UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Carlos Eduardo Zimmerle de Lima

Unveiling the Usability of Reactive Programming APIs: Findings, Tools, and Recommendations

Recife

2024

Carlos Eduardo Zimmerle de Lima

Unveiling the Usability of Reactive Programming APIs: Findings, Tools, and Recommendations

Tese de Doutorado apresentada ao Programa de Pós-graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação

Orientador (a): Kiev Santos da Gama

Recife

2024

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Lima, Carlos Eduardo Zimmerle de.
Unveiling the usability of reactive programming APIs: findings, tools, and recommendations / Carlos Eduardo Zimmerle de Lima. - Recife, 2024.
170 f.: il.

Tese (Doutorado) - Universidade Federal de Pernambuco, Programa de Pós-Graduação em Ciência da Computação, 2024.
Orientação: Kiev Santos da Gama.
Inclui referências e apêndice.

1. Programação reativa; 2. Usabilidade de APIs; 3. Mineração de repositório de softwares; 4. Avaliação centrada no usuário. I. Gama, Kiev Santos da. II. Título.

UFPE-Biblioteca Central

Carlos Eduardo Zimmerle de Lima

"Unveiling the Usability of Reactive Programming APIs: Findings, Tools, and Recommendations"

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovada em: 18/12/2024.

I dedicate this work to my parents, friends, advisor, and all of those that helped me through this journey.

ACKNOWLEDGEMENTS

I would first like to thank my advisor. Without his insistency and patience throughout all those years, I probably would not have made it this far. I was one of his students who took many of his courses, and my research journey with him started since my undergraduate thesis. Throughout the years, I have collaborated with him in many works, and he could also observe my ups and downs. Nevertheless, he always finds ways to convince you that it will work fine in the end. For those and next, future endeavors, I truly thank you!

I want to express my gratitude to all the professors that had the chance to study with. In special, I would like to include my personal thanks to professor Fernando Castor who I had the chance to meet during the last years and share some collaborating works with.

To my parents, I thank you for all the dedication and support in my life. I would like to thank my mother from the bottom of my heart for raising and educating me; my father, as well, who always believed in me and made it clear that he would always be my friend, that I could count on him whenever I needed.

I would like to extend my thanks to my long list of colleagues and friends (old and new ones). I thank you for understanding my absence from many meetings and parties during this journey. Moreover, I want to include a special thanks to Alexandre Vianna who I had the chance to include in my friendship and collaborate in some researches.

Finally, I want to express my faith by thanking God for providing my existence, guiding my path, and blessing my family.

Thank you very much to all of you.

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” (FOWLER, 2018, p. 10).

ABSTRACT

Reactive Programming (RP) has gained traction for its ability to simplify the development of event-driven and asynchronous applications. Despite its growing popularity, the usability of application programming interfaces (APIs) of RP remains a significant challenge for developers, with issues ranging from steep learning curves to inconsistent design practices (e.g., divergent number of operators and interfaces). This thesis explores the usability of RP APIs through a combination of approaches: mining studies, metrics, and user-centered evaluations. The first part of the study leverages data from GitHub and Stack Overflow to analyze how developers interact with RP APIs, identifying operators' frequency (a common design problem) and recurring pain points. The second part employs a mixed-method approach, combining structural, computed metrics with qualitative user study to assess API usability, employing a refined Cognitive Dimensions framework (CDN). Metrics are computed using our implemented tool, UAX (Usability Analyzer Experience), which embodies a set of six metrics explored in other studies. The user-centered evaluation further examines aspects like understandability, learnability, and expressiveness through task-based experiments and user feedback. Results highlight significant disparities between API design and usability, providing a clearer understanding of the real-world challenges users encounter. The thesis culminates in a set of practical recommendations for the designers, aimed at enhancing RP API usability and aligning it with users' needs. Contributions include a comprehensive usability analysis of RP APIs, empirical findings from the open-source community, answers for recurrent problems (i.e., excessive number of operators), the UAX tool, the first appliance of a user-centered evaluation with CDN and RP, recommendations for API improvements, and a foundation for future RP usability researches. This work lays the groundwork for enhancing the developer experience in RP interfaces and contributes to the broader field of software engineering.

Keywords: Reactive Programming. API Usability. Mining Software Repositories. User-Centered Evaluation.

RESUMO

Programação Reativa (RP) vem ganhando força por sua habilidade de simplificar o desenvolvimento de aplicações dirigidas a eventos e assíncronas. Apesar de sua crescente popularidade, a usabilidade de interfaces de programação de aplicativos (APIs) de RP continua sendo um desafio significativo para desenvolvedores, com problemas que vão desde curvas de aprendizado acentuadas até práticas de design inconsistentes (por exemplo, número divergente de operadores e interfaces). Esta tese explora a usabilidade de APIs de RP por meio de uma combinação de abordagens: estudos de mineração, métricas e avaliações centradas no usuário. A primeira parte do estudo aproveita dados do GitHub e do Stack Overflow para analisar como os desenvolvedores interagem com APIs de RP, identificando a frequência dos operadores (um problema comum de design) e pontos problemáticos recorrentes. A segunda parte emprega uma abordagem de pesquisa baseada em método misto, combinando métricas estruturais computadas com estudo qualitativo com usuários para avaliar a usabilidade das APIs, empregando uma estrutura refinada de Dimensões Cognitivas (CDN). As métricas são computadas usando nossa ferramenta implementada, UAX (Usability Analyzer Experience), que incorpora um conjunto de seis métricas exploradas em outros estudos. A avaliação centrada no usuário examina adicionalmente aspectos como compreensibilidade, capacidade de aprendizado e expressividade por meio de experimentos baseados em tarefas e feedbacks de usuários. Os resultados destacam disparidades significativas entre os designs das APIs e usabilidade, fornecendo uma compreensão mais clara dos desafios do mundo real que os usuários encontram. A tese culmina em um conjunto de recomendações práticas para os designers, visando aprimorar a usabilidade das APIs de RP e alinhá-las com as necessidades dos usuários. As contribuições incluem uma análise abrangente de usabilidade de APIs de RP, descobertas empíricas providas pela comunidade de código aberto, respostas para problemas recorrentes (i.e., número excessivo de operadores), a ferramenta UAX, a primeira aplicação de uma avaliação centrada no usuário com CDN e RP, recomendações para melhorias das APIs e uma base para futuras pesquisas de usabilidade em RP. Este trabalho estabelece fundamentos para aprimorar a experiência do desenvolvedor em interfaces de RP e contribui para o campo abrangente da engenharia de software.

Palavras-chaves: Programação Reativa. Usabilidade de APIs. Mineração de Repositório de Softwares. Avaliação Centrada no Usuário.

LIST OF FIGURES

LIST OF SOURCE CODES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
CDN	Cognitive Dimensions of Notations
FP	Functional Programming
FRP	Functional Reactive Programming
GH	GitHub
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JS	JavaScript
JSON	JavaScript Object Notation
LDA	Latent Dirichlet Allocation
OO	Object-Oriented
REST	Representational State Transfer
RP	Reactive Programming
Rx	ReactiveX
SEDE	Stack Exchange Data Explorer
SO	Stack Overflow
TS	TypeScript
UAX	Usability Analyzer Experience

CONTENTS

1 INTRODUCTION

1.1 CONTEXT AND MOTIVATION

In the area of software development, event-driven applications, also called interactive or reactive applications (e.g., graphical user interface apps), have relied on callbacks and inversion of control as means to structure their logic (DRECHSLER et al., 2014; BAINOMUGISHA et al., 2013). As those applications evolve, the code tends to become an "asynchronous spaghetti" with deeply-nested dependent callbacks, a problem often called callback hell or pyramid of doom (KAMBONA; BOIX; MEUTER, 2013). The Observer pattern, the object-oriented approach to event-driven programming, raises the abstraction level and decouples different application parts, but it also brings its set of complexities and readability problems (SALVANESCHI; HINTZ; MEZINI, 2014; SALVANESCHI; MARGARA; TAMBURRELLI, 2015; SALVANESCHI et al., 2017). As a result, event-based programs are known to be difficult to design, implement, and maintain (SALVANESCHI; MARGARA; TAMBURRELLI, 2015).

The Reactive Programming (RP) paradigm was conceived to facilitate the construction of interactive applications through the use of dedicated abstractions (BAINOMUGISHA et al., 2013; MARGARA; SALVANESCHI, 2014; SALVANESCHI; MARGARA; TAMBURRELLI, 2015). These abstractions (BAINOMUGISHA et al., 2013; SALVANESCHI, 2016), behaviors (values that vary continuously) and event streams (values that vary discretely), are then formulated through three concepts (MARGARA; SALVANESCHI, 2014): (*i*) time-changing values, (*ii*) dependency tracking and (*iii*) automatic propagation of changes. RP use allows better comprehension compared to the Observer pattern (SALVANESCHI et al., 2017). Additional qualities are also noted like its composability, reduction of code lines and boilerplate code, better readability, among others (SALVANESCHI; HINTZ; MEZINI, 2014; SALVANESCHI et al., 2017).

Given the advantages of RP, many libraries and extensions have been incorporated in varying languages (BAINOMUGISHA et al., 2013). The family of libraries called Reactive Extensions (ReactiveX (Rx)) is among the most popular families of libraries nowadays, specially after the Netflix's successful use case (SALVANESCHI et al., 2017). As of the JVM version 9, a set of reactive interfaces were defined under the class *java.util.concurrent.Flow*, all inspired by the specification for the JVM called Reactive Streams[1]. The specification details a minimal set of interfaces with non-blocking back-pressure (a control flow mechanism) which many libraries

[1] <https://www.reactive-streams.org/>

currently comply to (e.g., Akka Streams[2], Reactor[3], and RxJava[4]). Under the category of web frameworks and technologies, popular tools[5] like React, Angular and Vue.js, either incorporate reactive primitives or ideas. React library, for instance, employs concepts like immutability, pure functions, and automatic propagation of updates, and has third-party tools[6,7] that explore Rx capabilities in React. Angular, on the hand, has RxJS (Rx for JavaScript) as an integral part of its components since early versions[8], and, more recently, it added the Signal (behavior) abstraction to its API[9].

Despite the fact that the traditional domain of RP correspond to user interface and animations (SALVANESCHI, 2016; MOGK; SALVANESCHI; MEZINI, 2018), its application has been expanding since its conception to other areas as well. In fact, we are living in a connected world where events are coming from multiple places (e.g., clicks on a web page, updates on a social network or stock markets, and even smart components), so RP offers a compelling, declarative model to handle those interactive applications. RP has been explored for instance in the realm of Internet of Things (PROENÇA; BAQUERO, 2017; DIAS; FARIA; FERREIRA, 2018; ZIMMERLE; GAMA, 2018b) where more and more objects are connected everyday to the global network and with an ever-growing market (15% increase only in 2021 and with projections to continue expanding in the following years[10]). Zimmerle and Gama (2018a), based on the ideas of Margara and Salvaneschi (2013), created a prototype library where they showed it was possible to express complex event processing (CUGOLA; MARGARA, 2012) with RP. RP has even made its way to tiny elements like WiFi chips/firmware (STERZ et al., 2021) and high performance computing (SOKOLOWSKI; MARTENS; SALVANESCHI, 2019) (important in areas like meteorological forecasts and particle simulations) aiming to raise their abstraction level while allowing the construction of programs more easy to create and less error-prone. Other domain areas explored by RP include (BAINOMUGISHA et al., 2013; SALVANESCHI; MARGARA; TAMBURRELLI, 2015; SALVANESCHI et al., 2017; MARUM; JONES; CUNNINGHAM, 2019): modeling and simulation, robotics, computer vision, web applications, sensor networks, game engines, embedded systems, and control and monitoring systems.

2 <https://akka.io/>
3 <https://projectreactor.io/>
4 <https://github.com/ReactiveX/RxJava>
5 https://survey.stackoverflow.co/2024/technology
6 <https://react-rxjs.org/>
7 <https://redux-observable.js.org/>
8 <https://angular.io/guide/rx-library>
9 <https://angular.dev/guide/signals>
10 Available at: <https://iot-analytics.com/iot-market-size/>. Accessed in: Oct 12, 2024.

In spite of their clear benefits, dataflow-based solutions such as RP have been supported by little research and literature evidence (SALVANESCHI, 2016; MOGK; SALVANESCHI; MEZINI, 2018), leading to a scarcity of the actual programming experience (MOGK; SALVANESCHI; MEZINI, 2018). RP has received much more attention from programmers and practitioners than the software engineering community (SALVANESCHI; MARGARA; TAMBURRELLI, 2015). Some exceptions include studies related to aspects such as high composability (SALVANESCHI; HINTZ; MEZINI, 2014) and improved comprehension when compared to the traditional Observer pattern (SALVANESCHI et al., 2014; SALVANESCHI et al., 2017). On the other hand, it was also noted that the learning curve and RP's relation to functional programming can be challenging for anyone interested in the paradigm (SALVANESCHI et al., 2017; MOGK; SALVANESCHI; MEZINI, 2018). Furthermore, there is not a consensus about the APIs offered by the implementations, so RP libraries commonly adopt different interfaces (MOGK, 2015). In fact, the excessive number of operators is a recurring design issue discussed by researchers (SALVANESCHI, 2016; SALVANESCHI et al., 2017). This prompted researches to propose systematic investigation about the designs of such solutions to better understand their impact and usability (MOGK, 2015; ZIMMERLE et al., 2022) and to fulfill lacking guidelines to drive better design choices for future RP developers (SALVANESCHI, 2016; MOGK, 2015).

1.2 RESEARCH GOAL

Modern software development heavily depends on the reuse promises of APIs (MYERS; STYLOS, 2016; RAUF; TROUBITSYNA; PORRES, 2019; VENIGALLA; CHIMALAKONDA, 2021). Their ubiquity, however, also introduced many complexities as good APIs are often hard to develop (HENNING, 2009), given the many design decisions are involved (HENNING, 2009; STYLOS; MYERS, 2007) and bad designs can easily impact thousands of programmers (HENNING, 2009). Programmers often found APIs hard to learn and use (MYERS; STYLOS, 2016; MURPHY et al., 2018), costing productivity, code inefficiencies, bugs, and, more worrying, significant security flaws (RAMA; KAK, 2015; HENNING, 2009; MYERS; STYLOS, 2016). Due to those problems, API usability has become a fundamental attribute for the API success. Usable APIs give joyness to programmers and enhance their productivity (HENNING, 2009; PICCIONI; FURIA; MEYER, 2013). They are well-documented, easy to use and memorize, and encourage reuse (HENNING, 2009; PICCIONI; FURIA; MEYER, 2013). Still, usability is often difficult to gather and understand (user studies are often considered expensive during API design (ZHANG et al., 2020)) or simply ignored by the API designers (BURNS et al., 2012; ZHANG et al., 2020).

Given the importance of API usability and difficulty of gathering related information, aligned with the RP importance throughout many areas (domains and platforms), the need for researches and investigation to attest the quality of the dataflow-based approaches (SALVANESCHI, 2016), the challenging learning process of RP, the disconnection between RP design and user experience, and the different design choices adopted by different RP APIs (e.g., divergent number of operators), the primary objective of this thesis is to measure the level of usability offered by those APIs, identify aspects that need improvements, and the main problems that are becoming an obstacle for providing better API usability to RP tools. Those problems, combined with the already observed complexity of RP, may be collaborating to make the paradigm even more challenging without the designers even be aware of them. Besides, an understanding of the current usability level of RP can help offering a better perspective at the exact level and points of improvements.

This thesis is divided in two studies. During the first study, we focus on information provided by the open-source community. Essentially, we conduct a mining study in GitHub (GH) repositories and the Stack Overflow (SO) platform. First, we concentrate on the API Size of RP tools. The number of operators has been a source of discussion in the reactive community (SALVANESCHI, 2016; SALVANESCHI et al., 2017), and it is a consequence of the strong connection with Functional Programming (FP) in which functions (combinators) are used to drive the business logic. This in turn can complicate the library usage (MOGK, 2015) and the mastering of the API (SALVANESCHI, 2016). Second, we uncover the problems RP users are facing and which of those are the most relevant (i.e., popular and difficult). Finally, we compare the most frequent operators found in the GH repositories and the most relevant SO topics.

The second study focuses on understanding the current level of usability offered by RP APIs, the strongest and weakest areas (e.g., understandability and learnability), and the problems observed. The study is carried by a mixed-methods research by exploring the use of computed metrics and a user-centered study. The explored metrics are based on previous studies (RAMA; KAK, 2015; VENIGALLA; CHIMALAKONDA, 2021) and their implementation generated an open-source tool which we called Usability Analyzer Experience (UAX). The user study was conducted by mostly following a qualitative approach to capture a detailed understanding of the developers' accounts. All the process was approved by our university's ethics board.

The studies and corresponding chapters have been published or submitted at important Software Engineering conferences (ZIMMERLE et al., 2022; ZIMMERLE; GAMA, 2024) and Journals (ZIMMERLE; GAMA, 2025). The explored approaches and results have also influenced other published researches (PEREIRA et al., 2023; FARIAS; ZIMMERLE; GAMA, 2024) and undergraduate thesis (PEREIRA, 2022; BARROS, 2022; FARIAS, 2023).

1.3 RESEARCH QUESTIONS

To achieve the research goals and aligned with the performed studies, the thesis aims to answer the following research questions:

RQ1. How frequently are RP operators used in open-source projects, and what does their usage reveal about usability and adoption?

RQ2. What challenges are RP users facing, and how they relate to API usability?

RQ3. To what extent are RP APIs usable, and what aspects are most affected?

The first two research questions are addressed by the results collected in the first study (Chapter 3), while RQ3 is the main focus of the second study (Chapter 4). To support the studies, a number of secondary research questions were defined and are explored in subsequent chapters (3 and 4):

RQ1.1 How much are the Rx operators being used in open source projects?

RQ2.1 What problems are RP developers facing?

RQ2.2 How do the operators present in the most relevant Stack Overflow questions and the usage frequency of Rx operators in open source projects relate?

RQ3.1 How easily can developers learn and understand RP APIs?

RQ3.2 To what extent do RP APIs contribute to code cleanliness, reliability, and abstraction from low-level complexities?

RQ3.3 To what extent do RP APIs enhance code reuse and maintainability?

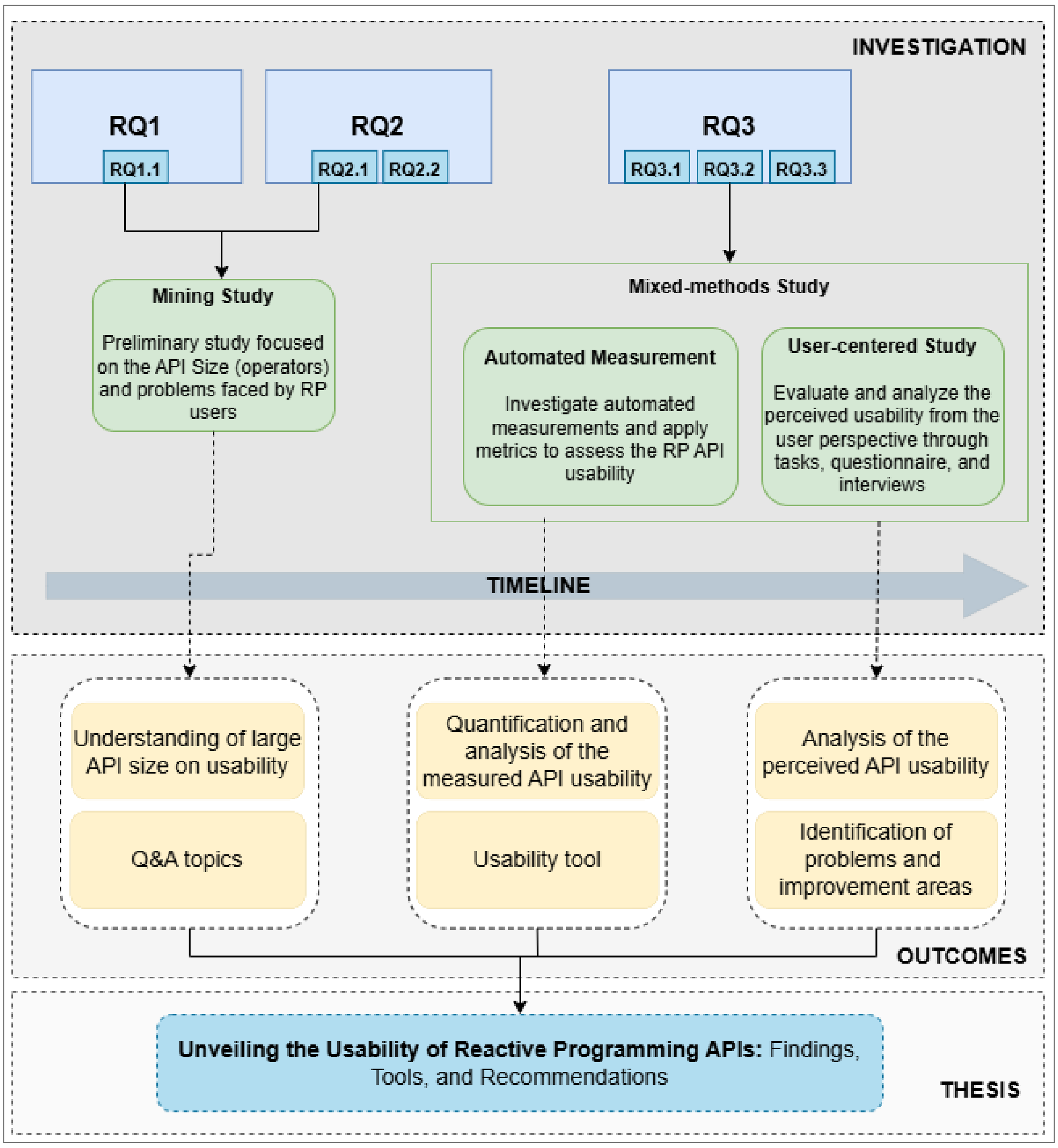
In the process of answering those questions, we explore different RP APIs, always looking into API with distinct characteristics such as syntax, semantics, and the number of operators for instance. Most of the work has been concentrated on JavaScript (JS) libraries given its inviting environment for event-driven, reactive tools.

1.4 RESEARCH METHODS

Figure 1 depicts a comprehensive overview of the leveraged methods and the expected outputs which ultimately comprise the thesis. The blue boxes with RQ1, RQ2, and RQ3 represent the research questions which are linked to the research methods (green boxes) described in Chapters 3 (mining study) and 4 (mixed-methods study). The boxes in the outcomes stage describe the results obtained from the research methods. Finally, the lowest blue box illustrates the thesis that includes all the produced findings.

All scripts and materials used and produced during the execution of the methods are publicly available at (<https://github.com/carloszimm/thesis>). The repository has been archived through Software Heritage and Zenodo to allow transparency and future reproducibility.

Figure 1 – Overview of the Research Methods.


INVESTIGATION
RQ1
RQ1.1
RQ2
RQ2.1
RQ2.2
RQ3
RQ3.1
RQ3.2
RQ3.3
Mining Study
Preliminary study focused on the API Size (operators) and problems faced by RP users
Mixed-methods Study
Automated Measurement
Investigate automated measurements and apply metrics to assess the RP API usability
User-centered Study
Evaluate and analyze the perceived usability from the user perspective through tasks, questionnaire, and interviews
TIMELINE
Understanding of large API size on usability
Q&A topics
Quantification and analysis of the measured API usability
Usability tool
Analysis of the perceived API usability
Identification of problems and improvement areas
OUTCOMES
Unveiling the Usability of Reactive Programming APIs: Findings, Tools, and Recommendations
THESIS

Source: Elaborated by the author (2024)

1.5 DOCUMENT STRUCTURE

The remainder of this document is structured as follows:

- **Chapter 2** exposes the fundamental areas behind the concepts explored in this thesis.
- **Chapter 3** outlines the methodology and results of our mining study linked to RQ1 and RQ2.
- **Chapter 4** presents the research methods and results of the mixed-methods research connected to RQ3.

- **Chapter 5** highlights the related works.
- **Chapter 6** addresses the final remarks, including the contributions and future works.

2 FUNDAMENTAL CONCEPTS

This chapter lays down some concepts necessary to fully understand the ideas explored in this thesis. Section 2.1 presents the core ideas behind the event-driven paradigm and the Observer pattern, often explored in object-oriented area. Section 2.2 covers the RP paradigm, presenting its definition, basic abstractions, usage domains, and state of art. Finally, Section 2.3 overviews the API, elucidating its definition, benefits, difficulties, and the usability topic, central to the thesis.

2.1 EVENT-DRIVEN PARADIGM AND THE OBSERVER PATTERN

Reactive applications, also called interactive or event-driven, is mainly supported by the event-driven paradigm which advocates a program defined as a reaction to events (TUCKER; NOONAN, 2002). This contrasts to the usual control sequence often found in imperative programs in which the control is defined by the sequence of statements (TUCKER; NOONAN, 2002). Instead, in the event-driven model, data arrival is the fundamental actor that drives the execution sequence of operations (TUCKER; NOONAN, 2002). A number of challenges accompanies such programs given the asynchronous nature of events (TUCKER; NOONAN, 2002) and the impossibility of controlling the order of events aligned with the manual (often complex) handling of dependency on state changes (BAINOMUGISHA et al., 2013). Callbacks and inversion of control have been the main approach to structure the event-driven logic (DRECHSLER et al., 2014; BAINOMUGISHA et al., 2013). Callbacks, however, can introduce a lot of complexities to code such as low undestandability and composability, boilerplate code, and side effects (KAMBONA; BOIX; MEUTER, 2013). In fact, as the application scales, it is common to evolve into a tangling, spaghetti code, a pattern called callback hell or pyramid of doom (KAMBONA; BOIX; MEUTER, 2013; BAINOMUGISHA et al., 2013), deeming callbacks as the modern 'goto' (KAMBONA; BOIX; MEUTER, 2013). As a result, event-based programs are known to be difficult to design, implement, and maintain (SALVANESCHI; MARGARA; TAMBURRELLI, 2015).

The object-oriented approach to event-driven programming often relies on the Observer pattern (SALVANESCHI; HINTZ; MEZINI, 2014; SALVANESCHI; MARGARA; TAMBURRELLI, 2015). Such pattern raises the level of abstraction by decoupling observables (producers) from observers (consumers) (SALVANESCHI; MARGARA; TAMBURRELLI, 2015; SALVANESCHI et al., 2017) and allowing a one-to-many interaction. However, as pointed by Salvaneschi, Hintz and Mezini (2014), the contributions of the pattern to managing complexity are minimal, and a number of problems have been pointed out that can even hurt the readability of code: cluttering of code, dynamic registration, side effects within callbacks, inversion of control and logical connection among entities, and absence of composability (SALVANESCHI; HINTZ; MEZINI, 2014; SALVANESCHI; MARGARA; TAMBURRELLI, 2015; SALVANESCHI et al., 2017). Given all the difficulties of coding reactive applications, a need for new abstractions to both manage event handling logic and state changes was created (BAINOMUGISHA et al., 2013) and a number of different approaches have emerged in the last years, including RP.

2.2 REACTIVE PROGRAMMING

RP is a paradigm which was conceived to facilitate the construction of interactive applications through the use of dedicated abstractions (BAINOMUGISHA et al., 2013; MARGARA; SALVANESCHI, 2014; SALVANESCHI; MARGARA; TAMBURRELLI, 2015); however, it is often simply defined as programming with asynchronous data stream. Two basic abstractions are normally included (BAINOMUGISHA et al., 2013; SALVANESCHI, 2016; BLACKHEATH, 2016), *behavior* (also called cell, property, or signal) for continuous time-changing values and *event* (also known as event stream, stream, observable, or signal) for discrete time-changing values. These abstractions designed after three concepts (MARGARA; SALVANESCHI, 2014): (*i*) time-changing values, (*ii*) dependency tracking, and (*iii*) automatic propagation of updates. Furthermore, both abstractions are complementary or dual and one could be used to represent or produce the other (MEYEROVICH et al., 2009; BAINOMUGISHA et al., 2013). Theoretically, *behaviors* always have a value and acts mostly like a variable equipped with automatic propagation of changes, while *event streams* represent a potently infinite source of discrete events (MEYEROVICH et al., 2009).

The paradigm origins can be traced back to Functional Reactive Programming (FRP) where it was offered through Haskell in the context of strictly functional programming and primarily used to modeling animations (SALVANESCHI et al., 2017; MOGK; SALVANESCHI; MEZINI, 2018). Besides, its theory is also rooted in the area of signal processing (MEYEROVICH et al., 2009). A survey of the RP languages is presented by Bainomugisha et al. (2013), classifying the languages under three categories: The FRP siblings, the cousins of RP, and synchronous, dataflow, synchronous dataflow languages. In its essence, RP belongs to a class of languages based on the dataflow style which have permeated various areas in the last years such as Big Data, stream processing, streams for collection, real time analytics, and RP itself (SALVANESCHI, 2016). In fact, the implementation of RP corresponds to a dataflow graph (MEYEROVICH et al., 2009; SALVANESCHI; HINTZ; MEZINI, 2014; BLACKHEATH, 2016).

Given the advantages of RP, many libraries and extensions have been incorporated in varying languages (BAINOMUGISHA et al., 2013). The family of libraries called Reactive Extensions (ReactiveX or Rx) is among the most popular families of libraries nowadays, specially after the Netflix's successful use case (SALVANESCHI et al., 2017). As of the JVM (Java Virtual Machine) version 9, a set of reactive interfaces were incorporated under the class *java.util.concurrent.Flow*, all inspired by the specification for the JVM called Reactive Streams[1]. That specification details a minimal set of interfaces with non-blocking back-pressure and many libraries currently comply to (e.g., Akka Streams[2], Reactor[3], and RxJava[4]). In the realm of JavaScript, a number of different libraries can be found for RP and a proposal[5], based on the Rx's stream type named Observable, is slowly making its way to the language. Furthermore, a number of tools have been inspired by the ideas of RP like the famous React.js library (SALVANESCHI et al., 2017) and the Vue.js framework.

The classical usage domains of RP correspond to animation and user interfaces (SALVANESCHI, 2016; MOGK; SALVANESCHI; MEZINI, 2018), but its application has been expanding since its conception to other areas as well. For instance, Zimmerle and Gama (2018a), based on the ideas of Margara and Salvaneschi (2013), created a prototype library where they showed that it was possible to express complex event processing (CUGOLA; MARGARA, 2012) with RP. Other applications of RP include (BAINOMUGISHA et al., 2013; SALVANESCHI; MARGARA; TAMBURRELLI, 2015; SALVANESCHI et al., 2017; MARUM; JONES; CUNNINGHAM, 2019; PROENÇA; BAQUERO, 2017; DIAS; FARIA; FERREIRA, 2018; ZIMMERLE; GAMA, 2018b; STERZ et al., 2021; SOKOLOWSKI; MARTENS; SALVANESCHI, 2019): Internet of things, modeling and simulation, robotics, computer vision, web applications, sensor networks, WiFi firmware, game engines, high performance computing, embedded systems, and control and monitoring systems.

Recent efforts have focused on enabling RP in a distributed setting (MARGARA; SALVANESCHI, 2014; DRECHSLER et al., 2014; MARGARA; SALVANESCHI, 2018) with consistency guaran-

[1] <https://www.reactive-streams.org/>
[2] <https://akka.io/>
[3] <https://projectreactor.io/>
[4] <https://github.com/ReactiveX/RxJava>
[5] <https://github.com/tc39/proposal-observable>

tees. As noted by Bainomugisha et al. (2013), the appliance of RP in a distributed environment with consistency guarantees was an open challenge in the area and it was only supported partially (with no consistency guarantees) based on the support of the host language. Another area of interest was the support for debugging. The unsuitability of usual debugging tools was recognized (SALVANESCHI; MEZINI, 2016; MOGK et al., 2018; BANKEN; MEIJER; GOUSIOS, 2018), driving some research efforts. Salvaneschi and Mezini (2016) proposed a technique and tool (Reactive Debugging and Reactive Inspector, respectively) to help visualizing the dataflow of the application, with extensions to provide live programming (MOGK et al., 2018).

Salvaneschi, Hintz and Mezini (2014) presented the REScala library and, to validate the composability of their approach, they designed and executed an empirical assessment by contrasting program versions not using the library and those refactored with it. The result showed the increase of composable code while diminishing the use of callbacks. Salvaneschi et al. (2014), Salvaneschi et al. (2017) conducted a controlled experiment to attest the enhanced comprehension property often supported by RP proposers. Results demonstrated an enhancement of comprehension by using RP compared to the Observer pattern Through a qualitative examination of the participants' feedback, the authors also identified some points that collaborated to enhanced RP comprehension, like less boilerplate code and shorter code, better readability support, automated consistency of reactive state, declarative code, ease of composition and separation of concerns. Contrarily, the learning curve, the level of abstraction and FP aspects were the main obstacle.

2.2.1 Important Aspects

2.2.1.1 Interfaces and Operators

To fill the gap of lack of systematization and the diversity of interfaces employed by the many RP libraries, Mogk (2015) surveyed the most common interfaces explored. According to the survey, there are three types of abstraction interfaces (MOGK, 2015): event combinators (libraries based on combinators to support the event stream abstraction), signal (behaviors) combinators (libraries based on combinators to support the signal, also called behavior, abstraction), and signal (behaviors) expressions (expressions that seamlessly allows the use of signal and, at the same time, automatic propagate changes). From those, event and signal combinators are the most adopted style; while providing great benefits (declarative pipeline of operators, easier sharing of combinator syntax among languages, and extensibility through the addition of new combinators), they also carry many challenges like the high number of available operators, a possible obstacle for learning and understanding. Signal expression offers a simpler choice since they better integrate with the host language and does not require a extensive set of operators. However, this type of interface is not good to represent both event streams and signals, and it also has challenges such as dependency discovery in dynamic scope and implicit value access. Future libraries should support both types of interfaces (combinators and expressions), while considering the implications raised in the study: keeping the set of combinators focused, avoiding combinators with surprising behavior, and including safe and easy points of extension (MOGK, 2015).

Combinators are functions used in functional programming (FP) to drive the business logic; they often return a new instance of the same type as part of its immutable, composable property. Event combinators provide the greatest variety of operators, while signal combinators provide a few to combine signal values for instance (MOGK, 2015). In general, event streams have access to operations for transformation, filtering, splitting, merging, selection, and grouping (MOGK, 2015). Well-known functional combinators like `map`, `filter`, and `reduce` are examples of operators often available. According to Bonér and Klang (2017), stream-based operators like count, triggers, or windowing are frequently supplied along combinators. Signal combinators, on the other hand, have at their disposal operators to combining and flattening (MOGK, 2015). Flattening operators are commonly found in event combinators as well, so it is easy to flat a stream of streams. Moreover, there are combinators that allow the combinations and conversions between event streams and signals (behaviors) (MOGK, 2015).

Lifting is a FP concept that is directly related to the last categorization of interfaces: signal expressions. In the context of RP, it allows the signal (behavior) abstraction to use common expression operators (bring to its context) while having the automatic propagation of changes. There are three strategies of RP lifting (BAINOMUGISHA et al., 2013):

- **Explicit lifting:** More common in combinator libraries, it corresponds to providing a *lift* operator to allow the combination of behaviors.
- **Manual lifting:** Letting the developer responsible for acquiring the current behaviors' values and applying the desired logic.
- **Implicit lifting:** Commonly adopted by languages with builtin RP. It makes possible to

mix common variables with signals in the language expressions.

Implicit lifting is the strategy adopted by signal expression interface, and it makes the reactive code more transparent (BAINOMUGISHA et al., 2013). Nevertheless, this transparency can hinder the distinction between a normal variable and a behavior (MOGK, 2015), so some signal expression interfaces opt for manual lifting (acquiring the current value of the behavior manually). Event and signal combinators do not require any special support besides the library or extension for a language. Signal expressions, on the other hand, require some compilation (e.g., Flapjax when used as a language (MEYEROVICH et al., 2009)) or the use of Domain-specific languages (e.g., REScala (MOGK; SALVANESCHI; MEZINI, 2018)). Libraries like the famous Rx are classified under the event combinators, and this can be explained by the fact that Rx primarily treats everything as a stream. Only two libraries were included as presenting all interfaces in the work of Mogk (2015): REScala (SALVANESCHI; HINTZ; MEZINI, 2014) and scala.react (MAIER; ODERSKY, 2012).

Number of operators. Operators are a recurrent, debatable aspect in the RP area (SALVANESCHI, 2016; SALVANESCHI et al., 2017; MOGK; SALVANESCHI; MEZINI, 2018; ZIMMERLE et al., 2022). More specifically, researchers mostly discuss about the API size of RP languages based on combinators, which can overgrow and pose an obstacle to new comers and non-experts (SALVANESCHI, 2016; SALVANESCHI et al., 2017; MOGK; SALVANESCHI; MEZINI, 2018). To exemplify the situation, researchers often cite Rx which lists, in its website[6], 70 to 80 core operators and more than 400 (core and variant operators) in total. Meanwhile, other APIs like Fran and REScala (SALVANESCHI; HINTZ; MEZINI, 2014), employ less operators. REScala, for instance, has between 20 (SALVANESCHI et al., 2017) to 40 (MOGK; SALVANESCHI; MEZINI, 2018) combinators, and it focus on small, more general concepts, so programmers can compose derived operators from basic ones and easily explain the composed semantics (MOGK; SALVANESCHI; MEZINI, 2018). As result, researchers believe that the focus of future implementation should be on identifying the small set of concepts and keep the RP APIs small (SALVANESCHI et al., 2017; MOGK; SALVANESCHI; MEZINI, 2018).

[6] <https://reactivex.io/documentation/operators.html>.

2.2.1.2 Glitches

Glitches are an inconsistency during the updates related to the propagation of changes (BAINOMUGISHA et al., 2013). This terminology comes from signal processing (MEYEROVICH et al., 2009), and it can only occurs in push-based systems (BAINOMUGISHA et al., 2013). The property of not presenting glitches is called glitch avoidance or glitch freedom (MEYEROVICH et al., 2009; BAINOMUGISHA et al., 2013). To guarantee glitch freedom, RP solutions often employ a topological order of evaluation in the dataflow graph (MEYEROVICH et al., 2009; SALVANESCHI; HINTZ; MEZINI, 2014). Examples of RP libraries that guarantee consistent results include: Bacon.js[7], Flapjax (MEYEROVICH et al., 2009), and REScala (SALVANESCHI; HINTZ; MEZINI, 2014). Rx, in spite of its popularity, does not prevent inconsistencies ((BLACKHEATH, 2016)), for instance; however, if this can become a problem or not to a user seems to depend on the use of appropriate operators and abstraction[8].

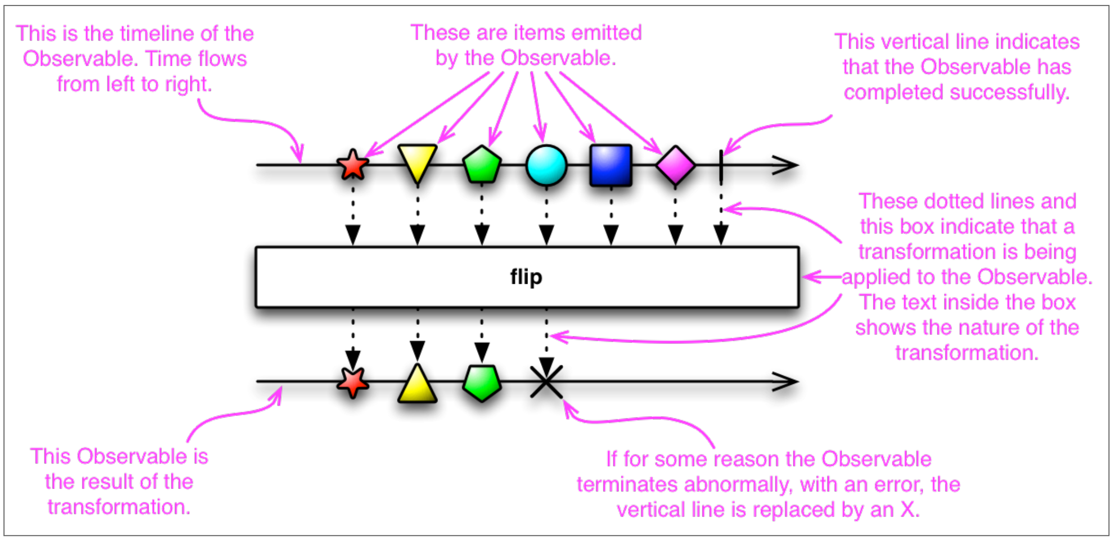
2.2.1.3 Visual Support

In the RP area, one of the few visual supporting tools is the well-known marbles diagrams. Those diagrams were a graphical depiction brought by the Rx project. They are specifically useful to provide a visual way of envision the stream and thus helping the mental model produced by the developer. As noted in the qualitative inquiry conducted by Salvaneschi et al. (2017), RP raises the level of abstraction in a way that sometimes may be hard to follow the code flow, requiring the user to heavily rely on the runtime. Therefore, a diagram aid is a very helpful mechanism in the RP world. Figure 2 explains how a marble diagram works: (*i*) timelines represent event streams (called *Observable* in Rx); (*ii*) each timeline is composed by items (marbles) representing the stream emissions; (*iii*) the rectangle denotes an operations that, when applied, creates a new stream (timeline) with emissions resulted from the appliance of the operator. An important point in this depiction is the presence of a vertical line in the upper straight line, and an X in the bottom line. Those are used to represent the events of completion and error. In Rx (as well some other libraries), streams can either emit a next event or data (similar to the nomenclature of the Iterator pattern), an error event (if some error is caught along the stream), or a completion event to signal the successful completion

[7] <https://baconjs.github.io/>
[8] <https://staltz.com/rx-glitches-arent-actually-a-problem>.

of a stream (if the current stream in fact can complete).

Figure 2 – Marble Diagram explained.


