Perspectives and Challenges of iOS Developers in Using Reactive Programming with RxSwift

Elaine Cruz Farias
Universidade Federal de Pernambuco (UFPE)
Recife, Brazil
ecf@cin.ufpe.br

Carlos Zimmerle
Universidade Federal de Pernambuco (UFPE)
Recife, Brazil
cezl@cin.ufpe.br

Kiev Gama
Universidade Federal de Pernambuco (UFPE)
Recife, Brazil
kiev@cin.ufpe.br

ABSTRACT

Reactive programming, which deals with asynchronous data streams and events, is gaining popularity but remains underexplored as a research topic. In the iOS ecosystem, RxSwift stands out as a widely used framework for reactive programming despite its challenges. This study investigates the difficulties faced by iOS developers when using RxSwift. Semi-structured interviews were conducted, and a code comprehension questionnaire was applied to map developers' perspectives and the main challenges encountered. The paradigm shift was identified as the primary obstacle, with specific difficulties in creating streams, managing memory, and handling concurrency. Architectural patterns and learning resources were also cited as significant barriers. This research provides an overview of reactive development with RxSwift in iOS, proposing ways to optimize the learning journey and maximize the benefits of this approach.

KEYWORDS

Reactive Programming, Reactive Extensions, Mobile Applications

1 INTRODUCTION

In the dynamic landscape of programming and projects aimed at mobile platforms, reactive programming emerges as an innovative approach to tackle increasing challenges [9]. Modern applications are becoming more robust and complex, requiring simultaneous management of tasks such as audio playback, interface manipulation, and network calls, all while maintaining responsiveness and speed for the user [22].

Many of these applications are reactive systems that respond to a series of events and perform actions based on them [21]. Reactive programming is a declarative programming approach that deals with constant data streams and the propagation of changes over time asynchronously [9]. This unique ability to manage these streams and events makes it a fundamental element in the development of mobile platform applications, producing more flexible and scalable systems that better handle changes and provide interactive and effective feedback to their users [4].

At the core of this research is RxSwift, a widely used framework that incorporates the principles of reactive programming in the development of projects for the iOS ecosystem [5]. However, despite the benefits that the reactive approach can bring, it does not come without challenges and difficulties in its implementation. These systems require a new way of interpreting and solving problems, which can be a difficult skill to master, as pointed out by participants in a study [28]. Thus, learning and using RxSwift is not just about acquiring a new tool but facing a learning curve that can be challenging.

Besides being suitable for event-driven applications on mobile devices (e.g., tablets, smartphones), reactive programming also follows this principle in web applications [14], forming the basis of various frameworks like Angular, React, and Vue.js [18]. Although widely adopted and recognized, reactive programming continues to be underexplored by the research community in Software Engineering, as pointed out in 2015 [27]. There is a scarcity of research articles on this topic, concentrated among a few authors [28, 29], often limited to repository mining [20, 29], and rarely adopting an approach that directly involves developers, and still less the RxSwift framework.

In this context, we conducted an exploratory study aiming to identify and map the difficulties faced by developers when using RxSwift. A mixed-method approach was used, combining interviews and a code comprehension questionnaire, which helped collect data from software industry professionals. By mapping these challenges, this research aims to create new perspectives and insights that not only enhance the state of practice in companies and software projects but also advance the state of the art in academic research on a topic that is underexplored and intersects with industry needs.

2 BACKGROUND

2.1 Reactive Programming

Reactive systems or applications are a vital and broad class of software that focus on reacting to various types of events, such as user interface interactions, Internet messages, and sensor stimuli [25, 26]. Most modern software encompasses reactivity [9, 27], making them challenging to design, implement, and maintain [26, 27]. Reactive programming is a paradigm built around continuous values that vary over time and the propagation of changes. It facilitates the declarative development of event-driven applications, allowing programs to be expressed in terms of intentions while the language automatically manages execution [9]. This approach increases abstraction and simplifies code, improving expressiveness and understanding.

The reactive paradigm has several principles present in the languages and libraries that implement it, such as declarativity; the abstraction in the propagation of changes; the existing abstractions in the composition of objects, facilitated by operators, for example, and the favoring of data flow over control flow [28]. These principles are present in various solutions that enable reactive development in imperative languages. One such example is Reactive Extensions [1], one of the most popular and widely used families of libraries, also known as ReactiveX or Rx. In 2012, the Rx.NET library became an open-source project, allowing the implementation of its functionality for various languages and platforms, such as Ada [16] and Java [19], among other languages that adopted this more reactive approach.

