# UAX: Measuring the Usability of TypeScript APIs

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

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

## ABSTRACT

Application Programming Interfaces (APIs) have become an asset used everyday by developers looking for opportunities of code reuse. However, this ubiquity has led to numerous complexities, making the construction of good APIs challenging. Developers in fact often find APIs hard to learn and use. Therefore, usability has become a fundamental attribute for APIs in the last years, but it is often neglected by API designers. In reality, traditional user studies are frequently deemed expensive and difficult to interpret which stimulated the search for objective, and less expensive ways to measure API usability like metrics. The studies of metrics revealed good ways to measure API structurally, but the availability of publicly tools are either lacking or not being maintained. In this paper, we present UAX, a public tool that implements API usability metrics initially for TypeScript (a superset of JavaScript). We discuss its implementation and show its application to three reactive programming (RP) APIs with the help of a dashboard we also produced to help interpret the results. The evaluation showed that the RP APIs presented a high level of usability but with areas of improvements, like parameter consistency and API documentation. This demonstrates that the tool could support API designers to better plan the API construction and software engineering process, especially before the APIs get even released.

## KEYWORDS

API Usability, Metrics, Tool, Reactive Programming

## 1 INTRODUCTION

Our everyday lives of modern software development are heavily dependent on the reuse offered by Application Programming Interfaces (APIs) [19, 25, 36]. From basic computer instructions [38] to web services [19, 20], most lines of code developers write involve API calls [19]. This ubiquity, despite its benefits, introduces many complexities as good APIs are often harder to develop [8] and developers frequently find APIs hard to learn and use [18, 19]. There are, in fact, many design decisions involved [8, 35] that could impact thousands of programmers [8]. The consequences of bad designs can vary, from productivity impact, code inefficiencies, bugs, to, more concerning and significant, security flaws [8, 19, 24]. As a result, API usability has become a paramount quality attribute for API construction and adoption, and it has been studied through research venues in the last years [25]. Usable APIs are well-documented, easy to use and memorize, and encourage reuse [8, 22], translating into better programmers' productivity [8, 22].

Despite its importance, API usability are in many cases ignored by API designers [4]. Reasons include the difficult to gather and interpret usability data [38], specially considering its subjectivity nature [22]. Besides, studies with users are often considered expensive during API design [38]. In this way, the last decade has sparked the search for usability metrics as an objective, cost and time effective way of measuring usability [25]. The proposed metrics are based on common beliefs [24], presenting mathematical formulations [24] and also considering the context of use [33]. Additionally, visualizations have been explored as complement to metrics [5] aiming to offload the cognitive load to human visual capabilities and facilitate the recognition of patterns and interpretation of results. However, all the benefits provided by metrics can only be leveraged through the availability of tools that implement them, which seems to be either lacking or not being maintained in the field [25].

In this work, we propose a tool to analyze the usability of APIs called UAX (Usability Analyzer Experience). As a start point, we directed the tool to TypeScript (TS) APIs. JavaScript (JS) is one of the most used languages in the world[1] and, along with TS (a JavaScript superset that, among other things, adds static typing to the language), top ranks the nowadays most popular technologies[2]. Moreover, one of the JS most used package managers, npm[3], counts with more than 2 million packages[4]; in other words, there are many, publicly available APIs that could benefit from usability tools. Finally, static typing can not only aid in implementing some metrics, but also help to find and fix errors faster [10], enhance the usability of unknown API [21], and act as an implicit documentation [23].

