Debugging of RxJS-Based Applications

Manuel Alabor
Eastern Switzerland University of Applied Sciences
Rapperswil, Switzerland
manuel.alabor@ost.ch

Markus Stolze
Eastern Switzerland University of Applied Sciences
Rapperswil, Switzerland
markus.stolze@ost.ch

Abstract

RxJS is a popular library to implement data-flow-oriented applications with JavaScript using reactive programming principles. This way of programming bears new challenges for traditional debuggers: Their focus on imperative programming limits their applicability to problems originated in the declarative programming paradigm. The goals of this paper are: (i) to understand how software engineers debug RxJS-based applications, what tools do they use, what techniques they apply; (ii) to understand what are the most prevalent challenges they face while doing so; and (iii) to provide a course of action to resolve these challenges in a future iteration on the topic. We learned about the debugging habits of ten professionals using interviews, and hands-on war story reports. Based on this data, we designed and executed an observational study with four subjects to verify that engineers predominantly augment source code with manual trace logs instead of using specialized debugging utilities. In the end, we identified the lack of fully integrated RxJS-specific debugging solutions in existing development environments as the most significant reason why engineers do not make use of such tools. We decided to elaborate on how to resolve this situation in our future work.

CCS Concepts: • **Software and its engineering**;

Keywords: reactive programming, debugging, empirical software engineering

ACM Reference Format:
Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-Based Applications. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '20), November 16, 2020, Virtual, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3427763.3428313

REBLS '20, November 16, 2020, Virtual, USA

ACM ISBN 978-1-4503-8188-8/20/11…$15.00
https://doi.org/10.1145/3427763.3428313

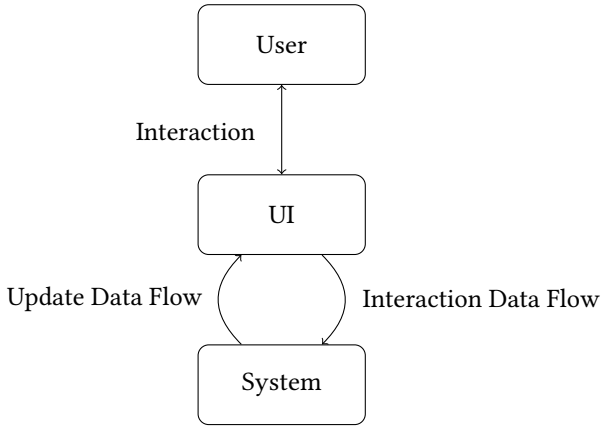

Figure 1. Basic data-flows in a UI Application.

1 Introduction

The (graphical) user interface (*UI* or *GUI*) of an application handles two constant flows of data: External user input (e.g. mouse, touch, or keyboard interaction) is interpreted and forwarded to the system. Once the system processed an interaction and updated its internal state accordingly, it notifies the UI about these changes, which are relayed to the user.

To implement the data-flows as shown in Figure 1 to drive a UI, the Observer design pattern[7] is often used and variations of the pattern are omnipresent today[4].

The Observer design pattern has its roots in the Object Oriented Programming paradigm (*OOP*), hence relies on imperative code constructs to handle a data-flow. Reactive Programming (*RP*) is another approach to realize such flows: It inherits the declarative way of implementing functionality from Functional Programming (*FP*), i.e., data-flows are described rather than implemented step by step[5]. RP functionality is usually available in the form of a library providing necessary abstractions, for imperative as well as declarative programming languages.

According to the IEEE Standard Glossary of Software Engineering, *debugging* is an activity "to detect, locate, and correct faults in a computer program."[1] From interpreting memory dumps, manually adding log statements to trace program execution up to the point where specialized debugging programs can interrupt a running process and interact with it on a low level, debugging utilities took different forms over time.

Modern IDEs and internet browsers ship with their own set of debugging tools. These debuggers are specialized in working with imperative, control-flow-oriented program code. The following example helps us to illustrate the implications of this: Assuming an engineer is inspecting a piece of code and wants to know which part of the program was executed right before. For a program implemented using the imperative paradigm, the call stack gives a clear answer to this question. Hence the stack frames represent each point in the program execution. In a data-flow-oriented program implemented using RP, the stack trace for a transformation function in the flow will not point to its logical predecessor. Instead, the stack frames lead to the internals of the RP runtime environment.