2.2 RxSwift

For the iOS platform [22], RxSwift was proposed in 2015 [5], the specific implementation of Swift for the Reactive Extensions (ReactiveX)

[1] https://reactivex.io

A

```swift
protocol Observer {
    var identifier: Int { get set }
    func update(temperature: Double) -> Void
}

protocol Subject {
    func registerObserver(observer: Observer)
    func removeObserver(observer: Observer)
    func notifyObserver()
}

class TemperatureSubject: Subject {
    private var observers: [Observer] = []
    private var temperature: Double = 0
    private var identifier: Int = 0

    func registerObserver(observer: Observer){
        observers.append(observer)
    }

    func removeObserver(observer: Observer) {
        observers.removeAll(where: { obs in
            obs.identifier == observer.identifier
        })
    }

    func notifyObserver() {
        observers.forEach { obs in
            obs.update(temperature: self.temperature)
        }
    }

    func setTemperature(temperature: Double){
        self.temperature = temperature
        notifyObserver()
    }
}

class TemperatureObserver: Observer {
    var identifier: Int = 0

    func update(temperature: Double {
        print("a temperatura mudou: \(temperature)")
    }
}


let temperatureSubject = TemperatureSubject()

let temperatureObserver = TemperatureObserver()

temperatureSubject.registerObserver(observer: temperatureObserver)

temperatureSubject.setTemperature(temperature: 25)
```

B

```swift
import RxSwift

let temperatureSubject = PublishSubject<Double>()

let disposable = temperatureSubject
                    .subscribe(onNext: { temperature in
                        print("A temperatura mudou: \(temperature)")
                    })

temperatureSubject.onNext(25.0)
```

Figure 1: Swift code (A) versus RxSwift equivalent (B)

standard. Like other Rx implementations, the intent of RxSwift is to enable the easy composition of asynchronous operations and data streams in the form of Observables, along with a set of methods to transform and compose these asynchronous work elements [6]. Figure 1 exemplifies Swift code using the Observer pattern (A) and equivalent code using RxSwift (B), showing a clear reduction in verbosity through the use of the library.

RxSwift has four main elements: observables, subscribers, operators, and schedulers. An Observable is a protocol that defines a type capable of transmitting a sequence of values over time [22], meaning it emits a sequence of values. On the other hand, there are subscribers, which use operators to consume these generated values. Operators are methods capable of abstracting some asynchronous activity, and they can be composed with each other by chaining the result of one operator to be used by another (for example, filter and map in Figure 2). Finally, Schedulers can be seen as enhanced dispatch queues. RxSwift provides predefined Schedulers and allows scheduling different parts of the work from the same signature on different Schedulers for better performance, sending parts of the work to the correct context and allowing them to work harmoniously with each other's output[22].

The relevance of RxSwift in the reactive development landscape for iOS is notable, even with the native option of Swift Combine [1] since 2019 and the release of the new Observation framework [3] in 2023. However, despite the many positive aspects and the extensive documentation available on the Internet, few studies investigate the use of reactive programming with RxSwift, and even more broadly in reactive development on mobile devices, especially studies involving developers directly in data collection.

3 METHOD

In this exploratory study scenario, interviews are fundamental for investigating practices and understanding perceptions, allowing for a deep exploration of relevant questions, topics, and experiences, capturing the complexity and richness of phenomena in specific contexts [15]. In addition, to support and complement the collected reports, multiple-choice questionnaires related to code comprehension were also conducted. Thus, this study adopted a mixed approach to data collection regarding RxSwift developers' perceptions of the technology. Data collection was conducted exclusively online, both for availability and practicality for participants from various cities.

3.1 Semi-structured interview

Since there are few studies on the topic [27], using semi-structured interviews served to avoid structural rigidity and allow for broader exploration [15]. The interview guide was designed to conduct the conversation around four main axes:

Experience, which will map the interviewee's background in programming, reactive programming, and RxSwift, providing context to understand their journey and level of expertise and how everything relates to and impacts the understanding of the subject being addressed;

Development and Challenges, which aims to validate the complexity associated with learning and applying RxSwift, exploring specific difficulties in nature and intensity through the reports;

Learning Resources, to obtain a practical perspective on the tools and materials available for learning. Identifying satisfaction with these resources and possible gaps in existing materials will contribute to improvements in learning RxSwift;

Suggestions and Tips, to consolidate data validation by collecting suggestions on how to make learning and using RxSwift more accessible and efficient.