This is the timeline of the Observable. Time flows from left to right.
These are items emitted by the Observable.
This vertical line indicates that the Observable has completed successfully.
These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.
flip
This Observable is the result of the transformation.
If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

Source: ReactiveX Documentation (2016)

2.2.2 Reactive Programming Examples

Listings 1 and 2 present examples of two equivalent programs constructed with the use of the Observer pattern and an RP API, respectively. The code is built in JS given its simplicity and since it is well explored during the thesis. Nonetheless, Appendices A.1 and A.2 contain equivalent Java and Swift versions, languages also explored in the present work (e.g., in Chapter 3). All examples use the versatile Rx library to demonstrate the reactive codes which provides implementation for many different languages including the ones in the examples. The codes focus on Hypertext Transfer Protocol (HTTP) requests similar to the tasks in Section 4.1.3.2, and they show fictitious HTTP requests with automatic retries (three in total). The differences are notorious. For example, different from the Rx APIs, the Observer pattern does not include a native, standard way to deal with errors and completion. This ability to deal with current emissions, errors and completion signals is possible thanks to the `Observer` interface provided in Rx; this interface represents the consumer part in the traditional Observer pattern, and, in Rx, an object implementing such interface (each method in the `Observer` interface normally receives a lambda expressions/arrow as detailed in lines 10 and 12 of Listing 2) is passed to a `subscribe` method of the stream type `Observable` (i.e., the producer of data – line 9 of Listing 2). Every data type is treated as a possible producer of data, and the API encourages the user to wrap those producers in a `Observable` type; thus, reactive codes are normally constructed as a composition of reactive (stream) types with many operators available to flat the possible nested structure and avoid problems found in the use of callbacks (e.g., callback hell).

In general, RP APIs often provides factory functions that create new streams according to the arguments provided. In line 5 of Listing 2, the code uses a helper factory function (`getJSON`) provided by the library that makes the HTTP request and automatically wraps it in an Observable (stream) type (alternatively, one could use the `from` factory function to wrap a `Promise` returned by the JavaScript function `fetch`); this wrapping in an `Observable` object allows the chaining or piping of functions provided by the Rx API to transform the emitted values (forming the so-called pipeline) before the final consumption by the `Observer` objects. In the RxJS example, we only used the `retry` function, but the ReactiveX project provides an extensive number of operators[9]; this in fact shows that the API allows different choices which is good (OUSTERHOUT, 2018), but may also impact the introduction of new users (PICCIONI; FURIA; MEYER, 2013).

The retry concept is more complex with the Observer pattern given the manual need of implementation; that can be seen in the scope of the function `fetchDataWithRetry` in Listing 1 which uses an inner function called `attemptRequest` to implement the requests with the help of a closure variable to track the number of attempts. As a result, the RP logic becomes more succinct (less boilerplate code) and clearer.

Source Code 1 – HTTP request simulation with automatic retries using the Observer pattern in JavaScript

```
class Subject {
  constructor() {
    this.observers = [];
  }

  subscribe(observer) {
    this.observers.push(observer);
  }

  notify(data) {
    this.observers.forEach(observer => observer.next(data));
  }

  error(error) {
    this.observers.forEach(observer => observer.error(error));
  }
```

[9] <https://reactivex.io/documentation/operators.html>.

```

  complete() {
    this.observers.forEach(observer => observer.complete());
  }
}

// Create a Subject instance
const subject = new Subject();

// Function to fetch data and notify observers
function fetchDataWithRetry(url, retries) {
  let attempts = 0;

  function attemptRequest() {
    attempts++;
    console.log(`Attempt ${attempts} to fetch data`);

    fetch(url)
      .then(response => {
        if (!response.ok) throw new Error(`HTTP error! Status:
            ${response.status}`);
        return response.json();
      })
      .then(data => {
        subject.notify(data);
        subject.complete();
      })
      .catch(error => {
        if (attempts < retries) {
          attemptRequest(); // Retry the request
        } else {
          subject.error(error); // Final failure
        }
      });
  }

  attemptRequest();
}

// Observers that subscribe to the subject
const observer1 = {
  next: data => console.log("Observer 1:", data),
  error: error => console.error("Observer 1 Final Error:", error.message),
  complete: () => console.log("Observer 1 Done"),
};

const observer2 = {
  next: data => console.log("Observer 2 received:", data),
  error: error => console.error("Observer 2 Final Error:", error.message),
  complete: () => console.log("Observer 2 Done"),
};

// Multiple observers subscribing to the same subject
subject.subscribe(observer1);
subject.subscribe(observer2);

// Start the fetch process
fetchDataWithRetry("https://api.example.com/data", 3);
```

Source: Elaborated by the author (2024)

Source Code 2 – HTTP request simulation with automatic retries using RxJS in JavaScript

```
import { ajax } from 'rxjs/ajax';
import { retry, catchError } from 'rxjs/operators';
import { of } from 'rxjs';

ajax.getJSON('https://api.example.com/data')
  .pipe(
    retry(3)
  )
  .subscribe({
    next: data => console.log(data),
    error: error => console.error(`Final Error ${error.message}`),
    complete: () => console.log("Done")
  });
```

Source: Elaborated by the author (2024)

2.3 APPLICATION PROGRAMMING INTERFACES

APIs correspond to the minimal information for the usage of program units or modules (well-defined parts of codes) in which is made available to application programmers (WATT, 2004). In other words, it is the piece of information that a developer needs to know in order to use a given module (OUSTERHOUT, 2018). They have become a key point for code reuse in which the majority of nowadays development is based upon (HENNING, 2009; MYERS; STYLOS, 2016). In fact, modern software design is heavily based on the modular design (OUSTERHOUT, 2018), allowing developers to better manage complexity. Also, it is through a module's interface that abstractions are provided (OUSTERHOUT, 2018), a concept that enabled the evolution of software by reducing complexity (HENNING, 2009). From simple functions to (web) services, lots of APIs are created and made available every day, becoming an important asset for companies and governments, a hundred billion dollar market (MURPHY et al., 2018).

Interfaces have two aspects: formal and informal (OUSTERHOUT, 2018). Formal information constitute the visible parts in code such as the signature of a method or the set of public methods of a class. All other aspects of the API (e.g., code behavior or constraints related to a module) belong to the informal part. Among those, only the formal aspects can be validated by the programming language (OUSTERHOUT, 2018). On the other hand, the informal information are the most numerous and complex part (often described by comments and documentation), and the language cannot enforce their completeness and accuracy (OUSTERHOUT, 2018).

There are a lot of design decisions involved in the creation of APIs. Each decision may impact the API in different ways, and Stylos and Myers (2007) conducted a mapping of the decisions and the corresponding recommendations in the context of object-oriented programming. For instance, one dimension of design decision involves architectural (e.g, which design pattern will be used) and language level (e.g., should the method be synchronized or not) decisions. The other dimension structural decisions (i.e., higher-level aspects like the the definition of the API classes or interfaces) versus class decision decisions (i.e., lower-level decisions such as the choice of class methods).

Given the many design decisions involved, it is much more easy to create bad APIs than good ones (HENNING, 2009). In fact, constantly APIs are complex, requiring sometimes significant time to learn and resulting in incorrect usages, bugs, and security flaws (MYERS; STYLOS, 2016). APIs are in the end a human-machine interface (HENNING, 2009; MYERS; STYLOS, 2016), where the main user are human begins, and the level of offered usability dictates how comprehensible and usable an API is (MCLELLAN et al., 1998). As a result, APIs built for the same purpose but with distinct design approaches can provide a better or worse experience to users.

2.3.1 Usability

APIs have become an ubiquitous asset that involves many quality attributes but usability has shown to be one of great and critical importance lately (RAUF; TROUBITSYNA; PORRES, 2019; BURNS et al., 2012). An API is an interface geared towards human developers (MYERS; STYLOS, 2016), similar to graphical interfaces, so usability considerations and human-computer interactions (HCI) principles are very valuable (MYERS; STYLOS, 2016; MURPHY et al., 2018), but often neglected by API designers (BURNS et al., 2012). The interest in API usability, in fact, only started to gain more attention after 2000 (MYERS; STYLOS, 2016), in which the activities were slowly being referred to as DevX or DX (developer experience), analogously to UX (user experience) (MYERS; STYLOS, 2016; MURPHY et al., 2018).

The definition of usability varies among standards and researchers given its subject nature (RAUF; TROUBITSYNA; PORRES, 2019; PICCIONI; FURIA; MEYER, 2013); nonetheless, in a summarized way, an API usability indicates how easy it is to use, by developers in a given context, and learn an API (RAUF; TROUBITSYNA; PORRES, 2019). A number of different attributes and factors are used throughout definitions and studies (RAUF; TROUBITSYNA; PORRES, 2019). Learnability, satisfaction and efficiency are the most used attributes found in definitions, whereas learnability, efficiency, and understandability are the most used factors present in studies (RAUF; TROUBITSYNA; PORRES, 2019).

The usability of an API can produce different impacts: it can encourage or discourage the use of an API (i.e., its adoption and retention), impact productivity and satisfaction of developers, and influence the quality of the produced code (MURPHY et al., 2018; RAUF; TROUBITSYNA; PORRES, 2019). Bugs and security problems are examples of poor or incorrectly usage of APIs (MYERS; STYLOS, 2016; MURPHY et al., 2018) that frequently impact negatively big organizations (WIJAYARATHNA; ARACHCHILAGE; SLAY, 2017). APIs are often difficult to use and learn (MYERS; STYLOS, 2016), and it is much more easy to create bad APIs than good ones (HENNING, 2009). A lot of causes have already been identified as reasons for difficulties, including API semantics, abstraction level, documentation's quality, error handling support, and confusing dependencies and preconditions (MURPHY et al., 2018). All of this corroborates to the importance of the API design in the process of creating usable APIs (MURPHY et al., 2018).

Different usability measurements methods exist, such as heuristic evaluation, thinking aloud, and surveys, but they incur in a set of constraints (e.g., experienced evaluators, representative set of users, functional product, etc.) and their applicability to the API area is still to be researched (SCHELLER; KÜHN, 2015). A very popular method in the area instead is the use of the CDN framework (LÓPEZ-FERNÁNDEZ et al., 2017). The CDN is a tool for designers that describe a set of vocabularies to measure the decision that most affect the usability of a notation (DIPROSE et al., 2017). This includes all sorts of notations including APIs (CLARKE, 2005). Despite its usability, the original dimensions and their applicability are often considered complex (e.g., the dimensions an their relations are hard to comprehend and assess (SCHELLER; KÜHN, 2015)), cost and time consuming, and very directed to the designers' perspective (DIPROSE et al., 2017; LÓPEZ-FERNÁNDEZ et al., 2017; SCHELLER; KÜHN, 2015). In this way, researchers have proposed variations of the framework more centered in the developers' perspective and dimensions that important qualities of software like understandability and learnability (PICCIONI; FURIA; MEYER, 2013; LÓPEZ-FERNÁNDEZ et al., 2017). For example, López-Fernández et al. (2017) refined the CDN version proposed by Piccioni, Furia and Meyer (2013) and included the following reasons for the practicability of their version for qualitative researches: (*i*) the questionnaire and the dimensions become more user-centered rather than being centered at the designer's perspective, allowing a better optimization for API users; (*ii*) the understanding and evaluation of the dimensions become simpler given its fewer number and its intimate and positive usability characteristic; this implies that the higher evaluation of dimension, the better is the usability in respect to that dimension; (*iii*) the new dimensions can be linked to developers' activities, simplifying more the questionnaire and the evaluation of the results; exploratory learning (i.e., understanding the API) encompasses understandability and learning dimensions, exploratory design (i.e., exploring the API to design applications) relates to abstraction and expressiveness, and maintenance (i.e., corrections and new features) is linked to reusability.

In spite of its popularity, CDN still relies on users performing some tasks (DIPROSE et al., 2017), which prompted some researches to invest in usability metrics in the last years (RAUF; TROUBITSYNA; PORRES, 2019). Souza and Bentolila (2009) defined the API usability as a function of its complexity based on three metrics and use a visualization tool, Metrix, to aid in the identification of complex API areas. However, the tool has not being maintained (RAUF; TROUBITSYNA; PORRES, 2019). Rama and Kak (2015) define a set of eight metrics to evaluate the usability of methods based on common beliefs and are accompanied by math formulas. The authors show their use in seven Java APIs by using their proprietary tool (RAUF; TROUBITSYNA; PORRES, 2019). That same set of metrics was latter used to analyze game engines' APIs (VENIGALLA; CHIMALAKONDA, 2021). A summary of those metrics is displayed in Table 1. The work of Scheller and Kühn (2015) seems to be a promise venue, in which they reused many metrics defined by Rama and Kak (2015) to create new metrics that consider other characteristics beyond methods (e.g., annotations) and the code context of use. However, the automated tool using those metrics is not available (RAUF; TROUBITSYNA; PORRES, 2019).

Table 1 – A summary of the eight structural metrics to evaluate API usability.

Metric	Definition
AMNOI (API Method Name Overloading Index)	It quantifies the degree to which the various overload definitions of a function yield disparate return types. The lesser the metric score, the greater is the number of overloads that return different types. As an example, Java 5 provided two methods called add in the class `javax.naming.directory.Attribute` that would return either `void` or boolean (RAMA; KAK, 2015).
AMNCI (API Method Name Confusion Index)	It is based on three name-abuse patterns listed by Rama and Kak (RAMA; KAK, 2015) which prescribes how to obtain the canonical forms of the functions identifiers and, from there, to generate a list of confusing function names. The greater the number of confusing names, the lesser tends to be the metric score. For instance, functions' names that only differ by numbers inserted at the end of their identifiers like `writeByteArray` and `writeByteArray2` may confuse the API users (RAMA; KAK, 2015).
AMGI (API Method Grouping Index)	It measures the extent to which semantically similar functions are grouped (e.g., in the same class) rather than dispersed. The semantic similarity is defined based on keywords extracted from the function names; for instance, functions called `mergeMap`, `concatMap` and `switchMap` could all be considered semantically similar.
APLCI (API Parameter List Consistency Index)	It assess the consistency in terms of parameter name ordering across functions' definitions. For example, for those API functions that share at least two parameter identifiers, if those parameters all follow the same order across the functions, this would result in a good score for this metric.
APXI (API Parameter List Complexity Index)	It deals with the length of function parameter and the runs of parameters of the same type. For example, across functions' signatures, long lists of parameters and sequences of parameters with the same data type are likely to worsen the user experience and produce lower score for the metric.
ADI (API Documentation Index)	It examines the number of words contained in the functions' documentation. The metric is based on a threshold, which defines a minimum number of words for every function documentation.
AESI (API Exception Specificity Index)	It deals with the specialization and generalization of checked exceptions. Specialized exceptions offer better usability than general ones for APIs (RAMA; KAK, 2015). So, the analyses revolve around an examination of the exception inheritance trees of exceptions declared by the functions exposed by the API.
ATSI (API Thread Safety Index)	It measures the usability regarding thread safety by analyzing the set of functions that have 'thread' or 'safe' in their declaration, more specifically in the documentation associated with the declaration, related the overall set of functions made available by the API.

Source: RAMA; KAK (2015) and VENIGALLA; CHIMALAKONDA (2021)

3 MINING THE USAGE OF REACTIVE PROGRAMMING APIS ON GITHUB AND STACK OVERFLOW

This Chapter presents the methods and results obtained trough the execution of the first study pointed at Section 1.2 and depicted in the first half of Figure 1. The study resulted in a publication titled "Mining the Usage of Reactive Programming APIs" which was presented in the Mining Software Repositories Conference (ZIMMERLE et al., 2022).

In this part of the thesis, we conducted a mining study on GitHub (GH) and Stack Overflow (SO) hoping to answer the following (secondary) research questions:

- **RQ1.** How frequently are RP operators used in open-source projects, and what does their usage reveal about usability and adoption?
- **RQ1.1** How much are the Rx operators being used in open source projects?
- **RQ2.** What challenges are RP users facing, and how they relate to API usability?
- **RQ2.1** What problems are RP developers facing?
- **RQ2.2** How do the operators present in the most relevant Stack Overflow questions and the usage frequency of Rx operators in open source projects relate?

To answer those questions, we explored the three Rx libraries with the most GH repositories: RxJava, RxJS, and RxSwift. Together, those libraries represent languages whose usage vary extensively, including UI, mobile, and web development.

3.1 METHODOLOGY

In this section, we explain how we mined GH repositories (Section 3.1.1) and SO questions and answers (Section 3.1.2) to address the three, secondary research questions. In general, the methodology applied to GH can be summarized as:

1. Search for Rx repositories applying the defined star filter and store the information.
2. Retrieve the repositories based on the stored information.
3. Search for Rx operators within the download repositories **(RQ1.1)**.

Conversely, the overall SO methodology follows the following stages:

1. Download Stack Exchange Data Explorer's data using Rx libraries' tags.
2. Remove duplicates and consolidate the result files.
3. Preprocess posts.
4. Run LDA followed by topics' inference **(RQ2.1)**.
5. Determine topics' relevance (popularity and difficulty) and search for operators among the posts **(RQ2.2)**.

Both Sections 3.1.1 and 3.1.2 access lists of operators of the different Rx libraries under analysis, and those lists were created by scraping official repositories of the distributions. The operators of RxJava and RxJS were extracted from their repositories on GH[1,2], while the ones from RxSwift were taken from the Rx website[3]. By the time of the last scraping (September 27, 2021), the libraries were in the versions: 3.1.1 (RxJava), 7.3.0 (RxJS), and 5.1.1 (RxSwift).

3.1.1 GitHub Mining

We used the GH API, which accounts for almost 40% of the solutions used in the mining field (COSENTINO; IZQUIERDO; CABOT, 2016) and allows to acquire repository data and metadata as well as commit messages, pull request information, etc. Researchers should take into account that many GH repositories are merely used to other concerns beside software development like storing personal data and (possibly inactive) repositories (KALLIAMVAKOU et al., 2016). Social features like stars were used as a selection filter previously (WEN et al., 2020; XU et al., 2020) as an indicative of a repository's popularity and a favoring factor among developers. Thus, we used it to exclude potentially unimportant repositories. Following other approaches (WEN et al., 2020; HENKEL et al., 2020), we set the exclusion threshold to consider repositories that have 10 or more stars. Table 2, column Repositories, outlines the total of repositories using the five most used Rx libraries along with the total of projects with zero stars and those with 10 or more stars. One can verify that repositories with 0 stars, and presumably low popularity, account for a big share of each library's total, supporting our choice of using stars as a filter.

1 <https://github.com/ReactiveX/RxJava/wiki/Operator-Matrix>
2 <https://github.com/ReactiveX/rxjs/blob/master/docs_app/content/guide/operators.md>
3 <https://reactivex.io/documentation/operators.html>

Table 2 – Information on the repositories using the five most used Rx libraries and the mined ones along with their star information, sorted by their total.

Library	Repositories			Mined Repositories			
	Total	Stars		Total	Stars		
		$= 0$	≥ 10		Min	Max	Median
RxJava	16,394	10,885	1,450	1,430	10	10,404	43
RxJS	16,380	12,740	818	797	10	16,835	30
RxSwift	5,505	3,833	402	401	10	13,644	37
RxKotlin	626	369	78	-	-	-	-
RxDart	493	274	73	-	-	-	-

*N*ote: Last updated on Jan 7, 2022.

Source: Elaborated by the author (2022)

The selection of the projects to be mined took into consideration the most used libraries. As depicted in Table 2, column Repositories, RxJava has the highest share of projects followed by RxJS and RxSwift. Considering the defined star filter (≥ 10) and aiming to obtain a significant sample size ($>$100), we decided to select the first three libraries (RxJava, RxJS and RxSwift). RxKotlin and RxDart were initially being considered to be included in the study but they would produce a small sample ($<$100 projects). Furthermore, RxKotlin takes most of its operators from RxJava, with only a small portion reserved for extension functions as delineated in its GH page[4]. Consequently, those libraries would not truly contribute to the study's objectives.

The actual sample size used corresponded to the de facto population of repositories with ≥ 10 stars for each investigated library. We selected the entire population willing not to incur in sampling errors and not to be unfair when defining a specific sample size for the three libraries when the population of dependent repositories has different dimensions.

The first step in our workflow was to look for Rx repositories, by using the name of the Rx libraries, along with the defined star filter. This search was then conducted by leveraging the 'search repositories' feature from the GH API. Afterwards, with that information saved in JavaScript Object Notation (JSON) files, we executed a script to download the repositories as tarball files. Given that it would not be feasible to store all the tarball files with our scripts for future replications, we stored the information about the downloaded files in a JSON file containing sufficient information to acquire them (e.g., a URL to download the file having the SHA1 hash of last commit already set) as well as details about the repository that file belongs to. This script took into consideration the exclusion of repositories that belong to

[4] <https://github.com/ReactiveX/RxKotlin>

official ReactiveX users[5] such as 'ReactiveX' or 'Reactive-Extensions'. The information about the retrieved and processed repositories is displayed in Table 2, column Mined Repositories[6]. Having the downloaded repositories, a final script was executed to search for the Rx operators among the project files. The search was conducted by using a regular expression (regex), looking for operator invocations, either method (for method chaining pattern found in RxJava and RxSwift, for instance) or function (for function used with RxJS pipe method) calls. Before the actual counting for operators, a series of filters were used. First, a file extension filter to consider only files linked to each Rx libraries' language. Second, we checked whether the file had any mention to the Rx distribution considered at that moment, which would correspond to some kind of import of the library in the file. Finally, we removed strings and comments of every file to avoid false positives from those constructs.

Similar operators of other APIs. A few operators like map and `filter` are often found in other APIs like the ones that deal with collections. In this way, there was a chance of introducing false positives coming from those interfaces; however, the probability of such occurrences should be very low. Libraries like Rx can wrap any type of value available in the target language (there are many creational methods for that purpose in the API), so the program works as a composition of those streams of values. In this way, it reduces the need to invoking methods from other libraries (e.g., Java `java.util.stream`). In fact, we checked Java files (the language with more mined projects – Table 2) that imported RxJava along with other Collection-like libraries: Java Streams[7], Eclipse Collections[8], Apache's CollectionUtils[9], and Guava's Collections2[10]. Results showed only 156 files out of 14,377, i.e., 1.09%. A random sample of 16 (≈10%) of those files (i.e., 156) yielded 62% of actual Rx operators, that is the great majority.

Forks. By including repositories with their forks or sibling forks, there was a chance of inflating the counting for a specific operator. In this manner, we checked if there was a fork among the investigated repositories and if either the parent was also present or there was a sibling in the set. Fortunately, by running a script, there was no such cases; instead, there was a

[5] The list of official Rx GH users can be obtained by inspecting the URLs in <https://reactivex.io/languages.html>.
[6] We were not able to retrieve the following project due to file corruption: <https://github.com/zwacky/game-music-player>.
[7] <https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>.
[8] <https://github.com/eclipse/eclipse-collections>.
[9] <https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/CollectionUtils.html>.
[10] <https://guava.dev/releases/23.0/api/docs/com/google/common/collect/Collections2.html>.

few instances of repositories not available anymore: 28 (RxJava), 23 (RxJS), and 7 (RxSwift). Since we cannot retrieve information of those missing repositories anymore, they could fit the fork cases. However, even if they were in fact forks, they only account for 1.96%, 2.88%, and 1.75%, respectively; in other words, there was little or no impact in terms of fork influence.

3.1.2 Stack Overflow Mining

The mining activities on SO were divided into two stages: topic modeling (Sections 3.1.2.1) and the definition of topics' relevance (3.1.2.2). Both stages use data collected via the Stack Exchange Data Explorer (SEDE)[11], one of the many tools to acquire SO data already used in existing work (REBOUÇAS et al., 2016; TAHIR et al., 2020). To have as much data as possible, we leveraged both questions and accepted answers, following Ahmed and Bagherzadeh (2018) and Bajaj, Pattabiraman and Mesbah (2014). We pulled questions, and their respective accepted answers, with the following tags: 'rx-java', 'rx-java2', 'rx-java3', 'rxjs', 'rxjs5', 'rxjs6', 'rxjs7', and 'rx-swift'; those are the tags directly related to the Rx libraries considered in this work: RxJava, RxJS, and RxSwift. Each query was executed separately for every tag and, afterwards, the results were combined, with the duplicated entries removed. A total of 47,404 records were fetched, with entries ranging from February 2011 to December 2021.

3.1.2.1 Topic Modeling

Topic modeling is a Natural Language Processing method based on the words' statistics that can be used to summarize documents through a set of topics (BAJAJ; PATTABIRAMAN; MESBAH, 2014). Latent Dirichlet Allocation (LDA), on the other hand, is one of the most commonly-used topic-modeling techniques (CAMPBELL; HINDLE; STROULIA, 2015; TREUDE; WAGNER, 2019). It has been employed in many previous studies targeting SO data (REBOUÇAS et al., 2016; AHMED; BAGHERZADEH, 2018; ABDELLATIF et al., 2020). It allows a more manageable overview of a large corpora of documents as it may be impractical to do a manual inspection (REBOUÇAS et al., 2016).

Posts Extraction. Data acquired through the SEDE comes in CSV format. It contains information about posts, such as titles, bodies, and types. For the purpose of this work, we initially decided to work with the posts' bodies. Working with the bodies gives us the opportunity to

[11] <https://data.stackexchange.com/>

extract code snippets and help with other phases of the research (Section 3.1.2.2). Nonetheless, we observed that some posts, either questions or accepted answers, after going through data preprocessing, were becoming too small to offer any intuition about the context of the post for future analysis; examples include the question #65813454 and the accepted answer #41908578. Therefore, based on Han et al. (2020) and Han et al. (2017), we mixed both title and body of the posts. For accepted answers, the titles of their respective questions were used to compose the effective post's text.

Data Preprocessing. A number of filters are commonly applied to documents before feeding them into the LDA for processing. We removed the following elements, based on (AHMED; BAGHERZADEH, 2018; REBOUÇAS et al., 2016; TREUDE; WAGNER, 2019; HAN et al., 2020): (1) code snippets, i.e. content inside <code>, <pre>, and <blockquotes>, (2) HTML tags, (3) Line breaks and sequence of whitespaces, (4) URLs, (5) one-letter words, (6) stop words, like *an* and *I*, (7) numbers, (8) punctuation marks, (9) non-alphabetical characters. Furthermore, we applied stemming (Snowball stemmer)(ALLAMANIS; SUTTON, 2013; CAMPBELL; HINDLE; STROULIA, 2015) to the remaining words (i.e., mapping the words to their root form) and removed common and uncommon words (i.e., words occurring in less than five posts or in more than 90% of those, respectively) (CAMPBELL; HINDLE; STROULIA, 2015). After a few executions, we also decided to drop common SO words such as 'answer', 'question', 'help', and 'solut'.

LDA Execution. For the LDA execution, we leveraged the NLP[12] library, which offers many machine learning and natural language processing algorithms. One of the key points in LDA usage is to find the appropriate number of topics (ABDELLATIF et al., 2020). As noted by Han et al. (2020), the exact quantity can impact the granularity of the result, yielding a too specific outcome, in case of a high number, or a too generic one, otherwise (ABDELLATIF et al., 2020). A common approach when deciding the number of topics is to vary this number (REBOUÇAS et al., 2016; ABDELLATIF et al., 2020) and determine the most coherent result either by manual inspection (REBOUÇAS et al., 2016; AHMED; BAGHERZADEH, 2018) or by some metric (ABDELLATIF et al., 2020) like coherence. Following Rebouças et al. (2016), we varied the number of topics between 10 and 35 and manually inspected the results. To help in the process, we also relied on the perplexity metric (TREUDE; WAGNER, 2019), which can be used to get some intuition about a possible good model fit. This metric tends become smaller as the number of topics grows (AGRAWAL; FU; MENZIES, 2018), i.e., smaller values usually indi-

[12] <https://github.com/james-bowman/nlp>

cate better models. However, the correspondence of good model fit and human assessment do not always correlate (TREUDE; WAGNER, 2019). Thus, we looked for outcomes yielding small improvements in perplexity with a different number of topics, but ultimately used the manual inspection to assess the results. Finally, we experimented with two combinations of values for hyperparameters: $\alpha = 50/k$, $\beta = 0.01$ (AHMED; BAGHERZADEH, 2018; ROSEN; SHIHAB, 2016)—k denotes the number of topics—and $\alpha = \beta = 0.01$ (CAMPBELL; HINDLE; STROULIA, 2015; RODRÍGUEZ; WANG; KUANG, 2018). The first combination is a common one used in other studies but, conventional standards for parameters are not fit for GH and SO texts (TREUDE; WAGNER, 2019). After a series of try-outs, the computed result with 23 topics, 1,000 iterations and $\alpha = \beta = 0.01$ showed the most coherent outcome.

Topic Inference. Topics produced by LDA correspond to a set of words and their proportion, with the name or label left to be inferred by who is applying the algorithm. To aid in this task, we resorted to the open card sorting technique (AHMED; BAGHERZADEH, 2018; ABDELLATIF et al., 2020; ROSEN; SHIHAB, 2016), which consists of analyzing the top words of a given topic and inspecting posts chosen randomly that have the topic as their dominant one (AHMED; BAGHERZADEH, 2018). The researchers analyzed the top 20 words and 15 random posts; each examiner labeled the topics individually and jointly discussed results supported by a mediator (a third researcher) to reach agreement when needed.

3.1.2.2 Defining Topic Relevance

In this study, we consider a topic as relevant based on its popularity and difficulty. To measure it, different metrics can be used. Popularity, for instance, can be calculated by taking the average value of three SO measures (ABDELLATIF et al., 2020; AHMED; BAGHERZADEH, 2018): (1) View, (2) Favorites, and (3) Score. Thus, a topic with high average view, favorites, and score is considered popular. Difficulty, conversely, has two metrics commonly employed (ABDELLATIF et al., 2020; AHMED; BAGHERZADEH, 2018; ROSEN; SHIHAB, 2016): (*1*) the percentage of questions with no accepted answer and (*2*) the median time it takes for an answer to be considered accepted; the time is calculated based on the creation dates found in the accepted answer and its respective question post (ROSEN; SHIHAB, 2016). As noted by Ahmed and Bagherzadeh (2018), topics showing a high rate of questions without answers and taking more time to get accepted answers are intuitively harder. Hence, the aforementioned metrics are also exploited in the study.

Additionally, aiming to find operators' occurrences among the SO posts, we conducted a search throughout the questions and accepted answers used in Section 3.1.2.1. This search is carried out to delineate a correlation between the most relevant topics and the usage frequency of Rx operators (considered in Section 3.1.1), the objective of RQ2.2. To access the code snippets, we extracted the contents inside the tags <code> of SO posts. As opposed to Section 3.1.2.1, we did not regard the <blockquote> and <pre> tags since <blockquote> is often used to include stack traces instead (RODRÍGUEZ; WANG; KUANG, 2018) and <pre> is usually applied to add formatting to <code>. Also, like Section 3.1.1, we removed comments and strings. The search in turn relied on a regex that either looked for the operator name or the operator invocation. The examination of only the operator name is necessary since, many times, text inside the <code> tag is used only to highlight a construct without actually showing its usage (e.g., post #40811273).

3.2 RESULTS

This section is organized according to secondary research questions (RQ1.1 - RQ2.2).

3.2.1 RQ1.1: How much are the Rx operators being used in open source projects?

To elaborate the results, we relied on the usage frequency of the operators from RxJava, RxJS, and RxSwift as detailed in Section 3.1.1. Figure 3 presents the percentage of operators' utilization according to their library. We can observe that the majority of operators are actually in current use. The greatest exception among the three libraries was RxJava presenting 94.1% (223 operators) of usage but 5.9% (14) of its 237 operators not actually being used in the GitHub projects of our sample. RxJS and RxSwift, on the other hand, presented 100% and 98.5% of employment of their 113 and 66 operators, respectively. Nonetheless, in general, the libraries showed a good measure of utilization. This fact is even more clear when combining the operators (merging those alike) and their frequencies from all three libraries and computing the percentage of utilization: 95.2% of general utilization. Thus, although Rx provides a great number of operators scattered through its different libraries, those operations showed a very considerable rate of utilization. From the 14 non-used operators of RxJava, we could notice that half were operators related to Java's concurrency API `CompletionStage`[13] whose support was

[13] <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html>

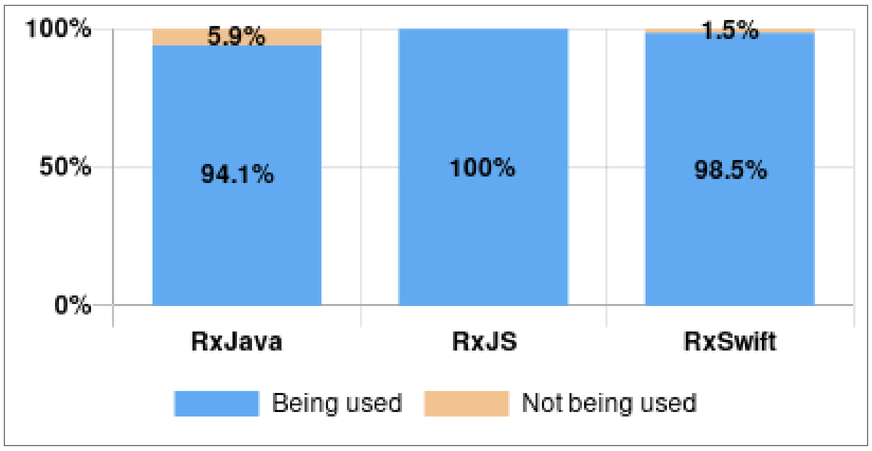
added in RxJava 3 (released on February, 2020). This may indicate that either the developers did not have time to use the feature yet or did not find a useful situation to apply it.

Figure 4, 5, and 6 depict the most and least frequently used operators in RxJava, RxJS, and RxSwift, respectively. For simplicity, we included only the 15 most and least utilized ones and removed those not in use. The complete list according to their frequency is available online. By inspecting the charts with the most used operators, one can perceive the occurrence of `subscribe` as one of the most utilized operators in the three libraries. This correlates to its importance in controlling the stream lifecycle. The stream becomes indeed active only when subscribed, given its lazy evaluation principle. Curiously, `map` is the most intensively used operator in RxSwift. Still, `map` is certainly an important transforming operator, and that can be evidenced by its presence in the list of most frequently used operators of the three libraries. Along with `map`, one can perceive the presence of other common functional operators like `any`, `filter`, and `reduce`. Concerning the creation operators, we can note that `just` and its equivalent `of` are one of the most present. It allows the creation of a one-element stream and can be useful for the construction of business logic that expects the emission of a single element in their composition. Apart from RxJava, there were few operators related to error handling/testing. In summary, the most frequently used operator comprises all categories[14] of operators such as creation (`of`, `from`), transforming (`map`, `flatMap`), filtering (`take`, `takeUntil`), combining (`merge`, `zip`), among others. A pattern that we could noticed is that the majority of the most used operations are largely composed of Rx core operators (e.g., `concat`, `filter`, `map`, etc.). The least frequently used, on the contrary, are mostly formed by library-specific variants (e.g., `concatMapSingle`, `concatMapMaybe`, `mapWithIndex`, etc.) which comprises all types of operators. An exception in the set of least utilized ones is the presence of core operators `buffer` and, specially, `window` (and some of its variants such as `windowTime`, `windowToggle`, and `windowWhen`). These correspond to an important class of operators since some computations require the accumulation of stream elements before execution. Table 3 presents a different perspective for the most used operators in the three libraries. It shows the average usage per analyzed repository. Those statistics indicate that even thought the operators have shown relative high frequencies, they are probably being used more in some projects than others (i.e., in an irregular way). A final worth observation is the scale of the least used RxJava operators. Its 15 least used operators did not exceed the limit of 10 usages, which is likely linked to its extensive API. However, when collecting operators

[14] Categories based on <https://reactivex.io/documentation/operators.html>.

with $\leq$50 uses throughout the analyzed projects, we could notice that RxJS presented the greatest percentage, corresponding to ≈32% of its operators against ≈22% from RxJava and only ≈7.5% from RxSwift.

Figure 3 – Percentage of operators' utilization per Rx library.



Source: Elaborated by the author (2022)

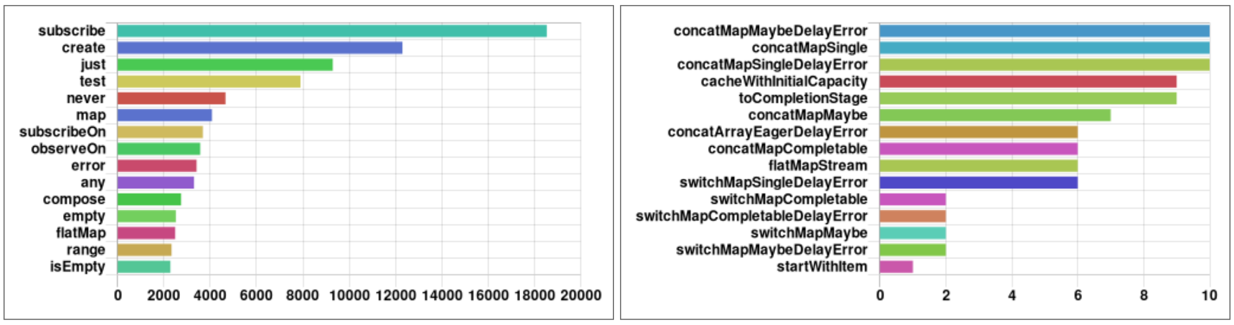
The great majority of Rx operators are being utilized in the libraries RxJava (94.1%), RxJS (100%), and RxSwift (98.5%). This percentage comes to 95.2% when merging the operators and their usage frequency from all three libraries. ***Finding 1***

Only RxJava presented more than one operator not being utilized (≈6%). ***Finding 2***

`subscribe` *appears as the most frequently used operation in two Rx libraries: RxJava and RxJS. Yet, it is the second most used in RxSwift, only behind the* `map` *operator.* ***Finding 3***

Functional operators, like `any`, `filter`, *and* `map`, *figure among the most used operators in all libraries.* ***Finding 4***

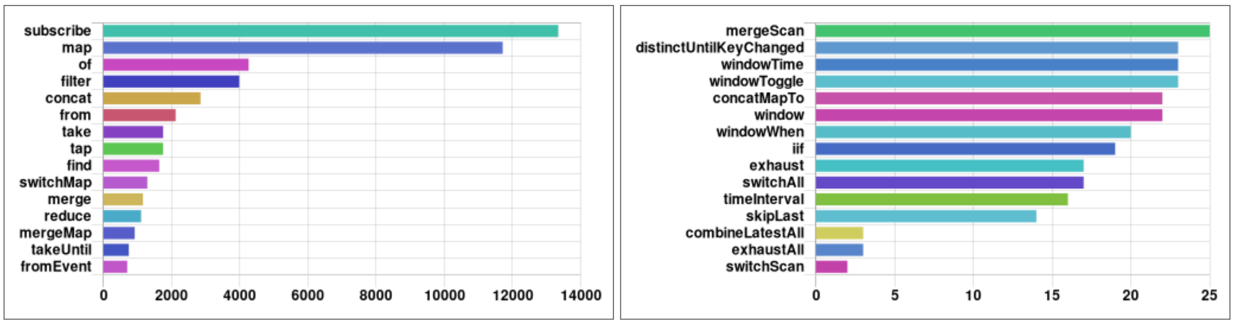
Figure 4 – The most and least frequently used operators of RxJava.



(a) 15 most used (b) 15 least used

Source: Elaborated by the author (2022)

Figure 5 – The most and least frequently used operators of RxJS.

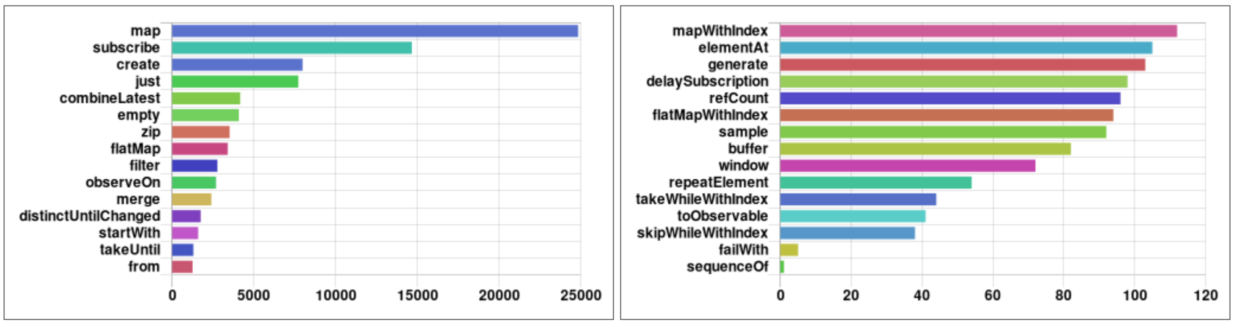


(a) 15 most used

(b) 15 least used

Source: Elaborated by the author (2022)

Figure 6 – The most and least frequently used operators of RxSwift.



(a) 15 most used

(b) 15 least used

Source: Elaborated by the author (2022)

Except for RxJava, there were few operators related to error handling/testing among the most used ones. ***Finding 5***

The most used operators mainly comprise core operators, while the least used ones are essentially composed of core variants. An exception to that observation was the presence of core operators like `buffer` *and, specially,* `window` *(and its variants like* `windowTime`*,* `windowToggle`*, and* `windowWhen`*).* ***Finding 6***

All 15 least used operators of RxJava showed less than or equal to 10 usages. However, RxJS presented the greatest percentage of operators with ≤50 usages (≈32%). ***Finding 7***

3.2.2 RQ2.1: What problems are RP developers facing?

As denoted in Section 3.1.2, we used the LDA topic modeling technique to uncover the problems, in a topic format, that reactive programming developers are facing. Table 4 presents

Table 3 – Average usage of the 15 most frequently used operators per analyzed repository.