This example demonstrates the limits of a traditional control-flow oriented debugger, which cannot interpret RP abstractions. As a result, these debuggers are not able to give the correct answer to a data-flow-specific inquiry. There have been numerous efforts to provide engineers with improved debugging utilities for RP [17] [18] [6]. However, none of these have seen broad adoption by practitioners yet. To gain a better understanding of the underlying root causes, we conducted interviews with several software engineers and collected "war stories" about the challenges they face in their day-to-day jobs when using RP. Based on this collected evidence, we will validate their statements in an observational study using RxJS and search for an answer to our first research question:

- *RQ1: What challenges do software engineers face when debugging RxJS-based applications?*

In response to this, we are going to present a concept on how to resolve previously identified challenges and answer the second research question:

- *RQ2: How can the experience of software engineers during the debugging process of RxJS-based applications be improved?*

The implementation and validation of these proposals lead to our third and last research question, which will be investigated in our future research:

- *RQ3: What is the impact of proposed solutions on the debugging experience of software engineers?*

We will conclude this introductory section with the clarification of important terms and a view on known RP debugging utilities. Section 2 gives an overview of the insights from the conducted interviews and the collected war story reports. We present our observational study intended to validate results from the interviews and reports in Section 3, which allow us to answer RQ1. Before our final conclusion, we will answer RQ2 in Section 4 "Future Work" and review the threats to validity regarding our study in Section 5.

1.1 Reactive Programming

RP is a declarative programming paradigm that is strongly influenced by FP. While engineers use imperative programming languages to specify every step *how* a program has to do something, declarative languages allow to describe *what* the program should achieve ultimately. A runtime system then figures out a way to satisfy that description and executes it. RP functionality is usually provided in form of a language extension for a specific programming language (e.g. REScala for Scala[17]) or as a library (e.g. RxJS for JavaScript[15])

Either way, both usually provide a (i) domain specific language (*DSL*) to describe data-flow graphs, how they depend on each other and how data flowing through should be transformed. At program execution, a (ii) runtime environment evaluates these descriptions and creates a representation of the specified graphs. It then takes care that values are processed and propagated correctly through them as well as that a consistent system state[5] is always maintained.

1.2 ReactiveX and RxJS

"Reactive Extensions" (*ReactiveX*) is an open-source project. Its members and contributors created a generic description of a RP API. They further provide reference implementations of this API along with RP language extensions for various programming languages like Java, C#, or JavaScript[1]. ReactiveX summarizes the API as "…a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming"[14]. The core concept of the API specification is the Observable[2]: An observable can be composed with other observables to form a data-flow graph. Once an observable gets subscribed, it might push ("emit") an arbitrary number of values to the subscriber until it either completes, fails, or gets unsubscribed again. There is a multitude of operator functions available which allow the transformation of values and composition with other (higher-order[3]) observables. The mechanism of subscribing to an observable is closely related to the Observer design patterns `attach` method.

RxJS[15] is the reference implementation of the ReactiveX API specification for JavaScript. Its current major version 6 is implemented using TypeScript and is used by large projects like Angular[8]. Listing 1 shows an example of RP using RxJS in TypeScript.

The RxJS community uses *marble diagrams* as shown in Figure 2 to document [19] the runtime behavior of an observable visually. Unit test libraries[16] use this abstraction to encode the behavior of mocked observables or to describe assertions.

[1] http://reactivex.io/languages.html

[2] There is no known relation between ReactiveX' concept of the Observable and the deprecated Java class `java.util.Observable`.

[3] An observable emitted by another observable is considered a Higher-Order observable. This naming is related to the concept of higher-order functions in mathematics and computer science.

```
1 import { of } from 'rxjs';
2 import { filter , map } from 'rxjs/operators';
3
4 of(0, 1, 2, 3, 4).pipe( // Create observable
5   filter (i => i < 4), // Omit integers >= 4
6   map(i => i * 2) // Multiply int. by 2
7 ). subscribe(console.log) // Logs: 0, 2, 4, 6
```

Listing 1. Basic RxJS example creating an observable emitting four integers. Each integer is processed by two operators and finally written to the console.

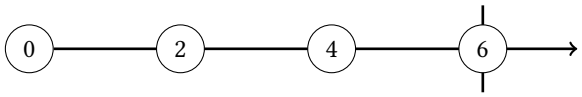

Figure 2. A Marble Diagram visualizing the observable in Listing 1. From left to right, each marble represents an emitted value. The vertical line at the last marble indicates that the observable completed after emitting 6.

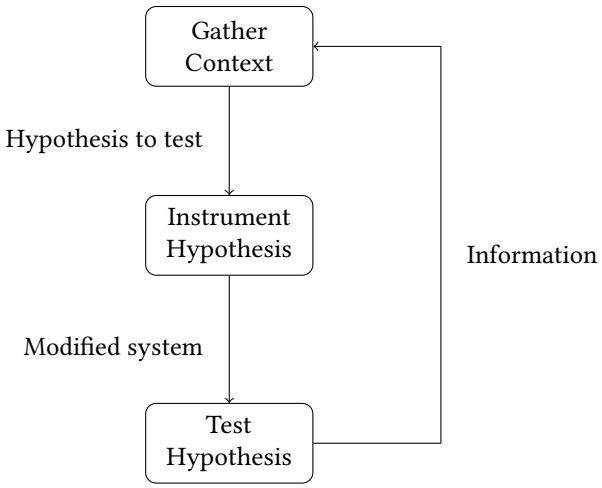

Figure 3. Debugging Process Model after Layman et al. [10]

1.3 Debugging Process Model

Layman et al. [10] use the debugging process model, an iterative hypothesis refinement process, to formalize the general activity of debugging a computer program in their paper.