3.2 Multiple-choice questionnaire

The questionnaire consisted of closed-ended questions and was divided into three main sections: Experience, Code Comprehension, and Operator Classification. In the Experience section, questions were asked about the duration of practice with programming, reactive programming, and RxSwift. In the Code Comprehension section,

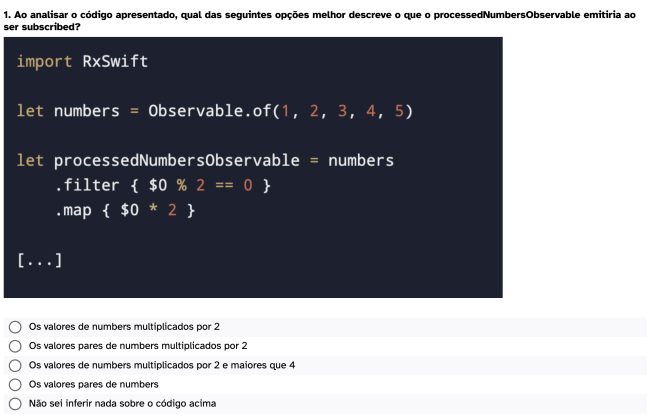

Figure 2: Example of a question about Stream Manipulation

1 to 3 questions were presented about: Stream Creation, Stream Manipulation, Dependency Management, Composition, and Disposable. These categories were derived from existing studies on repository mining and Q&A platforms focused on reactive programming [29] and from a specific study on the use of RxSwift [13]. In the Operator Classification section, based on the operators used in code examples, respondents were asked to categorize the difficulty (easy, very easy, neutral, difficult, very difficult, or unknown) of some RxSwift operators (map, subscribe, create, just, combineLatest, zip, flatMap, filter, observeOn, takeUntil, and share). A theoretical question without code to be interpreted, regarding the functionality of *DisposeBag*, complemented the analysis of code comprehension

The SoSci Survey platform [7] was used for the online research. In this setup, a code snippet was presented and participants were expected to identify the alternative that best expressed their interpretation and understanding of the provided snippet. An example question can be seen in Figure 2.

To mitigate the occurrence of false positives, each question included an option which participants could select if they did not know the answer or could not affirm any of the alternatives, indicating a lack of knowledge for interpreting the code. Participants were also encouraged and emphasized to avoid guessing the answers.

3.3 Sampling

The strategy of purposive sampling was used to select interviewees who well-represented the perceptions and experiences of RxSwift developers, aiming for a balanced profile between academic and professional experience. The sample consisted of 13 interviewees and 10 questionnaire respondents. The individuals were all Brazilians contacted among the professional network connections of the first author. As pointed out by Baltes and Ralph [10], this type of sampling is valuable in Software Engineering research because it allows for the selection of rich and relevant cases when a complete sampling frame is not available, enabling a focus on specific criteria to obtain deep and meaningful data, thus ensuring the relevance of the results for the study's objectives.

3.4 Daya analysis

3.4.1 Semi-structured interviews. The interviews were all conducted in Portuguese, and the recordings were transcribed with an automated tool (Whisper Transcription). The results were analyzed to correct minor transcription errors. To analyze the data collected from the interviews, we used deductive reasoning [23], meaning we started with categories gathered from the literature, analyzed individual reports, and made generalizations and classifications based on these inferences. We conducted three processes during data collection and analysis:

I. Analytical Memos. After each interview, we made Analytical Memos, which are concise notes about the thoughts, ideas, and questions that arose during data collection. Their purpose is to document and reflect on the development of the investigative process, including the analysis of patterns and categories [24]. This approach helped to observe repetitions that emerged in the data, contributing to the investigation.

II. In Vivo Coding. To categorize the difficulties pointed out by the developers, we performed a coding process on the interview transcriptions. Coding is a heuristic for the meanings of individual sections of data. These codes serve as a way to standardize, classify, and subsequently reorganize each piece of data into emerging categories for further analysis [23]. Among the different ways to apply coding, this study used In Vivo Coding, which emphasizes the participants' spoken words [23]. Thus, we sought to select excerpts that seemed significant or somehow summarized what was said.