RxJava	RxJS	RxSwift
`subscribe` (13.0)	`subscribe` (16.7)	`map` (62.0)
`create` (8.6)	`map` (14.7)	`subscribe` (36.6)
`just` (6.5)	`of` (5.3)	`create` (19.9)
`test` (5.5)	`filter` (5.0)	`just` (19.2)
`never` (3.3)	`concat` (3.6)	`combineLatest` (10.4)
`map` (2.8)	`from` (2.7)	`empty` (10.2)
`subscribeOn` (2.6)	`take` (2.2)	`zip` (8.8)
`observeOn` (2.5)	`tap` (2.2)	`flatMap` (8.5)
`error` (2.4)	`find` (2.1)	`filter` (6.9)
`any` (2.3)	`switchMap` (1.6)	`observeOn` (6.7)
`compose` (1.9)	`merge` (1.5)	`merge` (6.0)
`empty` (1.8)	`reduce` (1.4)	`distinctUntilChanged` (4.3)
`flatMap` (1.7)	`mergeMap` (1.2)	`startWith` (4.0)
`range` (1.6)	`takeUntil` (0.9)	`takeUntil` (3.2)
`isEmpty` (1.6)	`fromEvent` (0.9)	`from` (3.1)

*N*ote: Average usage shown in parentheses.

Source: Elaborated by the author (2022)

the 23 generated topics with the inferred labels and sorted by their number of posts. From the result, we can observe that the generated topics truly correspond to important matters that reactive programmers face daily. For example, the first two topics with the most posts are related to stream abstraction. We can also note presence of *Concurrency*, an important intrinsic concept in reactive programming given that its pipeline model allows the exploration of an asynchronous and non-blocking execution and the efficient use of threads (BONÉR; KLANG, 2017), either directly (as by schedulers in RxJava) or indirectly. One can also visualize elements revolving UI, a field from where most of the reactive programming research came from by the developing of the Fran language (BAINOMUGISHA et al., 2013). We grouped those topics under nine categories and briefly discuss each one.

Stream Abstraction (6 topics, 36.4% posts). This category encompasses six topics related to the stream abstraction (i.e., the main resource to structure the reactive logic): *Stream Manipulation*, *Stream Creation and Composition*, *Stream Lifecycle*, *Timing*, *Multicasting*, and *Control Flow*. Together, their posts account for 36.4% of the total of analyzed posts with the first two topics showing the biggest share (Table 4). Under the topic *Stream Manipulation*, for instance, there were many questions related to `Observable` – the main stream type in Rx –

Table 4 – Topics ordered by their number of posts.

Topic	# Posts	% Posts	Category
Stream Manipulation	6384	13.5	Stream Abstraction
Stream Creation and Composition	4872	10.3	Stream Abstraction
Array Manipulation	3146	6.6	Programming
Web Development	3049	6.4	Application Development
Data Access	2680	5.7	Persistence
Android Development	2551	5.4	Application Development
Concurrency	2251	4.7	Concurrency
HTTP Handling	2243	4.7	Networking
Stream Lifecycle	2141	4.5	Stream Abstraction
Error Handling	1825	3.8	Reliability
Timing	1813	3.8	Stream Abstraction
UI for Web-based Systems	1791	3.8	User Interface
Dependency Management	1710	3.6	Application Development
Typing and Correctness	1578	3.3	Programming
iOS Development	1536	3.2	Application Development
Multicasting	1305	2.8	Stream Abstraction
REST API Calls	1231	2.6	Networking
Testing and Debugging	1219	2.6	Reliability
State Management and JavaScript	1183	2.5	Application Development
Input Validation	1087	2.3	User Interface
Control Flow	735	1.6	Stream Abstraction
General Programming	588	1.2	Programming
Introductory Questions	486	1.0	Basics

*N*ote: 47,404 posts in total.

Source: Elaborated by the author (2022)

handling like converting it, accessing and passing values to it, returning it, among others. For example, #48601357 asks *"...convert Observable<string[]> into Observable<string>"* while #56610867 deals with *"A more succinct way to conditionally chain multiple observables."* Having this topic as the one with the higher number of posts is probably due to the shift that developers should face while structuring their code as compositions of streams. The second topic with more posts, *Stream Creation and Composition*, is very related to the first one and surprisingly showed many operators in their LDA top words, such as 'combin', 'merg', and 'combinelatest'. #38067532 is a typical example of questions found in this topic where the author asks about *"Combining two different observables."* The remaining topics cover other matter intrinsic to streams, like the use of time handling/control flow operations, subscribing and unsubscribing (i.e., lifecycle), the different types of stream "temperature" (hot or cold[15]), or the proper usage of `Subject`[16].

Application Development (5 topics, 21.2% posts). This category combines problems related to application development in general, totaling five topics. The addition of those topics' posts yields a share of 21.2% of all study's posts. From the posts in this category, we have 42.2% about Web development (considering *Web Development* and *State Management and JavaScript* together) and 40.8% about mobile development (*Android Development* and *iOS Development*). Most of the web development topics revolves around the Angular framework which is expected given that Angular both use RxJS internally and also makes it available to be utilized as part of its library[17]. Others, conversely, deal with state management, an important matter in nowadays JavaScript frameworks. In this regard, we have many mentions to the Redux-Observable (a Redux middleware) and, specially, NgRx (state management for Angular based on RxJS). In the mobile development, more posts linked to Android than iOS appeared, even though we did not include any SO tags related to RxKotlin or RxAndroid (RxJava bindings for Android). In the iOS posts, we could observe a prevalence of questions related to patterns embracing view models like MVVM (Model-View-ViewModel). The Android ones, on the contrary, included many questions about using RxJava with some framework or library like Retrofit (an HTTP client). The remaining posts within this category, curiously, related to dependency management problems like in #45516375 where the author complains about *"Problems with .maps and import 'rxjs/add/operator/map'"* or #46692177 with the issue *"RXJS Observable missing definitions."*

Programming (3 topics, 11.2% posts). Three topics are classified under this category: *Array Manipulation*, *Typing and Correctness*, and *General Programming*. The three topics represent 11.2% of the investigated posts, with *Array Manipulation* comprising most of the posts (59%), as expected. Dealing with stream manipulation often requires the conversion to or from an array (e.g., the `toArray` operator from RxJS) or even the accumulation of elements in arrays (the case of `buffer` operator) given that streams can be seen as a sequence of events. In #58657165, for example, the user complains *"Can not get array of Observables."* Other questions in the *Array Manipulation* revolved around manipulating collections of Observables

[15] Cold streams are those that produce data based on each subscription. Hot ones, on the other hand, do not depend on subscription to emit values and their values are normally produced from an outside source like WebSockets.

[16] An alternative to Rx `Observable` that is hot in nature (i.e., multicasting)

[17] <https://blog.angular-university.io/functional-reactive-programming-for-angular-2-developers-rxjs-and-observables/>

(streams) or the use of common collection-like functions (e.g., `filter`). The posts under *Typing and Correctness* entered this category since they mostly show problems of type systems like *"Property 'forEach' does not exist on type 'Object'"*(#46352289) or *"Type 'Observable< | T>' is not assignable to type 'Observable<T>'"*(#45273084). The rest of the posts under this category were classified as *General Programming*, given that they discuss primarily about handling of common programming constructs like objects, variables, null values, etc.

Networking (2 topics, 7.33%). The two topics under this category involve networking discussion, specifically targeting the exploration of HTTP handling and Representational State Transfer (REST) API calls. A variation of discussions can be found in the topic HTTP Handling which is the one with more posts. Some of them involve how to better manage concurrent requests and parallelism with the RP libraries (e.g., #56827462, #46047713, and #55558277). Another important aspect regarded asynchronous handling and orchestration with RP. A post (#54518039), for instance, was asking for a way to cache various HTTP responses using RxJS to improve performance and avoid redundant network calls. Others considered ways to cancel ongoing requests (#51620217), wait for emissions before requests (#64460331), and coordinate asynchronous events (#34287143). An interesting observation was the presence of error handling in network posts (e.g., #40348516 and #57084311) besides the ones included in the Reliability category; having those types of posts is actually expected given the unreliability of distributed systems (KLEPPMANN, 2017). Additional posts under this topic include: (1) `Observable` behavior and `Subscription` issues (e.g., #69636272 and #43785009), (2) Correct combination of RP with asynchronous constructs like JS `fetch`/Promise (#34512605), and (3) A lot of posts regarding Angular and Angular Interceptors[18] (e.g., #64807519, #56773770, #52947944). The posts of the topic REST API calls are very close to the ones observed in HTTP Handling, but they mostly concentrate on API Calls. #59903679 tries to discover how to ensure consistent calls and data flow, while #66418261 deals with how to minimize redundant calls. Error handling again was present among the discussions (e.g., #60093320 and #56040946). We noticed the presence of posts specifically related to operators like `zip` (#44840882 and #33793650); many posts in both topics actually relied on the exploration of the `flatMap` variations (e.g., #69618448, #64460331, #51620217, and #42888604). That demonstrates a frequent reliance on combination and dependency between streams in networking solutions.

[18] An Angular middleware that abstracts retrying, caching, logging, and authentication from HTTP requests – <https://angular.dev/guide/http/interceptors>.

Reliability (2 topics, 6.42%). The category *Reliability* concerns topics that make software more trustworthy like *Error Handling* and *Testing and Debugging*. The greatest topic, Error Handling (≈60%), mainly entails error handling aspects such as general error handling posts (e.g., #45416105 and #59946361), propagation and interception of errors (#53773926 and #44825585), error scenarios and debugging (#45750820 and #47972229), and handling error in Angular applications (#41911205 and #49779279). For instance, there were many questions regarding retrying on error like the post asking *"RxJava How to properly Resubscribe onError"* (#33072216); at least two posts (#42623002 and #48964735) presented solutions showing how to implement a backoff strategy with retry which Rx does not seem to include by default but it can be implemented with its API. A great majority of the posts under *Testing and Debugging* specifically target testing; for example, we saw posts dealing with unit testing reactive streams (e.g., #33896113 and #47989747), marble testing (e.g., #52081332 and #52707668), mocking (e.g., #52296336 and #63111840), and testing issues (e.g., #60690184 and #68535258). An interesting detail is the presence of marble testing among the discussed subtopics; marble testing is a feature that allows easily simulating stream data through an ASCII text version of marble diagrams (Section 2.2.1.3). However, that is only natively available in some Rx distributions like RxJS[19]. Additionally, the topic included some posts with respect to debugging and investigation of unexpected behaviors (e.g., #66749295, #60756850, and #42280054).

User Interface (2 topics, 6.07%). *User Interface* is the last category that encompasses at least two topics: *UI for Web-based Systems* (62%) and *Input Validation* (38%). A great part of the posts under *UI for Web-based Systems* are linked to Angular framework, specially concerning Async Pipe operator[20] in Angular templates. In a summarized way, we observed (mostly related to Angular) themes about *Observable* and Async pipe usage (e.g., #48346774 and #68762721), managing change detection (e.g., #57744502 and #53483895), handling route and navigation (e.g., #64822356 and #57115305), and data biding and template interaction (e.g., #47850488 and #41548035). There were, however, some posts beyond Angular like the post asking for "How to connect RxJs to React component" (#47832845) or "Retry a call with Retrofit 2 and RxJava2 after displaying a dialog" (#47672224). The *Input Validation* topic, on the other hand, concentrate on input and form validation (e.g., #62316601, #46258130,

[19] However, there are ports for RxJava and Rx.NET for instance.

[20] Angular Pipes

and #64778172). There were posts related to how to filter, debounce[21], and implement search functionality as well as stream combination and transformation (e.g., #56593091 and #52741334) which can be used during input validation. Two Rx operators were used recurrently in posts (e.g., #50739211, #53724674, and #44934279): `combineLatest` and `debounceTime`. An interesting post was given in #49423355 which involves the difference between `skipWhile` and `filter` operators; both functions can be used to control and filter stream emissions. Other posts under *Input Validation* comprise: managing HTTP Requests (e.g., #53432096 and #51989484), user input handling (e.g., #37273666 and #45651125), and some posts related with testing `Observables` (e.g., #48855982 and #36681078).

Persistence (1 topic, 5.65%). This category includes the *Data Access* topic and a total of 2680 posts. The discussions focus on efficient data fetching, simplifying database reads/writes, avoiding duplication, and ensuring data integrity in reactive applications. Overall, posts could be further categorized into: "data access and database operations" (e.g., #56853894, #47976737, and #48244139), "data transformation and fetching" (e.g., #40535704, #46520090, and #51870841) and "persistence integration" (e.g., #47474605 and #38900234). Among the posts, many of them dealt with manipulation of lists like #48671171 and #51870841.

Concurrency (1 topic, 4.75%). *Concurrency* posts are under this category, comprising 4.75% of all questions and answers analyzed. A great part of the posts deal with managing asynchronous tasks (e.g., #69206970 and #46541520), thread handling including scheduling of tasks (e.g., #29998856 and #36907451), data combination (#46888172, and interaction with native pool of threads (#43472947), and interruption of parallel execution (#63956012) to cite a few. There were actually themes concerned with the controlling of stream emissions including backpressure and flow control (e.g., #61224069, #66543084, #63117044). Additionally, we observed discussions regarding Blocking `Observables` (e.g., #31882325 and #48648156) which is a RxJava stream type that all of its operators are blocking[22]. A post (#52407096) was particularly interested in how to simulate and test blocking operations in RxJava using `TestScheduler`.

Basics (1 topic, 1.03%). The *Basics* category is the smallest category and it only encompasses the topic *Introductory Questions*; this topic mainly covers posts regarding basic concepts involving RP and the libraries. Typical examples in this category consist of posts asking about:

[21] Debouncing is a controlling technique used to improve the handling of user input.

[22] More information is available at <https://reactivex.io/RxJava/javadoc/rx/observables/BlockingObservable.html>.

- What Rx is and its uses (#55596877);
- How to introduce RP into existing projects (#31046584);
- Recipes for common operations like branching, filtering, and caching (e.g., #53860839, #36535716, and #64333692);
- Comparisons between operators (e.g., #69371484, #52861884, and #48889016[23]), similar APIs like the `Flow` interfaces in Java 9 (#46382341), and related tools (#28834829);
- Relation with web/UI architectures and frameworks (e.g., #38013873).

Additionally, users sought answers regarding Rx interoperations with event-driven/async features like Event Emitters[24] (#36999129) and listeners (#44833118). Interestingly, we also noticed the presence of error-handling and logging posts (e.g., #30975462 and #62301086).

Reactive developers ask about a multitude of topics, like Stream Manipulation, Web Development, Concurrency Networking, etc., but with higher interest in problems regarding the stream abstraction. ***Finding 8***

The topics with more posts concern the stream abstraction, having Stream Manipulation (13.5%) and Stream Creation and Composition (10.3%) occupying the first and second places. ***Finding 9***

Developers have asked less about basic matters as can be verified by the topics with fewer posts: General Programming (1.2%) and Introductory Questions (1%). ***Finding 10***

3.2.3 RQ2.2: How do the operators present in the most relevant Stack Overflow questions and the usage frequency of Rx operators in open source projects relate?

We considered a topic as relevant based on its popularity and difficulty. Table 5 shows the popularity of the topics according to the average views, favorites, and scores of their questions. Table 6 exhibits the difficulty of the topics according to their percentage of questions without an accepted answer and median time taken to receive an accepted answer. Tables 5 and 6 are

[23] This discussion touches an important aspect regarding the cancellation of a stream subscription in a more composable, reactive way versus doing it manually in an imperative-like fashion. The former way is in fact the most indicated way by lead developers of Rx.
[24] An implementation of the Observer pattern.

respectively sorted by the average views and the percentage of questions without an accepted answer.

The observation of Table 5 reveals that *Dependency Management* has the highest average number of views, while *Introductory Questions* shows the greatest average favorite and score. Hence, those topics are among the most popular ones. In contrast, *Data Access* exhibits low average view, favorite, and popularity, thus it is among the least popular questions. The topics in Table 6, in turn, show *Dependency Management* (highest rate of questions without accepted answer) and *iOS Development* (greatest median time) among the most difficult topics to answer. Conversely, *Array Manipulation* and *Web Development* are the easiest topics, with the least percentage of questions with no accepted answer and median time in hours, respectively.

Given that *Dependency Management* is classified as both popular and difficult, there are three most relevant topics: *Dependency Management*, *Introductory Questions*, and *iOS Development*. Figure 7 shows the similarities between the operators gathered through the posts of those topics and the ones collected in GH projects (Section 3.1.1) according to the order of their frequencies. This comparison was calculated based on the complement of Hamming Distance (HEMMATI; BRIAND, 2010). We can observe that very few operators share similarity when considered their position based on their frequencies. By looking at the sample of the 15 most used operators in both the most relevant topics and the GH projects and disregarding their order of appearance, we can notice many more matches as shown in Figure 8. Those matches can be observed in Table 7. We can also notice many of the most frequent ones out of the 15 most used in Figure 4, 5, and 6 also appearing in Table 7. Unsurprisingly, `subscribe` shows high frequencies as in the case of Introductory Questions, with 200 occurrences for RxJava posts. Well-known functional operators are also present, with some exhibiting great frequencies like `map` with 811 (RxJS) and 833 (RxSwift) for Dependency Management and iOS Development, respectively. Actually, this high frequency observed for `map` in RxSwift corroborates the findings in Section 3.2.1, giving more credibility, where `map` was also the most utilized operator in RxSwift. Also, following Section 3.2.1 findings, most of the operators are the ones considered as core; exceptions include, for example, the creation operator `fromEvent` and the utility `test` method.

Table 5 – Topics' Popularity.

Topic		**Average**	
	Views	**Favorites**	**Scores**
Dependency Management	**3453.6**	0.7	3.8
Web Development	2707.6	0.6	2.2
Stream Manipulation	2641.4	0.8	3.5
Typing and Correctness	2527.6	0.3	2.3
Stream Lifecycle	2503.1	1.1	4.2
Introductory Questions	2442.5	**2.2**	**6.5**
Multicasting	2369.9	1.1	3.9
Error Handling	2250.2	0.6	2.6
Control Flow	1923.0	0.4	2.2
Array Manipulation	1803.7	0.3	1.4
Android Development	1793.0	0.8	2.7
General Programming	1722.4	0.4	2.2
UI for Web-based Systems	1659.8	0.4	1.7
iOS Development	1634.3	0.5	1.9
Input Validation	1489.7	0.3	1.4
HTTP Handling	1470.8	0.5	1.7
Timing	1454.7	0.4	2.1
Stream Creation and Composition	1421.9	0.5	2.3
REST API Calls	1341.5	0.4	1.3
Concurrency	1308.9	0.7	2.6
Testing and Debugging	1237.9	0.3	1.8
State Management and JavaScript	1137.8	0.4	1.6
Data Access	1102.5	0.3	1.3

Source: Elaborated by the author (2022)

Dependency Management and Introductory Questions are among the most popular topics with the former having the greatest number of views and the latter presenting the highest favorites and score, on average. Problems regarding Data Access figure among the least popular. ***Finding 11***

Posts concerning Dependency Management are amongst the most difficult questions. iOS Development also figure as one of the most challenging topics, whereas Array Manipulation and Web Development appear among the easiest ones. ***Finding 12***

Table 6 – Topics' Difficulty

Topic	**w/o Acc. Answer(%)**	**Median Time(hr)**
Dependency Management	**50.3**	1.2
Testing and Debugging	49.0	3.3
Multicasting	48.1	2.7
iOS Development	47.8	**5.5**
Android Development	47.8	2.3
Concurrency	47.1	3.5
Data Access	46.5	1.9
UI for Web-based Systems	46.5	0.9
HTTP Handling	46.2	1.2
Error Handling	46.0	1.6
REST API Calls	44.2	1.0
Input Validation	43.9	1.2
Introductory Questions	43.3	4.8
Stream Lifecycle	43.3	1.5
Web Development	43.2	0.5
Control Flow	42.2	1.4
State Management and JavaScript	41.9	2.2
Timing	41.0	2.2
Stream Creation and Composition	38.4	1.8
Stream Manipulation	37.7	0.8
Typing and Correctness	37.3	0.8
General Programming	37.0	1.2
Array Manipulation	35.7	0.8

Source: Elaborated by the author (2022)

Although the usage frequency of operators of the most relevant SO topics and the GH projects do not share much similarity when comparing the order of appearance according to their frequency, the most frequently used operators of both sources showed a high percentage of similarity when considering only their matches regardless of their frequency position. ***Finding 13***

The majority of the most frequently used operators in SO posts which share similarities with the most frequently used operators in GH repositories is also mostly composed of core operators. ***Finding 14***

The least frequently used operators in SO showed small percentage of similarity when compared to the least utilized ones in GH projects. ***Finding 15***

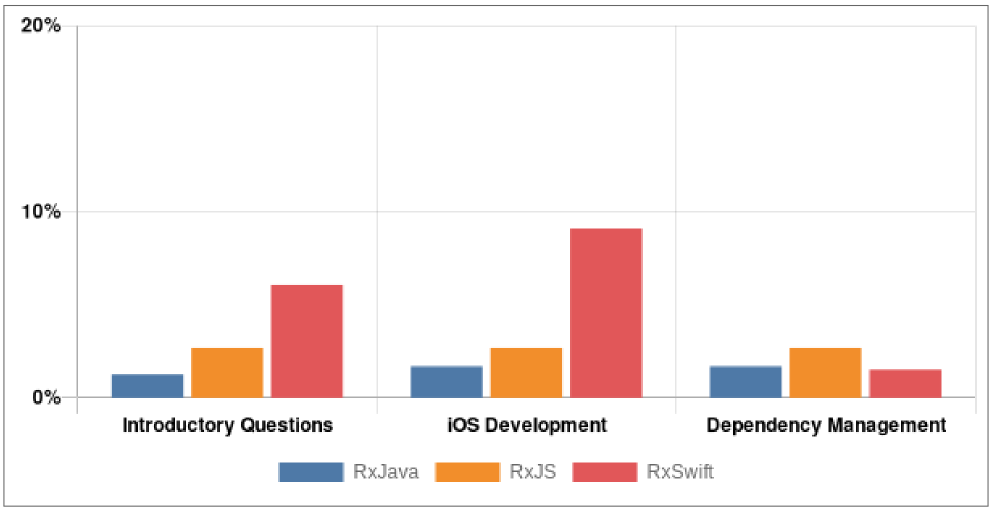
Table 7 – Operators most frequently used in the most relevant SO topics matching the most frequently used operators in GH projects.

Rx Library	**Operators**		
	Dependency Management	**Introductory Questions**	**iOS Development**
RxJava	`test` (66)	`subscribe` (200)	`subscribe` (110)
	`subscribe` (65)	`map` (95)	`error` (35)
	`create` (37)	`create` (72)	`create` (32)
	`error` (27)	`flatMap` (63)	`observeOn` (27)
	`map` (23)	`just` (39)	`subscribeOn` (26)
	`flatMap` (18)	`error` (35)	`map` (17)
	`any` (16)	`subscribeOn` (27)	`flatMap` (13)
	`observeOn` (13)	`test` (26)	`test` (10)
		`observeOn` (21)	
RxJS	`from` (2311)	`subscribe` (159)	`subscribe` (71)
	`map` (811)	`map` (154)	`map` (56)
	`subscribe` (649)	`from` (124)	`from` (49)
	`of` (644)	`of` (82)	`of` (36)
	`merge` (263)	`switchMap` (45)	`filter` (13)
	`mergeMap` (189)	`filter` (44)	`tap` (9)
	`find` (185)	`merge` (33)	
	`filter` (159)	`fromEvent` (31)	
	`switchMap` (116)	`take` (29)	
	`take` (107)	`mergeMap` (23)	
	`tap` (91)	`concat` (15)	
	`fromEvent` (86)		
RxSwift	`from` (15)	`map` (24)	`map` (833)
	`flatMap` (10)	`subscribe` (21)	`subscribe` (733)
	`map` (10)	`create` (19)	`create` (295)
	`subscribe` (6)	`filter` (16)	`filter` (291)
	`combineLatest` (3)	`flatMap` (12)	`flatMap` (290)
	`observeOn` (2)	`just` (4)	`just` (167)
	`create` (1)	`empty` (3)	`from` (139)
	`empty` (1)	`from` (3)	`combineLatest` (114)
	`filter` (1)	`startWith` (3)	`observeOn` (95)
			`empty` (77)
			`distinctUntilChanged` (71)

*N*ote: Operator's frequency in the SO posts shown in parentheses.

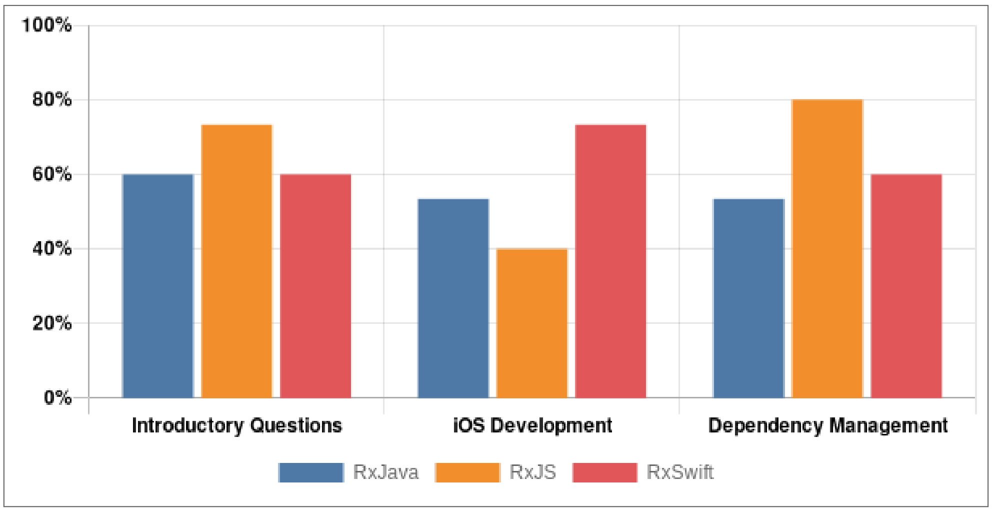
Source: Elaborated by the author (2022)

Figure 7 – Similarity between the operators from the most relevant SO topics and the ones from open source projects based on their frequency position.



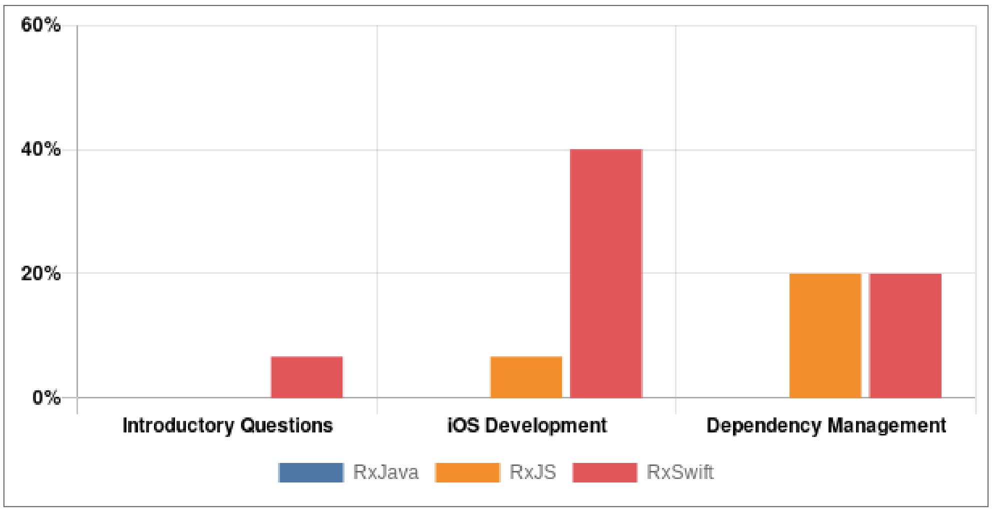
Source: Elaborated by the author (2022)

Figure 8 – Similarity between the 15 most frequently used operators in the most relevant SO topics and the 15 ones from open source projects regardless of their frequency position.



Source: Elaborated by the author (2022)

Figure 9 – Similarity between the 15 least frequently used operators in the most relevant SO topics and the 15 ones from open source projects regardless of their frequency position.



Source: Elaborated by the author (2022)

3.3 IMPLICATIONS

Developers. The findings delineated in Section 3.2.1 and 3.2.3 can serve as a start point for those novice developers that are trying RP. As noted, the great majority of the most used operators both in open source projects and SO forum are mostly comprised of common core operators. So, concentrating their efforts in some of those operators could facilitate their learning path, with support from both platforms.

Maintainers. API call frequencies offer the opportunity for API designers to comprehend the effects of API deprecation and direct efforts (ZHANG et al., 2020). Although, the Rx operators demonstrated a great usage (Section 3.2.1), some actually showed low frequency. Among the RxJava operators, for instance, there were 14 (5.9%) with no usage and, of the 15 with lowest usage, all demonstrated ≤10 calls. Besides, 32% and 22% correspond to the percentage of RxJS and RxJava operations with ≤50 calls, respectively. This contrasts to others operators displaying much more than 1,000 calls amongst the most used ones (Figure 4, 5, and 6). This give rise to a possible consideration of API reduction and the suggestion provided by some researchers of shifting focus from specialization to core concepts (SALVANESCHI et al., 2017; MOGK; SALVANESCHI; MEZINI, 2018).

As exposed in Section 3.2.3, *Dependency Management* topic was both classified as popular and difficult. This prompts a possible future evaluation to investigate why a topic that represents only 3.6% of the posts is receiving both classifications. Among the posts we could perceive, as noted in Section 3.2.2, many problems related to the building process, dependency handling, imports, etc. Having *Introductory Questions* as one of the most popular topics is possibly a sign that newcomers are showing interested in RP but may be encountering problems in the process of understanding it. Section 3.2.2 detailed that users are looking for a better comprehension of the technology, how to integrate it into existing codebases, and recipes for common operators. However, as we dive into the posts, we start to witness concerning accounts. In #36535716, for instance, the user states "I read a ton of literature about the Rx and, on the one hand, everything is clear, but on the other hand nothing is clear." A similar statement is provided by #55596747: "I have read some articles on the internet about RxJava in Order to have a better understanding of what it is but i'm still finding it pretty hard to understand." This perception of understanding problem is also extended to users not comprehending the situation where RP would be best used (#31046584): "From most of the tutorials I could read, I do not really see where rxjs would shine most." Those facts are showing that RP tools may be producing a general perception of tools that are hard to learn and use. In fact, Salvaneschi et al. (2017) detailed three points about RP, collected through a qualitative analysis, that may be very linked to this phenomenon: learning curve, higher level of abstraction (reliance on the runtime), and connection to functional programming. Giving that RP slightly changes the way that most programmers are used to design their programs, it is paramount to provide good documentation and resources in order to enhance the usability of such tools and APIs.

Researchers. API size is a valid concern, specially when usable APIs should be easily memorizable (HENNING, 2009). From a cognitive perspective, memorizing things require extra practice and only 25% of information remains in our minds after two days (HERMANS, 2021), so a long API may in fact turn the learning process hard. However, deciding between general and specialized APIs is an active design process (OUSTERHOUT, 2018), and many APIs from diverse paradigms suffer from huge sizes like the Java and .NET platform (MYERS; STYLOS, 2016).

The findings discovered in Section 3.2.1 show that even though Rx owns an API that is largely extended by its libraries, i.e., by the addition of many variants, we can conclude that developers are actually using a large portion (>90%) of those operators, and from those, the majority of the most used ones is composed by core operators (e.g., `just`, map, and `filter`). However, a great part of those usage present low frequencies when compared the most used operators. RxJava, for instance, exhibited low frequencies ($\leq$10) for its least used operations and no frequency at all for 14 operators (5.9%). Also, both RxJS and RxJava have a lot of operators with $\leq$ 50 usages (32% and 22%); which represents a great difference in usage when compared to the most used ones. Additionally, the core operators actually present a considerable API surface ($\approx$70-80 operators (MOGK; SALVANESCHI; MEZINI, 2018; SALVANESCHI, 2016)). Hence, by visiting the question suggested by Salvaneschi (2016) "Do data flow languages provide a 'simple enough' solution for the common case without excessive proliferation of overspecialised operators?", from the perspective of RP APIs, it does seem that the interfaces are suffering from overspecialization and even the simple enough solution, without a quality supporting material (e.g., a good, complete documentation with examples and extensive, usage guidance), may be making it difficult the life of RP users. It is our belief that RP designers should utilize the outcomes provided in this research and revise the offered API surface. In that process, they should evaluate which interface operators could be either removed or if a set of operators could be generalized; in practice, they should assess how much complexity is in fact avoided by keeping or removing some of their functionalities (OUSTERHOUT, 2018). Nonetheless, the removal may not be necessary as long as the designers identify a few set of core features (possibly less than the list of core operators) and make the users aware of only those; in this way, the effective complexity becomes the complexity of those frequently used operators (OUSTERHOUT, 2018). For instance, as a starting point, the three RP libraries explored could use the set of operators listed in Table 8 which were taken from Table 7.

Table 8 – Suggestive initial set of operators to start with according to three APIs explored.

RxJava	RxJS	RxSwift
`subscribe`	`subscribe`	`subscribe`
`just`	`of`	`just`
`test`	`from`	`from`
`create`	`merge`	`combineLatest`
`error`	`find`	`empty`
`map`	`map`	`map`
`flatMap`	`mergeMap`	`flatMap`
`any`	`filter`	`filter`
`observeOn`	`switchMap`	`observeOn`
`subscribeOn`	`take`	`create`
	`tap`	`startWith`
	`fromEvent`	`distinctUntilChanged`
	`concat`	

Source: Elaborated by the author (2024)

In the same vein, the topic *Introductory Questions* figured as a popular one that could, among various reasons (e.g., documentation (PICCIONI; FURIA; MEYER, 2013), learning curve and relation with FP (SALVANESCHI et al., 2017), or diverse API design decisions), also be related to this API surface. For example, in Section 3.2.2, there are examples of posts asking about methods for expressing common operations and comparisons among operators. Future studies could shed more light into *Introductory Questions* by, for instance, understanding its temporal trends and technical challenges (BAJAJ; PATTABIRAMAN; MESBAH, 2014; KOCHHAR, 2016).

An important observation taken from Table 6 is the presence of the *Testing and Debugging* with the second highest percentage of question without an accepted answer. Moreover, from the mining process, there seems to be few operators in this area. This prompt us to question if this topic is showing relevance due to a lack of dedicated facilities for testing and debugging or difficulty for using the API, and, thus, preparing testable code. In theory, testing should not be such an issue given that there is a separation of concerns of the main entities in the stream model (i.e., producers and consumers of data and, in-between, a pipeline of pure functions) which could facilitate the process. Mogk, Salvaneschi and Mezini (2018), for instance, report no specific obstacles with testing in REScala. Besides, Rx offers schedulers and textual marble diagrams for testing purposes to give more control over time and represent more easily stream events, respectively. However, marble diagrams to simulate stream emissions is a feature that is not implemented in every Rx library. RxJS [25] and RxSwift[26] have native support, while RxJava's users need to use a port for that[27]. Furthermore, for an effective testable code in the stream model, it may require a certain discipline from developers (e.g., decouple stream parts properly, keep pipeline functions free of side effects, etc.). It is our belief that a deeper understanding of how RP code is tested, including a distributed setting (SALVANESCHI; DRECHSLER; MEZINI, 2013), and how they have actually been tested in projects is an important research endeavor; that was in fact the effort of Vianna et al. (2023) in the related area of data stream processing applications.

For debugging reactive programming, conversely, the unsuitability of usual tools has already been recognized (SALVANESCHI; MEZINI, 2016; MOGK et al., 2018; BANKEN; MEIJER; GOUSIOS, 2018). Salvaneschi and Mezini (2016) proposed a technique and tool (Reactive Debugging and Reactive Inspector, respectively) to help visualizing the dataflow of the application, with extensions to provide live programming (MOGK et al., 2018). Cycle.js[28], a JavaScript functional and reactive framework, also provides a similar tool for Internet browsers that can be used to view the dataflow graph and, as a result, help the debugging and metal model process. The lack of integration with usual debugging tools of common IDEs and browsers was actually a downside for adoption of some RxJS debugging tools as pointed by Alabor and Stolze (2020). Consequently, we believe that the area of *Testing and Debugging* constitutes a rich and prominent direction for future exploration and investigation.

[25] Testing RxJS Code with Marble Diagrams.
[26] RxSwift unit tests.
[27] <https://github.com/alexvictoor/MarbleTest4J>.
[28] <https://cycle.js.org/>

3.4 THREATS TO VALIDITY

Internal Validity. Internal validity may be described as aspects that could influence our outcomes (ABDELLATIF et al., 2020) or possible mistakes in execution and experiments (TREUDE; WAGNER, 2019). To search for Rx operators, we looked through every file identifying by specific file extensions (according to the language of the analyzed Rx distribution) and relied on the use of regular expression (regex). By using a regex, this could indicate a threat concerning some Rx operators that may have the same name as well-known functional operators like `filter` and `map`. To reduce this threat, we checked if every inspected file had any mention to the investigated libraries (e.g., rxjava or rxjs) which would correspond to some import of library's package in that specific document. Besides, we examined RxJava files, the library with most mined projects, to verify how much false positives was possibly being introduced. In that case, the number of files that could account for false positives was very low (1.09%) in which a 10% sample showed majoritively Rx operators (62%). The verification also extended to the number of forks considered in the operators' count, which showed no forks identified and only a few projects that could not be verified due to unavailability. However, it is our belief that future replications should strive to use alternative methods that diminish false positives like the semantic tool explored in Xu et al. (2020). Finally, we also removed both comments and strings, so code snippets inside those constructs would not be counted.

According to Abdellatif et al. (2020), the choice of the optimal amount of topics for the LDA algorithm could constitute a threat since it is recognized as a difficult task and directly impacts the quality of the generated LDA topics. To mitigate that possible threat, we followed Abdellatif et al. (2020) and Rebouças et al. (2016) by experimenting with a range of topics; besides, we resorted to the Perplexity metric to aid the manual definition of the optimal number of topics. Still according to Abdellatif et al. (2020), the inference of the resulting LDA topics could also represent a threat given its subjectiveness. We countered this threat by having more than one author evaluating the topics using the open card sort technique and trying to reach some agreement afterwards, similar to the studies (ABDELLATIF et al., 2020; AHMED; BAGHERZADEH, 2018).

External Validity. This validity refers to the generalization of our discoveries (CAMPBELL; HINDLE; STROULIA, 2015; ABDELLATIF et al., 2020; TREUDE; WAGNER, 2019). The present work focused on two prominent platforms, GH and SO. Their widely usage in the development setting gives us a certain confidence about our findings, though other sources like alternative forums or hosting code platforms could improve the final result. Thus, we believe that further incorporation of other sources can complement our findings as well as the inclusion of surveys and qualitative studies. Nonetheless, we tried to include as much data as possible, working with all data in GH and SO regarding the chosen Rx libraries.

Construct Validity. Construct validity is about the fitness of the metrics used in the evaluation (CAMPBELL; HINDLE; STROULIA, 2015; TREUDE; WAGNER, 2019). Treude and Wagner (2019) report that the Perplexity metric and human assessment do not oftentimes correlate. Thus, we merely used it as an aid to identify the number of topics, and we mostly relied on the manual inspection carried out by the first two authors, similar to Han et al. (2020). Future work may incorporate better metrics (e.g., coherence).

The metrics used to determine the popularity or difficulty of the SO topics could represent a threat as pointed by Abdellatif et al. (2020). In this regard, we resorted to metrics used in previous studies (ABDELLATIF et al., 2020; AHMED; BAGHERZADEH, 2018; ROSEN; SHIHAB, 2016) as a mean to counter the threat. Yet, the difficulty aspect, despite being explored in other studies, could actually indicate other circumstances such as lack of popularity or interest in the topic from the community; thus, the clarification of the different possible meanings for difficulty may be a relevant research endeavor.

4 EVALUATING THE USABILITY OF REACTIVE PROGRAMMING APIS THROUGH A MIXED-METHODS STUDY

This Chapter presents the methods and results obtained trough the execution of the second study detailed in Section 1.2 and depicted in the second half of Figure 1. The study resulted in an article titled "On the usability of Reactive Programming APIs: A Mixed Evaluation" which was submitted to the Journal of Software: Practice and Experience (ZIMMERLE; GAMA, 2025).

To answer the following (secondary) research questions, we recurred to a mixed approach by combining information gathered through computed metrics and a usability study conducted with users.

- **RQ3.** To what extent are RP APIs usable, and what aspects are most affected?
- **RQ3.1** How easily can developers learn and understand RP APIs?
- **RQ3.2** To what extent do RP APIs contribute to code cleanliness, reliability, and abstraction from low-level complexities?
- **RQ3.3** To what extent do RP APIs enhance code reuse and maintainability?

4.1 METHODOLOGY

This section describes the methodology adopted in the study. First, we present a brief overview of the JS APIs (Section 4.1.1), along with the reasons we chose Bacon.js and RxJS for the API usability evaluation. Next, we delineate the metrics we explored to analyze those APIs (Section 4.1.2). Finally, the user-centered study (Section 4.1.3) that complements the metric results.

4.1.1 Reactive Programming APIs

The interactive and asynchronous nature of JS created a fruitful environment for RP libraries. Table 9 depicts some of those libraries with their GH repository and stats. As noted by Mogk (2015), the libraries usually differ in terms of focus, custom interfaces, and adopted abstractions. For instance, while RxJS and Bacon.js seem to have a more general-purpose focus, both Most.js and Kefir.js focus on performance. xstream, on the other hand, concentrates on offering a tiny library, mostly targeted to be used with the functional and reactive framework called Cycle.js. They differ in many more aspects, and an entire section could easily be devoted to expose them. However, due to the constraints of our methodology which also include analyzing the usability of RP APIs with users, we only focus on two RP libraries: Bacon.js and RxJS[1] (the JS version of the well-known Reactive Extensions[2]). Both of them are possibly the most popular JS RP libraries according to their GH star count[3] (Table 9) and number of forks. RxJS and Bacon.js have been available and maintained for over 10 years, with RxJS being the most active project: currently, it is in its 7th stable version and going to its 8th. On the surface, they can be used to solve the same problems with similar solutions. Nonetheless, each library has its own peculiarities, some of them briefly summarized below:

- **Abstractions.** They take different approaches to the basic RP abstractions (BAINOMUGISHA et al., 2013): Behaviors and Events. Bacon.js explicitly makes clear distinction and emphasizes the support of both Behaviors (called `Property` in the API) and Events (called `EventStream`) abstractions more aligned with the first ideas of FRP delineated by Conal Elliott (ELLIOTT; HUDAK, 1997). RxJS, on the other hand, puts more emphasis on the Events abstraction (named Observable); however, a close abstraction can be obtained by using either operators like `startWith` and `reduce` or a variation of Subject (i.e., `BehaviorSubject`), a secondary Rx abstraction explored only in specific circumstances.