The paper thus presents the tool UAX.js, discussing its implementation and the set of metrics already implemented, and showing a complementary user interface (dashboard) that can help in the interpretation of the results. To complete our presentation, we demonstrate the tool's feasibility by evaluating three reactive programming (RP) APIs (RxJS, Bacon.js, and xstream), abundant in JS given its asynchronous, event-driven nature. RP is a paradigm that aims to facilitate the development of reactive applications [30, 31], but its learning curve and relation to functional programming can be challenging [17, 32]. Moreover, RP libraries commonly adopts different interfaces [16], and many design decisions are disconnected from the user experience [17]. The metrics' results showed that both RxJS and Bacon.js presented a very high level of usability (inline with observations from previous studies relating high levels of usability with the number of GitHub stars and forks [36]), followed by xstream with a high level score. In metric terms, the APIs depicted mainly good results, but we could observed that at least two APIs scored from moderate to low in terms of documentation, a possible area of improvement.

[1]https://octoverse.github.com/2022/top-programming-languages
[2]https://survey.stackoverflow.co/2023/#technology-most-popular-technologies
[3]https://www.npmjs.com/
[4]One of the greatest in the world according to https://nodejs.org/en/learn/getting-started/an-introduction-to-the-npm-package-manager

We believe that our tool has the potential to help API designers to better plan the design and construction process of APIs by providing the easy access to usability metrics, something that has been lacking in the field [25]. Also, the tool may better support the software engineering process before the API even get released; this way, designers can avoid the problematic process of changing the APIs afterwards [24]. The remainder of the paper is organized as follows. Section 2 delivers the background and related work necessary for the rest of the study. Section 3 presents our methodology, detailing the metrics we used, the implementation of the tool, and an additional user interface we built. Section 4 presents our usage scenario with the three RP APIs, while Section 5 discusses the current limitations and opportunities for improvements. Section 6 presents the final remarks and perspectives for future works.

# 2 BACKGROUND AND RELATED WORK

## 2.1 API 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 [4, 25]. An API is an interface geared towards human developers [19], similar to graphical interfaces, so usability considerations and human-computer interactions (HCI) principles are very valuable [18, 19], but often neglected by API designers [4]. The interest in API usability, in fact, only started to gain more attention after 2000 [19], in which the activities were slowly being referred to as DevX or DX (developer experience), analogously to UX (user experience) [18, 19]. Given its subjective nature [22, 25], the usability definition varies among standards and researchers. However, in a summarized way, API usability indicates how easy it is to use, in a given context, and learn an API [25].

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 [18, 25]. Bugs and security problems are examples of poor or incorrect usage of APIs [18, 19] that frequently impact negatively big organizations [37]. APIs are often difficult to use and learn [19], and it is much more easy to create bad APIs than good ones [8]. 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 [18]. All of this corroborates to the importance of a good API design [18] and early usability tests in the API life cycle.

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.), but their applicability to the API area is still to be researched [33]. An exception is the popular method known as the Cognitive Dimensions of Notations (CDN) framework [11], which is a tool for designers that describe a set of vocabularies or dimensions to measure the decision that most affect the usability of a notation [6]. In spite of its popularity, the dimensions and their applicability are often considered complex (e.g., the dimensions an their relations are hard to comprehend and assess [33]) and the method is still cost and time consuming, specially considering that it relies on users to perform some tasks [6] normally in a lab setting.

The need for more objective and less expensive methods prompted some research investments in usability metrics in the last decade [25]. Souza and Bentolila [5] defined the API usability as a function of its complexity based on three metrics and used a visualization tool, called Metrix, to aid in the identification of complex API areas. However, the tool has not been maintained [25]. Rama and Kak[24] define a set of eight metrics, detailed through math formulas, to evaluate the usability of class methods based on common beliefs. Table 1 depicts a summary of the eight Rama and Kak [24] metrics; a thorough examination of the metrics can be in found elsewhere [24, 36]. The authors show the usage of the metrics by testing them in seven Java APIs with the help of their proprietary tool [25]. The work of Scheller and Kühn [33] seems to be a promise venue, in which they reused many metrics defined by Rama and Kak [24] to create 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 [25].