III. Categorization of Codes. After conducting the In Vivo Coding process, we categorized the extracted codes. For the categories that would support the deductive reasoning employed in the analysis, we surveyed topics already identified as popular or difficult within the literature on reactive programming and RxSwift: such as Dependency Management [29], Functional Programming [28], and Perception of time [12]. These topics formed the initial list of categories, subject to change during the analysis phase. During this process, new topics were added based on elements identified in the data, reflecting recurring themes present in the extracted excerpts:

- **Stream Creation:** Difficulties involved in creating observables. Notably regarding the different types of Subjects that exist, the various ways to create each one, and when to use each type.
- **User Interface (UI):** Complexities related to integrating reactive logic with user interactions.
- **Paradigm Shift:** Challenges of perceptual and conceptual changes faced when adopting the reactive paradigm, examining how this transition impacts the mental framework in approaching problems.
- **Functional Programming:** Obstacles when applying functional programming concepts in RxSwift, identifying how the composition of reactive operators reflects the principles of this approach.
- **Dependency Management:** Challenges related to effectively managing streams and the dynamic relationships between components, considering the subtle management of internal connections.
- **Memory Management:** Difficulties in managing memory, focusing on preventing leaks and optimizing performance. This category is sometimes also referred to as Lifecycle or Stream lifecycle.
- **Testing and Debugging:** Problems encountered in identifying and debugging issues, as well as in manually and automatically testing RxSwift code with unit tests.
- **Perception of time:** Complexity of understanding and structuring Perception of times in RxSwift implementation. This includes issues of state management and the complexity of tracking value changes over time.
- **Type Inference:** Obstacles dealing with type inference and type compatibility errors, especially in long chains of operators.

- **Error Handling:** Challenges in strategies for effectively managing the data flow, especially in handling the onError event to deal with exceptions and unexpected failures

3.4.2 Multiple choice questionnaires. To examine the data from the multiple-choice questionnaire, the accuracy and error rates for each question related to Code Comprehension were assessed, along with the response times. Alongside these indices, the categories established during the data collection structuring phase were examined, allowing for a detailed view of performance in specific areas, with special attention given to categories with higher error rates, indicating potential difficulty in those topics.

Thus, the combination of qualitative and quantitative methods allowed for an understanding of the challenges faced by developers when applying RxSwift in their projects. Additionally, it enabled an exploration of perceptions about learning resources and suggestions for improving the efficiency of using RxSwift, providing a comprehensive and practical view for professionals and researchers interested in this domain.

4 RESULTS

4.1 Semi-structured interviews

Participants details. As detailed in Table 1, the 13 participants had varying levels of experience in RxSwift, which were divided into 3 expertise categories: Beginner (1 to 3 years), Intermediate (3 to 5 years), and Advanced (more than 5 years). The interviews lasted, on average, 21 minutes, with longer duration observed for more experienced interviewees, reflected by the greater range of reports and testimonies.

Table 1: Profile of the interview participants

Participant	Experience with RxSwift	Interview Duration (min.)
P1	Beginner	18:05
P2	Beginner	17:30
P3	Beginner	08:30
P4	Beginner	10:02
P5	Beginner	16:50
P6	Beginner	17:15
P7	Intermediate	20:50
P8	Intermediate	30:05
P9	Intermediate	22:30
P10	Intermediate	18:15
P11	Advanced	24:04
P12	Advanced	27:05
P13	Advanced	21:50

Coding and categorization An In Vivo Coding approach was used to select excerpts spoken by the participants [23], as shown in Table 2. The aim was to identify the most representative segments, summarizing or concisely expressing the testimony presented. After the In Vivo Coding of the interview transcripts, the extracted codes were categorized. Starting from the initial 11 categories listed in Table 3, a cross-validation of the recurring topics was performed, resulting in the removal of one category (Type Inference) and the inclusion of four other categories, detailed below, which resulted in 14 categories being used (Figure 3).

a) **Architectural Patterns:** Obstacles faced in understanding and structuring architectural patterns that fit well with the use of RxSwift, considering scalability and code maintenance. Barriers in decision-making regarding structures and code organization were recorded here, as exposed in "Difficulty in passing Rx information between classes and screens" (P1) and "it became a very large file that we didn't know how to structure" (P2).
b) **Insufficient Learning Resources:** Challenges faced by developers in finding limited educational resources to learn and resolve doubts in RxSwift.
c) **Implicit Behavior:** Barrier regarding an 'implicit automation' in RxSwift snippets, referring to behavior that occurs automatically, without clear explicitness in the source code. This characteristic, although convenient, was identified as a challenge manifested in "Things seemed to be done by magic, I didn't understand well" (P2) and "A lot of things I used without really understanding how it works behind the scenes." (P3).
d) **Others:** Consideration of challenges and aspects not addressed in the previous categories, providing space for diverse topics in the context. Some issues raised included difficulty in knowing how to search when there was a doubt or performance problems.