- **API Size.** The libraries present distinct API surface. Bacon.js presents the greatest number of operators, 149, if considering both factory operators and the `EventStream` instance operators. RxJS has constantly evolving and, in its stable 7th version, it counts with 113 (`subscribe` method included) operators. In its next 8th version, this quantity of operators will slightly decrease, from 113 to 106 (`subscribe` included).

- **Syntactical composition**. Syntactically, Bacon.js uses the method chaining pattern to compose operators, commonly used in both functional and object-oriented programming. RxJS, on the hand, takes a more functional approach by exploring function composition with the help of a `pipe` combinator and standalone functions/operators.

[1] We extended this evaluation to a third, tiny library called xstream in Zimmerle and Gama (2024).
[2] <https://reactivex.io/>.
[3] A common social, selection factor in studies (WEN et al., 2020; XU et al., 2020; ZIMMERLE et al., 2022) as it indicates the repositories' popularity and developers' most probable tool choice.

- **Semantics.** Bacon.js defines a more precise semantic, only supporting hot streams (multicast)[4]. RxJS, conversely, has both cold and hot streams[5] (unicast and multicast, respectively) for the same abstraction: `Observable` type. Furthermore, subscriptions on RxJS `Observables` may execute synchronously or asynchronously, depending on the complexity of the stream (e.g., basic ones created with the help of the `of` operator are most of the time executed synchronously); Bacon.js' subscriptions, on the other hand, always executes asynchronously.

- **Error behavior.** Both libraries have different strategies on the presence of errors: while a stream in RxJS stops working when an error occurs, Bacon.js takes the opposite strategy and does not stop emitting items in the presence of errors; that is the default behavior of those libraries, unless provided an operator to alter that semantic in Bacon.js (e.g., `endOnError`). Besides, any error thrown inside the pipeline of operators is automatically caught by RxJS and forwarded as an error event, while Bacon.js requires either mapping the error with a library `Error` type or wrapping the potently dangerous functionally with a helper operator (e.g., `try`).

- **Glitches.** Only Bacon.js provides information about glitch avoidance, an important characteristic which guarantees consistency (BAINOMUGISHA et al., 2013) if the application requires dealing with changing state.

Table 9 – Reactive JavaScript libraries organized according to their number of stars.

Library	**Repository**	**Stars**	**Forks**	**Commits (last commit date)**
RxJS	ReactiveX/rxjs	30,743	3,003	5,460 (Jun 28, 2024)
Bacon.js	baconjs/bacon.js	6,472	330	3,002 (Jul 1, 2024)
Most.js (version 1)	cujojs/most	3,492	231	800 (Oct 8, 2020)
xstream	staltz/xstream	2,375	136	574 (Feb 7, 2022)
Kefir.js	kefirjs/kefir	1,873	97	1,223 (Mar 12, 2023)
Most.js (version 2)	mostjs/core	403	36	1,115 (Mar 28, 2024)

*N*ote: Last updated on Oct 24, 2024.

Source: Elaborated by the author (2024)

An important aspect of both libraries is the fact that they are currently written in TypeScript (TS), a superset of JS that mostly add static typing to the language. This was a very important

[4] This behavior is also noted in other libraries.

[5] Cold streams are those that generate data in response to individual subscriptions. Hot ones, in contrast, emit values independently of subscriptions and their values are typically generated from an outside source.

aspect for the metrics implementation (Section 4.1.2). Additionally, studies indicate that static typing systems help to find and fix errors faster (KLEINSCHMAGER et al., 2012), enhance the usability of unknown API (PETERSEN; HANENBERG; ROBBES, 2014), and can act as an implicit documentation (PIERCE, 2002). In the context of augmenting JS with TS, recent study results have shown that the use of TS improved code quality and understandability when compared to JS codes (BOGNER; MERKEL, 2022); however, bug proneness and resolution were actually worse in TS projects (BOGNER; MERKEL, 2022). Anyhow, during the study, we did not force any participant to use either JS and TS; the course and the lectures were actually presented in plain JS. Among the task solutions (Section 4.1.3.2), only two solutions used TS; nevertheless, none of them showed many traces of type exploration, allowing a rapid conversion from TS to JS.

Finally, in the metrics explored in the study (Section 4.1.2), there are indications of a possible relation between projects with the greatest number of stars and forks (combined) and the average metric score (i.e., the average of all computed metrics) (VENIGALLA; CHIMALAKONDA, 2021). Therefore, by choosing the most popular libraries, we can assess if this is true for RP APIs and provide more evidence to this tendency.

4.1.2 Metrics

To evaluate the APIs with the help of metrics, we decided to use the ones proposed by Rama and Kak (2015) which are summarized in Section 2.3.1. Exploring those metrics is advantageous given that the authors extensively survey well-accepted beliefs of good design, and the metrics are very general and detailed, with mathematical rigor. Also, they have already been applied in other studies (VENIGALLA; CHIMALAKONDA, 2021).

To implement the metrics, we took advantage that the explored APIs (Section 4.1.1) are implemented in TS, which facilitates the implementation due to the static typing system and function overload support. In this way, we utilized the ts-morph[6] tool, a wrapper to the TS compiler that facilitates code manipulation, to analyze source files and implement the computations. Not all metrics were implemented however, only the first six listed in Table 1. In the case of exceptions (i.e., AESI), it is a little complicated in TS since they are not part of the functions' and methods' signatures yet[7]. Besides, the catch clause does not differentiate

[6] <https://github.com/dsherret/ts-morph>.
[7] Related discussion can be found in <https://github.com/microsoft/TypeScript/issues/13219> and

types of custom exceptions, and it is common the use of the general `Exception` class on most occasions. Thread support in JS/TS, conversely, is used only in specific cases like CPU-bound tasks as most of the computations are handled within an event loop. Therefore, we decided not to include AESI and ATSI in the present study.

The code was implemented by examining the exported, public declarations declared in files as files would correspond to JS module boundary (i.e., the highest-level modular unit). One of the main sources of difficulties stemmed from incorporating type assignability, since TS has not made the function for testing type assignability publicly available yet[8]. To address this challenge, we explored a library called type-plus[9], that implements type checks, to insert dynamic type verifications inside in-memory temporary files created with ts-morph. This file manipulation proved to be a CPU-intensive task (specifically for the metrics AMNOI and APXI), so we exploited a pool of worker threads by using the Piscina[10] library. The metrics APXI and ADI rely on parameters that must be set for the execution of the metric. In APXI case, we set the threshold for the number of function parameters to four (i.e., functions with parameter list greater or equal to four would be considered long) following the same value defined in Rama and Kak (2015). For ADI, we set the threshold to 50 as specified in Venigalla and Chimalakonda (2021).

4.1.2.1 UAX

The implementation of the metrics resulted in a tool we called UAX[11] (ZIMMERLE; GAMA, 2024), intended to resemble the term "developer experience"(DevX) which is often linked to API usability (MYERS; STYLOS, 2016; MURPHY et al., 2018). The tool works as a command-line interface and generates the results for each metric as JSON files. Visualizations can help offloading the cognitive load to human visual capabilities (SOUZA; BENTOLILA, 2009), which in turn may facilitate the recognition of patterns and interpretation of results. In this way, we in parallel created a basic user interface (UI) in the format of a HyperText Markup Language (HTML) dashboard[12] to ease the interpretation of the tool's output. The dashboard is served

<https://github.com/microsoft/TypeScript/issues/52145>.

[8] Related discussion can be found in <https://github.com/microsoft/TypeScript/pull/9943> and <https://github.com/dsherret/ts-morph/issues/357>.

[9] <https://github.com/unional/type-plus>.

[10] <https://github.com/piscinajs/piscina>.

[11] <https://github.com/uax-analyzer/uax>.

[12] <https://github.com/uax-analyzer/uax-ui>.

and accessed through an HTTP server, working as a command-line program.

The main page of the dashboard shows an overview of the computed scores obtained by every API analyzed (Figure 10). We tried to include as much charts and visual elements like radial and progress bars instead of simple tables, so the user can quickly perceive the big picture expressed by the scores. The first part shows not only the average of individual, analyzed APIs, but also the average of all of them; this helps to visualize the individual API perspective and possibly how a category of APIs scored together. We mainly explored the mean statistics to be inline with the computations presented by Rama and Kak (2015); on the other hand, the authors did not indicate how one could get a perception of the results based on categories, serving mainly as a comparative score. In this way, we borrowed the range of categories commonly applied to Likert-based scales (ALKHARUSI, 2022) and adapted to our scale (which varies from 0.0 to 1.0) by means of simple linear transformation. The interpretation of the Likert-based scores in shown in Figure 10. Thus, we used those ranges to map the results between very low usability and very high usability; each range of values was also mapped to a color scale, ranging from red to (dark) green (shown in Figure 11). For instance, a red color indicates a very low level of usability, while dark green specifies a very high level of usability. Throughout the dashboard, many data points are actually colored by using that color scale. The middle part of the main page shows not only the general result obtained by each API, but also depicts the metrics' results by using a radar chart following the suggestions of Souza and Bentolila (2009). It is notable that different colors are used for the data points depending on the metric score. Finally, the last, bottom part shows the metrics' scores of the APIs side by side, allowing one to assess how every API scored from the metric perspective.

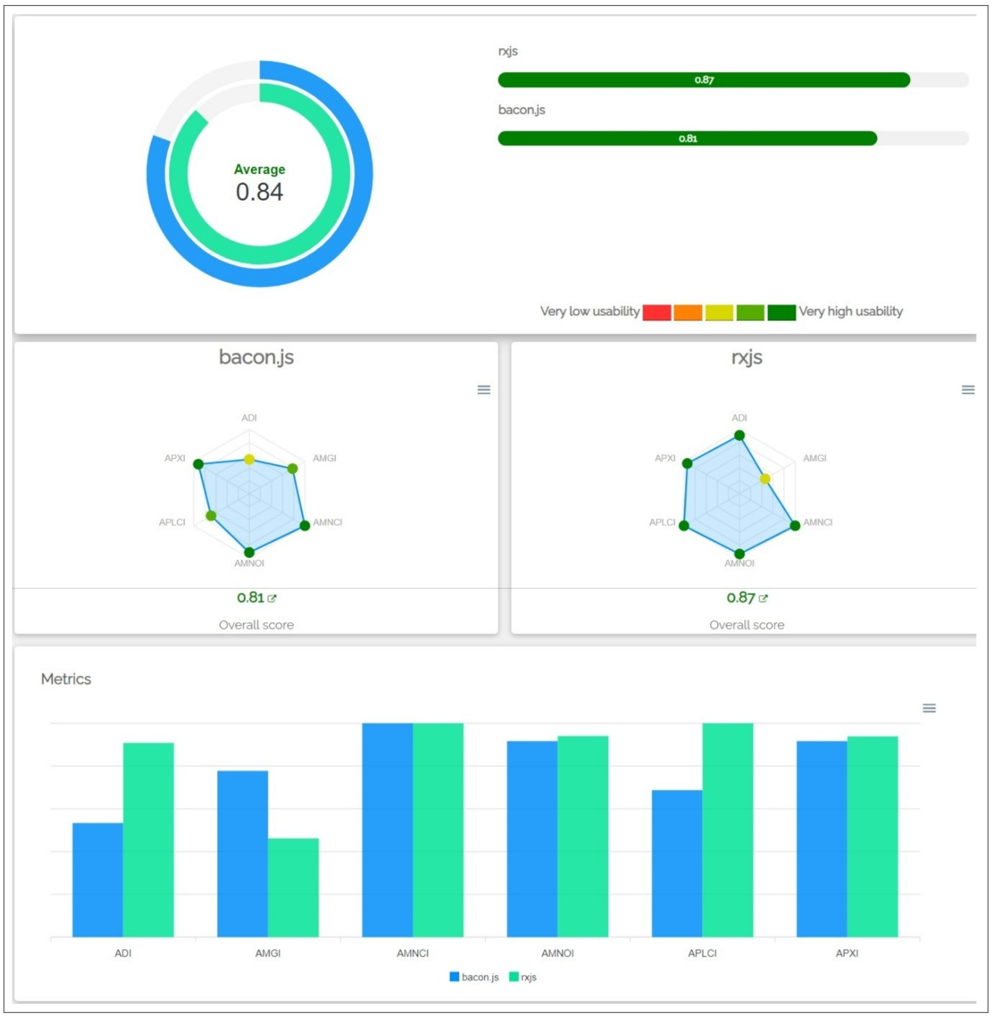
Table 10 – Interpretation of the average Likert-based scores according to equally spaced intervals.

Interval	Interpretation
1 - 1.80	Very low level of usability
1.81 - 2.61	Low level of usability
2.62 - 3.42	Moderate level of usability
3.43 - 4.23	High level of usability
4.24 - 5.00	Very high level of usability

Source: Elaborated by the author (2024)

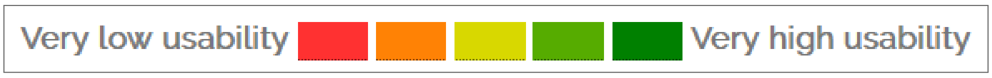
Additionally, we included a second page that displays, with more details and exclusiveness, the information about one of the analyzed APIs as shown in Figure 12. Among the information

Figure 10 – Main page overview of the UAX dashboard.



Source: Elaborated by the author (2024)

Figure 11 – Color scale indicating the level of usability according to scores.



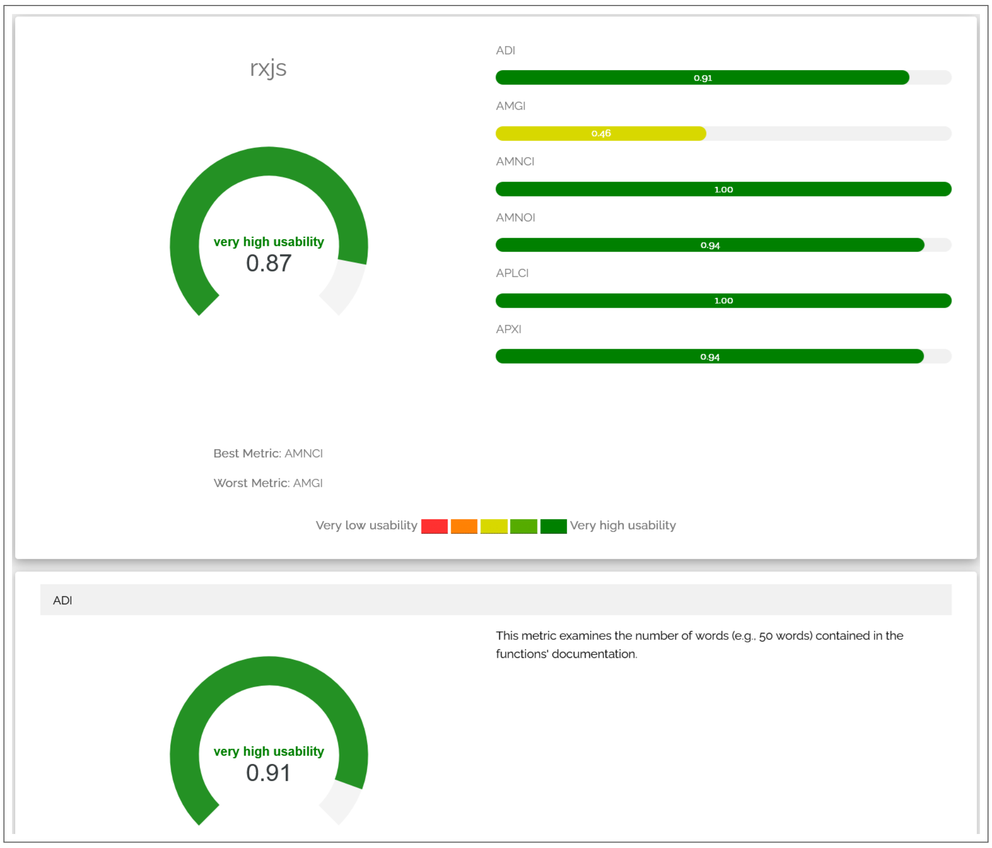
Source: Elaborated by the author (2024)

displayed in that page, we have the overall score of the API, the individual scores for the metrics (informing the highest and lowest values), along with a summary for every metric.

4.1.3 Usability Study

As part of our journey in discovering how usable RP APIs are, which areas are the most problematic that designers should put more efforts, and what usability problems there are, we organized a user-centered study. The study was formulated following usual steps of a usability study which starts by inviting participants to perform programming tasks (DUALA-EKOKO; RO-

Figure 12 – Snippet of the API detailing page in the dashboard.



This metric examines the number of words (e.g., 50 words) contained in the functions' documentation.

Source: Elaborated by the author (2024)

BILLARD, 2012; BRUNET; SEREY; FIGUEIREDO, 2011; ELLIS; STYLOS; MYERS, 2007) and asking them to answer a post-task questionnaire (DUALA-EKOKO; ROBILLARD, 2012; PICCIONI; FURIA; MEYER, 2013). The study took place during a course of introduction to distributed applications (Section 4.1.3.1) where students have the opportunity to learn the basics of RP. Actually, previous to the execution of the tasks, all participants attended to a lecture about RP, so they could have a minimal training (common in empirical studies (PICCIONI; FURIA; MEYER, 2013; SALVANESCHI et al., 2017)) on the subject and APIs. The contents of the lecture mainly included:

- The problems with event-driven programming (e.g., callback hell);
- Definition of RP and their main characteristics;
- Examples of available reactive tools;
- Main RP abstractions and their usual structure; this topic mainly focused on the structure found in combinator-based libraries (MOGK, 2015), which are the most numerous ones in the market, including the APIs used during the study (i.e., Bacon.js and RxJS);

- Marble diagrams, which are a visual way to represent event streams (specially in Rx community);
- Examples of reactive code written in diverse APIs like Bacon.js, RxJS, and most.js;
- Briefly talk about other concepts that libraries may adopt, such as stream multicasting (cold and hot streams), backpressure, glitches, and scheduling. We also briefly showed the rx-fruits[13], a game for learning Rx that could be useful in their learning.

As part of the invitation and convincing process, we had to take into account some considerations: (*i*) As dictated by the directives of the ethics board (Section 4.1.5) of our institution, students should not be put into a situation at which they feel uncomfortable, stressed, or obligated; Also, no type of reward could be offered to the participants and they could leave the study at any time; (*ii*) Participants may have decided not to participate in the study, so we had to make the process as appealing as possible. Thus, we applied the tasks in the form of assignments, in which the participants would have three days to complete them. Moreover, for the same reasons, we did not recurred to video recording or the thinking-aloud protocol, which consists in making the mental process visible through speech (BOREN; RAMEY, 2000), applied sometimes in usability studies (ELLIS; STYLOS; MYERS, 2007; PICCIONI; FURIA; MEYER, 2013). By giving up the recording session and offering more time to participant to accomplish the tasks might be beneficial as stressors such as time constraints and social pressure (e.g., observing person) can have an impact on participants' performance (JANNECK; DOGAN, 2013), which in turn could prompt participants not to cooperate and offer all, possible feedback. Furthermore, previous studies suggest that RP can be hard to master (SALVANESCHI et al., 2017), so an interval time can also contribute to the learning process.

The tasks were elaborated based on related content the students were seeing during the course (Section 4.1.3.2). Before we hand out the tasks to the participants (Section 4.1.3.1) and after the lecture, we explained to the students we were conducting a study, showing its importance and how it would work, and we asked for their cooperation with the promise that they could leave the study at any moment respecting the ethics protocol. For those who accepted, we divided the number of participants equally and sent the tasks with the instructions, which included the API they should work with. For the purposes of our study, we used the two most popular RP APIs for JS (Section 4.1.1), Bacon.js and RxJS, and the

[13] <https://www.rxjs-fruits.com/>.

APIs were assigned randomly to every student. In the task instructions, we emphasized the importance of the study, and that the work on the tasks should be their own, regardless whether or not they were able to finish all of them. Also, we asked the participants to estimate the time they took for each task, and requested that they send the time along with every code solution. By requiring their solution, we aimed to examine the solutions more deeply, to see for instance which features the participants recurred to and how they performed during the tasks. Additionally, the speed of task execution (i.e., time to perform a task) is an important factor to measure (SHNEIDERMAN; PLAISANT, 2010), and has been used in previous studies (ELLIS; STYLOS; MYERS, 2007; STYLOS; MYERS, 2008). Furthermore, this offers an opportunity take a look into the frequency of operator usage. The proliferation of combinators in RP APIs has been a debatable source in the RP area (SALVANESCHI et al., 2017; SALVANESCHI, 2016; MOGK, 2015; ZIMMERLE et al., 2022).

The approach taken to evaluate the perceived usability and their problems corresponded to a post-task questionnaire (Section 4.1.3.3). We explored the well-known CDN questionnaire in which we adapted to our needs. As part of the questionnaire, we also included a demographic section to better understand the participants' profile. Along with the questionnaire and demographic section, we additionally added a satisfaction survey. Comprehension of satisfaction is an important factor in usability (SHNEIDERMAN; PLAISANT, 2010) and the satisfaction level can be used to quantify the user opinion and create a base for comparison and improvement among APIs (MACVEAN et al., 2016). We used the Google Forms service[14] to create and share the questionnaire and made available to participants right after the period of task submissions (remaining available during three days). In the end of the questionnaire, we placed an open-ended section that would allow the participants to insert any relevant details not covered in the previous items of the questionnaire. The data collected from this open-ended space was further incorporated to an interview (Section 4.1.3.4) that took place after the questionnaire's period of response. With the interview, we aimed to complement our findings and obtain a deeper comprehension of the matter by triangulating the data acquired from different methods (JONSEN; JEHN, 2009). Interviews are useful to understand developers' actions (ROEHM et al., 2012), were used in API usability studies (PICCIONI; FURIA; MEYER, 2013; STYLOS; MYERS, 2008), and have been advocated as an important tool to future RP studies (SALVANESCHI et al., 2017).

The questionnaire was analyzed quantitatively, following preceding studies involving CDN (PICCIONI; FURIA; MEYER, 2013; DIPROSE et al., 2017; LÓPEZ-FERNÁNDEZ et al., 2017). The interview

[14] <https://www.google.com/forms>.

was analyzed qualitatively by transcribing the recorded audio and using the coding technique commonly employed in Ground Theory (SALDAÑA, 2021; CORBIN; STRAUSS, 2014). We resorted to an inductive approach and, to increase trustworthiness, two researchers conducted the process. The resulting coding was then discussed, compared, and merged if needed. Afterwards, the categories extracted from the codes are used for further discussion.

4.1.3.1 Participants

The sample of the study correspond to students who were taking the course on introduction to distributed applications. The course is offered for undergraduate students during their third and fourth year, and it includes basic concepts and technologies to build distributed applications, such as Sockets, HTTP/REST, message-oriented middleware, WebSockets, etc. RP was added in the last course offerings where students receive an overview of the paradigm, highlighting its importance to dealing with event-driven, asynchronous programming. The choice of the course was intentional given its linkage with RP and the experience of class students, which ensures a good skill level including object-oriented programming and software engineering. The study was applied throughout three semesters (i.e., 2022.2, 2023.1, and 2023.2).

4.1.3.2 Tasks

To elaborate the tasks, we focused on the context of distributed applications, which is an important area surrounding RP (MARGARA; SALVANESCHI, 2014; DRECHSLER et al., 2014; KAMBONA; BOIX; MEUTER, 2013). Moreover, the participants were recruited during a course in that specific area (Section 4.1.3.1). In total, we defined five tasks (Table 11) all revolving HTTP requests; this topic corresponds to a type of basic distributed interaction that most participants would probably be accustomed either inside or outside the course given its nowadays commonality. For every task, we tried to simulate some close-to-real scenarios, many of them even including comments to exemplify some corresponding situations. Also, we worked with an increasing level of difficulty along the tasks, always starting with a simpler task and asking for evolving the same code in the next exercises; in doing so, we pursued to stimulate the participants through a gradual learning rather than throwing a lot of difficult tasks and ending up risking a high dropout rate (Table 17 of Section 4.2.2.1 provides further details on how many students managed to submit the tasks on time and how complete the solutions were).

Table 11 – Description of the tasks used during the evaluation.

Task	Description
1	Using the endpoint <https://jsonplaceholder.typicode.com/users/[id]>, where id varies from 1 to 10, write a reactive code to consume all user data (1-10) in JSON format; each request must be followed by a time interval (delay) of 3 seconds (that is, there must be an interval of 3 seconds between requests). Data from all 10 users should be shown in the console's standard output (`console.log`).
2	Modify task 1 so that the code consumes the following endpoint <`https://httpbin.org/status/[status_code]`>, where status_code must vary between 401 and 410. Note that requests will generate errors. Therefore, modify your code so that each request is re-attempted at least 3 times. The objective is to simulate the consumption of endpoints that are temporarily offline or not found, that are presenting authorization/authentication problems (e.g., due to some error on the server), or that are providing values in an unexpected/requested format (thus generating parsing problems). For these error cases, the following should be issued through `console.error()`: "An error occurred when requesting the URL [URL consulted] (number of attempts: 3)." NOTE: The same delay (logic) between requests, from task 1, must be maintained in the logic of this exercise as well.
3	Use the endpoint <`https://dummyjson.com/products/[product_id]`>, where product_id varies between 1 and 100, to randomly consume products (simulating, for example, a promotions feed randomly showing products). The product_id must be generated randomly (that is, it must be a number between 1 and 100) always within a 10-second interval (so that the user can appreciate the data for each product/request), thus generating a stream of random ids. The data (products) must be shown in the console's standard output (`console.log`).
4	Extend task 3 so that the main logic continues to make product requests randomly until it reaches a maximum time of 15 seconds. This time could represent, in a real application, a user interaction, such as clicking a button (web page) or activating a sensor (IoT), which interrupts the main flow of requests. The logic must be structured in a reactive way (through the use of some operator to control the end of emissions) and not in an imperative way (by calling the unsubscribe() method within a `setTimeout`, for example) to end the stream.
5	Modify task 3 so that, in addition to displaying product data, the average price of the last 3 products is displayed (that is, you will have to find a way to accumulate stream data relating to the latest requests/products in order to calculate the average price). The average must be shown with the sentence "Average price of the last 3 products: [average price]". All data must be shown in the console's standard output (`console.log`).

Source: Elaborated by the author (2024)

Both tasks 1 and 3 could be considered as the simpler ones and they serve as a base for task 2 and tasks 4 and 5, respectively. In this way, we expected simpler and quicker solutions for both tasks 1 and 3 and a more challenging scenario for the remaining. Tasks 1 and 3 are in fact similar, requiring basic endpoint consumption. Task 2 extends previous tasks by adding error scenarios and error-handling requirements. Things can easily go wrong in distributed environments (KLEPPMANN, 2017) and incorrect error handling was the cause of 90% or more fatal disasters in distributed data-intensive systems (YUAN et al., 2014); thus, it is important to include such scenarios right away in the tasks and that RP APIs encompass usable facilities to deal with those cases. Task 4 augmented task 3 with a time interruption (simulating for instance an external agent like user interaction, sensor reading, etc.) and we asked the participants to do it in a reactive way; in other words, the participant should look for an appropriate operator to break the logic without needing to invoke some method or function in a separate, possibly imperative way afterwards. Task 5 also reuses the task 3's logic, but it introduces some state handling to the scenario. State handling is a recurrent problem for those who have an imperative background (MOGK; SALVANESCHI; MEZINI, 2018), often relying on shared states and side effects. The accumulation logic required for the task offers also an opportunity to deal with a scenario very common in data stream application (AKIDAU; CHERNYAK; LAX, 2018) (it shares the dataflow style with RP), in which we tried to show that, given the unbounded nature of streams, some operators have to operate on small sections of data. Overall, we tried to express every task in a clear way.

4.1.3.3 Questionnaire

For the main tool for data collection, we adopted the CDN questionnaire (Section 2.3.1). Nevertheless, given that the questionnaire was applied online and unsupervised, one fundamental aspect should be the simplicity to obtain good answers in terms of quantity and precision (GANASSALI, 2008). The complete set of CDN dimensions (questionnaire) are often deemed as complex and time-consuming (LÓPEZ-FERNÁNDEZ et al., 2017; DIPROSE et al., 2017). Besides, the dimensions were planned for independence instead of clarity and simplicity (LÓPEZ-FERNÁNDEZ et al., 2017). In this way, by taking inspiration from previous studies (PICCIONI; FURIA; MEYER, 2013; LÓPEZ-FERNÁNDEZ et al., 2017), we formulated and utilized a modified version of the CDN questionnaire. More specifically, we leveraged the structure presented by López-Fernández et al. (2017) which expresses every item of the CDN as an assertion that are answered through a 5-point Likert-based scale. The questionnaire proposed by López-Fernández et al. (2017) was based on the one presented by Piccioni, Furia and Meyer (2013) and groups the questionnaire items into five distinct dimensions: understandability, abstraction, expressiveness, reusability, and learnability. Those dimensions translate into important API usability aspects which are more relevant from the developers' viewpoint (LÓPEZ-FERNÁNDEZ et al., 2017). López-Fernández et al. (2017) provides the mapping between the new dimensions and the original dimensions (i.e., the ones defined by Blackwell and Green (2000)); by using this information, we tailored the questionnaire to our needs and the final result, containing 24 assertions, can be found in Table 12. The evaluation of the questionnaire data follows the additive nature of the Likert-based scale. In this way, each alternative of the scale is associated with a score, normally ranging from one to five, where one is equivalent to the least positive attitude (i.e., strongly disagree) and five to the most positive one (i.e., strongly agree). Some of the assertions of Table 12 are prefixed with an N to denote the ones with negative connotation; in that case, the results for those assertions are then inverted to normalize the evaluation.

Table 12 – Questionnaire based on five dimensions derived from the Cognitive Dimensions Framework. (continue)

Dimension	**ID**[a]	**Assertion**
Understandability	*U1*	The API I used is, in general, easy to understand.
	U2	[N][b] The API proved to be confusing and laborious during use, even for developing simple applications/programs.
	U3	Objects, types, functions and primitives in general of the API appropriately represent (or map) the domain concepts (event-driven programming; handling HTTP requests; etc) in the way I expected. For example, an API for handling IO that exposes a class of type File would probably map well to a representation of a file.
	U4	[N][b] I needed to track information during development that was not clearly represented in the API (e.g. details only present in the documentation; class, method or function information only available in the library implementation; etc.).
	U5	The code required to solve the tasks met my expectations.

Table 12 – Questionnaire based on five dimensions derived from the Cognitive Dimensions Framework. (continued)

Dimension	**ID**[a]	**Assertion**
Abstraction	*A1*	I found the API abstraction level appropriate for the tasks.
	A2	I found it simple to model my application as a composition of streams.
	A3	[N][b] I needed to adapt the API (extend a Class, override predefined behaviors, provide new types, add new features, etc.) to meet my needs.
	A4	[N][b] I felt like I needed to understand the implementation behind the API in order to use it.
Expressiveness	*E1*	I was able to easily transcribe into code, using the API, what was required in the tasks.
	E2	By reading the code produced in tasks using the API, I can easily understand what certain parts of the code do, as well as explain it to other people.
	E3	The API presented descriptive and unambiguous names/identifiers for its various functionalities, such as functions, operators, data types, among others.
	E4	It was relatively easy to find the right operator for a given task.
	E5	It was relatively easy to use different API operations (e.g. easy parameter passing and consistency between different calls, predictable behavior, etc.).
	E6	I had no problems understanding similar features of the API, such as similar operators or data types.
	E7	[N][b] It's easy to make mistakes when using the API.
Reusability	*R1*	[N][b] Developing the tasks required a lot of code (e.g. long or verbose code).

Table 12 – Questionnaire based on five dimensions derived from the Cognitive Dimensions Framework. (continued)

Dimension	**ID**[a]	**Assertion**
	R2	It was easy to evaluate my progress (intermediary results) as I solved the tasks with the API.
	R3	[N][b] There are a lot of different ways to solve certain tasks using the API.
	R4	It becomes easy to maintain and evolve (change) the code by structuring it using the API.
	R5	I can (would) reuse the API features (e.g. operators, data types, etc.) in a simple way.
Learnability	*L1*	The learning/development was incremental. Once solved a given task (or part of it), it was easy to solve either other parts of the same task or other tasks.
	L2	[N][b] I felt I needed to learn many concepts (Classes, operations, dependencies, etc.) to solve the tasks.
	L3	[N][b] I needed to delve into the documentation to be able to develop my solutions (tasks).

Source: Elaborated by the author (2024)

[a] The ID column indicates a unique identification to be used as reference.

[b] Assertions that comprise a negative meaning.

Before answering the CDN questionnaire, we also collected some demographic information to help characterize the experience of participants in the research. The demographic items applied to the participants are found in Table 13.

4.1.3.4 Post-Task Interview

After the questionnaire's period of response, we invited some participants for an interview session, believing that it could help to better understand and confirm the participants'

Table 13 – Demographic items applied to the questionnaire's participants.

Item	Description	Accepted answers
1	How long have you been programming	Less than 1 year; 1-2 years; 2-5 years; more than five years
2	Total of experience with oriented-object programming	None; Less than 1 year; 1-2 years; 2-5 years; more than five years
3	Total of experience with functional programming	None; Less than 1 year; 1-2 years; 2-5 years; more than five years
4	Total of experience with reactive programming	None; Less than 1 year; 1-2 years; 2-5 years; more than five years

Source: Elaborated by the author (2024)

responses. To do so, we formulated a structured interview, composed mostly of open-ended questions, which were conducted trough remote, individual, video sessions. Table 14 shows the script used during the interview. Item 1 is a more open and general item, giving a chance to the interviewee summarize their experience and following the interviews conducted by Robillard and DeLine (2011) and Duala-Ekoko and Robillard (2012). With items 2 and 3, we tried to explore the interviewee's experience. Following, items 4 and 5 were based on the tokens "surprise" and "choice" leveraged in Piccioni, Furia and Meyer (2013); since we did not record the participants' task execution due to privacy and ethical concerns and possible impact on participant performance (JANNECK; DOGAN, 2013), we selected those two tokens as they could be explored without necessarily depending on participants' video recordings. Item 6 take a look into marble diagrams, a visual representation of a stream and operations disseminated by Rx, while item 7 asks about a game, called rx-fruit, presented in class to the participants before the usability testing (Section 4.1.3); together, both marble diagrams and games like rx-fruits can be an important instrument for the visual learners and to aid the users in constructing mental models, so they can better reason about the presented problem (HERMANS, 2021). Besides, one of usability attributes includes the measurement of how well the API mirrors its users' metal models (MYERS; STYLOS, 2016). The goal of the rx-fruit game, on the contrary, is to learn RxJS in a didactic way by completing 16-level tasks, with each level corroborating to make a mix of fruits (juice). Two observations are worth noting. First, although marble diagrams are a compelling way of representing streams and stream transformation, it is most leveraged by the Rx family of libraries. Other libraries do not adopt directly this way of visualization, but one can still find similar illustration. For example, this Bacon.js page <https://github.com/baconjs/bacon.js/wiki/Diagrams> can only be found by going into

Bacon.js GitHub repository; yet, it is very short. Second, the rx-fruit tool specifically targets RxJS; nevertheless, we presented the tool as an optional source of learning that could help elaborating the tasks regardless of the API used. Finally, the last items of the interview, 8 and 9, interrogates whether the participant would consider using RP in the future and if they still have anything to add to the discussion.

Table 14 – Post-Task Interview Guide

Discussion item	Description
1	Please, summarize your experience with the API, including facilities and obstacles.
2	Which points did you like the most when using the API? E.g. syntax, semantics, documentation? What are the main facilities?
3	Which points did you like the least when using the API? E.g. syntax, semantics, documentation? What are the main difficulties?
4	Were there any moments when you felt surprised while using the API? This includes aspects contrary to your expectations. For example, an API class or object only accepts (or works with) objects but does not accept primitive types in which you, according to your understanding, would make sense to use them.
5	Were there times when you were faced with countless choices and had to decide whether the two were equivalent or which was the more appropriate choice?
6	How do you evaluate marbles diagrams for understanding reactive programming? For example, did you use them or did they help you in some way in conceptualizing your solutions? In terms of API documentation, do you think they are an essential mechanism or do you have another visual model in mind?
7	How do you evaluate the didactics of the rx-fruits game for understanding reactive programming and the task development? (if applicable)
8	Do you consider using reactive programming in future projects?
9	Do you have anything else to add?

Source: Elaborated by the author (2024)

In the middle of the interview, we also included additional points to be discussed, which were collected from the open-ended space offered at the end of the questionnaire (Section 4.1.3.3). In total, we elicited ten points to be discussed and, to balance the dynamics of the interview (between open-ended and close-ended questions), we asked the interviewees to answer those points in the Likert-based scale (the same they used during the questionnaire); we welcomed any extra detail that the participant would like to share as a complement to their answer. Each of the elicited point was then organized as assertions (presented in Table 15 and enumerated as A1-A10), in first person, to be akin to the way a questionnaire with a Likert-based scale would be formulated.

Table 15 – Elicited assertions discussed during the interview.

ID	Assertion
A1	I thought the number of operators was a positive point of the API.
A2	I found it easy to find the operation I wanted.
A3	I thought the functions/operations were not very explanatory.
A4	I thought there was a lack of documentation examples.
A5	I thought there was a lack of tutorials related to the tasks.
A6	I had the need to look at the source code behind the API.
A7	I felt a lack of support from the community.
A8	I think the API forced me to think differently.
A9	I think there was a lack of a good practice guide.
A10	I had difficulty visualizing what was in fact reactive.

Source: Elaborated by the author (2024)

We estimated the interviews would take from 20 to 30 minutes. In practice, it took approximately 33 minutes on average.

4.1.4 Questionnaire Dimensions and Metrics Intersection

Both the metrics (Section 4.1.2 and the dimensions (Section 4.1.3.3) of the CDN framework contribute in the creation of usable and high-quality software. Although not directly mentioning factors either measured through the metrics or detailed through the questionnaire items, it is our belief that the metrics can also impact the different dimensions. Table 16 presents the intersection between the CDN dimensions and the metrics along with the supporting reasoning.

4.1.5 Ethics

This research protocol was rigorously reviewed and approved by our university's ethics board. This approval included validation of our informed consent process, ensuring participants were fully aware of the study's scope and their rights. By adhering to these ethical standards, we safeguard participant well-being and data confidentiality, reinforcing the integrity and credibility of our research findings. The data collection only started after the full approval of the study under the CAAE (Certificate of Presentation of Ethical Review) ID 67965523.0.0000.5208, accessible through the *Plataforma Brasil*[15] portal maintained by CONEP, which is the highest body for ethical evaluation in research protocols involving human beings in Brazil.

[15] <https://plataformabrasil.saude.gov.br/>.

Table 16 – Intersection between CDN dimensions and the explored metrics.

Dimension	Metric	Reasoning
Understan-dability	AMNOI	When overloaded functions return different types, it can confuse users and make the API harder to understand. Keeping return types consistent can make APIs more predictable and easier to follow.
	AMNCI	Logical, expected function names make it easier for users to understand what a method does, improving readability and reducing confusion.
	AMGI	Grouping similar methods together helps users find related functionality more easily, making the API more intuitive.
	ADI	Good documentation is a direct way to aid users understand what an API does and how to use it.
Abstraction	AMNOI	Overloaded functions with inconsistent return types can make an API feel messy and less abstract. Clearer, more distinct functions contribute to cleaner and more specialized abstractions.
	APLCI	Keeping parameter names and order consistent across functions reduces the mental effort needed to work with an API, letting users focus on higher-level tasks instead of low-level details parameter particularities.
Expre-ssiveness	AMNCI	Clear, consistent naming makes an API more expressive by allowing users to understand its functionality quickly.
	AMGI	Grouping related functions together better conveys the API's structure, making it easier for users to see how everything fits together.
	ADI	Good documentation makes an API more expressive by complementing what naming conventions or code structure might not cover.
Reusability	AMNOI	Overloading methods with varied return types can make it harder to reuse them effectively if users cannot anticipate their behavior in different contexts.
	AMGI	When related methods are grouped logically, it becomes easier for users to discover and reuse them.
	APLCI	Consistent parameter lists make it easier to reuse functions because users can rely on the same pattern throughout different contexts.
Learnability	AMNCI	Clear, consistent naming directly impacts learnability, as users can quickly understand the purpose of various functions without guessing or extensive documentation.
	ADI	Detailed, well-written documentation makes it much easier for users to understand how to use an API, especially novices.
	APXI	Keeping parameter lists short and avoiding sequences of similar types helps new users pick up the API faster, reducing the chance of mistakes.

Source: Elaborated by the author (2024)

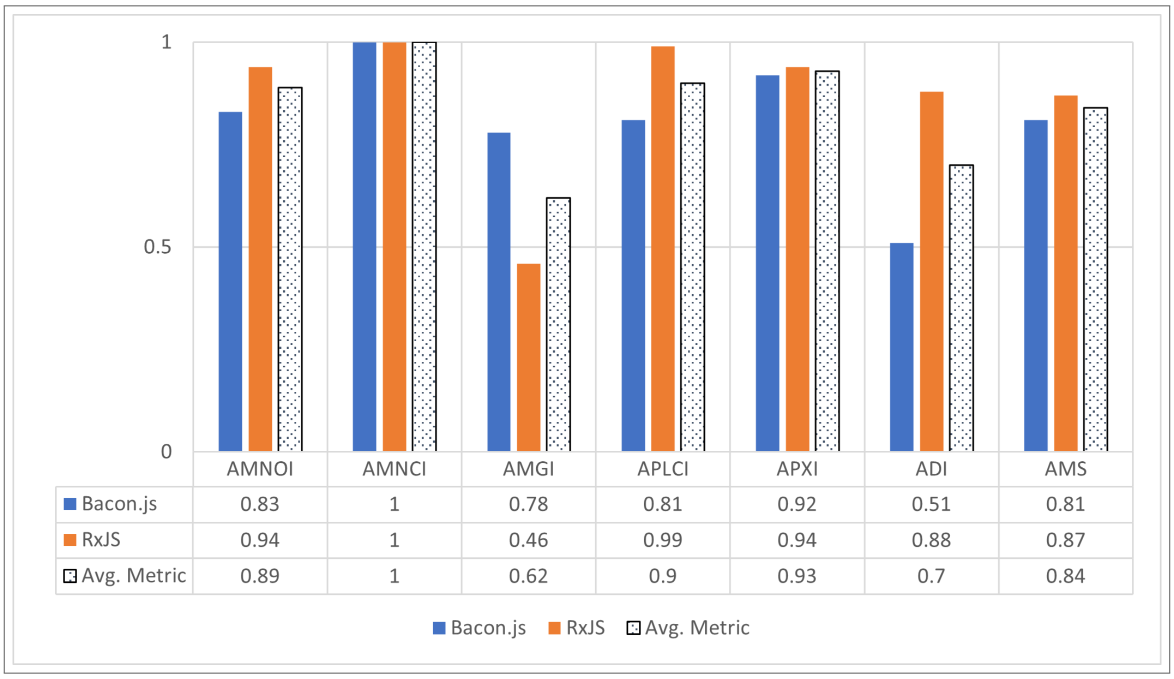
4.2 RESULTS

The following sections analyze the collected data from the metrics' execution and the usability study. Throughout the text, to protect the privacy of the contributors (Sections 4.1.3.1 and 4.2.3.1 describe their profiles), participants who did and submitted the tasks in time are referred as P[1-18].

