is inactive and/or personal [16]. We mitigate this treat by selecting only active and engineered GitHub projects. We have used the Reaper framework [19]. However, Reaper is not 100% accurate (the authors reported a precision of 82% and a recall of 86%). There might be valid software engineering projects which we have excluded (false negatives) while we might also have reported on GitHub projects that do not represent engineered projects (false positives). However, in such large scale corpus of data, we do not expect major distortions in the presence of false positives. Second, we have analyzed a corpus of publicly GitHub projects, however, =other source code hosting platforms exist. Therefore our results might not generalize to the wider Swift community.

Another limitation is regarding the parser used (SWIFT-AST). Using this parser, we were unable to parse all Swift files in the 2,733 parseable repositories. This happens due to several reasons. Over the past three years, several versions of the Swift programming language and its standard library have been released, and, surprisingly, none of the major releases have been backwards compatible. As the 2,733 projects in the dataset represent a wide spread of Swift projects there are still repositories written in Swift 2 or 3, leading to situations where some files cannot be parsed by the Swift 4 compiler. Secondly, even if a project is written in Swift 4 there is no requirement that all Swift 4 files in the repository should contain syntactically valid Swift 4 code. Therefore, even for Swift 4 projects, there are unparseable files . Finally, there are several known issues in SWIFT-AST, which means that some syntactically correct Swift 4 source files cannot be parsed by SWIFT-AST. To deal with these limitations of parsing Swift source files, we have extended the metrics extractor to report on the amount of files that cannot be parsed. If SWIFT-AST fails to parse a file, the file is fed to the Swift 4 compiler, to test whether the code is syntactically valid. If the Swift 4 compiler cannot parse the file, then the file is not syntactically

valid for Swift 4 code. If the Swift 4 compiler parses the file without failure then this file cannot be parsed by SWIFT-AST because of a bug. Thereby rendering it impossible for the metric extractor to extract metrics from the file. However, a syntactically valid file that could not be parsed by SWIFT-AST does not necessarily have to contain error handling code. Therefore, we search for EH primitives using the following regular expression: `(do\s*{)|(try!)|(try\?)`. If this pattern is not matched then it is highly likely that the file contains no error handling code. Consequently, if there is at least one match of the pattern in the file the failure is recorded as a parser failure where potentially error handling code has been missed. If the pattern is not matched, then the file is marked as a parser error where no error handling code has been missed.

Out of the 84,982 Swift source code files, 78,760 ones have been parsed by the metrics extractor. From the 6,222 files that could not be parsed, 2,641 could not be compiled by the Swift compiler; i.e., these 2,641 files do not contain syntactically valid Swift 4 code. The remaining 3,581 files are syntactically valid. When applying our search with the regular expression, we found 723 files that potentially contain error handling code (false positive are possible, e.g., a `try!` statement that have been commented out). However, these files account for only 0.8% of the total files investigated.

Some limitations pertain to the qualitative part of this study. The guides found are likely to change in a near future, as new guides are written and become popular. Due to the rapidly changing nature of Swift and programming practices in general we preferred professional (*e.g.,* the Apple developer guide) and popular guides (the ones that were top ranked in a Google search) to books and MOOCs. Manual analysis of the content was conducted by individual authors (*e.g.,* analysis of the guides and interview data). We adhered to the best practices in qualitative research methods as *e.g.,* defined for card sorting. During the process, we also reviewed the preliminary findings with the other authors.

Finally, one might suggest that some of our recommendations or anti-patterns are well-established in the error handling community (*e.g.,* with `TryWithinDoCatch`, we suggest that methods that throw errors should be called with a `try` statement). However, for novice programmers such well-established recommendations might not be straightforward to understand. As we found in our interview, novice developers face problems related to use of `try!` or `try?`, as well as compiler errors when implementing error handling code. Therefore, such guidelines are still useful.

7 RELATED WORK

Swift usage. Despite the crescent adoption in the software development arena, the Swift programming language received limited attention from the research community. By implementing the same functionality in Swift and Objective-C, González García *et al.* observed that Swift is less verbose than Objective-C due to changes in keywords and access operators, as well as lack of pointers [12]. Rogers *et al.* [26] discussed how Swift can be used in the educational context. In a previous effort, some of us investigated the most common problems faced by Swift developers [24]. We found that optionals and error handling are among the most common ones. However, none of these studies discuss how Error Handling constructs are being used in the Swift programming language.