Table 2: Example of in Vivo code extraction and categorization (translated from PT-BR)

Transcript Excerpt	Extracted Code	Category
"I couldn't get it to work at all because I didn't understand what DisposeBag was until then, and I think I still don't understand what DisposeBag is" (P3)	"I think I still don't understand what disposebag is"	Memory Management
"When we write object-oriented code, you know the order in which it will be executed. When you start using this type of framework, looking at the reactive part, you no longer have control over when things will be executed" (P9)	"you no longer have control over when things will be executed"	Concurrency

Table 3: Categories of codes and their respective origins

#	Category	Reference of Origin
1	Stream Creation	Zimmerle et al. [29]
2	User Interface (UI)	Pereira et al. [20]
3	Paradigm Shift	Salvaneschi et al. [28]
4	Functional Programming	Salvaneschi et al. [28]
5	Dependency Management	Zimmerle et al. [29]
7	Memory Management	Holopainen [13], Pereira [21], and Zimmerle et al. [29]
8	Testing and Debugging	Zimmerle et al. [29], Pereira et al. [20]
9	Concurrency	Zimmerle et al. [29], Pereira et al. [20]
9	Perception of time	Dinser [12]
10	Type Inference*	Zimmerle et al. [29] and Mäata [17]
11	Error Handling	Zimmerle et al. [29]
12	Architectural Patterns	This study
13	Insufficient Learning Resources	This study
14	Implicit Behavior	This study
15	Others	This study

**Not identified/used in data categorization*

In the other pre-selected categories, there are, for example, "inconsistency in the collection view" (P1) and "difficulty in getting information and displaying it the way I wanted in the UI" (P6) categorized as challenges faced by participants in the User Interface (UI) topic. Also, "Sometimes it is very difficult to reproduce how

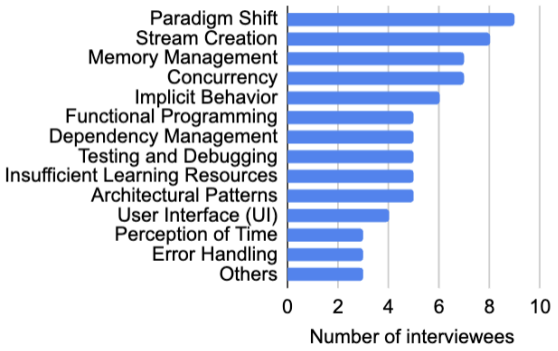

Figure 3: Mentions of each category by the interviewees (n=13)

the stream works in the test" (P12) categorized as a Testing and Debugging topic. Table 2 provides more examples. The complete list of codes is available in the artifact availability section.

Analysis of interviews. The frequency with which each category was mentioned by the participants can be observed in Figure 3. "Paradigm Shift" stands out as the most referenced category, representing 69% (9 people) of the participants. Mentions of this topic highlight the importance attributed to the paradigm shift and its impact on the mental framework for approaching problems. This finding also correlates with the discussion about the learning curve in reactive technologies and the intrinsic paradigm shift in them, as indicated by Salvaneschi et al [28], who also point out these factors as the main obstacles faced by developers of reactive applications, reinforcing this finding. "Stream Creation" was cited by 8 participants (61.5%), appearing more frequently in the statements of people new to reactive technology and RxSwift. The topics of "Memory Management" and "Concurrency" also had high citations, mentioned by 7 (53.8%) of the participants. These occurrences align with Zimmerle et al [29], who state that "the topics with the most publications address stream abstraction, with Stream Manipulation (over 13%) and Stream Creation and Composition (over 10%) being in first and second place". Additionally, "Concurrency" appears in another study [21] that addresses difficulties with Combine, Apple's reactive framework, being pointed out as the most challenging topic, which can also be seen with the RxSwift framework. The topics of "Functional Programming" and "Testing and Debugging" were also frequent in the interviewees' statements and should be considered frequent challenges. The functional programming theme is also identified as a major difficulty by Holopainen [13]. The category of Architectural Patterns identified in the reports highlighted a gap in guidelines and design patterns aimed at the reactive universe.