4.2.1 Metrics

The evaluation of the metrics (Section 4.1.2) for the APIs is presented in Figure 13.

Figure 13 – The results of each evaluated metric according to the APIs. The last chart denotes the Average Metric Score (AMS) of the APIs.



	AMNOI	AMNCI	AMGI	APLCI	APXI	ADI	AMS
🟦 Bacon.js	0.83	1	0.78	0.81	0.92	0.51	0.81
🟧 RxJS	0.94	1	0.46	0.99	0.94	0.88	0.87
▨ Avg. Metric	0.89	1	0.62	0.9	0.93	0.7	0.84

Source: Elaborated by the author (2024)

The great majority of the scores remained above the middle point (0.5), which can be considered a reasonable usability value; in fact, converting the metrics' results to the classification displayed in Table 10 used in UAX, most of the metrics scored high usability level. The individual results for each metric can be summarized as listed below.

- **AMNOI (overloaded functions with disparate return types):** Both APIs exhibited very good results, which is demonstrated by the average metric of 0.89 (very high usability - 4.56 in Likert-based scale). RxJS in fact had the greatest score of 0.94, followed by Bacon.js that also depicted a very high value of 0.83. The scores illustrate good definitions of return types for function overloads of the APIs. For example, by opening RxJS repository, we can observe few operators (functions) with overloads, and those that have overloads like `zipAll` return the same type (e.g., `OperatorFunction`). Bacon.js also has few overloads, but we could note few examples of function returning different types like the method `sampleBy` which returns `EventStream`, `Property`, and `Observable`.

- **AMNCI (name-abuse patterns):** In the case of the RP APIs evaluated, the lists of confusing names were empty; i.e., both Bacon.js and RxJS scored 1. This demonstrates an excellent result for both tools and a considerable effort applied by the API designers in defining good names (i.e., names that do not only differ by: a number placed at their end like `merge` and `merge2`, the presence of underscores like `trace` and `_trace`, and the case of its characters like `setTimezone` and `setTimeZone`).
- **AMGI (grouping of semantically similar functions):** Among the metrics, AMGI was the lowest, but it was still a high usability result (0.62 – 3.48 converted). RxJS was the responsible for pulling down the average metric and it had the worse API score among all metrics (0.46 – moderate level); Bacon.js, conversely, had a more reasonable score (high usability). The reason is explained by the RxJS strategy of modularizing most of its functionalities as standalone functions instead of class methods. Following this strategy, the RxJS team, for example, preferred to defined operators in their own separate files[16]. Interestingly, Bacon.js also takes the same strategy of defining operators in separate files[17], but, it ends up populating those operator in a type `Observable` which is what is in fact exported in the library (thus collaborating to a stronger score).
- **APLCI (consistency among parameter name ordering across functions' signatures):** The APIs combined and alone presented a great level of usability, meaning that a considerably set of their functions shows consistency ordering regarding the parameters. Nonetheless, despite Bacon.js demonstrating a high score for this metric (0.81), UAX reveled some case of inconsistencies like the `delay` and `take` functions that share the same sequence of parameter types, `number` and `Observable`, but it does not follow the same order.

[16] <https://github.com/ReactiveX/rxjs/tree/master/packages/rxjs/src/internal/operators>
[17] <https://github.com/baconjs/bacon.js/tree/master/src>

- **APXI (length of function parameters and runs of parameters of the same type):** Both APIs had an overall, very high level of usability (0.93 – 4.72 converted) in this metric, and the differences among the APIs remained very low. RxJS, in fact, scored slightly higher (0.94), but it was closely followed by Bacon.js (0.92). Looking at the sub-metrics Parameter Length Complexity (C_l) and Parameter Sequence Complexity (C_s) for each API, C_s was the one responsible for pulling down the APXI for the APIs, with Bacon.js and RxJS scoring 0.84 and 0.89, respectively. Yet, both C_s scores could be considered high, indicating that there were few sequences of parameters presenting the same data type. In terms of C_l, the APIs practically maximized the sub-metric, with Bacon.js and RxJS scoring 1 and 0.99, respectively. This indicates that both APIs do not have lengthy parameter lists; a lengthy parameter list being one with four or more parameter elements (RAMA; KAK, 2015). One of the few exceptions for RxJS, for instance, was the `Observable` `pipe` method that is used to chain a set of operators; this method has overloads exceeding five parameters of the same type. In Bacon.js, we could not observe functions and methods exceeding three parameters for example (reason why it scored 1 for C_l); however, there were a few occurrences of parameters lists having the same data type like the `slidingWindow` method or the `takeUntil` function.
- **ADI (number of words in functions' documentation):** This metric was the second lowest overall (0.7 – moderate), only greater than AMGI. RxJS was the least impacted by this metric and provided a very high amount of words in the documentation (0.88 score), greater than the quantity offered by Bacon.js. The outcome for Bacon.js, which was the lowest among its metrics' results, shows that the API has a moderate level of usability, indicating it is an area that possibly need improvement (more API documentation in that case). In fact, it is not difficult to find elements of Bacon.js documentation with very few or no words[18].

According to Figure 13, the highest Bacon.js metric was AMNCI (relative good names), followed APXI (parameter length and sequence of the same type) and AMNOI (good overload return types). AMNCI was also the strongest metric score for RxJS, showing that both APIs have function names that do not make usage of abuse patterns and are less likely to confuse the user. Along with AMNCI, RxJS API also scored great APLCI (consistency of parameter

[18] e.g., <https://github.com/baconjs/bacon.js/blob/4ab1b7d3/src/observable.ts#L477> or <https://github.com/baconjs/bacon.js/blob/4ab1b7d3/src/observable.ts#L1338>.

label ordering), followed equally by AMNOI and APXI with the same score. Therefore, with the exception of APLCI, the metrics AMNCI, AMNOI, and APXI were the most predominant among the APIs. The result was relatively close to the one pointed by Venigalla and Chimalakonda (2021), in which the highest metrics for game engines were AMNCI, APLCI, and APXI. Conversely, the APIs did not have metrics with such a low score that could be considered either low or very low level of usability (Table 10). The lowest score of Bacon.js API, ADI (words in functions' documentation), presents moderate level of usability, revealing that there are spaces for improvements in terms of documentation. This tendency of problems with documentation also seem to affect game engines (VENIGALLA; CHIMALAKONDA, 2021). The lowest RxJS score, on the other hand, regarded the AMGI (grouping of semantically similar functions). As already pointed, the RxJS designers separate most of the API implementation scattered in different files/modules. It is our believe, however, that such a repository structure would not have a great impact on the API users as related concepts are actually grouped at the folder level. Interestingly, AMGI was one of the metrics directly cited as the least satisfied usability metric for the game engines by Venigalla and Chimalakonda (2021) as well. We believe that, for future evaluations, a configuration parameter could be made available for the implementation (Section 4.1.2), so the evaluator can decide if the analysis should proceed at the file level, i.e. the JS module scope, or at the folder level, closer to configuration used by Rama and Kak (2015) which used Java package scope.

4.2.2 User Study

4.2.2.1 Task Completeness

Table 17 presents the percentage of task completeness according to the APIs, together and individually. We classified the tasks in complete, incomplete and not done[19,20]. The complete tasks were those that delivered a complete solution for what was asked. The incomplete ones were those that were either incomplete, had some error, or deviated from what was required. Finally, not done denote solutions not turned in by the participants. Only participants who had at least delivered two solutions (1/3 of the five tasks) on time were considered.

Overall, the participants showed a positive completion rate of more than 60% on average.

[19] The thesis repository includes links to the study materials which encompasses the the source codes of the tasks as well as a spreadsheet used during the tasks' evaluation.

[20] All the process of tasks' analysis was done manually by code inspection and execution.

Table 17 – Percentage frequency of tasks' completeness.

API	Task	Turned in	Complete	Incomplete	Not Done
All	#1	18 (100%)	**15 (83.33%)**	3 (16.67%)	0
	#2	17 (94.44%)	5 (27.78%)	**12 (66.67%)**	1 (5.56%)
	#3	18 (100%)	**14 (77.78%)**	4 (22.22%)	0
	#4	16 (88.89%)	**11 (61.11%)**	5 (27.78%)	2 (11.11%)
	#5	14 (77.78%)	**10 (55.57%)**	4 (22.22%)	4 (22.22%)
	Avg.	16.6 (92.22%)	**11 (61.11%)**	5.6 (31.11%)	1.4 (7.78%)
Bacon.js	#1	5 (100%)	**5 (100%)**	0	0
	#2	5 (100%)	**3 (60%)**	2 (40%)	0
	#3	5 (100%)	**4 (80%)**	1 (20%)	0
	#4	4 (80%)	**4 (80%)**	0	1 (20%)
	#5	4 (80%)	**4 (80%)**	0	1 (20%)
	Avg.	4.6 (92%)	**4 (80%)**	0.6 (12%)	0.4 (8%)
RxJS	#1	13 (100%)	**10 (76.92%)**	3 (23.08%)	0
	#2	12 (92.31%)	2 (15.39%)	**10 (76.92%)**	1 (7.69%)
	#3	13 (100%)	**10 (76.92%)**	3 (23.08%)	0
	#4	12 (92.31%)	**7 (53.85%)**	5 (38.46%)	1 (7.69%)
	#5	10 (76.92%)	**6 (46.15%)**	4 (30.77%)	3 (23.08%)
	Avg.	12 (92.31%)	**7 (53.85%)**	5 (38.46%)	1 (7.69%)

*N*ote 1: The last line of every section has the calculated average for the statistics.
*N*ote 2: Bold numbers indicate the highest value/percentage among the tasks' classification (i.e., complete, incomplete, and not done).

Source: Elaborated by the author (2024)

The incompletion rate unfortunately was higher than we expected, showing a value of ≈30%. The not done percentage fortunately was very low, only getting around 8%. RxJS participants depicted the worst completion rate, ≈50% on average, while Bacon.js users, the least numerous, had a greater rate (80%). As expected, both tasks 1 and 3 had the best rate scores, given their basic level of difficult. A surprise and a common point between the APIs was task 2 (error-handling scenario), which showed a drop in completion rate, specially for RxJS. Participants also demonstrated struggling with task 5, which involved stateful operations.

Table 18 – Time spent per task.

API	Task	Count	Time (minutes)		
			Median	Min.	Max.
All	#1	16	**56**	15	210
	#2	15	40	19	**230**
	#3	16	17.5	5.53	60
	#4	15	30	5	120
	#5	13	23	10	90
	Avg.	-	33.3	10.91	142
Bacon.js	#1	4	33.5	20.98	52
	#2	4	**40**	19	41.2
	#3	4	11.48	10	27
	#4	4	30.51	20	**114**
	#5	4	13.5	10	16.5
	Avg.	-	25.8	16	50.14
RxJS	#1	12	**68.5**	15	210
	#2	11	50	20.27	**230**
	#3	12	22.5	5.53	60
	#4	11	30	5	120
	#5	9	28	15	90
	Avg.	-	39.8	12.16	142

*N*ote 1: Column Count indicates the total of records considered for the statistics calculation.
*N*ote 2: The last line of every section has the calculated average for the statistics.
*N*ote 3: Bold numbers indicate the highest, median and maximum time.

Source: Elaborated by the author (2024)

4.2.2.2 Time of Task Execution

Table 18 presents the time spent during the tasks in terms of median, minimum, and maximum statistics. We preferred not to report the average time[21] given the accentuate variation of the measures; instead, we mostly based our analysis on the median statistic due to its resistance to outliers (ROUSSEEUW, 1990; DAS; IMON, 2014), especially considering that the time was actually informed by the participants and it could be susceptible to some level of variation from the de facto spent time.

On average, the participants spent a total of 33.3 minutes during the elaboration of the solutions. RxJS users took the most time (≈40 min.) on average with the tasks, while Bacon.js

[21] The remaining statistics are rather informed through the supplementary material in the thesis repository.

spent only ≈25 minutes. It was notable and somewhat expected that the volunteers spent more time in the first two tasks; afterwards, the participant took relatively less time (with some variations). RxJS was the API that registered the shortest (task 4) and the longest (task 2) times. Task 2 showing long time is in line with the observation of Section 4.2.2.1 which pointed that participants had trouble with that task. Task 5 also depicted a long time for RxJS, which had more problems in completing the task (Section 4.2.2.1).

4.2.2.3 Code Observations

This section describes important observations we noticed while manually inspecting and executing the source codes of the tasks produced by the participants.

Different paths, the same outcome. We notice within the tasks (e.g., tasks 1, 3, 4, and 5) that participants were capable of combined different operators to get to the same logic. This was more evident with Bacon.js users, and it is was largely anticipated given its extensive range of operators (Section 4.1.1). Bacon.js P8 user, for instance, provided two versions for tasks 3 and 5. There were also a few RxJS codes that presented this characteristic. P13 (RxJS), for example, created a solution for task 4 that diverged from the others and was still correct.

Problematic operators. We identified two Bacon.js operators with uncataloged behavior: `fromPromise` and `fromPoll`. `fromPromise` is used to wrap any JavaScript *Promise* as a Bacon.js `EventStream`, and it was used a lot as many of the tasks required HTTP requests. At least two solutions (i.e., P1 and P8) used the (Bacon.js) `flatMapError` operator to map the cases of errors that could happen inside the `fromPromise` function. However, other two codes did not use the `flatMapError` and still succeed. By taking a look at the `fromPromise` implementation [22], one can find out that the operator actually automatically wraps any error case in a Bacon.js Error[23] instance, but it is not described in the documentation. A great majority of the solutions also exploited the factory operator `fromPoll`, which allows repeatedly invoking a function according to a given delay. The repetition is controlled by returning two different types of Bacon.js objects: `Next`[24], representing the next emissions, and Bacon.js `End`[25] to halt the repetition. Surprisingly, the operator automatically wraps any return as a `Next` object (e.g., P1, P8-v2), making the code more compact and less verbose, but the

[22] <https://github.com/baconjs/bacon.js/blob/master/src/frompromise.ts#L25>.
[23] A class that represents a event of error in Bacon.js: <https://baconjs.github.io/apP8/classes/error.html>.
[24] <https://baconjs.github.io/apP8/classes/next.html>.
[25] <https://baconjs.github.io/apP8/classes/end.html>.

behavior is also not covered in the documentation.

We also found that factory operators that rely on return statements like Bacon.js `fromPoll` and `repeat` can stimulate the production of more imperative code as the user is forced to introduce some branch logic to control their behavior. The code ended up being more verbose and complicated (e.g., it could be observed in many P5 codes). The return values are also inconsistent. While `fromPoll` depends on `Next` and `End`, which are subclasses of `Event`[26], to control the iteration, `repeat` relies on a `EventStream` return or a falsy value (like `null` or `undefined`). Without care, a developer may assume that the `repeat` iteration can be break with the `never` stream[27], which would be more consistent. Overall, we believe that solutions that relied on `interval` (both found in Bacon.js and RxJS) for example produced clearer reactive codes and have better mapping with the language constructs like the JavaScript global function `setInterval`, that would probably be used in the codes without the RP APIs.

We observed the use of two Bacon.js operators that could be problematic as the user can be either misled based on the name or have trouble to find it; yet, there were no complaints from the participants. `bufferingThrottle` throttles the emissions without discarding by buffering the values for future emissions. The problem is that it can be name misleading, as some API users can think that the operator produces Arrays just like Bacon.js `bufferWithCount`/`bufferWithTime`/`bufferWithTimeOrCount` (they are even displayed closely in the library documentation) or RxJS `bufferCount`/`bufferTime`/`bufferToggle`, when in fact it does not. The `slidingWindow` was used by at least two solutions (P1 and P8) and it works akin to the sliding window concept found in data stream applications (AKIDAU; CHERNYAK; LAX, 2018). `slidingWindow` is actually interesting as the API does not make any distinction between the buffer class of operator and the window operators. For instance, in RxJS, buffers are utilized to produce results as array of elements, while the window class of operators is reserved for those that return the result as (sub)streams of data; moreover, RxJS keeps a corresponding version for each one of those classes. Hence, if there is a RxJS `bufferCount`, equivalent to a fixed window or Bacon.js `bufferWithCount`, there is also a RxJS `windowCount`. In Bacon.js, there is a full class of buffer operators like `bufferWithCount`, `bufferWithTime`, and `bufferWithTimeOrCount`, but only `slidingWindow` has a window nomenclature.

Error-handling. We observed a lot of problems with RxJS error-handling required for task 2.

[26] <https://github.com/baconjs/bacon.js/blob/4ab1b7d36cdfe49dc5474099d1ff4c959064b56a/src/event.ts#L9>.

[27] <https://baconjs.github.io/api3/globals.html#never>.

RxJS codes in task 2 were in general less readable and longer than Bacon.js solutions. RxJS offers a good arsenal of error-handling operators. Some of the codes were actually in the right path, but missed or misplaced essential operators (e.g., `catchError` and `retry`) either to retry the stream or prevent the stream from stopping work on the presence of error (e.g., P3, P14, and P15). Others (e.g., P2, P3, and P11) used operators in a wrong way (i.e., `retryWhen` and `onErrorResumeNext`), while some developed their own implementation (e.g., P17). We think that Bacon.js semantics regarding errors (i.e., not killing the stream in the presence of errors) may have helped Bacon.js participants; yet there was a Bacon.js participant (i.e., P18) who preferred to create his own implementation. Anyhow, it is important to emphasize that neither API provide a dedicated space to guide the users to correctly deal with error handling; better guidance could probably had rendered better RxJS solutions.

RxJS Subject. Subjects were barely talked about during the course class (training), and, still, P4 used it in task 4. By combining a `Subject` instance with a `takeUntil`, P4 explored the `Subject` to create a imperative mechanism that was used to control the `takeUntil` with the help of a global JavaScript `setTimeout`. In fact, Subjects (`Bus` in Bacon.js) can lead to a more imperative code and they are a debatable concept in the Rx world[28,29], reserved only for special occasions. The APIs should strive to clarify when better use such concept in code.

Lack of best Practices. During code inspection, we observed some instances of code poorly designed which we denoted as lack of best practices. While some of them helped a few students to produce the asked outcomes, the final product resulted in code hard to understand and often nested that would probably not scale well and offer the wrong perception about the API (a tool to fight the complexities of event-driven programs). In what follows, we condensed some of the design issues we could perceive:

- **Side effects:** Some participants resorted to side effects inside pipelines, either to log intermediate emissions or store data for logic control or further access. A few even recurred to the concept of hot streams (multicast consumer), barely touched in class, to attach two consumers, one for logging and another for carrying out the task demands.
- **Global variables.** A couple of students relied on global variables, aligned with side effects, to control their logic or store data. Task 5, for instance, had some implementations using external variables (numbers and arrays) to serve as a temporary buffer.

[28] <https://stackoverflow.com/questions/9299813/rx-subjects-are-they-to-be-avoided>.
[29] <https://stackoverflow.com/a/31467377>.

- **Mix of Async. APIs:** There were some solutions that specifically used the Promise API facilities to bundle a lot of the logic instead of relying on the RP API operators. Some of those mixtures ended up producing nesting, hard-to-understand codes with long callbacks and uses of side effects and global variables. This is specially dangerous, given the peculiarities of asynchronous programming.
- **Long callbacks:** Among the problems observed, this one undermines the readability of RP code. We could observe many instances of long callbacks, concentrating a lot of concerns.
- **Nesting subscription:** This was more noticed in one of the participants' code. Rather than using an operation to deal with stream of streams and flattening, the student recurred to create streams inside the consumer part or subscription, creating a pattern of nesting, indented code.
- **Unnecessary use of operators:** A student repeatedly overused RxJS `pipe` combinator, by calling it every step of the pipeline. In fact, the participant use it to pass only one parameter or operator, not taking advantage of its variadic property. Strangely, there was also excessive use of creation function to wrap intermediate results along the pipeline course. As a result, this forced the participant to many times recur to the `concatMap` operator to streamline the logic (i.e., to flatten the intermediate streams).

4.2.2.4 Operators

Figure 14 depict the frequency of the operators used during the solutions using both APIs. Bacon.js has an extensive catalog of operators, 149 as detailed in Section 4.1.1, and considering the operators utilized during the tasks, approximately 19% (29 operators) of the Bacon.js operators were used for the tasks. RxJS, on the contrary, has undergone some interface changes (Section 4.1.1), and its version 7 counted with 113 operators (*subscribe* included), while the next version, 8, has 106 (`subscribe` included). Considering the operators used throughout the tasks are in both RxJS versions, the participants required ≈28% and ≈30% (32 operators) of the operators of the API versions 7 and 8, respectively.

By examining the operators and their frequencies, a series of patterns can be uncovered. For instance, the most utilized Bacon.js operator was `fromPromise`, while the most explored for RxJS was `subscribe`. Given the nature of the task, it was expected that operations like `fromPromise` or similar ones were at the top of the most requested functions. In RxJS, the function used for wrapping *Promises* is the `from` operator and it appears as the 4th most utilized one, only behind `subscribe`, `interval`, and `map`. Having those operations as the most used for RxJS instead of `from` is not strange as demonstrated by Zimmerle et al. (2022) in which showed `subscribe` and functional combinators like `map` as the most requested RxJS operators in open-source projects. However, taking into account the big difference between Bacon.js `fromPromise` and RxJS `from` frequencies and considering there were more RxJS delivered tasks, we took a look into RxJS solutions. A close inspection revealed that many source codes (e.g., P7 in all tasks and P9 in tasks 3 and 4) utilized the `concatMap` operator to map stream emissions to *Promise*s but without wrapping the *Promise* object with the appropriate RxJS operator as one would expect. This behavior is actually not cataloged in the official documentation (<https://rxjs.dev/api/index/function/concatMap>), but, in fact, revealed through other non-official resources[30]. The same situation can be observed with `mergeMap` (P10 in tasks 4 and 5) and `switchMap` (P3 in task 4). Although apparently not so important for small tasks, this behavior and interaction with the *Promise* API should have been detailed clearly in the documentation, so all users are aware of this possibility.

The second most exploited RxJS operator was `interval`, which matches the majority of the task requirements that asked for some type of interval or repetition interaction. As already commented, some participants recurred to alternative ways, in special with Bacon.js. Figure 14 exhibits that Bacon.js users preferred to use the `fromPoll` operator to create the interval behavior. However, Section 4.2.2.3 observed that in some situations, `fromPoll` did not collaborate to create some clean code and it has behavior not described in the documentation.

The majority of the Bacon.js task implementations picked the `onValue` to consume the elements of the streams, while RxJS users greatly depended upon the `subscribe` operation; RxJS only counts on the `subscribe` to consume the stream, with the variation of either informing an `Observer` object implementing at least a `next` method or a callback equivalent to the `next` method. Bacon.js, on the contrary, has different ways of subscribing to the stream (e.g., `onValue`, `log`, and `subscribe`). Bacon.js version of the `subscribe`, however, receives only one parameter, an `EventSink`[31], and the user has to introduce branch test to identify the type of current emission (a value, an error, or an end signal); this may explain why no

[30] E.g., <https://www.learnrxjs.io/learn-rxjs/operators/transformation/concatmap>
[31] <https://baconjs.github.io/api3/globals.html#eventsink>.

Bacon.js participant explored the `subscribe` method, aligned with the lack of examples [32] and `EventSink` description.

RxJS `of` and Bacon.js `once` are among the most basic factory operator to wrap single values. Still, RxJS `of` stood out as a very requested operator, even though there were not much space for its use in the tasks. Participant P3, for example, many times used it unnecessarily.

Given the more strict requirement for error handling, it was expected that operator frequencies showed some considerable utilization of error handling operators for RxJS, specially considering the demands of task 2. However, as Figure 14 reveals and complemented by the data displayed in Table 17 (task 2), the exploration of such operators was way below the expected. Two Bacon.js functions had some usages, `onError` and `flatMapError`, but the API's default approach to error handling (Section 4.1.1) aligned with non-described behavior of `fromPromise` (Section 4.2.2.3) may have helped Bacon.js participants. An interesting point in the RxJS code was the presence of the `onErrorResumeNext`, which is not properly indicated in the documentation section of operators (<https://rxjs.dev/guide/operators>); one can only find it by searching for it in the search bar of the web documentation.

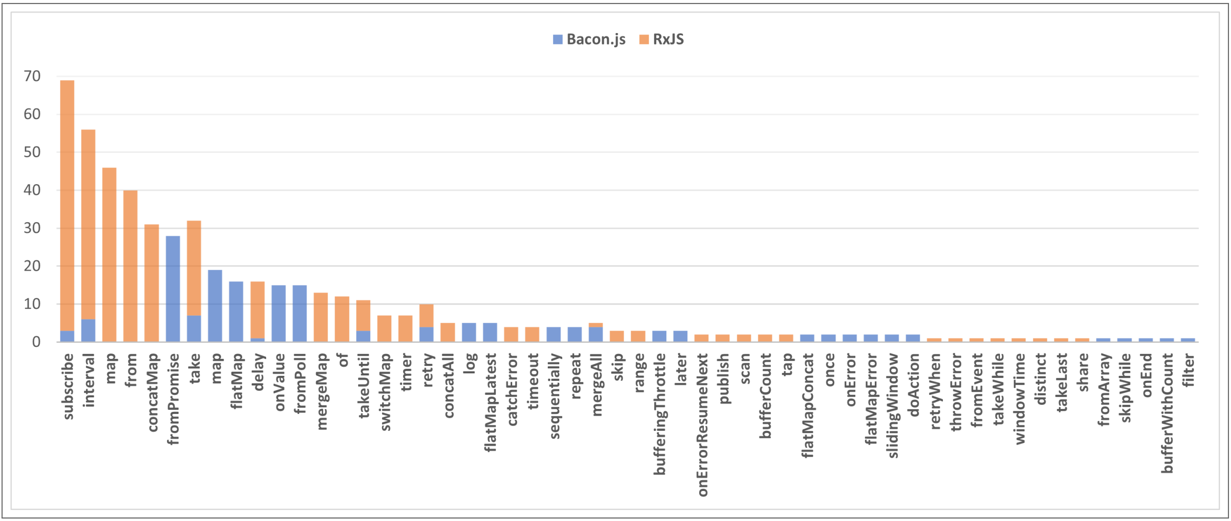
Debugging, as well as testing, was pointed as a concerning area in RP by Zimmerle et al. (2022), and, by inspecting Figure 14, one can notice the low frequency of the `doAction` and `tap` operators. In fact, the solutions not necessarily had to include those functions as most of the side effects could be concentrated at the consumer/subscription part of the applications. Nonetheless, a lot of participants wrongly mixed side effects inside the pipeline of operators, not realizing the implications of introducing side effects (e.g., difficult testing or memoizing of the functions). Unless one already had a basic functional background (i.e., understanding the advantages of side-effect free functions and the use of `tap` combinator), the APIs do not make clear efforts to clarify the importance of fitting those debugging functions inside the pipeline in the presence of side effects.

4.2.3 Questionnaire

This section examines the answers for the questionnaire applied after the conclusion of the tasks. In total, 12 participants (Section 4.2.3.1 describes their profile), who did the tasks, made themselves available to answer the questionnaire. From those, four experimented with Bacon.js, whereas eight explored RxJS. The following sections explore the results in accordance

[32] <https://baconjs.github.io/api3/classes/eventstream.html#subscribe>.

Figure 14 – Usage frequency of the Bacon.js and RxJS operators used during the tasks.



*N*ote 1: A total of 29 and 32 operators were utilized from Bacon.js and RxJS, respectively.
*N*ote 2: Operators sharing the same name that were placed in the same bar also share the same semantics.

Source: Elaborated by the author (2024)

with the main parts of the questionnaire.

4.2.3.1 Demographics

To characterize the participants, we ask them to provide some information about their experience in the field. Table 19 depicts the profile of the questionnaire respondents. Regarding experience, we asked about their general experience with programming, as well as, with the paradigms: Object-Oriented (OO) programming, FP, and, finally, RP, in case they had any previous exposure to the area. Half of the volunteers declared having more than five years of programming experience, while the other half answered having from two to five years. The set of more-than-five programming experience includes all of the Bacon.js participants, while the RxJS ones are divided between more than five and two to five years. Regarding OO paradigm, at least four had been exposed to it for more than five years, whereas other four had between 2 and 5 years of OO experience; the remaining three feel into the 1-2 years category. As expected and given its popularity, the respondents claimed to possess more OO knowledge than FP. In this way, only one participant answered having more than five years of exposure with FP; four of them had two to five years of experience, while the great majority (five) showed one to years of practice. Surprisingly, two students had less than one year of FP. Regarding RP, a positive aspect is that half of the participants had some contact with the paradigm before, albeit six declared had no exposure. Four acknowledged having one to two years of experience with RP; on the other hand, other two only had less than one year of RP contact. In general, it was a very diverse group, with varying levels of experience among the API users. Bacon.js participants presented the most experienced group, albeit, in terms of paradigm experience, the scenario was a little mixed.

Table 19 – Experience of the participants who answered the questionnaire according to the explored API.

API	**Participant**	**Interview**	**Experience (years)**			
	ID	**Participation?**	**Programming**	**OOP**	**FP**	**RP**
Bacon.js	P1	yes	> 5	> 5	1-2	1-2
	P5	yes	> 5	> 5	1-2	1-2
	P8	yes	> 5	2-5	2-5	1-2
	P12	-	> 5	> 5	> 5	0
RxJS	P3	yes	> 5	1-2	1-2	0
	P4	-	> 5	> 5	< 1	< 1
	P6	-	2-5	2-5	2-5	0
	P9	-	2-5	2-5	2-5	1-2
	P10	yes	2-5	2-5	1-2	0
	P11	-	2-5	1-2	1-2	< 1
	P13	yes	2-5	< 1	< 1	0
	P14	-	2-5	1-2	2-5	0

*N*ote 1: OOP stands for object-oriented programming.
*N*ote 2: FP stands for functional programming.
*N*ote 3: RP stands for reactive programming.

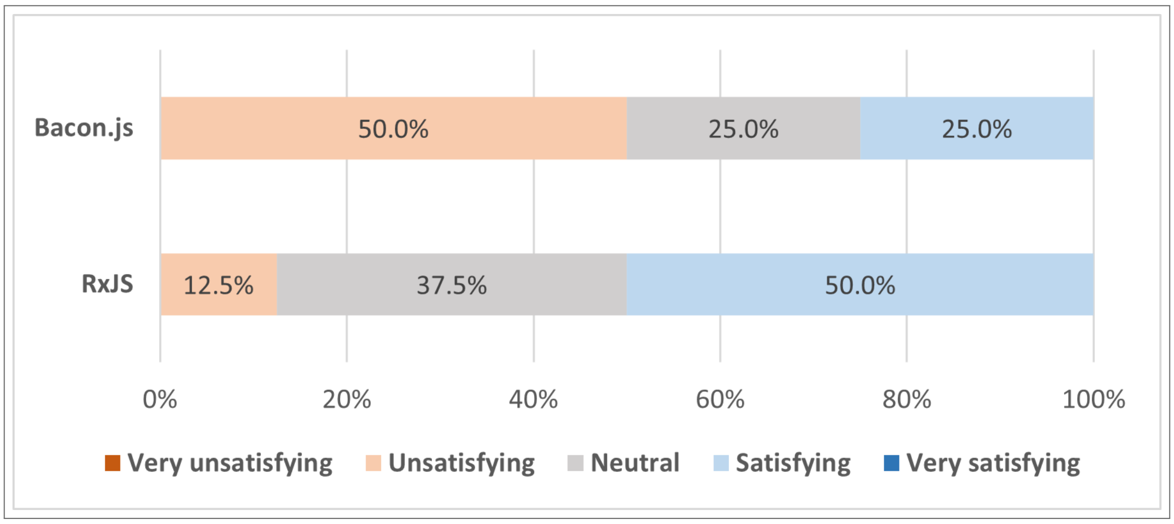
Source: Elaborated by the author (2024)

4.2.3.2 General Satisfaction

One of the first items of the questionnaire (Section 4.1.3.3) corresponded to collect the general level of satisfaction from the participants. Figure 15 summarizes the respondent answers according to the API explored. The results show half of the four, Bacon.js users reported having an unsatisfying experience with the API; only one (25%) user considered having a satisfying experience while the other one (25%) neither considered satisfying or unsatisfying (neutral). RxJS, on the other hand, had four participants (50%) of the eight respondents who declared having a satisfying experience. Interestingly, other three (37.5%) were neutral, neither considering an unsatisfying or satisfying moment. Unlike Bacon.js, only one (12.5%)

participant stated having a disappointing experience.

Figure 15 – Answers to the Likert-based question that measures the API satisfaction level according to the questionnaire's participants.

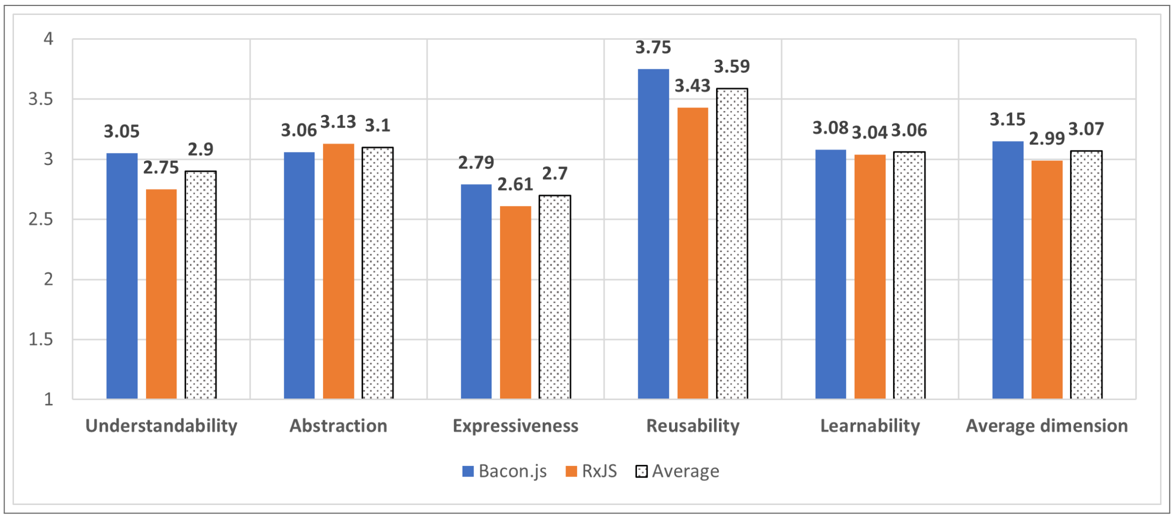


Source: Elaborated by the author (2024)

4.2.3.3 Cognitive Dimensions Questionnaire

The result for every assertion, grouped according to their associated dimension and APIs, of our cognitive dimensions questionnaire (Table 12) is displayed in Table 20. The calculation followed the usual procedure applied to Likert-based scale (ALKHARUSI, 2022), consisting in mapping every response to a predefined weight, ranging from 1 (strongly disagree) to 5 (strongly agree), and calculating the average of those values. For the assertions with negative connotation like U2 and U4, we inverted the values of weights; so, the results are expressed in a positive direction and higher values indicate better usability for the assertion. We also calculated the average values for every dimension with results from both APIs (Table 20) and the average obtained by each API (i.e., the average of the dimension values obtained individually by each API), which are displayed in Figure 16. Considering the interpretation of Table 10 (also used in UAX), the majority of the dimensions scored very closely (Figure 16), both in terms of API or general average dimension, and most of them could be interpreted as presenting a moderate level of usability. The only exception was reusability in which scored high (3.75), and it was the highest dimension for both APIs. Actually, the APIs also shared the lowest dimension as well, expressiveness, having RxJS exhibiting the lowest average score among all computed scores. According to Table 10, RxJS expressiveness is actually at the upper limit of the low level interpretation. If we draw a line at the middle point (2.5) of the scale and we face the interpretation from that angle, we can actually interpret the results more toward a positive attitude but still far from the highest value and very near the middle, neutral point. The mean of the average dimension, displayed in Figure 16, summarizes our observations by showing that both APIs in general scored moderate; Bacon.js had a value slightly greater than RxJS, but, still, both scores are within the moderate level.

Figure 16 – Average rankings on the dimensions explored in the questionnaire. Maximum scale reduced from 5 to 4 to aid visualization.



Source: Elaborated by the author (2024)

4.2.4 Interview

Based on their inclination and collaboration during previous stages, a total of eight people were invited to be interviewed after the questionnaire. To be fair, we balanced the groups, so four were from Bacon.js, and the others from RxJS. From the eight, six accepted the invitation, three from each group. According to Table 19, participants P1, P5, and P8 were the Bacon.js participants interviewed; conversely, P3, P10, and P13 were the RxJS interviewees. As detailed in Table 19, the interviewees from Bacon.js presented a close profile, with all of them having more than five years of programming experience along with a great OO level of experience. The RxJS group was most varied, but still presenting an elevated experience.

Table 20 – Results for every questionnaire assertion according to their dimension and the API explored by the participants.

Dimension	**ID**	**APIs**		
		Bacon.js *(n = 4)*	**RxJS** *(n = 8)*	**Avg.**
Understandability	U1	2.5	2.75	2.63 *(moderate)*
	U2	2.5	1.88	**2.19** *(low)*
	U3	3.5	3.13	3.32 *(moderate)*
	U4	3.25	2.75	3 *(moderate)*
	U5	3.5	3.25	3.38 *(moderate)*
Abstraction	A1	2.75	3.38	3.07 *(moderate)*
	A2	2	2.63	**2.32** *(low)*
	A3	4.5	3.38	3.94 *(high)*
	A4	3	3.13	3.07 *(moderate)*
Expressiveness	E1	2.5	2.63	**2.57** *(low)*
	E2	4	4	4 *(high)*
	E3	3	2.63	2.82 *(moderate)*
	E4	2.25	1.75	**2** *(low)*
	E5	3.25	3.38	3.32 *(moderate)*
	E6	2.5	2.5	**2.5** *(low)*
	E7	2	1.38	**1.69** *(very low)*
Reusability	R1	4.5	4.13	4.32 *(very high)*
	R2	4.25	3.75	4 *(high)*
	R3	2.25	2	**2.13** *(low)*
	R4	3.5	3.63	3.57 *(high)*
	R5	4.25	3.63	3.94 *(high)*
Learnability	L1	3.5	4.13	3.82 *(high)*
	L2	3.5	2.5	3 *(moderate)*
	L3	2.25	2.5	**2.38** *(low)*

*N*ote 1: *n* is the number of questionnaire respondents.
*N*ote 2: Bold numbers indicate the most concerning assertion, either low or very low level, for the dimension.

Source: Elaborated by the author (2024)

4.2.4.1 *Categories*

Documentation. Documentation was the category most cited by the participants. The majority of the discussion concentrated on bad aspects of the APIs' documentations. Both interviewees P1 and P5 considered Bacon.js poor documented. P1 argued the API could have been better documented, considering one of the main sources of difficulty during the tasks. P5 also complained and felt limited by the documentation, arguing the documentation was too simplistic and messy in a way it was not able to truly expose the API. P5 insisted there was a chance that the library indeed offers a really good set of tools and does not need to be enhanced, but, since he could not understand it and situate himself while using the API, many of his complaints could possibly be due to the quality of the documentation.

Numerous documentation problems revolved around the poor resource description described specifically in the Bacon.js docs. Both P1 and P5 participants pointed to the fact many resources like major objects, constants, and functions do not include either good descriptions or any description whatsoever. P1 provided as an example the `retry`[33] function, which, besides presenting a strange syntax, was not good documented. P5 also suggested a few examples of bad documented resources, such as `Spy`, `takeWhileT`, and `toDelayFunction`[34]. As stated by P5, for someone who is experienced in the area, having short descriptions or none may be enough, but, for someone just starting, it is not and leads to a lot of guessing. In the below passage, P5 recognized some names could be somewhat obvious and the work in open-source project may not be easy, but, to attract people, the usability is key: *"I even understand that there are some names that are a bit obvious, but man ... [...] I normally understand that there are few people, I understand that it's not easy to do this work, really. But, if you set out to do this, if you want us to use it, you will only get people using this thing if it has good usability, you know? And if it's easy for you to learn. Easy...relatively easy."* (P5)

Another major pain point brought by the interviewees was the insufficient and often low-quality examples. P10 (RxJS) felt the documentation examples were not enough, mainly in specific cases like when he was dealing with error-handling scenarios (task 2). Two Bacon.js users complained about the quality and lack of examples as well. P5 repeatedly called attention to the fact that many resources do not present an example in Bacon.js docs. For instance, P5 commented about the `interval`[35] operator that had at least a description but totally missed the usage example; also, Bacon.js `retry`[36], needed during the task 2, includes a poorly formatted example. For P8, the lack of examples was his biggest criticism toward Bacon.js; according to him, RxJS has more examples than Bacon.js, and, although not all of them are

[33] <https://baconjs.github.io/api3/globals.html#retry>.
[34] All of three resources are available at <https://baconjs.github.io/api3/globals.html>.
[35] <https://baconjs.github.io/api3/globals.html#interval>.
[36] <https://baconjs.github.io/api3/globals.html#retry>.

good, they are still there. P8's comments pinpoint the importance of introducing examples during the learning process: *"I think it's a point they could work on more; especially if you feel like "ah, I want to learn how to use this library, I'm not here only to do the tasks", which was the case with us there. Having examples makes it much easier."* (P8)

In the same direction, Bacon.js participants were discontent about the quality of the API tutorial. P1 suggested the introduction of a kick-start tutorial for Bacon.js, as he considered the documentation tutorial bad. P5 considered Bacon.js tutorial bad, extensive, and very specific, recommending a more segmented tutorial with diverse cases like HTTP requests, error-handling scenarios, etc. P10 (RxJS) missed more cases as well, regarding more specific error-handling scenarios specially like when things go wrong.