## 2.2 Reactive Programming

Reactive systems or applications are a vital and wide class of software that focus on the reaction of all sorts of events like UI interactions, Internet messages, sensor stimuli, and others [29, 30]. Most modern software encompass reactivity [2, 31], which makes them challenging to design, implement, and maintain [30, 31]. In this way, the reactive programming (RP) paradigm has gained popularity in the last years as an alternative to facilitate the construction of such systems by directly represent reactive values in a intuitive, composable, and declarative way[29, 32]. The paradigm is influenced by signal processing [15] and has origins in the functional programming area where its was primarily used to model animations [17, 32]. Nowadays, its applicability has been extended to a myriad of areas like distributed applications, game engines, WiFi firmware, among others [7, 12–14, 34].

Recent studies have attested the high composability of RP abstractions [30] and the increase in comprehension compared to the Observer pattern [28, 32]. Nonetheless, there still a lack of understanding between the RP design choices and the user experience [17], and the benefits of widespread adoption of solutions based on the dataflow style are still unclear [27]. Observation showed that the learning curve and its relation to functional programming can be challenging for the paradigm adoption [32], which is exacerbated by the lack of consistency among RP interfaces [16]. Research has proposed systematic investigation about the designs of such solutions to better understand their impact and usability [16, 27, 39] and to fulfill the lacking of guidelines to drive better design choices for future RP developers [16, 27].

# 3 METHODOLOGY

## 3.1 Metrics

To evaluate the APIs with the help of metrics, we decided to use the ones proposed by Rama and Kak (Section 2.1). An advantage of using such metrics is that the authors extensively surveyed well-accepted beliefs of good designs. Also, the metrics are very general and detailed, with mathematical rigor, and have already been applied in other studies [36]. A total of eight metrics were

**Table 1: Usability metrics defined by Rama and Kak [24]**

| Metric | Description |
|---|---|
| API Method Name Overloading Index (AMNOI) | It quantifies the degree to which the various overload definitions of a function yield disparate return types [24]. The lesser the metric score, the greater is the number of overloads that return different types. |
| API Method Name Confusion Index (AMNCI) | It is based on three name-abuse patterns listed by Rama and Kak [24] 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. |
| API Method Grouping Index (AMGI) | It measures the extent to which semantically similar functions are grouped rather than dispersed [36]. The semantic similarity is defined based on keywords extracted from the function names [24]; for instance, functions called `mergeMap` and `concatMap` could be considered semantically similar given the shared keyword "map". The metric is then calculated based on the number of runs of similar keywords, where runs are groups of semantically similar functions placed together (i.e., in a class or file). |
| API Parameter List Consistency Index (APLCI) | It assesses the consistency in terms of parameter name ordering across functions' definitions [36]. |
| API Parameter List Complexity Index (APXI) | This metric deals with the length of function parameter and the runs of parameters of the same type [24]. Long lists of parameter and sequences of parameters with the same data type are likely to worsen the user experience [24]. |
| API Documentation Index (ADI) | The metric examines the number of words contained in the functions' documentation [36].It is important to emphasize that the metric is based on a threshold, which defines a minimum number of words for every function documentation. |
| API Exception Specificity Index (AESI) | This metric deals with the specialization and generalization of checked exceptions. As pointed by Rama and Kak [24], specialized exceptions work better than general ones for APIs [24]. So, the analyses revolve around an examination of the exception inheritance trees of exceptions declared by the functions exposed by the API. |
| API Thread Safety Index (ATSI) | 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. |

proposed by Rama and Kak [24] and a summary of them is displayed in Table 1.

## 3.2 Implementation

We implement the metrics in a tool we called Usability Analyzer Experience (UAX), intended to resemble the term "developer experience"(DevX) which is often linked to API usability [18, 19]. The implementation works a command-line interface and requires that the user inform the path to a JSON containing basic configuration information like an optional name for the library to be tested, its base path, the path for the *tsconfig.json* file, and the file containing the exported API. The user can also provide a path for the output results (defaults to *./output/*), which consist of a series of JSON files following the pattern name "name of the tested API - metric name". Each corresponding file carries information about the metric evaluated along with the library/project tested and usability information regarding the metric execution like the final score. A complete description is found at the GitHub repository[5].