Error/Exception Handling. Among the contributions in this area, researchers have been investigating how software developers use error and exception handling in different programming languages, such as Java, C, C++, C#, and PHP [2–5, 15, 20, 21]. Another branch of study is related to understanding how error handling techniques impact different concerns, such as modularity and reuse [7], evolution [5], and even bugs [10]. Other studies were interested in understanding the perception of developers on the use of error/exception handling mechanisms [27, 28]. This work differs from the literature in terms of 1) its focus on the Swift programming language, which is not only new in the software development arena, but also introduces additional features to the traditional error handling mechanisms, and 2) its mixed-methods approach, which relies on large scale corpus of Swift projects and in a set of interviews with practitioners, grouped by experience.

8 CONCLUSIONS

In this paper we report on the error handling patterns of 78,760 Swift source files from 2,733 open source Swift projects, the experiences of 10 Swift developers with the error handling mechanisms of Swift, and the qualitative analysis of 789 changesets authored by expert Swift developers. Among the findings, we observed that while error handling guides provide different recommendations about using Swift error handling mechanism, 50.71% of the projects do not exhibit any error handling code. However, the Swift projects that do employ the error handling mechanisms tend to follow some of the recommendations of the guidelines, for instance, 85.86% of the projects that employ error handling follow guidelines on how to call code that throws errors. This percentage drops to 24.70%, when considering more complex recommendations, such as declaring a custom `enum` error. Several projects do contain anti-patterns explicitly discouraged by guides; we find that 11.24% of projects exhibit anti-patterns related to the usage of `try!` or `try?`. Similarly, 22.84% of the projects contain an anti-pattern related to catch handlers.

Most of the interviewees mention problems when using Swift's error handling mechanisms. Novices mention confusion and experienced developers mention technical limitations they encountered. This is corroborated by the fact 50.71% of the projects do not utilize any of the error handling mechanisms of Swift. When investigating commits made by experts related to error handling, we rarely observed the introduction of anti-patterns in Swift code. Moreover a thorough analysis of Swift error handling guides found that all guides discuss the basics of Swift error handling. However, the examples used in the guides are contrived and adequate real world examples of how to deal with errors are not given.

As *future work* we intend to conduct a survey with Swift developers to understand the reasons for introducing some of the anti-patterns described. We also plan to create static analysis tools to automatically identify the anti-patterns. As a complement of this tool, refactoring and recommendations tools can also be proposed. These tooling might be useful for novice and experts alike, when dealing with error handling in Swift.

ACKNOWLEDGMENTS

This work is supported by CNPq (406308/2016-0);

REFERENCES

[1] Apple Inc. 2018. The Swift Programming Language 4.1 – Error Handling. (February 2018). https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ErrorHandling.html

[2] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K. Roy, and Kevin A. Schneider. 2016. How Developers Use Exception Handling in Java?. In *MSR*. ACM, 516–519.

[3] Rodrigo Bonifácio, Fausto Carvalho, Guilherme N. Ramos, Uirá Kulesza, and Roberta Coelho. 2015. The use of C++ exception handling constructs: A comprehensive study. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015*. Bremen, Germany, 21–30.

[4] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. 2006. Discovering faults in idiom-based exception handling. In *28th International Conference on Software Engineering (ICSE 2006)*. Shanghai, China, 242–251.

[5] Nélio Cacho, Eiji Adachi Barbosa, Juliana Araujo, Frederico Pranto, Alessandro F. Garcia, Thiago César, Eliezio Soares, Arthur Cassio, Thomas Filipe, and Israel García. 2014. How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C# Applications. In *ICSME*. 31–40.

[6] Nélio Cacho, Thiago César, Thomas Filipe, Eliezio Soares, Arthur Cassio, Rafael Souza, Israel García, Eiji Adachi Barbosa, and Alessandro Garcia. [n. d.]. Trading robustness for maintainability: an empirical study of evolving c# programs. In *ICSE*. 584–595.

[7] Fernando Castor, Nélio Cacho, Eduardo Figueiredo, Alessandro Garcia, Cecília M. F. Rubira, Jefferson Silva de Amorim, and Hítalo Oliveira da Silva. 2009. On the Modularization and Reuse of Exception Handling with Aspects. *Softw. Pract. Exper.* 39, 17 (Dec. 2009), 1377–1417.

[8] Roberta Coelho, Awais Rashid, Alessandro Garcia, Fabiano Ferrari, Nélio Cacho, Uirá Kulesza, Arndt Staa, and Carlos Lucena. 2008. Assessing the Impact of Aspects on Exception Flows: An Exploratory Study. In *ECOOP*. 207–234.