The tasks' solutions left some Bacon.js participants unsure if they have followed the best practices. P8, who had a great performance during the tasks (Section 4.2.2), stated there were many ways to accomplish the tasks and, during the process, he was not certain he was following the good practices of the API. P5 was more emphatic about his solutions' problems, explaining he likes to doing things the best way possible, but his codes with Bacon.js seemed to be workarounds. In this way, P5 admitted it was even hard to affirm there are API usability problems since he did not even know if his work was done following good practices.

Not every commentary relating to the documentation was pessimistic. P1 acknowledged that there were methods relatively well documented with examples. P3 also stated the RxJS introduction pages were simple, and P8 (Bacon.js) said once the required operations were found in the documentation, the explanation was clear about what they do.

Operators. Along with documentation, operators was a recurrent source of discussion during the interview. A considerable part of the discussion concerned the usage of some operators, which we tagged as confusing ones. Interviewees reported a few operators' names were not so clear and needed more information or naming enhancement, specially the class of `flatMap` operations (i.e., to deal with stream of streams). This type of nomenclature confusion regarded the `flatMap` operators was observed in both groups of APIs. P1 reported feeling confused about the difference between Bacon.js `flatMap` and `flatMapConcat` at the beginning, but he ended up using both during his solutions (all of them considered complete in the analysis). However, P1 also mentioned, by the time of the interview, he did not remember the difference anymore, and, based on the nomenclature, he would expect that *flatMapConcat* concatenates the results in some form but it did not seem to be the case. P10 also confirmed the confusion with the `flatMap` operators in RxJS. While understanding their need, P10 described moments of either constantly visiting the documentation to ensure the right operator or wrongly using one instead of the other; so, P10 thought the nomenclature for those types of operators could be different to help a little more. Even P8, who had previous experiences with RxJS, also considered the nomenclature for RP `flatMap` operators not good for both APIs; yet, P8 thought the other Bacon.js operators had good naming and matching description.

During P1's interview, the participant often demonstrated dissatisfaction with the Bacon.js `retry` (task 2) calling it confusing and obscure, completely different from Apple Combine one-parameter version[37] (very similar to RxJS); P8 also reported trouble with that Bacon.js operator. Based on the task description, P8 agreed he needed the `retry` function to do the task, but it was hard to make the operator work correctly. Contrarily, P5 did like Bacon.js `retry`, arguing it was well-defined and covers all use cases; P5's solutions, however, were always accompanied by problems and mostly resembled an imperative code, and he, during interview, doubted the quality of his produced codes. P5 actually did not praise all operations, specially taking into account the nomenclature used in which he deemed confusing or possibly having RP jargon. In syntactic terms, for instance, P5 thought it was strange the way he managed to stop Bacon.js `fromPoll` loop, and it would be more intuitive to have some function to control the life cycle of `fromPoll` instead of a return statement with a Bacon.js End instance.

There were indecisive situations concerning the choice of the correct operators. P1, for instance, took at least three different approaches to get to his final version of task 4 (i.e., creating a timer to break the main logic). P3 felt operators like RxJS `takeWhile` and `takeUntil` were too close, so depending how one organize the logic and the experience level, both operators could apparently produce the same outcome. Closely, P3 also reported feeling unsure of the best way of initialize a stream giving the many creation operators that were available; P3 argued the problem should actually be due to his experience with the API, but he missed a step by step guidance of best operator choice. The following P3's analogy illustrate his dilemma:

"For me, many paths led to Rome, but I didn't know if the paths that led to Rome really led to Rome or if they only seemed to lead to Rome, but I was going to Palestine, I don't know." (P3)

Besides reporting problems about choosing the correct flatMap variation, P10 demonstrated insecurity whether the RxJS `interval` was the correct decision to create both an interval and a counter during the tasks as he personally felt there were probably clearer ways of expres-

[37] <https://developer.apple.com/documentation/combine/publisher/retry(_:)>.

sing those two different things. In terms of clarity, P13 did not understand how to properly mix RxJS Observable with Promises; in this sense, P13 complained about the constant usage of operators like `concatAll` or `mergeAll` every time there was a Promise, but, according to his understanding, it did not make sense as those functions seemed to be more linked with windowing. Moreover, P13 described moments where he considered to create his own operators as he did not find the proper one, but it was not so straight forward.

A very praise aspect was the composition of operators. Interviewees emphasized the easiness, legibility, readability, and linearity of the codes. P1, in particular, commented the way the operators are structured (i.e., in terms of a pipeline) makes the code easier to understand and read as one can visualize each step on the way; however, one has to first understand how each needed operator works as warned by P1. Still, there were debatable opinions regarding the pipeline syntax. P8, who had previous experience with RxJS, preferred the way Bacon.js allows function composition through method chaining, arguing the code becomes more usable and is one less syntactical element polluting his code (i.e., the RxJS `.pipe` method); yet, P8 understood the reason why RxJS decided to adopt the current pipeline syntax a few years ago. P13, contrarily, liked RxJS way of passing standalone functions, praising the components isolation and less chaining.

Learning and Understanding Aspects. Some participants provided insightful accounts about their progress and initial learning and understanding. P1 described he started disliking Bacon.js and felt upset for having to do the tasks with the API. However, as detailed by P1, the introduction was reasonable by taking a look at the documentation and the API turned out to be similar to Rx, which he had previous contact, and other RP libraries (e.g., Apple Combine); by the end, after acquiring more knowledge about how the API worked, he described the API use as amusing. The same gap until a better understanding of the API usage was recorded by P10, who recounted taking longer time during the first tasks, but starting making better progress after changing his way of thinking by putting the datastream concept upfront. Another key factor, for P10, to get a better understanding of the API was to make some parallel (i.e., function composition, syntax, operators, etc.) with another library he knew called Ramda[38].

P13's account considered the learning of the first concepts as the greatest obstacle, which he managed to understand in a little more than one hour by doing the rx-fruits game (discussed below) and reading the overview provided in the RxJS documentation pages; however, as also

[38] a functional programming, JavaScript library which is available at <https://ramdajs.com/>.

pointed by him, the overall use was simple. For P8, the experience with RxJS became an obstacle for using Bacon.js during the task 1; as stated by P8, he knew how to approach the problem from a RxJS point of view, but his bias toward RxJS function nomenclature made him taking a while to find the right functionalities in Bacon.js. Participant P8, in fact, stated, after taking a good look at Bacon.js' documentation, he felt that the API offered a more complete and practical set of tools compared to RxJS.

Among RxJS users, P3 seemed not to have grasped the basic concepts as well as the reactive terminology. In fact, P3 ensured to emphasize he spent five hours during the tasks and did not have many facilities with RxJS. To exacerbate his frustration, P3 did not like the Stack Overflow restriction during the tasks. According to him, he understood more or less the concepts, but he did not understand why using an `of` operator for instance or the structure of the RxJS `Observable`. Thereby, P3 described many situations of copying and pasting examples, trying to escape to its comfort zone by inserting raw JavaScript in the middle of the pipelines, and even forcing to fit a new `Observable` instance within the logic as workarounds. Another source of confusion for P3 was the reactive term as he was not sure he was reacting to anything. Delving into P3's narration, the source of all confusion seemed to be the fact a great majority of RxJS initial examples focus on UI reactions[39], while most of the tasks asked for reactions to number sequences, HTTP requests, etc. In this sense, P3 naively considered the task context as a simulation of real reaction.

P5 experience with Bacon.js did not convince him of the advantages of using a tool like Bacon.js. As put by P5, although describing some nice situations with the API, he kept asking why he would use the library as he did not feel the need for it. P5, indeed, detailed he was able to code with API, but the problem lied in situating himself while coding.

Visual Diagrams. Within this section, we approached the Marble Diagrams, highly promoted in the ReactiveX community, aiming to understand, from the participants' point of view, their practical usability. All participants agreed marble diagrams are a reasonably helpful mechanism to understand RP. Some praised for their existence, affirming they were essential to learn a few operators. P1, for instance, emphasized he is a visual learner and the diagrams are very practical when he does not know or have not used an operator. P3 praised the RxJS documentation for providing the diagrams: *"...this was a really cool thing of the documentation. In all documented operators there was a marble diagram that explained more or less, abstractly, how it worked,*

[39] This can be seen by taking a look at the overview page: <https://rxjs.dev/guide/overview>. Last visited at Oct. 10, 2024.

and that I liked." (P3)

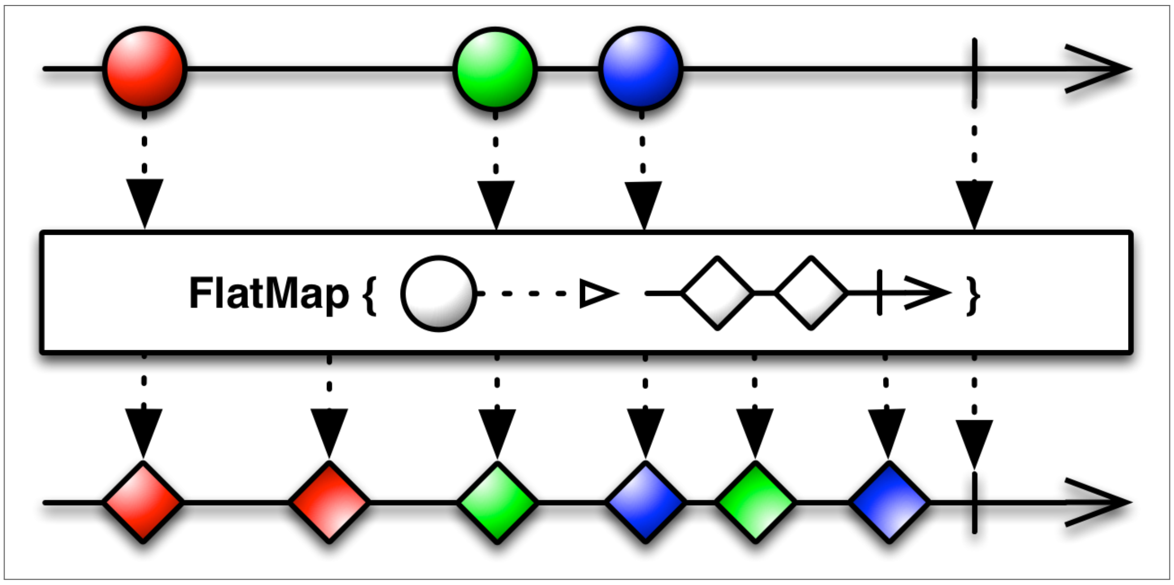
Despite the recognition of the diagram's helpfulness, two Bacon.js participants reported there is no such a visual aid in the Bacon.js documentation indeed, but it would make an important addition. When asked about if marble diagrams were essential for RP understanding, five disagreed arguing they help a lot and it is something that works; nonetheless, they do not replace textual documentation like function descriptions or usage examples. In fact, according to P1, the diagrams only offer an initial understanding and intuition. P10 followed the same reasoning of P1 explaining they are good to get an initial sense and change the way of thinking, but it is difficult to utilize them as the application become more complex: *"Well, I think that in the beginning they help, especially in changing the way of thinking [...] But I felt that when I was going to things that were a little more complex, and really more situational, like, doing the fetch [operation], taking the result of that fetch and to do the pipe of operations. I had difficulty taking this diagram and placing it in these moments. So, I think that for basic examples it helped a lot, but for more complex examples I couldn't visualize it."* (P10)

P3 emphasized the quality of the diagram, affirming all depends on whether it is well done as they are often oversimplified. To prove his point, he presented the `flatMap`, also known as `mergeMap`, diagram retrieved from the ReactiveX website (<https://reactivex.io/documentation/operators/flatmap.html>) and also reproduced here in Figure 17; in fact, without the textual support from the documentation, it is hard to tell that the operation depicted in the diagram is doing two things: mapping the stream emission into streams (rectangle in the diagram), followed by flattening the streams into a final stream (last part/timeline of the marble). Alternatively, P3 pointed to a diagram he came upon awhile ago while reading about JavaScript Array `flatMap` in an online article (<https://dev.to/charlottebrf_99/visualising-documentation-javascript-array-flatmap-1pcj>). According to him, the diagram is clearer as it dissect the two operations happening in a `flatMap`: a map followed by a `flat`.

When questioned about alternative ways of visually representing RP, most the participants did not provide much feedback. P10 commented an animated diagram like a GIF (Graphic Interchange Format) could perhaps be more useful, but it should be accompanied by more concrete and specific examples. Finally, when asked about the stream representation as a conveyor belt, sometimes used by diverse sources, as an alternative to be included in the API documentation, P8 replied the representation is a ludic, nice tool, but it should not directly be included in the API documentations.

Learning game. At the end of the training class, we rapidly presented the rx-fruits game, and,

Figure 17 – Marble diagram for the FlatMap operator taken from the ReactiveX official website.



Source: REACTIVEX (2024)

despite not being a required task, we suggested as an optional homework, so they could have one more (possibly amusing) source of learning. In this way, we asked the users if they had finished or at least tried the tool and how they evaluate it for the RP understanding. From the six interviewees, three said they completed the game, two played partially, and one admitted his only contact was what was shown in the end of the course. One of the participants (P5) who did not complete the game said since it did not targeted Bacon.js and there was no certainty that the game could actually help during the tasks, he decided not to spend too much time on it. Another participant (P10) only completed to the 10th level, considering it was probably enough to do the tasks; in reality, he was able to really complete 60% of the tasks.

All participants confirmed the game is a very nice and helpful tool for RP understanding. P1 agreed it helps to make clearer how the operators work, while P13 commented it helps to remember the library. P8 praised its ludic aspect, arguing if he had to encourage someone to learn about RP and RxJS, he would probably recommend the game. P5, despite having few moments with the game, enjoyed the game more than the Flexbox Froggy[40].

In terms of game experience, the participants provided some insightful accounts. At least three participants reported a not so pleasant start, in special at the first two exercises. According to P3, the first two exercises were a little difficult as he did not know exactly what was happening. Particularly, this seems to be caused by many already done parts (P10), demanding little initial efforts, and some not clear instructions (P5). However, as pointed by P3 and P10,

[40] a game to learn CSS flexbox which is available at <https://flexboxfroggy.com/>.

as the game progresses, it becomes more dynamic and interesting, with helpful feedback in error cases. For those with some level of experience, it is important to consider if it is in fact beneficial, as it can offer a too simple practice as observed by P1; nonetheless, P1 generally liked the game and recommended. Overall, it is our vision that, in spite of the previous points, the general experience of the game can be summarized as detailed by P13: *"...it is very easy to follow. It's a good introduction, although there isn't so much complex [advanced] things, but it's good to remember the library."* (P13)

A valuable feedback was given by P8 regarding possible improvements of the game. According to P8, bringing more real context, like consuming an (RESTful) API instead of fruits, would make a better connection between the ludic and real cases: *"What would be cool is to have some parallel of some code a little more, let's say, real ... more useful with that [...] Here we are doing it with fruit, but if I have an [RESTful] API here that serves a given thing, you can use a similar structure to achieve such a [learning] goal and then you can make a connection between what you saw in the game with something more real."* (P8)

RP Future Usage. As one the final interview sections, we interrogated the participants whether they consider to use RP in future projects. With the exception of P3, all interviewees either confirmed they intend to use RP in future projects or said they already somewhat used it, even if it is not the RP library they have explored during the study. P1 is starting to use RP in his daily work; besides, P1 had recent experience with SwiftUI, which utilizes the Apple reactive framework Combine, and had very basic experience with RxSwift. P8 has already done projects utilizing RxJS in his work, and, from time to time, P8 has to do some maintenance or improvements on those codes. P5, in addition to working with functional programming concepts daily, has recently worked with Vue.js, a front-end framework that includes reactive ideas, and had previous experiences with Spark streaming, a streaming tool built on top of Apache Spark. P10 felt stimulated to continue to use RP in professional projects, specially in the context of data stream processing (AKIDAU; CHERNYAK; LAX, 2018). P3, unfortunately, did not show signs of a great experience, arguing that he felt very tied in the style and it was very different from that he was used to; in this sense, P3 disregarded the chances of using RP in bigger projects and affirmed he would probably forget he used this at some point. In fact, by inspecting more closely, all P3's solutions presented some errors or aspects that prevented them to be considered complete (Section 4.2.2); it is also an interesting case as P3 fully completed the rx-fruits game.

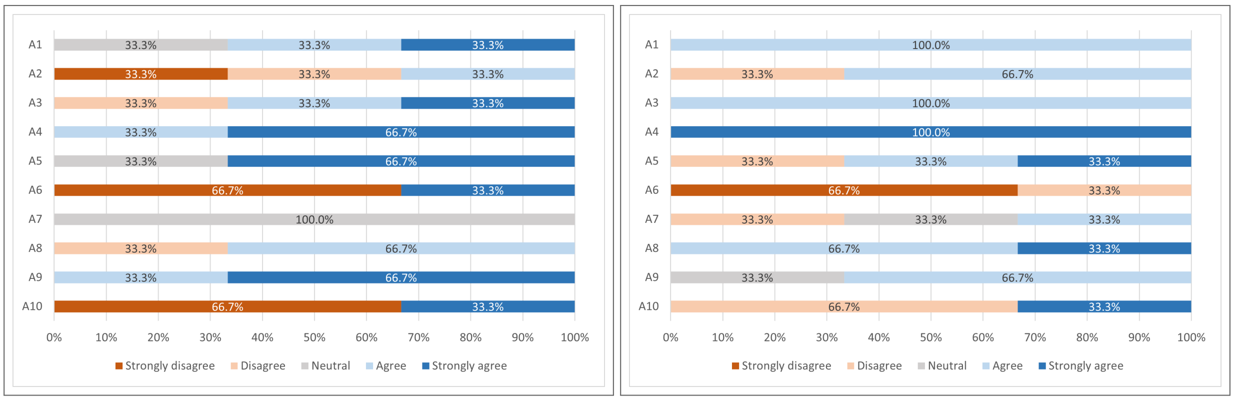
In support of their idea of using RP in future projects, the majority of the interviewees recognized the value of using the APIs and RP style. P5, for example, declared that RP is a powerful concept and it is basically essential for nowadays front-end development. P10 sees more value in the RP production and thinking at the moment, specially now that he understands more about it and how to use it rather than only having heard of. Surprisingly, even P3 admitted that if one day he becomes very good at RxJS, it would be a very interesting tool. However, possibly the best account came from P8 who has been working with RxJS for a while. According to P8, he was introduced to RxJS due to his job, and his first impression was of a complicate and horrible tool. It was only after some time that P8 started to recognize the value added to the projects, affirming that in fact, the tool helped a lot, and, after getting used the API, the codes became more simple and cohesive; as he stated: *"There are things that are much easier to do with this [RP]. I have faced callback hell in the past [...] this is much more practical to do."* (P8)

Given the recurrent experience with RxJS and, in the study, Bacon.js, we also questioned P8 about touching a code after some while. By asking that, we aimed to analyze P8's perception of knowledge retention, an important aspect of usability (NIELSEN, 1994), provided by the code produced with the API. As stated by P8, a code produced with a tool like RxJS still makes sense after some while, specially knowing it is going to have mostly code produced with that type of style or tool; in this way, the code is more consistent and easier to carry out maintenance. P8 also held his opinion for Bacon.js code, confirming that his codes, open during the interview, was still clear.

4.2.4.2 Interview Assertions

We complemented the interview (Section 4.1.3.4) with close-ended questions organized as assertions, which were answered trough a 5-point Likert-based scale. Figure 18 summarizes the answers of the interview participants according to the API used; labels A1-A10 refer to the assertions of Section 4.1.3.4. The answers were diverse amongst the APIs; A4, A6, and A8 were the ones that seemed closest. Interestingly, there was almost no case we can say it was completely opposite (i.e. with contrary tendencies). The closest exception was the assertion A2, in which two Bacon.js users either disagree or strongly disagree and only one agreed; on the contrary, two RxJS agreed on the same assertion, whereas one of them disagreed. In spite their close profile (Section 4.1.3.4), the Bacon.js interviewees did not show a complete agreement for the majority of the assertions. The RxJS participants, on the contrary, offered

Figure 18 – Answers to the Likert-based scale items of the interview.



(a) Bacon.js (b) RxJS

Source: Elaborated by the author (2024)

very close answers, with only minor disagreements.

A1. The positive aspect of the number of operators offered by the used API had all three Bacon.js participants given different answers. P1 (neutral) insisted that it is hard to evaluate that due to the total of tasks and only a few were sufficient to accomplish the tasks, a possible positive aspect according to him. In reality, P1 believes: *"...quantity doesn't mean much."* (P1 - A1 Neutral)

Closely, P5 agreed (but tending to Neutral) and commented that the number of operators was adequate for the tasks, but the real issue was how they were being taught in the documentation. P8 (strongly agree), in turn, affirmed that the quantity of operators offered by the API was enough and not inflated, otherwise, it could be difficult to find them.

RxJS users showed a full agreement for A1. The following quotation summarizes why interviewee P10 only agreed with the assertion which is probably the same view held by the other RxJS interviewees: *"I agree. I don't completely [strongly] agree, because I think that when you have such a large number of operators, sometimes you end up losing effectiveness and end up having those naming problems [...] I agree that there are many options, but I think that sometimes these options can have problems as well."* (P10 - A1 Agree)

A2. A2 (the facility to find the proper operator) had most of the Bacon.js answers in the disagreement direction. P1 (disagree), for instance, resorted to a lot of "Ctrl + F" [41] to carry manual searches and that proved to be very time consuming; to make matters worse, P1 had to trust their own naming guess instinct which was not always right. Similarly, P5 (strongly disagree) also found the operator searching problematic, blaming the documentation to be

[41] A keyboard shortcut for textual search.

unhelpful, not only to find the proper operator but also how to use it. P8 (agree) was in the opposite direction, probably due to previous experience with RxJS; however, P8 recognized that it would depend on the user level, and one-page documentation can be helpful but also disorganized: *"RxJS [documentation] is everything... one little thing inside another but that makes it difficult if you don't know what you're looking for. In the case of Bacon, you have a whole page and you scroll through it and you end up finding what you want there. So I think that if you are already an API user and want to find a specific thing that you already have a good idea of what it is, documentation like RxJS where things are more hierarchical is easier for you to find. But if you fell into the API and never used it and you want to learn something that you don't even know what it is yet, the way bacon shows things makes things easier but then it makes it difficult in the other case because one page is just kinda messy."* (P8 - A2 Agree)

The majority of the RxJS users agreed with A2. P3 (agree) declared the operators were actually well-named, and it was easy to find the proper one for the task; in terms of usability, however, P3 considered them confusing. P10 (disagree), conversely, went into the opposite direction, arguing that it was difficult to locate the proper operation based on the expected name, requiring a lot of manual search in the documentation: *"...because I knew what I wanted to do [...] but I didn't know exactly the name, and the documentation didn't help me much in that sense. I had to go to each of the methods and read its description and see if it was really what I wanted to do, because I felt like I wasn't able to, either through the name of the method or through the documentation, quickly find the method that I wanted."* (P10 - A2 Disagree)

A3. Most Bacon.js agreed with A3 and considered the functionalities not very explanatory. P1 (agree) testified that some function names were actually clear (e.g., `fromPoll`), but, in general, other names were not so clear, mostly due to the documentation and its lack of examples. P5 (strongly agree) declared that the names were actually fair, and, despite some of operators have lead to errors in some cases, it was due to his inexperience and, especially, the poor documentation: *"...the names are quite plausible. The documentation should tell me which is which, right? That was the problem, like, I thought the documentation was terrible. So I would say I completely agree because of the documentation and I only agree because of the names [...] If I open some random function, there is a real chance that it won't have any description [...]* `isRawPattern`, *description none, parameters* `Pattern<any>` *[...] With a documentation like that, I wouldn't need it. I could have written nothing there, and that would* *have done the same thing."* (P5 - A3 Strongly agree)

P8 (disagree) praised the code legibility, arguing that the code makes sense even not consulting the documentation, but all depends on the viewer's experience as the code for the inexperienced ones can be a bit strange. Contrary to P1 and P5 argument, P8 considered the operators well covered by the documentation, but they could have more examples (reason why he did not strongly disagreed).

All three RxJS participants agreed with A3. Interestingly, everyone's feedback targeted the lack of examples. As pointed by P3 and P10, there was some examples, but they wanted more in the documentation.

A4. There was a complete agreement among Bacon.js interviewees towards the lack of examples. P1 (Strongly agree) reported that there was some basic examples, but it could be better if there was both more complex examples and examples for every operator (with perhaps some visual diagrams). P8 was the only one that did not fully agree, contending that it could be worse.

Regarding the answers for RxJS, everyone strongly agreed. P13 observed that the documentation use a lot of marble diagrams and, sometimes, some simple examples, but it could have more use cases.

A5. Given its proximity, A5 (lack of tutorials) had a close result to A4 among Bacon.js users. While P1 (strongly agree) reported that the text provided in the documentation is too initial, P5 (strongly agree) stated that he was not satisfied by the provided tutorial. Instead, a better alternative would be to provide more concrete examples with context and varying situations (P1).

In the same vein, RxJS interviewees mostly agreed with A5 with only one user disagreeing. P3 (agree) detailed that there is a basic tutorial in RxJS, but one cannot go far with it. P10, the only one who disagreed, stated the tasks were not complex and what was offered by the documentation, along with reading class material and examples (external resources), should be enough. Also, both P3 and P13 asked if rx-fruits (Section 4.1.3.4) could be considered in this assertion, i.e., an outside material not present in the RxJS documentation as well.

A6. Most of the Bacon.js interviewees did not need to look at the source code implementation, presenting mainly a complete disagreement. The only exception was the participant P5 (strongly agree), who emphasized that the source code was more helpful than the documentation. While the documentation offered summaries, they were not enough, forcing P5 to look into the source code many times.

The results for RxJS were not different, showing almost a complete disagreement as well. P3 (disagree) expressed a slight desire to look at the implementation in moments of errors, but this desire was accompanied by hesitation as P3 recognized that if there was an error in code, there is a great chance it was due to something done wrongly or misunderstood: *"...there were times when errors would occur, and I wanted to go into the source code to understand what the error was, but... Then at the same time something appeared saying "Ah, if you're having to go into the source code it's because you got a wrong thing". So go there, come back and see what you're doing."* (P3 - A6 Disagree)

P3 concluded that the desire of going to the source code would probably be circumvented by gaining more experience with the API. P10 (strongly disagree) stated that, in spite of the lack of examples, the concepts were clear, thus there was no need to going to the source code.

A7. All Bacon.js interviewees were neutral concerning A7 (lack of support from community). Most of them (P1 and P8) commented that they did not go after the community, considering that they were restricted to the code and documentation and not to look into the Stack Overflow (or ChatGPT[42] usage). Even after insisting if they have seen any alternative channel displayed in the web documentation like Discord[43] or mailing list, they did not notice or go after it. P5, however, raised an interesting point, saying that the quality of the documentation did not seem to indicate a good support of the community. Nonetheless, P5 was not sure about the level of openness concerning community participation: *"I would say, just in case, neutral. If the community has the freedom to get there and make a request on GitHub to improve something, then I thought it was somewhat insufficient. But it's difficult for me, you know, in that sense. So it would be like a neutral [tending] to agree, right? I agree because maybe the community, as I said, the community could enrich it [documentation] better."* (P5 - A7 Neutral)

RxJS users, conversely, all expressed different answers. P3, for instance, was neutral since they were not allowed to using Stack Overflow (i.e., the main source of community contribution according to P3's viewpoint). Likewise, P13 did not go after it (disagree). P10 (agree) offered a different answer, declaring that there could have other channels, apparently not present in the web documentation: *"I agree because although there was the documentation, I said before that perhaps there was a lack of a forum, perhaps a discussion part of the community itself there in the documentation that would be a little more active and effective as well. [...] I'm*

[42] <https://chatgpt.com>.
[43] <https://discord.com/>.

not going to completely agree because I think there is a level of discussion about this [in some place], but I agree that there should be more." (P10 - A7 Agree)

A8. A8 (different way of thinking) displayed an inclination toward the positive side for Bacon.js, demonstrating the API/paradigm forced the participant to think a little differently. All three interviewees reported previous experience with functional programming and P8 had previous experience with RxJS, factor that seems to have helped their production. P1 previous experience with functional programming influenced the disagreement with A8; in fact, P1 liked that way of structuring the code, with more succinct and manageable steps (readable code): *"I think it forces you to think in a more compositional way, right, but I was already relatively used to it, so I didn't work as much, but I had already worked. Regardless, even if I'm working in a more imperative way, I try to divide things very small, like this, to ensure readability. So, for me, I think it didn't affect me that much. [...] even if you're not using reactive programming, I think I'm very used to, instead of a* `for`*, using a* `map`*, or using functional programming, like, methods linked to functional programming, like a* `filter` *and such, which is very related. So, for me, I would say I disagree."* (P1 - A8 Disagree)

P5 (agree) and P8 (agree), despite their experience, recognized that, indeed, the API forced them to think in a different way; both of them cited it was like to program in Haskell. It changed the way P5 thinks naturally as it was not a common way of structuring code, but that did not actually represent something bad: *"[...] because of the habit with functional programming [...] seeing some things as a certain chain of events is OK for me but, in any case, nevertheless, it still broke the ways in which I think naturally. And this matter of looking at things like stream composition is not that common. [...] Overall, like, it forced me to think differently, but, [...] all of that isn't necessarily a bad thing."* (A8 - Agree, P5)

Most RxJS answers were focused on the agreement side as well. P3 (agree) believed that the use of the pipe operator was an enhanced version for the "then" method of the "Promise" object; in this sense, P3 felt trapped with syntactical constraints, although somewhat familiar, forcing to rethink somethings: *"I partially agree. Because, as I said, what I used most was* `pipe`*, and* `pipe` *improved just a little the use of several* `.then` *[...] on the one hand I had to stick to a syntax, so I had to rethink my things. And for another, they were a little familiar."* (A8 - Agree - P3)

P10 (strongly agree) end up being more effect in solving the tasks after starting thinking differently. Contrarily, P13 only agreed with the assertion, but remarked that it was differently from day-to-day practice; by questioning about previous functional experience to investigate if it could have any influence with the answer, P13 took a functional programming course with Haskell, which probably was recently (Table 19).

A9. Bacon.js respondents were almost unanimous regarding the lack of good practices; P8 (strongly agree) detailed there is not anything related to it offered by the API, but it is not something so severe (possibly due to his previous experience with RxJS). Nevertheless, this contradicts a little with P5 (agree) account, who worryingly claimed: *"I agree, because I couldn't say what the best practice was. I don't know to this day, you know? I don't know if I made reasonably clean code or not. [...] The code is reasonably clean because it is small. But I don't know how it would scale."* (P5 - A9 Agree)

A9 only had one neutral answer for RxJS, with two respondents agreeing. P3 (neutral) confidently answered that this thought did not occur. Following closely, P13 (agree) considered not having missed so much good practices, but it certainly could help, especially by providing some examples of standardization. P10 (agree) did not feel comfortable with the tasks' solutions, with perhaps some instances of probable workarounds. P10 confidently completed: *"I wouldn't feel comfortable, for example, putting these codes I made into production, because I don't know if there would be any case, any exception that could break them or that could generate a problem in the future."* (P10 - A9 Agree)

A10. The majority of the answers indicate that Bacon.js interviewees had no problem distinguishing what was in fact reactive. P5 (strongly agree) was the only one who disagreed with the assertion. According to P5 feedback, though, the uncertainty was very close to the way the code was being organized, without necessarily being sure that was the correct way or was following the best approach.

A10 had mainly disagree votes in RxJS, with both P10 and P13 disagreeing with the assertion. P10 (disagree) considered the beginning difficult; nonetheless, after expending some time with the tasks and changing the way of thinking about the solutions, P10 reported starting to understanding (reactive code) in a better way. P3, however, was not sure if the built solution was in fact reactive, arguing: *"Is what I'm doing here really reactive? Just using RxJS am I already reactive?"* (P3 - A10 Strongly agree)

4.3 DISCUSSION

4.3.1 RQ3: To what extent are RP APIs usable, and what aspects are most affected?

By combining the appliance of metrics along with a usability study with users, we could obtain results from different perspectives. From the metrics' point of view (Section 4.2.1), which mostly evaluate structurally the APIs, both APIs had a very high usability level, either individually or combined. That means that in fact there were areas where API designers have direct good efforts to deliver very usable APIs. Those results also collaborate with the observations pointed by Venigalla and Chimalakonda (2021), in which those repositories with the higher number of stars and forks tend to have a higher AMS score. In this case, RxJS scored higher and had the greatest number of stars (>30,000) and forks (≈3,000). However, this does not mean that there is no places where the API could not be improved. The ADI metric, the only one less linked with the syntactical aspect, demonstrates an area that the APIs should improve, specially Bacon.js.; in fact, popular APIs are expected to have better and extensive documentation with lots of code comments (LIMA; HORA, 2020).

Other factors can also impact usability beyond structure (ROBILLARD, 2009), and the results were complemented with a usability study involving users. According to the data detailed in Section 4.2.3, the majority of the dimensions indicate that the APIs present a moderate level of usability. That is in fact what is shown by the average of the average dimension (depicted in Figure 16), displaying a value of 3.07. While reusability (high-level) and abstraction (moderate) were the highest dimensions, understandability (moderate) and, specially, expressiveness (moderate), were the ones with the lowest scores (i.e., most affected ones) and should be focus of RP API designers to increase the overall usability.

The individual questionnaire results (Figure 16) for the APIs demonstrated a very close, moderate usability, with Bacon.js (3.15) scoring slightly higher than RxJS (2.99). The same pattern was also observed for task completeness (Table 17) and time spent in the tasks (Table 18). Nonetheless, RxJS participants seemed more satisfied than Bacon.js ones (Section 4.2.3.2). We hypothesize that this discrepancy of results and perceptions was due to the differences in size of API users among the groups and level of experience; the Bacon.js participants who decided to participate and continue in the study (Table 19) demonstrated an elevated level of programming and OO experience. We believe the results would have been more balanced among the APIs if the study had counted on more willing volunteers, specially for Bacon.js (in which case, it would be more aligned with the metrics' results). Therefore, it is hard to point which one is better, and it is outside the scope of the study.

In summary, by reviewing **RQ3**, we can come to the conclusion that RP APIs present a very high level of usability from a structural viewpoint. However, in practice, this usability level did not translate into the same perceived level from the users' point of view: a moderate one. A good aspect that the metrics revealed, which matched many observations we complemented through code inspection and interviews for instance, was the documentation problem. Bad documentation are an obstacle for effectiveness (PICCIONI; FURIA; MEYER, 2013), and good ones should contain code snippets and tutorials (ROBILLARD, 2009). It was recurrent the problems with documentation (most cited category during the interviews) involving supporting materials, descriptions, lack of examples, and bad tutorials. It is paramount that designers direct more efforts in those areas, not only with more words (mostly considered by the metric ADI), but with quality materials in the documentation, which could not be verified only by the application of the metrics. This is particularly important, given that the RP APIs approach a different style which is probably difficult to be self-documenting by itself and is in fact hard to learn and master (SALVANESCHI et al., 2017). We could synthesize some of the observed findings and relationships noted throughout the sections:

- **Flexibility versus Complexity:** While diverse operator sets enable multiple solutions to a problem, they also intensify the learning curve and create inconsistency, especially with undocumented behaviors.

- **Documentation gaps:** Poorly documented operators exacerbate the cognitive load, making APIs less accessible and reliable. These issues repeat across interview themes and participant code evaluations.

- **Confidence and Maintainability:** Both the observed code patterns and participant feedback highlight a lack of confidence in long-term code maintainability due to incomplete guidance (e.g., when things does not work properly which is particularly related to error handling, a topic that users demonstrated problems).

- **API maturity:** Bacon.js and RxJS present contrasting strengths and weaknesses, with RxJS displaying more signs of "maturing" through its usability and documentation improvements.

4.3.2 RQ3.1: How easily can developers learn and understand RP APIs?

Understandability. At least, four metrics impact that dimension: AMNOI, AMNCI, AMGI, ADI. An average of their value suggest a high level of usability. However, that contrast with the user perception. Indeed, understandability was the second lowest score (2.9) in the questionnaire, only above expressiveness, representing a moderate perception of usability. The low score was even more accentuated for RxJS (2.75), presenting a percentage difference of 10% when compared to Bacon.js. In interpretation terms, however, both APIs scored under the moderate range (Table 10), demonstrating that the difference was in reality minimal.

In general, participants felt a moderate feeling (2.63) about the easiness of understanding the API (U1); Bacon.js was actually a bit worse, presenting a low sentiment (2.5). U5 was the assertion that must contributed positively toward understandability; it is related to the participants' expectation regarding their produced solution and the participants were, on average, neutral (3.38) about their solution. Still, the score was almost high, demonstrating that there is a possibility that some developers may have agreed with the assertion. The perception for confusion and difficulty while using the APIs (U2), on the other hand, scored the lowest for both APIs (2.19), specially RxJS (1.88). Even though the mapping of domain context by the API primitives (U3) had a moderate perception (3.32), we considered a positive result as it scored higher than other items and it was the first time that many developers were experimenting with RP. U4 deals with the need for going after information that it was not clearly represented by the API, and the result indicates a moderate feeling (3) about this assertion. This expresses that the participants had possibly to go sometimes to the documentation to reason about their code or to find certain concepts or operators that they could not identify or understand only based on their naming or behavior description, for instance.

Learnability. Learnability is an important quality attribute of usability (MYERS; STYLOS, 2016) and, along with understandability, it is linked to the learning of an API (LÓPEZ-FERNÁNDEZ et al., 2017). Three metrics are linked to this dimension: AMNCI, ADI, APXI. Together, those metrics showed an an excellent level of usability, with AMNCI and APXI presenting the greatest values. From those, the most problematic was ADI, particularly Bacon.js' score showing a moderate result. This result of Bacon.js was very close with the general perception of learnability expressed in the questionnaire. The learnability score (3.06), close to understandability (2.9), denoted a moderate perception of usability. The APIs scored individually almost the same, differing only by 0.04.

L1 was the assertion with the greatest score in both APIs, showing a general high-level perception (3.82) of usability regarding incremental learning. In terms of L2, the participants were more moderate concerning the need for learning a lot of concepts. L3 attracted our attention as a great majority of the respondents agreed they had to go deeper into the documentation to do the tasks, demonstrating a low level of usability. Overall, this implies that the learning process was incremental and involved reasonable amount of resources, but it was at the same time difficult, forcing participants to deeply inspect the documentation.

4.3.3 RQ3.2: To what extent do RP APIs contribute to code cleanliness, reliability, and abstraction from low-level complexities?

Abstraction. This is a very important dimension as it abstracts the handling of low-level complexity while helping to create clean and bug-free code (LÓPEZ-FERNÁNDEZ et al., 2017). The two metrics, AMNOI and APLCI, that most impact this dimension reveals a very high level of usability (0.9 on average or 4.6 converted to Likert-based scale), close to the maximum level. From the user point of view, abstraction was actually the second largest dimension (3.1). The difference was minimal (≈2%) between the APIs, with RxJS showing a better abstraction support (3.13). Nevertheless, those (questionnaire) scores support a moderate perception.

The answers were very close among the APIs, with A3 (no need for modifying the API) being the best evaluated; this means that developers basically did not have to change the APIs. The lowest answer was given to A2 (2.32), which also hold for both APIs and specially Bacon.js (2). This possibly shows that the participants found difficult to modeling their programs as composition of streams. A1 exhibited a moderate, neutral feeling (3.07) regarding the abstraction level for the tasks; the results comes unsurprisingly as most students were experiencing their first contact with the API and paradigm. However, Bacon.js was more impacted by A1 (2.75) than RxJS (3.38), depicting a difference of ≈20%. A4 also revealed a moderate perception (3.07) about not needing to understand implementation details, but with a lower API difference. This possibly indicates that some participants may have encountered a few situations where things were not so clear, and they may have considered sometimes going into the implementation. This could be seen as very liked with the result of A1, which some students may have believed that the abstraction level was neither appropriate nor inappropriate.

Expressiveness. According to the related metrics (i.e., AMNCI, AMGI, ADI), the APIs should present a high level of usability on average in spite of some problems observed in AMGI and

ADI. Nonetheless, expressiveness was the lowest dimension among the APIs based on the questionnaire result, expressing a moderate level of usability like most of other dimensions. The results for Bacon.js were in the same line of moderate feeling while RxJS, on the other hand, was evaluated at the upper limit of the low level of usability.

Looking into the assertions, E7 was the one that most cooperate for pulling the results down. The average for the dimension reported the very low value of 1.69, the lowest value in all dimensions' assertions. Both APIs' users showed unsatisfactory toward E7, specially RxJS (1.69), denoting a high probability of committing mistakes with the APIs. E4 (2) was another concerning assertion as it pointed that it was actually hard to find the proper, required operator; this affected much more RxJS participants with a very low evaluation (1.75). This situation could be linked with the proliferation of operators (common in those libraries (MOGK, 2015; SALVANESCHI et al., 2017), naming conventions (e.g., the use of jargon), or the intermediate scores of AMGI and ADI. E2 top ranked the assertions of expressiveness with a high level of usability, illustrating the readable quality of RP codes. The results for E1 (2.51) indicated that some respondents found difficult to translate the tasks' requirements into code using the API, specially in the Bacon.js case (2.5). Regarding E3, the participants were undecided about the quality of naming exposed by the APIs. The participants exhibited a more positive point of view about E5 (ease of using API operators), yet it was still within the neutral range. Apparently, a problematic area for both APIs was the distinction of similar features, in which both APIs scored the same, low value (i.e., 2.5).