To implement the metrics, we utilized the ts-morph[6] library, which is a wrapper over the TypeScript compiler that facilitates code manipulation. Given the early stages of the tool development, we have implemented only the first six metrics listed in Section 3.1. The Exception case (AESI) is a little complicated in TypeScript, since there is not yet a syntax declaration of the exceptions raised by a function or method[7]. Besides, the catch clause does not differentiate between different types of custom exceptions, and it is common that TypeScript (JavaScript) developers use the general `Exception` class most of the time. Thread support (ATSI) in JavaScript/TypeScript is something that is explored only in specific cases like CPU-bound tasks, since most of the computations are handled within a single-threaded, event loop. Therefore, our focus was on the other metrics, leaving AESI and ATSI for future evaluation if applicable.

The code was implemented by examining the exported, public declarations, and the inspection proceeded according to files as they would correspond to JavaScript module boundary (i.e., the highest-level modular unit). One of the main sources of difficulties was to incorporate type assignability, since TypeScript did not make publicly available yet a function to test that[8]. Instead, we recurred to a library called type-plus[9] that implements a series of type checking operations; with type-plus, we explored the ts-morph API to create temporary files to check for type assignability. Due to the manipulation of temporary files, some metrics like AMNOI and APXI proved to be CPU-intensive; so, for those metrics, we exploited

[5]https://github.com/uax-analyzer/uax
[6]https://github.com/dsherret/ts-morph.

[7]A complete discussion can be found in https://github.com/microsoft/TypeScript/issues/13219 and https://github.com/microsoft/TypeScript/issues/52145.
[8]A full 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.

a pool of worker threads with the help of the Piscina[10] library. Both APXI and ADI metrics rely on parameters that must be set for the execution of the metric. In the case of APXI, we set the parameter (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) in accordance to the value defined in Rama and Kak [24]. For ADI, we considered a threshold of 50 following the work of Venigalla and Chimalakonda [36].

## 3.3 Complementary Visualization

Visualizations can help offloading the cognitive load to human visual capabilities [5], 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 HTML dashboard to ease the interpretation of the tool's output. The dashboard is served and accessed through an HTTP server, and it also works (i.e., it is executed) as a command-line program. To run it, the user mainly needs to inform the location of the JSON files generated by UAX.js, and the dashboard parses and computes all the available, metrics' information.

The main page of the dashboard shows an overview of the computed scores obtained by every API analyzed (Figure 1). We tried as much to include 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 [24]; 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 [1] and adapted to our scale (which varies from 0.0 to 1.0) by means of simple linear transformation. 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 2). 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 this 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 [5]. 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 from the APIs side by side, allowing one to assess how every API scored from the metric perspective.

Additionally, we included a second page to display, with more details and exclusiveness, the information about one of the analyzed APIs. This page can be accessed through various links available on the main page or by using the endpoint "/[API_name]". For instance, in Figure 1, the name of one of the APIs is rxjs, so by accessing the URL "/rxjs" one would view a similar page as displayed in Figure 3. First, the overall score of the API is shown, describing its level of usability; also, the scores for the metrics are outlined, informing the highest and lowest evaluated metric. Following, the results of every metric are depicted along with a summary for the given metric.