[9] Massimiliano Di Penta and Damian Andrew Tamburri. 2017. Combining quantitative and qualitative studies in empirical software engineering research. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 499–500. https://doi.org/10.1109/ICSE-C.2017.163

[10] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2015. An exploratory study on exception handling bugs in Java programs. *Journal of Systems and Software* 106 (2015), 82–101.

[11] K. R. Gabriel. 1969. Simultaneous test procedures—some theory of multiple comparisons. *The Annals Mathematical Statistics* 40, 1 (1969), 224–250.

[12] Cristian González García, Jordán Pascual Espada, B. Cristina Pelayo García Bustelo, and Juan Manuel Lovelle Cueva. [n. d.]. Swift vs. Objective-C: A New Programming. *International Journal of Artificial Intelligence and Interactive Multimedia* 3, 3 ([n. d.]), 74–81.

[13] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

[14] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. 2014. Lean GHTorrent: GitHub data on demand. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, Premkumar T. Devanbu, Sung Kim, and Martin Pinzger (Eds.). ACM, 384–387. https://doi.org/10.1145/2597073.2597126

[15] Benjamin Jakobus, Eiji Adachi Barbosa, Alessandro Garcia, and Carlos Jose Pereira de Lucena. 2015. Contrasting Exception Handling Code Across Languages: An Experience Report Involving 50 Open Source Projects. In *ISSRE*. 183–193.

[16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *MSR*. 92–101.

[17] Frank Konietschke, Ludwig A. Hothorn, and Edgar Brunner. 2012. Rank-based multiple test procedures and simultaneous confidence intervals. *Electronic Journal of Statistics* 6 (2012), 738–759.

[18] Wenkai Mo, Beijun Shen, Yuming He, and Hao Zhong. 2015. GEMiner: Mining Social and Programming Behaviors to Identify Experts in Github. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware (Internetware '15)*. 93–101.

[19] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (01 Dec 2017), 3219–3253. https://doi.org/10.1007/s10664-017-9512-6

[20] Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E. Hassan. 2015. An empirical study of goto in C code from GitHub repositories. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 404–414.

[21] Juliana Oliveira, Deise Borges, Thaisa Silva, Nélio Cacho, and Fernando Castor. 2018. Do android developers neglect error handling? a maintenance-Centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software* 136 (2018), 1–18.

[22] Dennis Pagano and Walid Maalej. 2011. How Do Developers Blog?: An Exploratory Study. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. ACM, New York, NY, USA, 123–132. https://doi.org/10.1145/1985441.1985461

[23] Lawrence A. Palinkas, Sarah M. Horwitz, Patricia Chamberlain, Michael S. Hurlburt, and John Landsverk. 2011. Mixed-Methods Designs in Mental Health Services Research: A Review. *Psychiatric Services* 62, 3 (2011), 255–263.

[24] M. Rebouças, G. Pinto, F. Ebert, W. Torres, A. Serebrenik, and F. Castor. 2016. An Empirical Study on the Usage of the Swift Programming Language. In *SANER*. 634–638.

[25] Martin P. Robillard and Gail C. Murphy. 2003. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.* 12, 2 (2003), 191–221.

[26] Michael P. Rogers and William Siever. 2015. Switching to Swift: Instructional Issues and Student Sentiment. *J. Comput. Sci. Coll.* 30, 5 (2015), 144–150.

[27] Hina Shah, Carsten Görg, and Mary Jean Harrold. 2008. Why Do Developers Neglect Exception Handling?. In *Proceedings of the 4th International Workshop on Exception Handling (WEH '08)*. 62–68.

[28] Hina Shah, Carsten Görg, and Mary Jean Harrold. 2010. Understanding Exception Handling: Viewpoints of Novices and Experts. *IEEE Trans. Software Eng.* 36, 2 (2010), 150–161. https://doi.org/10.1109/TSE.2010.7

[29] Susan Elliott Sim and Kavita Philip. 2010. Software Source Code Mashups as Transnational Circuits of Know-How. In *Tinkering, Tailoring, and Mashing: The Social and Collaborative Practices of the Read-Write Web*. 1–4.

[30] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[31] Alaaeddin Swidan, Alexander Serebrenik, and Felienne Hermans. 2017. How do Scratch Programmers Name Variables and Procedures?. In *SCAM*.

[32] Bogdan Vasilescu, Andrea Capiluppi, and Alexander Serebrenik. 2014. Gender, Representation and Online Participation: A Quantitative Study. *Interacting with Computers* 26, 5 (2014), 488–511.

[33] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Inf. & Softw. Technology* 74 (2016), 204–218.

[34] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM Press, New York, NY.