4.3.4 RQ3.3: To what extent do RP APIs enhance code reuse and maintainability?

Reuse is one of reasons for the existence of APIs (STYLOS; MYERS, 2007; MYERS; STYLOS, 2016), and usable APIs stimulate code reuse (PICCIONI; FURIA; MEYER, 2013). In this way, we were expecting an elevated value for this dimension, specially given the high number of operations offered by the APIs (Section 4.1.1). Marvelously, the score for this dimension was the highest among all, including individually for the APIs, and the result is classified within the range of high-level usability. The average value of the metrics (AMNOI, AMGI, and APLCI) in that dimension demonstrates that same level of high usability, scoring, on average, 0.8 (i.e., 4.2 converted to a Likert-based scale).

In the questionnaire, there were five assertions under this dimension, and the first one (R1) scored the highest value (4.32) in the dimension and among the other dimensions' assertions. R1 deals with code length and the respondents gave a very high positive indication, agreeing that the codes produced by the APIs were tiny. This was specially true for tasks 1 and 3 according to the observations provided in Section 4.2.2.3. The R3 was also in line with the code observations, resulting in a weak level of usability (2.13). R3 relates to the capability offered by the API to solve the same problem by different paths, and it can become a liability if the alternatives are not actually complementary (experts tend to like choice more than novices) (PICCIONI; FURIA; MEYER, 2013). The remaining assertions, nonetheless, showed high-level results. R2 (4) concerns intermediary evaluation of progress, and it is close to the learnability assertion L1, which also scored high. The usability provided by R2 was even more apparent in Bacon.js, showing 0.50 points higher than RxJS (i.e., a difference of 12.5%) and receiving the classification of very high. R4 deals with the maintainability and evolvability of code, and the participants, from both APIs, evaluated positively this aspect (which they exercised during the solutions of the tasks). R5 (3.94) was even evaluated better than R4 and concerns the simple reuse of API features; this characteristic was well evaluated in both APIs, but Bacon.js respondents emphasized the aspect by considering very high.

4.4 IMPLICATIONS

This section presents the implications considering the findings of the Discussion section (4.3). We use this section structure to delineate a series of recommendations (Section 4.5) which take into account not only data presented in this chapter, but also data detailed in Chapter 3.

Most Concerning Issues. The evaluation of all dimensions revealed that RP APIs currently offer moderate usability in terms of understandability, learnability, abstraction, and expressiveness. The only exception was reusability, showing high usability. In other words, both APIs have a lot of space for improvements regarding four different dimensions. Within those dimensions, there were more concerning aspects as they demonstrated lower levels of usability; this was more visible in the expressiveness spectrum, which scored lowest for both APIs. From lowest to highest, the most concerning points were presented by: E7 (1.69), E4 (2), R3 (2.13), U2 (2.19), A2 (2.32), L3 (2.38), E6 (2.5), and E1 (2.57). From those, we believe that E4, E6, and R3 are very related to the offered operators by the APIs, so we dedicated an exclusive topic to this matter. The remaining, based on the data collected in the study, seems to be related to the learning and understanding process and the available resources provided to support the process.

All participants detailed taking some initial time to better understand the APIs, which was expected given it is not only an API but a different style (MOGK; SALVANESCHI; MEZINI, 2018); it also matched the observations from previous studies (SALVANESCHI et al., 2017) depicting that RP may take a time for some users, specially coming from other paradigms like OO. Experience with FP and other RP-based libraries as well as contact with other FP libraries helped many participants to get a better grasp of the ideas. Unfortunately, both APIs seem to assume equal level of FP background from the users, exploring concepts easily found in FP (e.g., purity, function composition, side-effects) but not dedicating simple sections or links to minimally explain those basic ideas. FP was actually pointed as a possible obstacle for RP (SALVANESCHI et al., 2017). The Rx team in fact recognized this need and created a mini series of (JS) exercises arguing that "it turns out that the key to learning Rx is training yourself to use functional programming to manipulate collections." However, it is only cited at the main page of the project[44] and not in the RxJS website/documentation. We believe that a better approach to basic concepts, including FP and the problems RP tries to solve, could have produced better impact for U2, A2 and even E1; after all, API learners like to understand some high-level details about the rationale and intents of the API in order to make better use of it (ROBILLARD, 2009).

Documentation is one of the most used resources during API learning (ROBILLARD, 2009) and its completeness and preciseness is fundamental (HENNING, 2009; PARNAS, 2010). Many of the participants reported accessing the documentation and reading the overviews. For some participants, the initial understanding was difficult but became simpler in the process. Many, however, did not had an enjoyable experience, considering themselves lost within the code and discouraged to use the API in the near future. A concerning point was that many had to delve into the documentation (L3) while facing poor resource description and presentation, specially for Bacon.js. Some Bacon.js users in fact considered that the documentation was too simplistic and messy, not really exposing the API. A common aspect among the APIs was the few or lack of examples. It should be unsurprising the importance of examples as a key learning asset (ROBILLARD, 2009; NASEHI et al., 2012): through examples, developers can better understand the libraries purposes and its usage protocols and context (MCLELLAN et al., 1998). Thus, it is no surprise the developers have found confusing and laborious to use the API (U2) and difficult to turn the tasks into code with the API (E1). In general, participants

[44] <https://reactivex.io/tutorials.html>.

classified many examples, including tutorials (e.g., in Bacon.js), as presenting low quality and very specific. RxJS users actually declared the API have both good and bad examples, but there is a lack of scenario variations. Many scenarios seemed to focus in UI, a factor that even confused a user about the reactive terminology (i.e., considering reactive only when there was UI interactions). Programmers tend to consider APIs easier when tasks follow the API examples (MCLELLAN et al., 1998), so diverse scenarios can impact their perception.

A complete documentation should not only include scenarios with the right behavior, but also when things go wrong (HENNING, 2009); this was also a vision shared by one the participants, and, as demonstrated by E7, the general consensus was that it was easy to make mistakes with the API. This fact indeed prompted many participants to doubt about the quality of their solution and declare not feeling comfortable to put them intro production. Two prominent areas were more susceptible to problems: error-handling (task 2) and state management (task 5). Those two areas were in reality shown previously as topics discussed in Q&A forums (Chapter 3). Aligned with that, we observed many problems within code that we named as a lack of best practices (Section 4.2.2.3). In task 5, for instance, we observed the used of global variables which could be avoided by better guidance on the topic and basic understand of FP state-handling. The interviews revealed that the participants missed good practices in the APIs and it could be very helpful, showing, for instance, how to better approach certain situations.

Operators. From what was observed in Section 4.2.2.3, both APIs presented a close profile in terms of operators required. However, the difference of API usage shows that Bacon.js provides a lot of functions. In fact, only 19% of operators were required from its extensive API. Contrarily, users explored $\approx$30% of RxJS API. The observations of Section 4.2.2.3 and the answers for the R3 item of the questionnaire indicate that users felt they could approach the problems by using different paths, and this was very explicit in many Bacon.js codes. Moreover, Bacon.js extensive set of operators also showed more signs of problematic operators (Section 4.2.2.3).

According to the participants' feedback, the numbers of operators from both libraries did not seem to be an issue for them. While some argued that the quantity was sufficient for the tasks and it was hard to evaluate this matter given the quantity used, others considered more problematic how the operators were being taught in the documentation. In terms of nomenclature, the participants were a bit divided, showing some instances of naming and searching problems with operators. Still, we had participants complaining about the class of *flatMap* operators from both APIs, judging their distinction often difficult in the short and long term, and finding hard sometimes to discern between similar types of operators like the different variations of `take` (e.g., `take`, `takeWhile`, `takeUntil`, etc.). It was not clear for some users why using factory function like `of` rather than instantiating an `Observable`. Furthermore, the great majority of the participants (from both APIs) agreed about the presence of non-explanatory functions; most reported problems with documentation like poor and lack of descriptions and absence of good, more complex examples.

Despite the participants' belief that the API size was not an impediment, we collected reports that the many choices causing difficulties to fully understand the concepts and causing insecurities regarding the qualities of the solutions, specially among Bacon.js users. As also observed, there were accounts of many features with no description or examples, leading to a lot of guessing. RxJS, on the other hand, depicted more signs of maturing, although with complaints about lack of examples, specially when things does not work well. Additionally, the API has been constantly updating (specially since its version 4), which shows more efforts from the maintainers. For instance, operators like `average`, still displayed in the official Rx website as implemented by RxJS, is not part of RxJS API since version 5. Instead, users are encouraged to use either `reduce` or `scan`, a variation of the general FP combinator `fold`, to express the functionality. Unfortunately, we do not envision easy ways of expressing some operations (e.g., RxJS `bufferTime` or `bufferToggle`) using more general operators like `fold`, unless the designers supplied recipes for such implementations. Therefore, it is our belief that Bacon.js should reevaluate its API size, and better organize its features, in order to become more approachable to newcomers. RxJS, on the other hand, seems to be actually offering a more stable number of operators when compared to Bacon.js. However, it should include more examples and add more support for similar operators (e.g., `flatMap` class of operator) in order to avoid confusion. Both APIs should strive to find better ways to minimize the problem pointed by R3 item of questionnaire which deals with different ways to solve certain tasks. As noted by (PICCIONI; FURIA; MEYER, 2013), choice is preferable only for experienced users. In this way, RP APIs should evaluate how much complexity is avoided by keeping or removing some of their functionalities (OUSTERHOUT, 2018). Also, they could identify API usage patterns (ZHONG et al., 2009; SAIED et al., 2015), so users can be aware of operators that usually come together. Finally, removing operators may not be necessary as long as the API designers identify a few set of core features and make the users aware of only those; in this way, the effective complexity becomes the complexity of those frequently used operators (OUSTERHOUT, 2018).

An aspect that we missed from the participants' feedback was the RxJS operator decision tree[45], which none reported or apparently did not use it. The decision tree offers some precompiled options that the developer can select to help find the better operator for the task. This can be a helpful tool for RP APIs, specially those APIs with an elevated number of combinators. Besides, the tree also demonstrates that more tools could be used to aid in the operator choice and alleviate the user cognitive load. For instance, large language models (LLM) integrated with IDEs like Visual Studio (VS) Code seems to be a prominent venue (NAM et al., 2024).

Additional resources. Additional resources can become a valuable learning asset, and many API resort to diagrams and games. Diagrams can help users conceptualize structures and their code, forming mental models. Accurate mental models can help to use systems more effectively (KULESZA et al., 2012). Games, in turn, can enhance the understanding (LEUTENEGGER; EDGINGTON, 2007). During the study, more specifically in the interviews, we tried to collect the participants' opinions regarding the practical usability of visual diagrams like marble diagrams in RP and the rx-fruits, a game we have rapidly shown and pointed as a possible additional source of learning. There was a unanimous sentiment that marble diagrams are a helpful and practical tool, specially to learn operators. Bacon.js participants also agreed with usefulness view of the diagram, but they were disappointed that their API did not include much visual aid. The diagrams, however, were not considered an essential tool, but a complementary one. They indeed offer a good initial understanding but are limited to small parts, not helping so much as the program scale. The quality of the diagrams are also an essential aspect, and complex operations should not be oversimplified in a small, compact diagram. Another possibility for enhancement includes changing the diagrams to become more dynamic and animated, with more concrete and specific examples.

There was an agreement about the benefits for learning and understanding RP through the rx-fruits. According to participants' feedback, the tool eases the learning of operations and helps to memorize the API. However, the game seem to become more interesting and dynamic only after the initial lessons. For newcomers, the game can be a good resource, but, people with more experience, may find the game not so helpful as it involves simple practices. The game developers should incorporate more real contexts (e.g., web API requests instead of fruits) to make a better connection between the ludic and real.

[45] <https://rxjs.dev/operator-decision-tree>.

4.5 RECOMMENDATIONS

Based on what was discussed in this chapter and also considering the outcomes of Chapter 3, we elaborated a set of practical, suggestive recommendations (Table 21) that RP APIs should consider to possibly improve their overall usability. To facilitate, they are grouped according to the implications of Section 4.4.

Table 21 – Recommendations for API designers according to the points and observations presented throughout the research. (continue)

Topic	**Recommendation**
Most concerning issues	- Consider the differences in user background and dedicate some sections for basic concepts like the problems that RP can prevent and FP topics.
	- Pay attention to resource descriptions, and make sure they are complete and clear. Only describe features that are actually part of the API, otherwise they are only adding more complexity.
	- The API should ideally try to add all types of examples in addition to code snippets, like tutorials and complete applications. Users' suggestions indicate the inclusion of kick-start tutorials and different scenarios and applications for the examples. The quality (formatting) and up-to-date content should also be considered. The Learn RxJS portal[46] counts with a lot of examples that could easily be adapted for the APIs' documentations.
	- Include key sections to better guide developers to deal with difficult topics like error-handling and state management. Chapter 3 details a list of most discussed topics in Stack Overflow that could as well be considered by the APIs.

[46] <https://www.learnrxjs.io/>.

Table 21 – Recommendations for API designers according to the points and observations presented throughout the research. (continued)

Topic	Recommendation
	- Provide a "best practices" section showing possible situations in which code can be structured wrongly or can lead to error conditions; the section should also include clear instructions and explanation about how to overcome those situations.
Operators	- Identify the most used operators and list them as the primary focus for a newcomer; this way, the APIs can minimize the effective complexity (the users will need to be aware of only a few core features/operators) and any discouragements from learners, specially considering that the RP is hard to learn and master (SALVANESCHI et al., 2017; MOGK; SALVANESCHI; MEZINI, 2018). Alternatively, API designers could also consider providing only the most general operators, and let the community supply the most specialized ones (an approach taken by Most.js v.2[47]).
	- API designers should also think about the use of recipes for some operators; this is in fact an approach taken by the Most.js API in its v.1[48]. For instance, topics about implementation of missing or removed features[49] could also be included in such recipes, so a user can easily found them.

47 <https://github.com/mostjs-community>.
48 <https://github.com/cujojs/most/wiki/Recipes>.
49 E.g., <https://github.com/ReactiveX/rxjs/issues/1295> and <https://github.com/ReactiveX/rxjs/issues/1542>

Table 21 – Recommendations for API designers according to the points and observations presented throughout the research. (continued)

Topic	Recommendation
	- Review all features exposed in the API and documentation. We observed many features listed in both APIs, specially in the online web documentation, that had obscure role (i.e., the reason it is in the public API) and description. For example, many Bacon.js features were listed in the API without a proper description, discouraging and infuriating some users. Other features presented surprising, uncataloged behaviors like Bacon.js `fromPromise` (Section 4.2.2.3) or RxJS `concatMap` (Section 4.2.2.4), for instance.
	- Explain clearer about the heavy reliance on factory functions and how much complexity would take not to use them; in other words, describe the instantiation of the main abstractions. This fact was actually previously studied, showing that constructors were the primary technique programmers try to use to create objects and factories are more difficult than constructors (ELLIS; STYLOS; MYERS, 2007). Ben Lesh, the RxJS technical lead, holds a series of posts and one of them specifically targets this subject [50]. We believe that such resource could easily be integrated to the RxJS API documentation or, at least, links to the posts. A similar asset could also be included for Bacon.js.

[50] <https://benlesh.com/posts/learning-observable-by-building-observable/>.

Table 21 – Recommendations for API designers according to the points and observations presented throughout the research. (continued)

Topic	Recommendation
	- Offer more helpful information about the distinction of similar operators (e.g., variations of `take`), specially for the case of flatMap. One trouble factor is the lack of consistency among RP APIs. For instance, both Bacon.js and Kotlin Flow [51] use the same prefix for flatMap operators, while RxJS include different prefixes like `mergeMap`, `concatMap`, and `switchMap`; thus, we particularly find the naming used by Bacon.js for flattening clearer since it can better identify operators belonging to this type. Moreover, an account we presented reported problems to adapt to Bacon.js nomenclature conventions having previous experience with RxJS. We think that the APIs should strive to standardize the terminologies, after all transference is fundamental both within and across APIs (HENNING, 2009).
	- Include better and more complex operator examples. API designers should not only add small code snippets, but also links for more complete, step-by-step tutorials and applications that include the given operators. The Learn RxJS portal specifically adopts this strategy by including related recipes and additional resources.
	- Encourage more the use of decision tools like the operator decision tree. Also, RP APIs could look for alternative ways the exploration of LLM aligned with IDEs like VS Code. In RxJS community, there is already a VS Code extension[52] that could be used as a base project to construct such a decision tool.

[51] <https://kotlinlang.org/docs/flow.html#flattening-flows>.
[52] <https://github.com/dzhavat/rxjs-cheatsheet>.

Table 21 – Recommendations for API designers according to the points and observations presented throughout the research. (continued)

Topic	Recommendation
	- Demonstrate why operators like `tap` and `doAction` exist in the API and when to properly use them (e.g., avoid side-effects scattered throughout the pipeline).
Additional resources	- Invest more in diagrams, preferably with more dynamic content and concrete examples. APIs should also pay attention to the quality, splitting diagrams for complex operations. There a number of online tools could either be recommended by the APIs or used for creating better tools: ThinkRx, Rx Visualizer, and Rx ASCII Visualizer[53].
	- ASCII marble diagrams could be better explored as a mechanism to express code logic reasoning as shown in many parts of the Most.js v2 documentation[54] and the Egghead course RxJS Beyond the Basics: Operators in Depth[55].
	Games have been shown as a good learning resource, so the APIs could either recommend or explore the ideas of rx-fruits. Desirably, the game should strive to go beyond the ludic, adding more concrete situations (e.g., interaction with real RESTful APIs).

Source: Elaborated by the author (2024)

4.6 THREATS TO VALIDITY

Internal Validity. The participants of the study were students, and to make sure that they had a considerable level of programming experience, the study was conducted in a course where students are in their 3th or 4th semester. In fact, a great portion of the questionnaire

[53] Available at <https://thinkrx.io/>, <https://rxviz.com/>, and <https://ascii-marble-diagrams.surge.sh/>, respectively.
[54] <https://mostcore.readthedocs.io/en/latest/notation.html#timeline-notation>.
[55] <https://egghead.io/lessons/rxjs-resubscribe-to-an-observable-on-error-with-rxjs-retry>.

participants (Table 19) reported either having two to five years or more than five years of general experience; a close observation can be seen for OO experience as well. Some of those students were actually already working for companies, ensuring a level of experience outside the academic context. However, it is important to note that previous study (SALMAN; MISIRLI; JURISTO, 2015) observed no significant difference between professional and student developers. Besides, API usability is a problem that affects both novices and experts (STYLOS; MYERS, 2007). In any case, future studies should try applying similar studies with a broader range of professionals.

There was a chance of bias in the participants' selection given the closeness of many of them with the second author (students enrolled in his course). To minimize it, the conduction of the usability testing was carried out by the first author. All participants showed enough background to undertake the tasks (Section 4.2.3.1), both in terms of general and OO experience. The users also had heterogeneous background regarding FP, diminishing the impact of previous experience due to the RP connection with the paradigm (SALVANESCHI et al., 2017).

The number of group participants in usability studies is a very debatable matter, raging from three or five (TURNER; LEWIS; NIELSEN, 2006; VIRZI, 1992) until 25 (MACEFIELD, 2009). The used number can impact the discovery of some usability problems (ALROOBAEA; MAYHEW, 2014), but it also depends on other factors like the purpose of the study and expertise of the participants. Our final number of the participants unfortunately was unbalanced (Sections 4.2.2 and 4.2.3.1), and, due to ethics constraints, we could not force user participation; the number of participants even decreased during the course of the study. However, RxJS questionnaire participants were within a quantity enough to uncover minor and major problems (ALROOBAEA; MAYHEW, 2014), while the Bacon.js participants showed great experience (Table 19). In any case, there is a chance that some usability problems went unnoticed. We, therefore, recommend future replications of the current study with more users aiming to uncovering more usability problems. Additionally, other contexts, besides distributed applications, may as well be beneficial.

By not administering the tasks with every participant in person, we deliberately minimized the possible interaction effect that can happen in usability sessions between the participant and the facilitator (also called proctor) (PICCIONI; FURIA; MEYER, 2013). The same effect can also happen during interviews (PICCIONI; FURIA; MEYER, 2013). To minimize it, during the tasks' assignment, we informed the users we would only be available to answer questions related to the interpretation of the tasks. For the interviews, we also tried to stick as much as possible to the interview script, while providing a relaxed environment. The absence of an observing facilitator during the tasks may also have indirectly relieved any psychological stress like the presence of a person observing or time pressure that could impact the participant performance (JANNECK; DOGAN, 2013). Nonetheless, choosing not to observe the participants came at the cost of allowing plagiarism and exchange of information. To reduce the threat, we constantly reminded the participants of the study's importance and how important it was to deliver their own work (including no involvement of LLMs like ChatGPT), even if they were incomplete within the given time. In any case, we measured the level of plagiarism with the help of the online tool Dolos (MAERTENS et al., 2022), a plagiarism detection tool. Solutions for both APIs presented low level of plagiarism, with Bacon.js and RxJS having a mean average similarity of only 24.6% and 19.8%, respectively. Bacon.js actually presented instances of strong plagiarism in tasks 3 and 5, but it was due to the fact that the participant P8 offered two different solutions for both tasks (so, we reconsidered only one of the P8 solutions to measure plagiarism). An inspection revealed that the plagiarism mostly corresponds to packages' importation or the use of the common functions like JS `fetch`.

External Validity. The choice of the APIs using GH stars and forks was motivated by the belief that they possibly offer the best usability among RP APIs (VENIGALLA; CHIMALAKONDA, 2021). However, it is important to emphasize that the results may or may not generalize to other APIs. Other studies should be conducted to test if the level of usability observed can actually be generalized to other APIs as well. Moreover, we do not try to make a generalizable comparison between the APIs, as it is not in the scope of the present study and the sample of participants does not allow a significant, statistical decision. Consequently, we always try to present an aggregated view of the APIs, and the comparisons are only to describe the observed strong and weak points of each.

Construct Validity. We recurred to triangulation (JONSEN; JEHN, 2009) to reduce the subjectivity in our results and increase confidence. More specifically, we utilized both questionnaire (Section 4.2.3) and interviews (Section 4.1.3.4), intending to acquire a more profound understanding of the problem. Moreover, our questionnaire was based on the well-known cognitive dimensions, successfully applied in numerous usability studies (PICCIONI; FURIA; MEYER, 2013; LÓPEZ-FERNÁNDEZ et al., 2017). Furthermore, the categorization of the interviews was analyzed by both authors; any disagreement were solved trough discussion and agreement.

The time spent displayed in Section 4.2.2.2 relied on the participants informing that information which may be susceptible to a certain degree of deviation from the de facto spent time.

To minimize that problem, we insisted, previous to the start of the tasks, that the participants considered the tracking of that information as an important factor for their task submission. Furthermore, predicting some possible variations, we relied on the median statistic for the result analysis due to its resistance to outliers (ROUSSEEUW, 1990; DAS; IMON, 2014).

Finally, any qualitative study may be subjected to research bias. In other words, different researches may produce diverse interpretation and conclusions with the same set of data, or even within different contexts (DENZIN, 2019). We, consequently, made publicly available any generated material[56] to mitigate the threat. In this way, researchers can replicate and examine our analysis and data.

[56] <https://github.com/carloszimm/thesis>.

5 RELATED WORK

5.1 REACTIVE PROGRAMMING

Empirical Studies. Salvaneschi et al. (2017) executed an experiment to verify if RP improves software comprehension when compared to the traditional OO approach: the Observer pattern. The experiment was an extension of a previous study (SALVANESCHI et al., 2014) and involved a total of 127 participants. The study, composed of 10 tasks, focused on three factors: correctness of comprehension, time for program comprehension, and skill level dependence. The answers demonstrated that RP increases the correctness of comprehension while not impacting comprehension time. Regarding skill level, the results suggest that RP lowers the barrier for understanding reactive (event-driven) applications, even for less experienced developers. A quantitative analysis drew the following observations from developers' perspective: (*i*) developers with minimal RP exposure found RP easier to comprehend, (*ii*) following data and control flow is simpler in RP, (*iii*) RP conciseness improves comprehension, and (*iv*) conversion functions (signals and events) increases comprehension. Aspects perceived supporting RP included: reduced boilerplate and shorter code, better readability, automatic consistency of reactive values, declarative nature, ease of composition, and separation of concerns. Conversely, the learning curve (i.e., it is harder to master) along with its (increased) level of abstraction (i.e., it takes time to understand and codeflow is abstracted), and relation to functional programming (e.g., excessive number of operators) can be challenging. The authors highlighted the need for enhanced debugging tools to improve codeflow visualization (e.g., they cite their related work (SALVANESCHI; MEZINI, 2016)) and the simplification of RP APIs, especially by limiting the proliferation of specialized operators. In our work, we witnessed users struggling with basic understanding both though user-centered study and Q&A topics. Also, our mining study shows that in fact many RP APIs are providing many specialized operators and could in fact consider reshaping their interfaces.

Data Flow Languages. Salvaneschi (2016) examines data flow programming languages, widely used in areas like reactive programming, Big Data analytics, and real-time systems. These languages prioritize data dependencies over control flow, offering declarative, functional-style abstractions to simplify complex systems. Despite their popularity, evidence on the usability and effectiveness of data flow languages remains limited. Challenges include mastering subtle semantic variations (e.g., lazy vs. eager evaluation) and managing a vast number of specialized operators. The paper proposes research questions around comprehension, maintainability, and the cognitive impact of data flow paradigms, urging systematic investigation into their usability and design. In Chapter 3 we based part of our analysis in one of the research questions proposed in the paper: "Do data flow languages provide a 'simple enough' solution for the common case without excessive proliferation of overspecialised operators?" Our results have demonstrated that some reactive interfaces are in fact providing an excessive number of operators, many of them with little or no usage.

5.2 MINING SOFTWARE REPOSITORIES

The work by Rebouças et al. (2016) investigates early adoption of the Swift language which was developed as a replacement for Objective-C by Apple. Particularly, the authors were interested in uncovering the most common problems faced by Swift developers, if the developers were encountering problems with the optional type and error-handling. Like our work, they used the LDA algorithm to analyze 59,156 Swift-related questions from Stack Overflow, and complemented their study with semi-structured interviews. The paper highlights the need for improvements in Swift's tools, clearer error messages, and better guidance for new developers.

In the paper "What Do Concurrency Developers Ask About? A Large-scale Study Using Stack Overflow", Ahmed and Bagherzadeh (2018) analyze over 245,000 posts from Stack Overflow. The study identifies 27 concurrency topics grouped into eight categories, including concurrency models, programming paradigms, and debugging. Some findings in the study revealed that developers ask most about basic concepts, concurrency correctness, and multithreading; besides, thread safety is a popular topic but relatively easy, while database management systems are both challenging and less popular. Similar to our work, the paper uses LDA along with questions (title and body) and accepted answers. We also explore metrics to classify topics according to their popularity and difficulty, showing *Dependency Management*, *Introductory Questions*, and *iOS Development* as relevant topics.

5.3 API USABILITY

User-centered Studies with CDN. Variations of the CDN framework have become commonplace in usability studies. The work of Piccioni, Furia and Meyer (2013) introduces a design of a empirical study to evaluate API usability. The study combines two methodologies: a set of interview questions based on CDN and systematic observations based on usability tokens (e.g., "surprise" or "unexpected") to analyze participants' behavior. The research questions and the CDN questionnaire are based on four dimensions: understandability, abstraction level, reusability, and learnability. The study was used to analyze the API usability of the Eiffel library (a persistence library), and it involved 25 participants performing five tasks. The tasks sessions were recorded and the participants followed the "thinking aloud" protocol, allowing the researchers to capture real-time feedback; that feedback was then categorized using five tokens: "Surprise," "Choice," "Missed," "Incorrect," and "Unexpected." Afterwards, structured post-task interviews were conducted. The results evidenced close outcomes reported in similar studies like the problematic task of finding descriptive, non-ambiguous names for API components and creating relations between types that are easily discoverable. Additionally, the findings unveiled the need for accurate and complete documentation and consideration for the user expertise during the API design, balancing flexibility with simplicity (i.e., experienced users tend to like API choice more than novice ones). In our study, we could perceive many of the reported problems presented in the paper like documentation and the problem of providing too much choice. Also, we reused the tokens "surprise" and "choice" for the interview, aiming to obtain a more profound understanding of their experience since we decided not to apply the thinking aloud method.

López-Fernández et al. (2017) presented a new API targeting multimedia technologies like Real-Time multimedia Communications (RTC). The authors organized a usability study by adapting the CDN framework from Piccioni, Furia and Meyer (2013), organized as a 28-assertion questionnaire targeting five dimensions (i.e., understandability, abstraction, expressiveness, reusability, and learnability) which was answered through a 5-point Likert-based scale. Results showed the proposed API can be helpful in the learning process, and the creation and maintenance of applications. The study also revealed some API weaknesses: (*i*) the need for better documentation to support initial learning and (*ii*) understanding of real applications' requirements to include missing characteristics. Our study reused and adapt a great part of those questionnaire items, taking into consideration the RP context. Also, we opted to use the Likert-based scale to create a more inviting environment for the respondents. This is especially important given the voluntary nature of the study, so participant would possibly not take a long time to answer the questions and not feel overwhelmed as pointed in the work of Diprose et al. (2017).

Metrics. The study of Rama and Kak (2015) elaborated a set of structural metrics based on commonly held beliefs about API design structure that provide the best usability. Based on those beliefs, the authors build a set of nine usability issues corresponding to structural defects that difficult APIs may present. Examples of such issues include inconsistent parameter ordering (e.g., it can lead to programmer errors), overloaded methods returning different types of values (i.e., it can confuse users), and the API documentation quality. A total of eight metrics were then defined, with mathematical formulations, based on the structural issues. The metrics were validated by analyzing seven, popular Java projects. The proposed metrics can be used to guide API design, highlighting areas for improvement and providing a comparative factor among API versions. An actual implementation for the metrics are not supplied; however, given the detailed presentation of the metrics, it becomes easier to implement and adapt the metrics. In our study, therefore, we could easily translate most of the metrics to actual implementation. That implementation was made publicly available (only for TS so far), so it can be reused in other studies and enhanced with the help of the open-source community.

Venigalla and Chimalakonda (2021) evaluates the usability of APIs in game engines by analyzing 95 publicly available repositories on GitHub written in C++. Using the eight structural API usability metrics of (RAMA; KAK, 2015), the study finds that 25% of the the game engines present minimal API usability, while only 3% shows good usability; the APIs most struggled in two areas: method grouping (AMGI) and thread-safety documentation (ATSI). The analysis highlights the importance of consistent naming, clear documentation, and better semantic grouping to enhance usability. The authors propose future directions, including developing game engine-specific usability metrics and conducting surveys with developers to refine API design. The findings in our study showed common points with the paper like the share predominance of the metrics AMNCI and APXI. Another shared fact was the problems with documentation (ADI) and grouping of sematically similar function (AMGI). Finally, we also observed the tendency of repositories with more stars and forks providing the best usability level.

6 CONCLUSION

In this thesis, we conducted an exploratory research to unveil the usability of RP APIs hoping to understand their current usability level and reveal the main problems and areas of improvements. This chapter summarizes our work (Section 6.1) and the contributions (Section 6.2). Furthermore, we suggest future directions (Section 6.3) that could enhance and extend the present work.

6.1 SUMMARY

RP offers clear benefits like high composability and improved comprehension (Observer pattern), but its adoption is possibly being obstructed by its steep learning curve, ties to functional programming, and inconsistent APIs. Researchers emphasized the need for systematic studies to address these challenges and guide its development and usability. The work contained in thesis aims to measure and reveal the usability of RP APIs, detailing findings (e.g., main problems and areas of improvements), tools, methods, and recommendations. To accomplish it, we conducted two major studies guided by three research questions: **RQ1.** *How frequently are RP operators used in open-source projects, and what does their usage reveal about usability and adoption?*, **RQ2.** *What challenges are RP users facing, and how they relate to API usability?*, and **RQ3.** *To what extent are RP APIs usable, and what aspects are most affected?*

The answers for first two research questions is the main focus of Chapter 3. The chapter conducts a mining study in both GH and SO. The operator data obtained through GH repositories showed that RP APIs are presenting bloated interfaces, with many instances of low frequencies of specialized operators. The SO mining, conversely, revealed a set of 23 topics grouped into nine categories by exploring the LDA algorithm. The largest category among RP posts is *Stream Abstraction*, which shows users struggling with stream manipulation and common operations. Among the relevant topics, we observed the presence of *Introductory Questions* and passages indicating RP developers finding it difficult to grasp basic RP understanding including operators.

Our second study focus on **RQ3** and the results are shown in Chapter 4. The chapter detailed a mixed-methods research that combines the use of metrics (applied with the use of an implemented tool called UAX), and a user-centered study (using the CDN framework). The great majority of the metrics' results indicate a high level of usability of APIs; nonetheless, that did not translate directly into users' perception, demonstrating a moderate level. Users, for instance, pointed that it was easy to make mistakes with the APIs as well as following different resolution paths, ultimately expressing low confidence regarding produced code. Nonetheless, users were not completely unsatisfied by their experience, and the majority recognized the importance of RP including considerations of future usages. To enhance usability, RP designers should focus especially in the lowest, concerning areas: understandability and expressiveness. Among the recommendations, the designers should consider interface reformulation (e.g., number of operators, nomenclature, and behavior description), inclusion of better examples showing diverse scenarios, addition of key sections like error-handling and state management, detailing of a best practices section, and investment in visual and alternative resources like learning games.

Software engineering approaches software development by exploring challenges and solutions that support quality and productivity (VALENTE, 2020). In that sense, tools should only be incorporated or explored in the software process when they cooperate to the efficiency of the development team, otherwise they likely add unnecessary cost. RP has the potential to improve event-driven and asynchronous software, especially those that explore the Observer pattern (SALVANESCHI et al., 2017). However, they also add risky factors, as demonstrated through the results of our thesis, that can directly impact the quality of the product under development. Therefore, the enhancement of RP usability will be a paramount goal for designers and researchers in the following years to make those interfaces more easy to use and learn and less costly from a software cycle perspective.

6.2 CONTRIBUTIONS

- **Comprehensive Usability Analysis of RP APIs:** This thesis evaluates the usability of RP APIs through multiple, mixed research methods such as mining researches, user-centered studies, and evaluation through metrics.
- **Empirical Findings from GitHub and Stack Overflow Mining:** The research leverages mining techniques to extract and analyze data from GH repositories and SO discussions. The analysis uncovered API users' preferences regarding the available ope-

rators and those less or not used that could be taken into account by the API designers to simplify the interfaces. The work also identifies common challenges faced by developers, and classifies those problems according to their popularity and difficulty to help API designers better direct their efforts.

- **Answer to Open Research Question from Literature:** Results of Chapter 3 targeted a recurrent problem pointed in previous researches about the excessive number of operators in RP APIs (SALVANESCHI, 2016; SALVANESCHI et al., 2017; MOGK, 2015). More precisely, we answered, from the RP's perspective, the following research question, "Do data flow languages provide a 'simple enough' solution for the common case without excessive proliferation of overspecialised operators?" (SALVANESCHI, 2016).

- **User-Centered Evaluation with Cognitive Dimensions:** We apply a refined version of the CDN framework to capture the developers' experience with the APIs; that provided a better perspective into important aspects like expressiveness and understandability that should be addressed by RP APIs' designers.

- **Usability Tool:** The thesis introduces an open-source tool called UAX, designed to measure usability through structural metrics, contributing as a practical asset for API evaluation. This becomes especially important given that most of the current tools are either not maintained or not available (RAUF; TROUBITSYNA; PORRES, 2019).

- **Recommendations for Improved API Usability:** Based on the thesis' findings, we proposed suggestions (Section 4.5) for possibly improving RP APIs' usability, which include enhancements in documentation quality, addition of code examples, and the development of best practices. All those recommendations are practical and address both the structural and informal usability aspects that participants found challenging.

- **Foundation for Future Usability Research in Reactive Programming:** The study introduce a mixed-method approach to evaluating RP API usability that could be reused for future studies in the RP area. In fact, the methods and findings of Chapter 3 served as a basis for the works of Pereira et al. (2023) and Farias, Zimmerle and Gama (2024).

6.3 FUTURE WORK

There are many ways in which the work conducted throughout the thesis could be enhanced or extended. Moreover, we envision ideas that could be further explored to improve the state of the art in the RP field. Section 6.3.1 lists enhancements to the work of this thesis, while Section 6.3.2 presents some ideas for future researches.

6.3.1 Thesis' Refinements

6.3.1.1 Mining Study

Operator mining. To mine the operators, we explored the use of regular expressions (regex) since it would be a convenient mean throughout the different languages explored. However, this is not the optimal approach since regex are designed for regular languages (WATT; BROWN, 2000) and lack context (e.g., a given method belongs to that class/object). We in fact demonstrated (Section 3.1.1) that noise in the counting was low when compared with actual Rx operators. Still, the use of a parser would increase the validity of our findings. For instance, the paper of Xu et al. (2020) used the Scala SemanticDB[1] tool to parse and extract Scala higher-order functions' information. For counting the operators in Chapter 4, we utilized the esquery[2] to query the AST generated by Esprima[3]. Nonetheless, a prominent approach is presented by the tool called Tree-sitter (BRUNSFELD et al., 2024) which is a parser generator tool that has many bindings, including Java, JavaScript, and Swift explored in our mining study.
Temporal trends and technical challenges. A missing aspect of our research was possibly not considering the understanding of time distribution among the topics reveled in Table 4. Future works, for instance, could try to divide the dataset of SO posts into time intervals, recategorize the posts based on the already identified topics and categories, and plot that information (BAJAJ; PATTABIRAMAN; MESBAH, 2014; KOCHHAR, 2016). This can be helpful to comprehend how the topics have been prevalent throughout a timeline. In addition to popularity and difficulty metrics (Section 3.1.2.2), a supplementary metric could be used to identify the technical challenges faced by the developers (BAJAJ; PATTABIRAMAN; MESBAH, 2014; KOCHHAR, 2016). This metric is computed based on SO statistics like upvotes, down-

[1] <https://scalameta.org/docs/semanticdb/guide.html>.
[2] <https://www.npmjs.com/package/esquery>.
[3] A parser for ECMAScript: <https://esprima.org/>.

votes, comments, answers, and favorite count[4].

6.3.1.2 Mixed-methods Study

Number of participants. Keeping a good quantity of willing participants was difficult during the study, especially considering the followed ethical principles (e.g., not offering rewards in exchange of users' participation). As a result, we could not generalize, for instance, which API provided the best usability from a users' perspective. In the future, it could be helpful to increase that number to collect more data, particularly if the research presents a more quantitative focus.

Additional APIs. We based the work in two popular JS tools. Nonetheless, it would be good to test if some of the findings are also present in other interfaces and how they would score in terms of metrics. As a result, we could build a better overview from the field. For example, in addition to Bacon.js and RxJS, we included the library xstream[5] in the paper (ZIMMERLE; GAMA, 2024); related to the other APIs, xstream showed a lower mean score of 0.73 (high level of usability). The library presented two, low and very-low scores: ADI (number of words contained in the functions' documentation) and APLCI (consistency among parameter name ordering across functions' definitions), respectively; the first result could be explained by some elements in its interface with short or no documentation, while the second may be related to the its few operators that is inline with the library design (in this case, it may not be a usability problem for the API, but consistency can alleviate the users' cognitive load (HERMANS, 2021; OUSTERHOUT, 2018) and it is often considered in usability test methods (BLACKWELL; GREEN, 2000)).

Additionally, it would be interesting to replicate the work with other interfaces beyond JS. For example, Pereira et al. (2023) explored the Swift Combine[6] framework, which is a declarative interface for processing asynchronous events over time and the base for SwiftUI[7]. The application and comparison of the results between RxSwift (explored in Chapter 3) and Combine could draw important research outcomes, especially considering their popularity and direct and indirect impact (e.g., there are possibly millions of apps[8] using those technologies).

[4] However, the favorite count was replaced twice: first by *bookmarks* and then by *saves* which is private. A full discussion can be found in #383706 and #383382.
[5] It is a small RP library specifically built for the framework Cycle.js: <https://github.com/staltz/xstream>.
[6] <https://developer.apple.com/documentation/combine>.
[7] <https://developer.apple.com/xcode/swiftui/>.
[8] <https://www.apple.com/app-store/>

In the same way, Kotlin, the Java alternative to Android development, figures as one of the most popular technologies[9] and includes its own version of a RP library called Flow[10]. Therefore, a similar comparison between Flow and RxKotlin would greatly cooperate in understanding the usability of RP interfaces and which points should be improved.

Validation of recommendations. As an outcome of the thesis, Section 4.5 presents some recommendations based on our findings. In the future, it would be important to validate those recommendations from the viewpoints of:

- **RP users:** to ensure that they agree with the proposed recommendations and there are no missing details.
- **RP API designers:** to capture their opinion regarding the recommendations (if all or part of them would in fact be helpful or not) and whether they would incorporate those into their tools.

6.3.1.3 UAX Tool

Improvements of implemented metrics. Some implemented metrics rely on specific options like the maximum number of parameters or the threshold used during documentation evaluation. While we have used values used in other studies and some of them are general preferences of developers or have a psychological explanation (RAMA; KAK, 2015; VENIGALLA; CHIMALAKONDA, 2021), there is still a lack of broad understanding of those values (VENIGALLA; CHIMALAKONDA, 2021). In this way, we plain to make those values configurable, so different scenarios will be possible to be tested. Furthermore, our tool only checks exported functions and classes (i.e., public methods of the class, either static or instance ones). Slowly, we plan to incorporate and investigate the addition of other exportable, languages constructs such as interfaces or objects.