**Figure 1: Main page overview of the dashboard tool.**

**Figure 2: Color scale indicating the level of usability according to scores.**

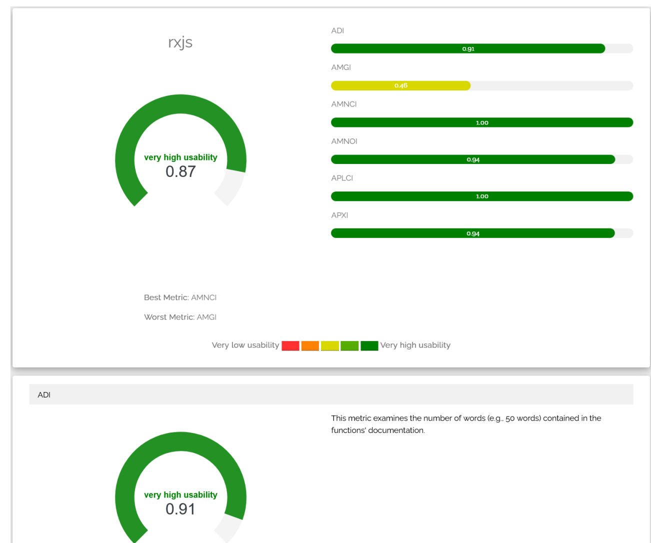


**Figure 3: Snippet of the API detailing page of the user interface tool.**

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

**Table 2: The explored RP APIs sorted (descending order) by their GitHub stars and forks.**

| API | Version | GitHub | |
|---|---|---|---|
| | | Stars | Forks |
| RxJS | 7.8.1 | 30,303 | 2,984 |
| Bacon.js | 3.0.17 | 6,463 | 330 |
| xstream | 11.14.0 | 2,366 | 136 |

*Note*: Last update on June 2, 2024.

# 4 USAGE SCENARIO

To test our tool, we selected three JS, RP APIs: RxJS, Bacon.js, and xstream. The first two are the JS RP libraries with the most GitHub stars and forks in that order. Venigalla and Chimalakonda [36] observed that repositories with both high numbers of stars and forks showed higher scores in the context of game engines; in this way, it was our chance to test this hypothesis for RP APIs. xstream, conversely, was built for the web framework Cycle.js and, different from the other RP libraries, it has a tiny API surface. Table 2 shows the some information about the tested APIs such as their version, number of GitHub stars and forks.

Figure 1 actually shows the results, depicted in the UI dashboard, obtained for the three RP APIs after running the metrics. Overall, the APIs presented high level of usability, scoring an average result of 0.8. From the three, RxJS scored highest (0.87), followed by bacon.js (0.81) and xstream (0.73). It could be observed that API repositories with highest stars and forks actually showed the highest level of usability for RP APIs. Moreover, both RxJS and Bacon.js had very high levels of usability, while xstream, the API with less stars and forks, was slightly lower but still presented a high usability level.

In metric terms, a great part of the metrics' results presented either high or very high results, with RxJS excelling at five metrics. The RxJS only exception was AMGI (grouping of semantically similar functions), which yielded a moderate value. A reason to this result may be explained by the RxJS strategy of modularizing most of its functionalities as standalone functions in separate files. For example, all of its operations reside in different files[11]. Both Bacon.js and xstream group their functionalities in class methods, reason that may justify their good results for AMGI. In future releases of the tool, we may analyze alternative ways of API analysis for AMGI, after all the RxJS operators may be not grouped at the module or class scope, but they are placed together in the same folder.

xstream was the API with the lowest overall score and its radar chart is reproduced in Figure 4. xstream actually presented very high results for four metrics: APXI (length of function parameter and runs of parameters of the same type), AMGI, AMNCI (name-abuse patterns), and AMNOI (overloaded functions with disparate return types). Contrarily, ADI (number of words contained in the functions' documentation) and, specially, APLCI (consistency among parameter name ordering across functions' definitions) were the metrics that most collaborated to lower xstream's overall score. The impact on APLCI may be explained by the small length of API operators (26 according to its repository), which implies few functions sharing consistent parameter ordering. RxJS and Bacon.js, conversely, offer more than 100 operators, which gives more space for consistent ordering of functions parameters, but it also introduces other challenges like possibly making the API more difficult to master [32]. Nonetheless, the API designers of xstream should evaluate if the consistency level is really a problem for the API, after all, consistency can alleviate the users' cognitive load [9, 20] and it is a property often considered in usability test methods [3]. Along with names and types, documentation is a resource that developers use to understand APIs and it seems that xstream did not include a lot of words in its documentation (ADI) related to the features exported in its API. For example, none of the methods in xstream `MemoryStream` class[12] has any documentation. We could also observe that Bacon.js did not show a high level of usability in terms of documentation, but a moderate one. Therefore, we think RP APIs should invest more in documentation, especially considering that it is a different paradigm with learning curve and functional programming concepts as possible obstacles [32].