4.2 Multiple Choice Questionnaire

The questionnaire was divided into three main sections: Experience, Code Comprehension, and Operator Classification, to explore different aspects of RxSwift usage. In the Code Comprehension section, 1 to 3 questions were presented on the topics of Stream Creation, Stream Manipulation, Dependency Management, Composition, and Disposable. The Stream Creation part had 1 question; Stream Manipulation, Composition, and Disposable are explored in 2 questions each, and Dependency Management had 3 questions.

Profile of Participants. The profile of the participants can be seen in Table 4, fitting individuals with little expertise in the reactive paradigm and the RxSwift framework. The interview participants with little experience and categorized as beginners answered the questionnaire, with this intersection between the samples.

Table 4: Profile of the Questionnaire Participants

Characteristics	n(%)
Experience with programming	
0-2 years	1 (10%)
2-3 years	1 (10%)
3-4 years	2 (20%)
4-5 years	2 (20%)
5+ years	4 (40%)
Experience with reactive programming	
0-1 years	5 (50%)
2-3 years	4 (40%)
4-5 years	1 (10%)
Experience with RxSwift	
None	1 (10%)
0-1 years	9 (90%)

Analysis of Responses. In the analysis of the responses, the options indicating a lack of knowledge about the correct alternative were treated as equivalent to incorrect ones. This was due to the interpretation that the question presented challenges for the respondent, which is recognized in both cases. The accuracy and error rates for each of the questions can be observed in Table 5. It is noteworthy that the questions related to the use of Disposables registered the lowest accuracy rate. Understanding these questions is intrinsically linked to understanding the lifecycle of streams, and is thus associated with Memory Management in the context of RxSwift. This finding reinforces the high incidence of mentions of Memory Management as a challenging area in the interviews. Additionally, challenges in the topics of stream manipulation and composition also had some errors. In the categorization of operator comprehension difficulty, out of the 10 responses, *share* was the least known operator, according to 6 respondents, and considered difficult by 3. Other notable figures include *takeUntil* being unknown to 5 people, while 4 were unfamiliar with *create*, *just*, and *observeOn*. On the other hand, *filter* and *map* were considered easy by 9 respondents, while *subscribe* was seen as easy by 5 people. Furthermore, 3 people considered *flatMap* to have some level of difficulty. It is important to note that difficulty is not necessarily related to popularity, as, with the exception of the *share* operator, the presented operators are among the 15 most popular [29].

Table 5: Rate of correct/incorrect questionnaire answers (n=10)

Question	Correct	Incorrect
1 - Stream Creation	90%	10%
2 - Stream Manipulation I	90%	10%
3 - Stream Manipulation II	50%	50%
4 - Dependency Management I	80%	20%
5 - Dependency Management II	60%	40%
6 - Dependency Management III	90%	10%
7 - Stream Composition I	60%	40%
8 - Stream Composition II	60%	40%
10 - Disposables I	40%	60%
11 - Disposables II	30%	70%

4.3 Limitations

One of the main limitations that can be raised for this work is the limited sample size, although this is an article focused on new ideas and emerging results. We understand that the findings cannot be

generalized, nor is it the purpose of this research, which presents emerging results. According to the research objectives, small samples are acceptable in qualitative research [11]. Regarding the purposive sampling approach, where items are selected based on a specific logic or strategy, it becomes appropriate for qualitative and interpretive research [10]. Thus, even with such a sample of developers with varied experience, it is possible to ensure the relevance and depth of the data, essential for understanding the challenges in using RxSwift.

Internal validity may have been compromised by participant bias, with more experienced developers potentially providing more detailed responses, and by the possible influence of the interviewer. External validity is limited by the generalization of the results, given that the sample has a small number of developers and the specific focus on the iOS context, not being applicable to other platforms.