New metrics. We did not implement all metrics defined by Rama and Kak (2015), but it is our belief that their contribution (AESI and ATSI) in the context of JS would be minimal (considering the arguments presented in Section 4.1.2). Nonetheless, in future releases, we will pursue the feasibility of not only those metrics, but additional metrics. The new for new metrics became more evident particularly considering the contrasting results shown in Section 4.3

[9] <https://survey.stackoverflow.co/2024/technology#most-popular-technologies>.
[10] <https://kotlinlang.org/docs/flow.html>.

that demonstrated, in most cases, that metrics linked to a dimension did not match the users' experience. As an example, the work of Scheller and Kühn (2015) presents additional, interesting aspects (e.g., fluent interfaces, which are commonly known as function composition in functional libraries) that could be adapted to our tool. Also, there are aspects pointed in other studies that could be used to drive new metrics. For instance, the well-known factory pattern was considered less usable and impacted programmer performance when compared to the direct usage of class constructors (ELLIS; STYLOS; MYERS, 2007); we could adapt this fact to create a metric that limits the number of factory functions allowed in a given interface. This, in turn, would impact RP APIs like Rx that often rely on factories. The majority of the implemented metrics deal with structural aspects, but there are more aspects that can also affect the user experience (ROBILLARD, 2009). This was in fact recognized by Scheller and Kühn (2015), giving as an example both naming (very dependent on many factors like the API purpose and language domain) and abstraction level. In fact, both Rama and Kak (2015) and Rauf, Troubitsyna and Porres (2019) express that metrics could be enhanced by techniques from machine learning and natural language processing, which is becoming more achievable in the present era of artificial intelligence. The ADI metric, for instance, is computed only based on the number of words related to a function documentation, not considering the quality of the text.

Dashboard. Other types of visualization are intended to be included in the dashboard such as the TreeMap chart (SOUZA; BENTOLILA, 2009) which may enable a better overview of grouping elements like results of all interface's functions, modules (i.e., the average score of the APIs within the module), and classes. In fact, we are currently enhancing the tool to include the results (in the generated JSON files) according to different language units/perspectives like modules (packages) and classes.

Linters. Static analysis tools like linters can help find opportunities for code improvement and refactorings, allowing an early, cost-effective modification of code (TÓMASDÓTTIR; ANICHE; DEURSEN, 2018). This is especially important for APIs, since, when they are defined, it is hard to modify them without impacting dependent codes (RAMA; KAK, 2015). With minor modifications or adaptations, we believe that the tool could be used as a linter, guiding API designers in the construction of more usable interfaces during the development process.

New languages. UAX has only being implemented to analyze TS projects. In the future, we intend to extend the implementation either for other JS type tools like Flow[11] or other

[11] It is a type static checker that allows type inference or annotation in JS: <https://flow.org/>.

statically typed programming languages. As long as all implementation agree on a common output, the same dashboard will be able to display a consistent visualization of the results.

6.3.2 Future Researches

API usage patterns. Given the extensive number of operators in RP APIs, it can be challenging for the user to choose the right operators for the given task. The Rx documentation usually group operators by categories[12] which helps the users to understand their intent. However, unveiling which of those operators are usually used together is still a missing fact. In this way, we believe that it could be an important research endeavor to understand API usage patterns (ZHONG et al., 2009; SAIED et al., 2015). Such a perspective would not only help users that could be aware of related set of operators that are mostly used together in a given context, but also the RP designers by better understanding the operators' usages and plan possible ways to simplify the APIs.

Recommendations in practice. The set of recommendations presented in Section 4.5 offers a practical opportunity to validate whether they can effectively impact the usability of RP APIs. To that end, future works could apply those recommendations to forks of RP tools and capture that phenomenon through both a quantitative and qualitative lens by leveraging user-centered studies. For example, those works could try to answer whether the recommendations, or a small set of them, can produce a statistically significant impact in the usability of RP APIs and which recommendation was the most impactful; in that sense, the qualitative perspective could complement the reasons behind those conclusions and possibly ways of improvements. Outcomes from those works could then be triangulated with the results from the validations pointed at Section 6.3.1.2 (Validation of recommendations) collected from both users and designers.

Learning game impact. In the present work, we approached the area of learning game which is still an area little explored in RP. As detailed in Section 4.2.4.1, there was an agreement that the explored game offered a good, basic introduction to the library and its API. The results presented only a first step of understanding its impact in the learning process. For example, future works could try to comprehend that learning mechanism in a larger scale. Moreover, according to a valuable feedback provided by one of the study participants, the game could be enhanced by bringing more real elements and context like the interaction of the game with

[12] <https://reactivex.io/documentation/operators.html#categorized>

available, RESTful APIs; in this way, it would become easier to make the transition between the game (ludic context) and code (real context). Hence, in the future, we plan to expand this line of investigation and possibly apply some of the suggestions to better understand their impact.

6.3.2.1 Software Engineering

Further understanding of RP testing. Section 3.3 observed the presence of the topic *Testing and Debugging* as the second highest percentage of questions without an accepted answer and few operators related with this area. Debugging is an important aspect under researching (SALVANESCHI; MEZINI, 2016; MOGK et al., 2018; BANKEN; MEIJER; GOUSIOS, 2018), but, as far as we know, there are no works specific targeting the understanding of reactive testing. Tests are an invaluable aspect in the modern software development (VALENTE, 2020), and even within the same family of RP libraries, it seems there is no consensus what method is the most indicated (e.g., there is only bindings of unit testing with marble text in RxJava[13] while RxJS brings that idea natively[14]). From the related area of data stream processing (DSP) applications, Vianna et al. (2023) examine challenges and practices for testing DSP applications, focusing on findings from 154 grey literature sources (e.g., blogs, forums, and documentation); the study outlined the testing challenges, purposes, approaches, strategies, and tools. The results of that study culminated in a series of testing guidelines for DSP applications[15]. Therefore, it is our belief that a deeper understanding of how RP code is tested, including a distributed setting (SALVANESCHI; DRECHSLER; MEZINI, 2013; DRECHSLER et al., 2014), and how they have actually been tested in projects is an important research endeavor.

Code maintenance and knowledge retention. Code maintenance is an inevitable part in the software lifecycle, and good software should promote maintainability (SOMMERVILLE, 2015). On the other hand, reuse can improve maintainability by allowing code reuse, modularity, consistency, among others. Section 4.3.4 examined the reusability dimension included in the questionnaire and covered by some metrics. The results indicated that the APIs excelled at the reuse aspect, displaying the highest value among all results. RP maintenance actually comprises an open research question: "Expressing computations with data flows involves mixing the right

[13] <https://github.com/alexvictoor/MarbleTest4J>
[14] <https://rxjs.dev/guide/testing/marble-testing>.
[15] Testing Guidelines for Data Stream Processing Applications.

operators in a combination that is specific for the problem at hand. Are such highly specialised pipelines more difficult to modify correctly than imperative code? What is the effect of the data flow style on maintainability?" (SALVANESCHI, 2016). The data presented in the thesis offers a first, positive glimpse towards that aspect, but further studies (e.g., mining studies) must be carry out for the complete understanding and response to that open question.

Knowledge retention is an important element in usability (NIELSEN, 1994) that may impact future maintenance of code. Usable APIs should thus allow easy comprehension of code, even after a long period since it was written. The account of an experienced participant during the interviews of Chapter 4 (Section 4.2.4.1) showed that he still considered his RP code clear. In the future, we intend to further explore if that point of view is the same for more users, or it actually depends on other factors like the user experience which may impact the quality of the produced code.

Code smells. The results of Chapter 4 revealed that the participants felt it was easy to make mistakes by using the APIs. In fact, we witnessed many instances of problems during the code observations (Section 4.2.2.3), and the interviews (Section 4.2.4) reinforced the need for best practices. Therefore, the identification of code smells and possible solutions could strongly benefit the area. Code smell is a term for design problems that are often mitigated by means of refactoring (FOWLER, 2018); by cataloguing those, it would possibly help improving the construction of programs built with those APIs and also serve as an important learning tool. Furthermore, it is known that code with smells is more prone to errors (LI; SHATNAWI, 2007; CAIRO; CARNEIRO; MONTEIRO, 2018) and is also more likely to undergo future modifications compared to code without smells (KHOMH; PENTA; GUEHENEUC, 2009).

BIBLIOGRAPHY

ABDELLATIF, A.; COSTA, D.; BADRAN, K.; ABDALKAREEM, R.; SHIHAB, E. Challenges in chatbot development: A study of stack overflow posts. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. [S.l.: s.n.], 2020. p. 174–185.

AGRAWAL, A.; FU, W.; MENZIES, T. What is wrong with topic modeling? and how to fix it using search-based software engineering. *Information and Software Technology*, Elsevier, v. 98, p. 74–88, 2018.

AHMED, S.; BAGHERZADEH, M. What do concurrency developers ask about? a large-scale study using stack overflow. In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.: s.n.], 2018. p. 1–10.

AKIDAU, T.; CHERNYAK, S.; LAX, R. *Streaming systems: the what, where, when, and how of large-scale data processing*. [S.l.]: "O'Reilly Media, Inc.", 2018.

ALABOR, M.; STOLZE, M. Debugging of rxjs-based applications. In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. [S.l.: s.n.], 2020. p. 15–24.

ALKHARUSI, H. A descriptive analysis and interpretation of data from likert scales in educational and psychological research. *Indian Journal of Psychology and Education*, v. 12, n. 2, p. 13–16, 2022.

ALLAMANIS, M.; SUTTON, C. Why, when, and what: analyzing stack overflow questions by topic, type, and code. In: IEEE. *2013 10th Working Conference on Mining Software Repositories (MSR)*. [S.l.], 2013. p. 53–56.

ALROOBAEA, R.; MAYHEW, P. J. How many participants are really enough for usability studies? In: IEEE. *2014 Science and Information Conference*. [S.l.], 2014. p. 48–56.

BAINOMUGISHA, E.; CARRETON, A. L.; CUTSEM, T. v.; MOSTINCKX, S.; MEUTER, W. d. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 45, n. 4, p. 1–34, 2013.

BAJAJ, K.; PATTABIRAMAN, K.; MESBAH, A. Mining questions asked by web developers. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2014. p. 112–121.

BANKEN, H.; MEIJER, E.; GOUSIOS, G. Debugging data flows in reactive programs. In: IEEE. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. [S.l.], 2018. p. 752–763.

BARROS, M. O. de. *Um Estudo do Uso da framework Combine em Projetos de Código Aberto em Swift*. Bachelor Thesis — Centro de Informática - Universidade Federal de Pernambuco (UFPE), 2022. Available at: <https://www.cin.ufpe.br/~tg/2022-1/tg_CC/tg_mob.pdf>.

BLACKHEATH, S. *Functional reactive programming*. [S.l.]: Simon and Schuster, 2016.

BLACKWELL, A. F.; GREEN, T. R. A cognitive dimensions questionnaire optimised for users. In: EDIZIONI MEMORIA. *12th Workshop of the Psychology of Programming Interest Group (PPIG 2000)*. [S.l.], 2000. p. 137–154.

BOGNER, J.; MERKEL, M. To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github. In: ACM. *Proceedings of the 19th International Conference on Mining Software Repositories*. [S.l.], 2022. p. 658–669.

BONÉR, J.; KLANG, V. Reactive programming versus reactive systems. *Dosegljivo: https://www. lightbend. com/reactiveprogramming-versus-reactive-systems.[Dostopano: 23. 08. 2017]*, 2017.

BOREN, T.; RAMEY, J. Thinking aloud: Reconciling theory and practice. *IEEE transactions on professional communication*, IEEE, v. 43, n. 3, p. 261–278, 2000.

BRUNET, J.; SEREY, D.; FIGUEIREDO, J. Structural conformance checking with design tests: An evaluation of usability and scalability. In: IEEE. *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. [S.l.], 2011. p. 143–152.

BRUNSFELD, M.; QURESHI, A.; HLYNSKYI, A.; THOMSON, P.; OBSERVEROFTIME; VERA, J.; DUNDARGOC; TURNBULL, P.; CLEM, T.; CREAGER, D.; HELWER, A.; RIX, R.; KAVOLIS, D.; ANTWERPEN, H. van; DAVIS, M.; LILLIS, W.; IKA; YAHYAABADI, A.; ÊN, T.-A. N.; KOLJA. *tree-sitter/tree-sitter: v0.24.4*. Zenodo, 2024. Available at: <https://doi.org/10.5281/zenodo.14061403>.

BURNS, C.; FERREIRA, J.; HELLMANN, T. D.; MAURER, F. Usable results from the field of api usability: A systematic mapping and further analysis. In: IEEE. *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. [S.l.], 2012. p. 179–182.

CAIRO, A. S.; CARNEIRO, G. d. F.; MONTEIRO, M. P. The impact of code smells on software bugs: A systematic literature review. *Information*, MDPI, v. 9, n. 11, p. 273, 2018.

CAMPBELL, J. C.; HINDLE, A.; STROULIA, E. Latent dirichlet allocation: extracting topics from software engineering data. In: *The art and science of analyzing software data*. [S.l.]: Elsevier, 2015. p. 139–159.

CLARKE, S. Describing and measuring api usability with the cognitive dimensions. In: CITESEER. *Cognitive Dimensions of Notations 10th Anniversary Workshop*. [S.l.], 2005. v. 16.

CORBIN, J.; STRAUSS, A. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. [S.l.]: Sage publications, 2014.

COSENTINO, V.; IZQUIERDO, J. L. C.; CABOT, J. Findings from github: methods, datasets and limitations. In: IEEE. *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. [S.l.], 2016. p. 137–141.

CUGOLA, G.; MARGARA, A. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 44, n. 3, p. 1–62, 2012.

DAS, K. R.; IMON, A. R. Geometric median and its application in the identification of multiple outliers. *Journal of Applied Statistics*, Taylor & Francis, v. 41, n. 4, p. 817–831, 2014.

DENZIN, N. K. Grounded theory and the politics of interpretation, redux. *The SAGE handbook of current developments in grounded theory*, Sage Publications, p. 449–469, 2019.

DIAS, J. P.; FARIA, J. P.; FERREIRA, H. S. A reactive and model-based approach for developing internet-of-things systems. In: IEEE. *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. [S.l.], 2018. p. 276–281.

DIPROSE, J.; MACDONALD, B.; HOSKING, J.; PLIMMER, B. Designing an api at an appropriate abstraction level for programming social robot applications. *Journal of Visual Languages & Computing*, Elsevier, v. 39, p. 22–40, 2017.

DRECHSLER, J.; SALVANESCHI, G.; MOGK, R.; MEZINI, M. Distributed rescala: An update algorithm for distributed reactive programming. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 49, n. 10, p. 361–376, 2014.

DUALA-EKOKO, E.; ROBILLARD, M. P. Asking and answering questions about unfamiliar apis: An exploratory study. In: IEEE. *2012 34th International Conference on Software Engineering (ICSE)*. [S.l.], 2012. p. 266–276.

ELLIOTT, C.; HUDAK, P. Functional reactive animation. In: ACM. *International Conference on Functional Programming*. 1997. Available at: <http://conal.net/papers/icfp97/>.

ELLIS, B.; STYLOS, J.; MYERS, B. The factory pattern in api design: A usability evaluation. In: IEEE. *29th International Conference on Software Engineering (ICSE'07)*. [S.l.], 2007. p. 302–312.

FARIAS, E.; ZIMMERLE, C.; GAMA, K. Perspectives and challenges of ios developers in using reactive programming with rxswift. In: *Proceedings of the XXXVIII Brazilian Symposium on Software Engineering*. Porto Alegre, RS, Brazil: SBC, 2024. p. 609–615. ISSN 0000-0000. Available at: <https://doi.org/10.5753/sbes.2024.3569>.

FARIAS, E. C. *Uma análise das perspectivas e desafios de desenvolvedores iOS ao incorporarem a programação reativa através do RxSwift*. Bachelor Thesis — Centro de Informática - Universidade Federal de Pernambuco (UFPE), 2023. Available at: <https://www.cin.ufpe.br/~tg/2023-2/TGs_CC/tg_ecf2.pdf>.

FOWLER, M. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 2018.

GANASSALI, S. The influence of the design of web survey questionnaires on the quality of responses. *Survey research methods*, v. 2, n. 1, p. 21–32, 2008.

HAN, J.; SHIHAB, E.; WAN, Z.; DENG, S.; XIA, X. What do programmers discuss about deep learning frameworks. *Empirical Software Engineering*, Springer, v. 25, n. 4, p. 2694–2747, 2020.

HAN, Z.; LI, X.; XING, Z.; LIU, H.; FENG, Z. Learning to predict severity of software vulnerability using only vulnerability description. In: IEEE. *2017 IEEE International conference on software maintenance and evolution (ICSME)*. [S.l.], 2017. p. 125–136.

HEMMATI, H.; BRIAND, L. An industrial investigation of similarity measures for model-based test case selection. In: IEEE. *2010 IEEE 21st International Symposium on Software Reliability Engineering*. [S.l.], 2010. p. 141–150.

HENKEL, J.; BIRD, C.; LAHIRI, S. K.; REPS, T. Learning from, understanding, and supporting devops artifacts for docker. In: IEEE. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. [S.l.], 2020. p. 38–49.

HENNING, M. Api design matters. *Communications of the ACM*, ACM New York, NY, USA, v. 52, n. 5, p. 46–56, 2009.

HERMANS, F. *The Programmer's Brain: What every programmer needs to know about cognition*. [S.l.]: Simon and Schuster, 2021.

JANNECK, M.; DOGAN, M. The influence of stressors on usability tests-an experimental study. In: SCITEPRESS. *Special Session on Socio-technical Dynamics in Information Systems*. [S.l.], 2013. v. 2, p. 581–590.

JONSEN, K.; JEHN, K. A. Using triangulation to validate themes in qualitative studies. *Qualitative research in organizations and management: an international journal*, Emerald Group Publishing Limited, v. 4, n. 2, p. 123–150, 2009.

KALLIAMVAKOU, E.; GOUSIOS, G.; BLINCOE, K.; SINGER, L.; GERMAN, D. M.; DAMIAN, D. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, Springer, v. 21, n. 5, p. 2035–2071, 2016.

KAMBONA, K.; BOIX, E. G.; MEUTER, W. D. An evaluation of reactive programming and promises for structuring collaborative web applications. In: ACM. *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. [S.l.], 2013. p. 1–9.

KHOMH, F.; PENTA, M. D.; GUEHENEUC, Y.-G. An exploratory study of the impact of code smells on software change-proneness. In: IEEE. *2009 16th Working Conference on Reverse Engineering*. [S.l.], 2009. p. 75–84.

KLEINSCHMAGER, S.; ROBBES, R.; STEFIK, A.; HANENBERG, S.; TANTER, E. Do static type systems improve the maintainability of software systems? an empirical study. In: IEEE. *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. [S.l.], 2012. p. 153–162.

KLEPPMANN, M. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. [S.l.]: "O'Reilly Media, Inc.", 2017.

KOCHHAR, P. S. Mining testing questions on stack overflow. In: *Proceedings of the 5th International Workshop on Software Mining*. [S.l.: s.n.], 2016. p. 32–38.

KULESZA, T.; STUMPF, S.; BURNETT, M.; KWAN, I. Tell me more? the effects of mental model soundness on personalizing an intelligent agent. In: ACM. *Proceedings of the sigchi conference on human factors in computing systems*. [S.l.], 2012. p. 1–10.

LEUTENEGGER, S.; EDGINGTON, J. A games first approach to teaching introductory programming. In: ACM. *Proceedings of the 38th SIGCSE technical symposium on Computer science education*. [S.l.], 2007. p. 115–118.

LI, W.; SHATNAWI, R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software*, Elsevier, v. 80, n. 7, p. 1120–1128, 2007.

LIMA, C.; HORA, A. What are the characteristics of popular apis? a large-scale study on java, android, and 165 libraries. *Software Quality Journal*, Springer, v. 28, n. 2, p. 425–458, 2020.

LÓPEZ-FERNÁNDEZ, L.; GARCÍA, B.; GALLEGO, M.; GORTÁZAR, F. Designing and evaluating the usability of an api for real-time multimedia services in the internet. *Multimedia Tools and Applications*, Springer, v. 76, n. 12, p. 14247–14304, 2017.

MACEFIELD, R. How to specify the participant group size for usability studies: a practitioner's guide. *Journal of usability studies*, Usability Professionals' Association Bloomingdale, IL, v. 5, n. 1, p. 34–45, 2009.

MACVEAN, A.; CHURCH, L.; DAUGHTRY, J.; CITRO, C. Api usability at scale. In: *PPIG*. [S.l.: s.n.], 2016. p. 26.

MAERTENS, R.; PETEGEM, C. V.; STRIJBOL, N.; BAEYENS, T.; JACOBS, A. C.; DAWYNDT, P.; MESUERE, B. Dolos: Language-agnostic plagiarism detection in source code. *Journal of Computer Assisted Learning*, Wiley Online Library, v. 38, n. 4, p. 1046–1061, 2022.

MAIER, I.; ODERSKY, M. Deprecating the observer pattern with scala. react. 2012.

MARGARA, A.; SALVANESCHI, G. Ways to react: Comparing reactive languages and complex event processing. *REM*, v. 14, 2013.

MARGARA, A.; SALVANESCHI, G. We have a dream: Distributed reactive programming with consistency guarantees. In: ACM. *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. [S.l.], 2014. p. 142–153.

MARGARA, A.; SALVANESCHI, G. On the semantics of distributed reactive programming: the cost of consistency. *IEEE Transactions on Software Engineering*, IEEE, v. 44, n. 7, p. 689–711, 2018.

MARUM, J. P. O.; JONES, J. A.; CUNNINGHAM, H. C. Towards a reactive game engine. In: IEEE. *2019 SoutheastCon*. [S.l.], 2019. p. 1–8.

MCLELLAN, S. G.; ROESLER, A. W.; TEMPEST, J. T.; SPINUZZI, C. I. Building more usable apis. *IEEE software*, IEEE, v. 15, n. 3, p. 78–86, 1998.

MEYEROVICH, L. A.; GUHA, A.; BASKIN, J.; COOPER, G. H.; GREENBERG, M.; BROMFIELD, A.; KRISHNAMURTHI, S. Flapjax: a programming language for ajax applications. In: ACM. *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. [S.l.], 2009. p. 1–20.

MOGK, R. Reactive interfaces: Combining events and expressing signals. In: ACM. *Workshop on Reactive and Event-based Languages & Systems (REBLS)*. [S.l.], 2015.

MOGK, R.; SALVANESCHI, G.; MEZINI, M. Reactive programming experience with rescala. In: ACM. *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*. [S.l.], 2018. p. 105–112.

MOGK, R.; WEISENBURGER, P.; HAAS, J.; RICHTER, D.; SALVANESCHI, G.; MEZINI, M. From debugging towards live tuning of reactive applications. In: *2018 LIVE Programming Workshop. LIVE*. [S.l.: s.n.], 2018. v. 18.

MURPHY, L.; KERY, M. B.; ALLIYU, O.; MACVEAN, A.; MYERS, B. A. Api designers in the field: Design practices and challenges for creating usable apis. In: IEEE. *2018 ieee symposium on visual languages and human-centric computing (vl/hcc)*. [S.l.], 2018. p. 249–258.

MYERS, B. A.; STYLOS, J. Improving api usability. *Communications of the ACM*, ACM New York, NY, USA, v. 59, n. 6, p. 62–69, 2016.

NAM, D.; MACVEAN, A.; HELLENDOORN, V.; VASILESCU, B.; MYERS, B. Using an llm to help with code understanding. In: ACM. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. [S.l.], 2024. p. 1–13.

NASEHI, S. M.; SILLITO, J.; MAURER, F.; BURNS, C. What makes a good code example?: A study of programming q&a in stackoverflow. In: IEEE. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. [S.l.], 2012. p. 25–34.

NIELSEN, J. *Usability engineering*. [S.l.]: Morgan Kaufmann, 1994.

OUSTERHOUT, J. K. *A philosophy of software design*. [S.l.]: Yaknyam Press Palo Alto, CA, USA, 2018.

PARNAS, D. L. Precise documentation: The key to better software. In: *The future of software engineering*. [S.l.]: Springer, 2010. p. 125–148.

PEREIRA, A.; GAMA, K.; ZIMMERLE, C.; CASTOR, F. Reactive programming with swift combine: An analysis of problems faced by developers on stack overflow. In: *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2023. p. 109–115.

PEREIRA, A. L. N. *UM ESTUDO SOBRE O USO DO FRAMEWORK COMBINE ATRAVÉS DA MINERAÇÃO DE PUBLICAÇÕES DO STACK OVERFLOW*. Bachelor Thesis — Centro de Informática - Universidade Federal de Pernambuco (UFPE), 2022. Available at: <https://www.cin.ufpe.br/~tg/2022-1/tg_SI/TG_alnp.pdf>.

PETERSEN, P.; HANENBERG, S.; ROBBES, R. An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In: ACM. *Proceedings of the 22nd International Conference on Program Comprehension*. [S.l.], 2014. p. 212–222.

PICCIONI, M.; FURIA, C. A.; MEYER, B. An empirical study of api usability. In: IEEE. *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.], 2013. p. 5–14.

PIERCE, B. C. *Types and programming languages*. [S.l.]: MIT press, 2002.

PROENÇA, J.; BAQUERO, C. Quality-aware reactive programming for the internet of things. In: SPRINGER. *International Conference on Fundamentals of Software Engineering*. [S.l.], 2017. p. 180–195.

RAMA, G. M.; KAK, A. Some structural measures of api usability. *Software: Practice and Experience*, Wiley Online Library, v. 45, n. 1, p. 75–110, 2015.

RAUF, I.; TROUBITSYNA, E.; PORRES, I. A systematic mapping study of api usability evaluation methods. *Computer Science Review*, Elsevier, v. 33, p. 49–68, 2019.

REACTIVEX. *ReactiveX Documentation*. 2016. <https://reactivex.io/documentation/>. Accessed: 2022-10-01.

REACTIVEX. *FlatMap Documentation*. 2024. <https://reactivex.io/documentation/operators/flatmap.html>. Accessed: 2024-09-12.

REBOUÇAS, M.; PINTO, G.; EBERT, F.; TORRES, W.; SEREBRENIK, A.; CASTOR, F. An empirical study on the usage of the swift programming language. In: IEEE. *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*. [S.l.], 2016. v. 1, p. 634–638.

ROBILLARD, M. P. What makes apis hard to learn? answers from developers. *IEEE software*, IEEE, v. 26, n. 6, p. 27–34, 2009.

ROBILLARD, M. P.; DELINE, R. A field study of api learning obstacles. *Empirical Software Engineering*, Springer, v. 16, p. 703–732, 2011.

RODRÍGUEZ, L. J.; WANG, X.; KUANG, J. Insights on apache spark usage by mining stack overflow questions. In: IEEE. *2018 IEEE International Congress on Big Data (BigData Congress)*. [S.l.], 2018. p. 219–223.

ROEHM, T.; TIARKS, R.; KOSCHKE, R.; MAALEJ, W. How do professional developers comprehend software? In: IEEE. *2012 34th International Conference on Software Engineering (ICSE)*. [S.l.], 2012. p. 255–265.

ROSEN, C.; SHIHAB, E. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, Springer, v. 21, n. 3, p. 1192–1223, 2016.

ROUSSEEUW, P. J. Robust estimation and identifying outliers. *Handbook of statistical methods for engineers and scientists*, McGraw-Hill, New York, v. 16, p. 16–11, 1990.

SAIED, M. A.; BENOMAR, O.; ABDEEN, H.; SAHRAOUI, H. Mining multi-level api usage patterns. In: IEEE. *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. [S.l.], 2015. p. 23–32.

SALDAÑA, J. *The coding manual for qualitative researchers*. [S.l.]: SAGE publications Ltd, 2021.

SALMAN, I.; MISIRLI, A. T.; JURISTO, N. Are students representatives of professionals in software engineering experiments? In: IEEE. *2015 IEEE/ACM 37th IEEE international conference on software engineering*. [S.l.], 2015. v. 1, p. 666–676.

SALVANESCHI, G. What do we really know about data flow languages? In: ACM. *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. [S.l.], 2016. p. 30–31.

SALVANESCHI, G.; AMANN, S.; PROKSCH, S.; MEZINI, M. An empirical study on program comprehension with reactive programming. In: ACM. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.], 2014. p. 564–575.

SALVANESCHI, G.; DRECHSLER, J.; MEZINI, M. Towards distributed reactive programming. In: SPRINGER. *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*. [S.l.], 2013. p. 226–235.

SALVANESCHI, G.; HINTZ, G.; MEZINI, M. Rescala: Bridging between object-oriented and functional style in reactive applications. In: ACM. *Proceedings of the 13th international conference on Modularity*. [S.l.], 2014. p. 25–36.

SALVANESCHI, G.; MARGARA, A.; TAMBURRELLI, G. Reactive programming: A walkthrough. In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.], 2015. v. 2, p. 953–954.

SALVANESCHI, G.; MEZINI, M. Debugging for reactive programming. In: IEEE. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. [S.l.], 2016. p. 796–807.

SALVANESCHI, G.; PROKSCH, S.; AMANN, S.; NADI, S.; MEZINI, M. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering*, IEEE, v. 43, n. 12, p. 1125–1143, 2017.

SCHELLER, T.; KÜHN, E. Automated measurement of api usability: The api concepts framework. *Information and Software Technology*, Elsevier, v. 61, p. 145–162, 2015.

SHNEIDERMAN, B.; PLAISANT, C. *Designing the user interface: strategies for effective human-computer interaction*. [S.l.]: Pearson Education India, 2010.

SOKOLOWSKI, D.; MARTENS, P.; SALVANESCHI, G. Multitier reactive programming in high performance computing. In: ACM. *6th Workshop on Reactive and Event-based Languages & Systems*. [S.l.], 2019.

SOMMERVILLE, I. *Software Engineering*. 10th. ed. [S.l.]: Pearson, 2015. ISBN 978-0-13-394303-0.

SOUZA, C. R. D.; BENTOLILA, D. L. Automatic evaluation of api usability using complexity metrics and visualizations. In: IEEE. *2009 31st International Conference on Software Engineering-Companion Volume*. [S.l.], 2009. p. 299–302.

STERZ, A.; EICHHOLZ, M.; MOGK, R.; BAUMGÄRTNER, L.; GRAUBNER, P.; HOLLICK, M.; MEZINI, M.; FREISLEBEN, B. Reactifi: Reactive programming of wi-fi firmware on mobile devices. *Art Sci. Eng. Program.*, v. 5, n. 2, p. 4, 2021.

STYLOS, J.; MYERS, B. Mapping the space of api design decisions. In: IEEE. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. [S.l.], 2007. p. 50–60.

STYLOS, J.; MYERS, B. A. The implications of method placement on api learnability. In: ACM. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. [S.l.], 2008. p. 105–112.

TAHIR, A.; DIETRICH, J.; COUNSELL, S.; LICORISH, S.; YAMASHITA, A. A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. *Information and Software Technology*, Elsevier, v. 125, p. 106333, 2020.

TÓMASDÓTTIR, K. F.; ANICHE, M.; DEURSEN, A. V. The adoption of javascript linters in practice: A case study on eslint. *IEEE Transactions on Software Engineering*, IEEE, v. 46, n. 8, p. 863–891, 2018.

TREUDE, C.; WAGNER, M. Predicting good configurations for github and stack overflow topic models. In: IEEE. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2019. p. 84–95.

TUCKER, A. B.; NOONAN, R. *Programming languages: principles and paradigms*. [S.l.]: McGraw-Hill, 2002.

TURNER, C. W.; LEWIS, J. R.; NIELSEN, J. Determining usability test sample size. *International encyclopedia of ergonomics and human factors*, CRC Press Boca Raton, FL, v. 3, n. 2, p. 3084–3088, 2006.

VALENTE, M. T. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. [S.l.]: Editora: Independente, 2020.

VENIGALLA, A. S. M.; CHIMALAKONDA, S. On the comprehension of application programming interface usability in game engines. *Software: Practice and Experience*, Wiley Online Library, v. 51, n. 8, p. 1728–1744, 2021.

VIANNA, A.; KAMEI, F. K.; GAMA, K.; ZIMMERLE, C.; NETO, J. A. A grey literature review on data stream processing applications testing. *Journal of Systems and Software*, Elsevier, v. 203, p. 111744, 2023.

VIRZI, R. A. Refining the test phase of usability evaluation: How many subjects is enough? *Human factors*, SAGE Publications Sage CA: Los Angeles, CA, v. 34, n. 4, p. 457–468, 1992.

WATT, D. A. *Programming language design concepts*. [S.l.]: John Wiley & Sons, 2004.

WATT, D. A.; BROWN, D. F. *Programming language processors in Java: compilers and interpreters*. [S.l.]: Pearson Education, 2000.

WEN, F.; NAGY, C.; LANZA, M.; BAVOTA, G. An empirical study of quick remedy commits. In: ACM. *Proceedings of the 28th International Conference on Program Comprehension*. [S.l.], 2020. p. 60–71.

WIJAYARATHNA, C.; ARACHCHILAGE, N. A.; SLAY, J. A generic cognitive dimensions questionnaire to evaluate the usability of security apis. In: SPRINGER. *International Conference on Human Aspects of Information Security, Privacy, and Trust*. [S.l.], 2017. p. 160–173.

XU, Y.; WU, F.; JIA, X.; LI, L.; XUAN, J. Mining the use of higher-order functions: An exploratory study on scala programs. *Empirical Software Engineering*, Springer, v. 25, p. 4547–4584, 2020.

YUAN, D.; LUO, Y.; ZHUANG, X.; RODRIGUES, G. R.; ZHAO, X.; ZHANG, Y.; JAIN, P. U.; STUMM, M. Simple testing can prevent most critical failures: An analysis of production failures in distributed {Data-Intensive} systems. In: ACM. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. [S.l.], 2014. p. 249–265.

ZHANG, T.; HARTMANN, B.; KIM, M.; GLASSMAN, E. L. Enabling data-driven api design with community usage data: A need-finding study. In: ACM. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. [S.l.], 2020. p. 1–13.

ZHONG, H.; XIE, T.; ZHANG, L.; PEI, J.; MEI, H. Mapo: Mining and recommending api usage patterns. In: SPRINGER. *ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*. [S.l.], 2009. p. 318–343.

ZIMMERLE, C.; GAMA, K. Reactive cep: Integrating complex event processing into web reactive languages. In: ACM. *Proceedings of the 24th Brazilian Symposium on Multimedia and the Web*. [S.l.], 2018. p. 69–72.

ZIMMERLE, C.; GAMA, K. A web-based approach using reactive programming for complex event processing in internet of things applications. In: ACM. *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. [S.l.], 2018. p. 2167–2174.

ZIMMERLE, C.; GAMA, K. Uax: Measuring the usability of typescript apis. In: *Proceedings of the XXXVIII Brazilian Symposium on Software Engineering*. Porto Alegre, RS, Brazil: SBC, 2024. p. 803–809. ISSN 0000-0000. Available at: <https://doi.org/10.5753/sbes.2024.3658>.

ZIMMERLE, C.; GAMA, K. On the usability of reactive programming apis: A combined approach. Under evaluation. 2025.

ZIMMERLE, C.; GAMA, K.; CASTOR, F.; FILHO, J. M. M. Mining the usage of reactive programming apis: a study on github and stack overflow. In: ACM. *Proceedings of the 19th International Conference on Mining Software Repositories*. [S.l.], 2022. p. 203–214.

APPENDIX A – REACTIVE PROGRAMMING CODE EXAMPLES

This Appendix shows the Java and Swift versions of the programs presented in the Section 2.2.2 of Chapter 2.

A.1 JAVA EXAMPLE

Source Code 3 – HTTP request simulation with automatic retries using the Observer pattern in Java

```java
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.time.Duration;

// Observable (Subject)
class ApiRequest {
    private final String url;
    private ApiObserver observer;
    private int retryCount = 3; // Maximum retries

    public ApiRequest(String url) {
        this.url = url;
    }

    public void subscribe(ApiObserver observer) {
        this.observer = observer;
        makeRequest(0);
    }

    private void makeRequest(int attempt) {
        new Thread(() -> {
            try {
                HttpClient client = HttpClient.newBuilder()
                        .connectTimeout(Duration.ofSeconds(10))
                        .build();

                HttpRequest request = HttpRequest.newBuilder()
                        .uri(URI.create(url))
                        .GET()
                        .build();
                HttpResponse<String> response = client.send(request,
                    HttpResponse.BodyHandlers.ofString());

                if (response.statusCode() >= 400) {
                    throw new IOException("HTTP error: " +
                        response.statusCode());
                }

                observer.onNext(response.body());
                observer.onComplete();
            } catch (IOException | InterruptedException e) {
                if (attempt < retryCount) {
                    System.out.println("Retrying... Attempt " + (attempt + 1));
                    makeRequest(attempt + 1);
                } else {
                    observer.onError(e);
                }
            }
        }).start();
    }
}

// Observer (Listener)
interface ApiObserver {
    void onNext(String data);
    void onError(Throwable error);
    void onComplete();
}

// Main class
public class ObserverPatternExample {
    public static void main(String[] args) {
        ApiRequest apiRequest = new ApiRequest("https://api.example.com/data");

        apiRequest.subscribe(new ApiObserver() {
            @Override
            public void onNext(String data) {
                System.out.println("Data: " + data);
            }

            @Override
            public void onError(Throwable error) {
                System.err.println("Final Error: " + error.getMessage());
            }

            @Override
            public void onComplete() {
                System.out.println("Done");
            }
        });
    }
}
```

Source: Elaborated by the author (2024)

Source Code 4 – HTTP request simulation with automatic retries using RxJava in Java

```java
import io.reactivex.rxjava3.core.Single;
import io.reactivex.rxjava3.schedulers.Schedulers;
import io.reactivex.rxjava3.core.Observable;
import io.reactivex.rxjava3.core.Maybe;
import io.reactivex.rxjava3.plugins.RxJavaPlugins;

import java.io.IOException;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.net.URI;
import java.time.Duration;

public class RxJavaRetryExample {
    public static void main(String[] args) {
        fetchJson("https://api.example.com/data")
            .retry(3) // Retries up to 3 times if an error occurs
            .subscribe(
                data -> System.out.println("Data: " + data),
                error -> System.err.println("Final Error: " +
                    error.getMessage()),
                () -> System.out.println("Done")
            );
    }

    private static Single<String> fetchJson(String url) {
        return Single.fromCallable(() -> {
            HttpClient client = HttpClient.newBuilder()
                .connectTimeout(Duration.ofSeconds(10))
                .build();

            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(url))
                .GET()
                .build();

            HttpResponse<String> response = client.send(request,
                HttpResponse.BodyHandlers.ofString());
            if (response.statusCode() >= 400) {
                throw new IOException("HTTP error: " + response.statusCode());
            }

            return response.body();
        }).subscribeOn(Schedulers.io()); // Perform network calls on IO scheduler
    }
}
```

Source: Elaborated by the author (2024)

A.2 SWIFT EXAMPLE

Source Code 5 – HTTP request simulation with automatic retries using the Observer pattern in Swift

```swift
/*** protocol defintion ***/
protocol Observer {
    func onNext(_ value: String)
    func onError(_ error: Error)
    func onCompleted()
}

/*** Observable/Subject implementation ***/

import Foundation

class NetworkObservable {
    private var observers: [Observer] = []

    // Add an observer
    func addObserver(_ observer: Observer) {
        observers.append(observer)
    }

    // Remove an observer
    func removeObserver(_ observer: Observer) {
        observers.removeAll { $0 as AnyObject === observer as AnyObject }
    }

    // Simulate an API request
    func fetchData(url: String, retries: Int = 3) {
        var attempt = 0

        func attemptRequest() {
            attempt += 1
            guard let requestURL = URL(string: url) else {
                notifyError(NSError(domain: "Invalid URL", code: -1,
                    userInfo: nil))
                return
            }

            let task = URLSession.shared.dataTask(with: requestURL) {
                data, response, error in
                if let error = error {
                    if attempt < retries {
                        print("Retrying... (\(attempt))")
                        attemptRequest() // Retry on failure
                    } else {
                        self.notifyError(error) // Notify observers
                            about the error
                    }
                    return
                }

                if let data = data, let jsonString = String(data: data,
                    encoding: .utf8) {
                    self.notifyNext(jsonString) // Notify observers
                        about the successful data
                    self.notifyCompleted() // Notify that the operation
                        is done
                } else {
                    self.notifyError(NSError(domain: "Invalid Data",
                        code: -2, userInfo: nil))
                }
            }

            task.resume()
        }

        attemptRequest()
    }

    // Notify observers about new data
    private func notifyNext(_ value: String) {
        observers.forEach { $0.onNext(value) }
    }

    // Notify observers about an error
    private func notifyError(_ error: Error) {
        observers.forEach { $0.onError(error) }
    }

    // Notify observers that operation is completed
    private func notifyCompleted() {
        observers.forEach { $0.onCompleted() }
    }
}

/*** Observer implementation ***/

class NetworkObserver: Observer {
    private let name: String

    init(name: String) {
        self.name = name
    }

    func onNext(_ value: String) {
        print("\(name) received data: \(value)")
    }

    func onError(_ error: Error) {
        print("\(name) encountered an error:
            \(error.localizedDescription)")
    }

    func onCompleted() {
        print("\(name) operation completed.")
    }
}


/*** Instatiating the objects and triggering the request ***/

let networkObservable = NetworkObservable()
let observer1 = NetworkObserver(name: "Observer 1")

networkObservable.addObserver(observer1)

// Fetch data (simulating an API call with 3 retries)
networkObservable.fetchData(url: "https://api.example.com/data")
```

Source: Elaborated by the author (2024)

Source Code 6 – HTTP request simulation with automatic retries using RxSwift in Swift

```
import RxSwift
import RxAlamofire
import Alamofire

/*
* This code uses an Rx wrapper around the elegant HTTP networking in
   Swift Alamofire
* https://github.com/RxSwiftCommunity/RxAlamofire
*/

let disposeBag = DisposeBag()

RxAlamofire.requestData(.get, "https://api.example.com/data")
    .map { _, data -> String in
        guard let jsonString = String(data: data, encoding: .utf8) else {
            throw NSError(domain: "Invalid Data", code: -1, userInfo:
                nil)
        }
        return jsonString
    }
    .retry(3) // Retry up to 3 times
    .subscribe(
        onNext: { response in
            print(response)
        }, onError: { error in
            print("Final Error: \(error.localizedDescription)")
        }, onCompleted: {
            print("Done")
        }
    )
    .disposed(by: disposeBag)
```

Source: Elaborated by the author (2024)