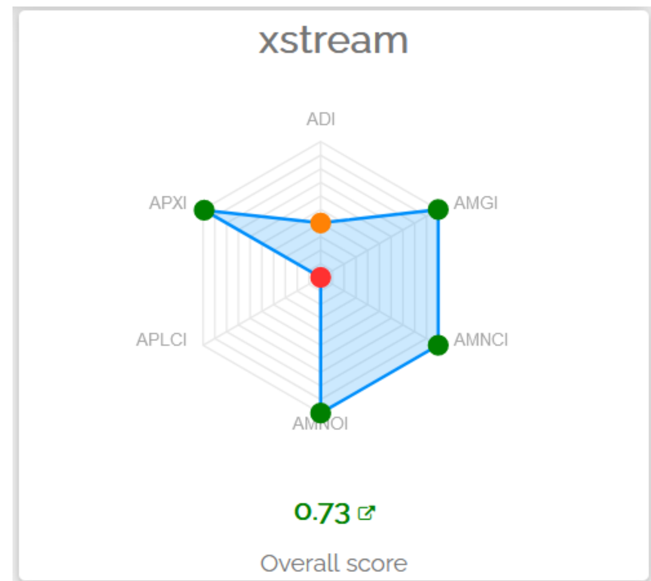


**Figure 4: Radar chart showing the overall scores and metric results of the xstream API.**

# 5 LIMITATIONS AND FUTURE IMPROVEMENTS

As discussed in Section 3.2, we did not implement all metrics listed by Rama and Kak [24], 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 3.2 as well). Still, in future releases, we will pursue the feasibility of not only those metrics, but additional metrics. For example, the work of Scheller and Kühn [33] presents additional, interesting aspects (e.g., fluent

[11] https://github.com/ReactiveX/rxjs/tree/7.8.1/src/internal/operators

[12] https://github.com/staltz/xstream/blob/v11.14.0/src/index.ts/#L1978

interfaces, which are commonly known as function composition in functional libraries) that could be adapted to our tool.

The majority of the metrics deal with structural aspects, but there are more aspects that can also affect the user experience [26]. This was in fact recognized by Scheller and Kühn [33], giving as an example both naming (very dependent on many factors like the API purpose and language domain) and abstraction level. Moreover, both Rama and Kak [24] and Rauf et al. [25] 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. We believe that only with the incorporation of such techniques the metrics may become a full replacement for user studies rather than a complementary tool.

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 [24, 36], there is still a lack of broad understanding of those values [36]. In this way, we plain to make those values configurable, so different scenarios will be possible to be tested.

Finally, 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. Also, other types of visualization are intended to be included in the dashboard such as the TreeMap chart [5] which may enable a better overview of grouping elements like results of all API 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.

Future studies should explore different types of APIs beyond RP. It would be interesting to compare how distinct classes of APIs score. Moreover, a greater number of APIs should also be tested to obtain more varying results. A limitation of our study was to test only three RP APIs; in this way, our results did not show great variation. Therefore, we cannot generalize our findings to all RP APIs that exist. However, it was not our intention to generalize the results; our goal was only to demonstrate the tool's usage as well as the supporting dashboard. Nevertheless, we strongly believe that the tool should be tried with a greater quantity of APIs as well as distinct classes of APIs.