5 DISCUSSION

The interviews highlighted that the **Paradigm Shift** is an area of notable difficulty, being mentioned by **69%** of the participants. This result emphasizes the significant mental transition that developers face when adopting the reactive paradigm, indicating that the main difficulty is more related to understanding the fundamentals of the approach rather than RxSwift itself. From the reports, it was observed that most developers started learning RxSwift due to professional demands, highlighting the lack of a solid foundation to support the daily use of the framework. This aspect was also emphasized in the tips and suggestions section of the interviews, where there were many mentions of the importance of studying and understanding the paradigm before using RxSwift in practice. This finding is aligned with the literature that addresses reactive programming [28].

Another category mentioned was **Stream Creation**, addressed by 61.5% of the interviewees. Of the two questions about **Stream Manipulation** in the questionnaire, one had a 90% success rate and the other only 50%. The topic was pointed out in the literature [29] as one of the most frequent in posts and questions on Stack Overflow and GitHub, although it is not listed among the most challenging. This seems to align with the frequency of this topic being a common doubt for beginners, but appears to decrease with higher expertise.

Other frequently cited categories were **Memory Management** and **Concurrency**, highlighted in the statements of **53.8%** of the participants. These topics are consistent with the complexities identified in reactive frameworks, emphasizing the need for a deep understanding of the lifecycle of streams and efficient resource management. The topic of **Memory Management**, besides being recurrent in the testimonies, had its complexity corroborated by the analysis of the questionnaire responses, revealing specific challenges in understanding the use of **Disposables**, which was the area with the highest error rate in the questions and naturally involves memory management and references. A study [13] points to the issues of reference cycles and memory leaks as one of the main challenges in developing with RxSwift, which confirms the findings. Furthermore, this theme is mentioned in other studies [21, 29] as one of the most popular topics for questions, although not identified as one of the main ones in terms of difficulty. The significant difficulty index with the **Concurrency** category reported by the interviewees supports the findings of two studies [20, 29] that pointed to it as one of the most difficult topics in reactive programming and Combine, respectively. Additionally, the exclusion of the "Type Inference" category from the classification of codes reflects its absence in the testimonies, although studies [17, 29] have pointed it out as a difficulty.

The categories that emerged in the analysis, such as **Insufficient Learning Resources** and **Architectural Patterns**, provide valuable insights as they are more difficult to identify in the counting and analysis of Stack Overflow [8] and GitHub [2] posts. Therefore, the presence of such themes highlights gaps in the availability of specific educational materials for RxSwift and underscores the importance of clear architectural guidelines.

6 CONCLUSION

Although RxSwift is not a very recent framework, having been released in 2015, there is currently scarce evidence of research addressing its use, and none have incorporated the direct opinions of developers. The mixed-method approach used in our research, combining qualitative and quantitative elements through semi-structured interviews along with a code comprehension questionnaire, provided a broad understanding of the challenges faced by developers in applying RxSwift. Thus, with the aim of investigating the difficulties faced by iOS developers when incorporating reactive programming through RxSwift, this research helps reveal these main challenges and how they relate to and differ from more general aspects of reactive programming, highlighting issues related to stream creation and manipulation, memory management, functional programming, testing, and debugging.

In this work, we collected and categorized topics related to the use of RxSwift, examining their appearance and frequency in developers' reports as well as in code comprehension responses. This approach provides a foundation for the research community to start deeper investigations into the understanding gaps of the framework in reactive projects. We contribute to the identification of common problems when the framework is employed, guiding suggestions for improvements to make the use of RxSwift more accessible to Swift developers interested in reactive programming. Ultimately, the results of this research provide valuable insights to improve the efficiency of learning and using RxSwift, benefiting developers and researchers interested in this domain.

This study serves as an initial step that can lead to broader investigation, in which we will conduct interviews with a larger sample of iOS developers and compare perceptions between RxSwift and Apple's reactive frameworks, Combine [1] and Observation [3]. Additionally, considering the learning difficulties pointed out, we aim to investigate how the use of assistants like GitHub Copilot and other tools based on Large Language Models can alleviate the learning curve for developers with less experience in reactive programming. Such approach would allow us to confront the current data, enriching the understanding of the challenges faced in using the technology. Furthermore, subsequent studies could bring interventions and materials aimed at addressing the major gaps and obstacles found.

ARTIFACTS AVAILABILITY

Supplementary material (in portuguese) containing a code book with the list of codes and in vivo codes; a copy of the questionnaire with questions and code examples; and a file with the questionnaire responses: https://doi.org/10.6084/m9.figshare.25952860

ACKNOWLEDGMENTS

This work is partially supported by INES (www.ines.org.br), CNPq grant 465614/2014-0, FACEPE grants APQ-0399-1.03/17 and APQ/0388-1.03/14, CAPES grant 88887.136410/2017-00.