## 6 FINAL REMARKS

APIs have become a ubiquitous tool in everyday lives of programmers as a way of reusing code and probably enhance code productivity. However, APIs are often hard to learn and use, which demands better support for tools to objectively measure API usability, something that has either been lacking or not maintained. In this paper, we presented a public tool called UAX.js, primarily targeting TypeScript APIs. With the help of a complementary, web dashboard, we demonstrated the tool by applying the implemented metrics to three reactive programming (RP) APIs. The results already showed that RP API designers have directed good effort to make the APIs very usable. However, the tool also revealed some areas of improvements like parameter consistency and, especially, API documentation. From all that was observed and demonstrated, it is our belief that the tool can support API designers to better plan the API design and construction as well as the overall software engineering process. This becomes especially important as the designers can get better insights and make the appropriate modifications even before the API get released, when it becomes harder to introduce changes to the API.

There are a lot of ways that the tool can still be enhanced, specially considering that it is in its early stages. For instance, the tool already implement six metrics, but there are others available in the literature that could be incorporated involving other aspects. Also, we plan to investigate and add other languages constructs that could be evaluated by the metrics beyond functions and class methods. Future releases of the tools will include a more detailed output, informing the results at the module (package) or class level. Additionally, we intend to include more types of visualizations (e.g., treemap charts) and configuration to help users better interpret the results. Finally, we envision that the metrics could be further enhanced by techniques of machine learning and natural language processing, something that is becoming more feasible in this artificial intelligence era.

## ARTIFACT AVAILABILITY

A demo video of the tool is available at https://youtu.be/qqLkTYIvnGQ and https://dx.doi.org/10.6084/m9.figshare.25971490. The repository of UAX.js is available at https://github.com/uax-analyzer/uax. The repository of the UI dashboard is available at https://github.com/uax-analyzer/uax-ui. All the repositories have been archived through software heritage (https://www.softwareheritage.org/).

## ACKNOWLEDGMENTS

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

## REFERENCES

[1] Hussain Alkharusi. 2022. A descriptive analysis and interpretation of data from likert scales in educational and psychological research. *Indian Journal of Psychology and Education* 12, 2 (2022), 13–16.
[2] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A survey on reactive programming. *ACM Computing Surveys (CSUR)* 45, 4 (2013), 1–34.
[3] Alan F Blackwell and Thomas RG Green. 2000. A Cognitive Dimensions questionnaire optimised for users.. In *12th Workshop of the Psychology of Programming Interest Group (PPIG 2000)*. Edizioni Memoria, 137–154.
[4] Chris Burns, Jennifer Ferreira, Theodore D Hellmann, and Frank Maurer. 2012. Usable results from the field of API usability: A systematic mapping and further analysis. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 179–182.
[5] Cleidson RB De Souza and David LM Bentolila. 2009. Automatic evaluation of API usability using complexity metrics and visualizations. In *2009 31st International Conference on Software Engineering-Companion Volume*. IEEE, 299–302.
[6] James Diprose, Bruce MacDonald, John Hosking, and Beryl Plimmer. 2017. Designing an API at an appropriate abstraction level for programming social robot applications. *Journal of Visual Languages & Computing* 39 (2017), 22–40.
[7] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An update algorithm for distributed reactive programming. *ACM SIGPLAN Notices* 49, 10 (2014), 361–376.
[8] Michi Henning. 2009. API design matters. *Commun. ACM* 52, 5 (2009), 46–56.
[9] Felienne Hermans. 2021. *The Programmer's Brain: What every programmer needs to know about cognition.* Simon and Schuster.

[10] Sebastian Kleinschmager, Romain Robbes, Andreas Stefik, Stefan Hanenberg, and Eric Tanter. 2012. Do static type systems improve the maintainability of software systems? An empirical study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 153–162.

[11] Luis López-Fernández, Boni García, Micael Gallego, and Francisco Gortázar. 2017. Designing and evaluating the usability of an API for real-time multimedia services in the Internet. *Multimedia Tools and Applications* 76, 12 (2017), 14247–14304.

[12] Alessandro Margara and Guido Salvaneschi. 2014. We have a DREAM: Distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 142–153.

[13] Alessandro Margara and Guido Salvaneschi. 2018. On the semantics of distributed reactive programming: the cost of consistency. *IEEE Transactions on Software Engineering* 44, 7 (2018), 689–711.

[14] Joao Paulo O Marum, J Adam Jones, and H Conrad Cunningham. 2019. Towards a reactive game engine. In *2019 SoutheastCon*. IEEE, 1–8.

[15] Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. ACM, 1–20.

[16] Ragnar Mogk. 2015. Reactive interfaces: Combining events and expressing signals. In *Workshop on Reactive and Event-based Languages & Systems (REBLS)*. ACM.

[17] Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. Reactive programming experience with rescala. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*. ACM, 105–112.

[18] Lauren Murphy, Mary Beth Kery, Oluwatosin Alliyu, Andrew Macvean, and Brad A Myers. 2018. API designers in the field: Design practices and challenges for creating usable APIs. In *2018 ieee symposium on visual languages and human-centric computing (vl/hcc)*. IEEE, 249–258.

[19] Brad A Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (2016), 62–69.

[20] John K Ousterhout. 2018. *A philosophy of software design*. Vol. 98. Yaknyam Press Palo Alto, CA, USA.

[21] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. 2014. An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 212–222.

[22] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. 2013. An empirical study of API usability. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 5–14.

[23] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

[24] Girish Maskeri Rama and Avinash Kak. 2015. Some structural measures of API usability. *Software: Practice and Experience* 45, 1 (2015), 75–110.

[25] Irum Rauf, Elena Troubitsyna, and Ivan Porres. 2019. A systematic mapping study of API usability evaluation methods. *Computer Science Review* 33 (2019), 49–68.

[26] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.

[27] Guido Salvaneschi. 2016. What do we really know about data flow languages?. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 30–31.

[28] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. 2014. An empirical study on program comprehension with reactive programming. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 564–575.

[29] Guido Salvaneschi, Joscha Drechsler, and Mira Mezini. 2013. Towards distributed reactive programming. In *Coordination Models and Languages: 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings 15*. Springer, 226–235.

[30] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th international conference on Modularity*. ACM, 25–36.

[31] Guido Salvaneschi, Alessandro Margara, and Giordano Tamburrelli. 2015. Reactive programming: A walkthrough. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 953–954.

[32] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2017. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1125–1143.

[33] Thomas Scheller and Eva Kühn. 2015. Automated measurement of API usability: The API concepts framework. *Information and Software Technology* 61 (2015), 145–162.

[34] Artur Sterz, Matthias Eichholz, Ragnar Mogk, Lars Baumgärtner, Pablo Graubner, Matthias Hollick, Mira Mezini, and Bernd Freisleben. 2021. ReactiFi: Reactive Programming of Wi-Fi Firmware on Mobile Devices. *Art Sci. Eng. Program.* 5, 2 (2021), 4.

[35] Jeffrey Stylos and Brad Myers. 2007. Mapping the space of API design decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. IEEE, 50–60.

[36] Akhila Sri Manasa Venigalla and Sridhar Chimalakonda. 2021. On the comprehension of application programming interface usability in game engines. *Software: Practice and Experience* 51, 8 (2021), 1728–1744.

[37] Chamila Wijayarathna, Nalin AG Arachchilage, and Jill Slay. 2017. A generic cognitive dimensions questionnaire to evaluate the usability of security apis. In *International Conference on Human Aspects of Information Security, Privacy, and Trust*. Springer, 160–173.

[38] Tianyi Zhang, Björn Hartmann, Miryung Kim, and Elena L Glassman. 2020. Enabling data-driven api design with community usage data: A need-finding study. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, 1–13.

[39] Carlos Zimmerle, Kiev Gama, Fernando Castor, and José Murilo Mota Filho. 2022. Mining the usage of reactive programming APIs: a study on GitHub and stack overflow. In *Proceedings of the 19th International Conference on Mining Software Repositories*. ACM, 203–214.