REFERENCES

[1] [n. d.]. *Combine framework.* https://developer.apple.com/documentation/combine
[2] [n. d.]. *Github.* https://github.com/
[3] [n. d.]. *Observation framework.* https://developer.apple.com/documentation/observation
[4] [n. d.]. *The Reactive Manifesto.* https://www.reactivemanifesto.org/
[5] [n. d.]. *RxSwift GitHub.* https://github.com/ReactiveX/RxSwift
[6] [n. d.]. *RxSwift Reference.* https://docs.rxswift.org/
[7] [n. d.]. *SoSci Survey.* https://www.soscisurvey.de/help/doku.php/en:start
[8] [n. d.]. *Stack OverFlow.* https://stackoverflow.com/
[9] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2012. A survey on reactive programming. *ACM Computing Survey* (2012).
[10] Sebastian Baltes and Paul Ralph. 2022. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering* 27, 4 (2022), 94.
[11] Clive Roland Boddy. 2016. Sample size for qualitative research. *Qualitative market research: An international journal* 19, 4 (2016), 426–432.
[12] Moritz Dinser. 2021. An Empirical Study on Reactive Programming. Master's thesis, Technische Universität Darmstadt. https://tuprints.ulb.tu-darmstadt.de/19901/8/An_Empirical_Study_on_Reactive_Programming.pdf
[13] Niko Holopainen. 2022. Reactive iOS Development with RxSwift. Master's thesis, Metropolia University of Applied Sciences. https://www.theseus.fi/bitstream/handle/10024/704093/holopainen_niko.pdf
[14] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. 2013. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. 1–9.
[15] Tim May. 2002. *Qualitative research in action.* Sage.
[16] Alejandro R Mosteo. 2020. Reactive programming in Ada 2012 with RxAda. *Journal of Systems Architecture* 110 (2020), 101784.
[17] Matti Määttä. 2017. Reactive Programming in iOS Application Development. Master's Thesis, Tampere University of Technology. https://trepo.tuni.fi/bitstream/handle/123456789/25137/M%C3%A4%C3%A4tt%C3%A4.pdf
[18] Chi Nguyen Huu Ngoc. 2017. Functional Reactive Programming in React Application. Bachelor's Thesis, Metropolia University of Applied Sciences.
[19] Tomasz Nurkiewicz and Ben Christensen. 2016. *Reactive programming with RxJava: creating asynchronous, event-based applications.* " O'Reilly Media, Inc.".
[20] Alessandra Pereira, Carlos Zimmerle, Kiev Gama, and Fernando Castor. 2023. Reactive Programming with Swift Combine: An Analysis of Problems Faced by Developers on Stack Overflow. (2023).
[21] Alessandra Luana Nascimento Pereira. 2022. Um Estudo Sobre O Uso Do Framework Combine Através Da Mineração De Publicações Do Stack Overflow. Bachelor's Thesis, Universidade Federal de Pernambuco. https://www.cin.ufpe.br/~tg/2022-1/tg_SI/TG_alnp.pdf
[22] Florent Pillet, Junior Bontognali, and Marin Todorova nand Scott Gardner. 2019. *RxSwift. Reactive Programming with Swift.* Razeware LLC.
[23] Johnny Saldaña. 2011. *Fundamentals of Qualitative Research - understanding Qualitative Research.* Oxford University Press, Inc.
[24] Johnny Saldaña. 2013. *The coding manual for qualitative research.* SAGE Publications.
[25] Guido Salvaneschi, Joscha Drechsler, and Mira Mezini. 2013. Towards distributed reactive programming. In *Coordination Models and Languages: 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings 15.* Springer, 226–235.
[26] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th international conference on Modularity.* ACM, 25–36.
[27] Guido Salvaneschi, Alessandro Margara, and Giordano Tamburrelli. 2015. Reactive Programming: A Walkthrough. (2015).
[28] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2017. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on Software Engineering PP(99):1-1* (2017).
[29] Carlos Zimmerle, Kiev Gama, Fernando Castor, and José Murilo Filho. 2022. Mining the Usage of Reactive Programming APIs: A Study on GitHub and Stack Overflow. (2022).