The process consists of three steps and includes a feedback loop: After the engineer (i) gathered sufficient context information (e.g. ways to reproduce the failure or details about external factors) and understands the situation satisfactory, they generate a hypothesis on the origins of the bug or what impact a change made to the program might have. With the intent to proof their hypothesis, the engineer then (ii) instruments the defective program using suitable tools (e.g. adding log statements, setting breakpoints or removing code parts). Finally, the instrumented system gets (iii) challenged against the formed hypothesis. E.g., code statements are executed step by step using a debugger or trace logs are analyzed and compared against expected behavior. If the hypothesis turns out to be correct, the debugging process stops. If not, the newly gained knowledge about the problem is used to build a refined hypothesis and start a new iteration.

1.4 Debugger Concepts

Software engineers have many tools and utilities at hand, which help them to interact with and gain insight on the behavior of a defective program. Tools range from the instrumentation of source code with trace log statements (manually or automated) to specialized utilities allowing them to directly interact with a program at runtime.

Many of these specialized utilities differentiate themselves fundamentally in regards of the concepts they are built upon. We identified and will use the following three categories to structure them:

Traditional (i) *imperative-focused debuggers* provide the functionality to interact with programs at runtime: Once a breakpoint pauses the program execution, they provide access to the current call stack and the values assigned to variables of a given stack frame. Manual control of the program execution allows inspecting its behavior step by step as well as assigning new values to variables "on-the-fly."

RP provides its own set of challenges to debuggers: Call stacks expose internal invocations of the RP runtime system rather than, e.g., the predecessor transformation according to the data-flow graph. Further, breakpoints can only be used on the imperative parts of transformations and lack the functionality of interrupting execution when the RP runtime hits a specific node within the graph. A (ii) *reactive debugger* can interpret the underlying graph model of a RP runtime system. It leverages on it and provides specialized tools e.g., to navigate, visualize or instrument the data-flow graph [18] [6] [3].

Traditional, as well as reactive debuggers work with the *current* state of a program's execution only. They lack information about what happened before or what is going to happen in the future. This shortcoming is tedious, especially when debugging a problem depending on many complex circumstances in a system. An (iii) *omniscient debugger* [13] [12] does not interact with the executed program directly. Instead, it records runtime telemetry and provides an interface for later inspection. Engineers can "time travel" back and forth through the program execution trace without the hassle of reconstructing a given failure situation over and over again.

1.5 RxJS Debugging Utilities

1.5.1 rxjs-spy. *rxjs-spy*[2] is a logging library specialized on RxJS observables. Once an observable is tagged with an arbitrary string identifier, a monitor can generate trace logs whenever a value is emitted as well as when individual life cycle events occur (i.e. subscribe, unsubscribe, complete and error). Ideally, tagged observables are created during development when the data-flows are composed for the first time. Like `tap`[4], the `tag` operator in Listing 2 is completely transparent to the actual data-flow.

```
1  import { create } from 'rxjs-spy';
2  import { tag } from 'rxjs-spy/operators';
3
4  const spy = create(); // Create monitor
5  spy.log(/ multiply /);   // Log tags matching
6                           // RegEx /multiply/
7   interval (1000). pipe(
8     map(i => i * 2),
9     tag('multiply'),    // Tag with "multiply"
10    map(i => i - 1),
11    tag('subtract')     // Tag with "subtract"
12    take(2)
13  ). subscribe ();
```

Listing 2. Application of rxjs-spy using its `tag` operator on Line 9 and 11.

The data-flow configuration in Listing 2 will produce a trace log as shown in Listing 3 eventually.

```
1  Tag = multiply; notification = subscribe
2  Tag = multiply; notification = next; value = 0
3  Tag = multiply; notification = next; value = 2
4  Tag = multiply; notification = unsubscribe
```

Listing 3. rxjs-spy execution trace log generated by default monitor in Listing 2 on Line 4.

Additional features are available through the library's console interface. E.g., a tagged observable can be paused, so values get collected rather than being emitted immediately. The engineer can then emitted these values one after another manually or resume all of them at once.

1.5.2 rxfiddle. *rxfiddle*, as proposed by Banken et al.[6] is the first reference implementation of their RP debugger architecture for the ReactiveX API specification. They describe a software design consisting of two independent components: The (i) *host instrumentation* augments a ReactiveX API implementation to emit events at runtime (e.g., emitting a value or life cycle events) and forwards them to the second component. The (ii) *visualizer* interprets the events and displays them along two dimensions: The StoryFlow graph[11] shows when an observable is created and how it interacts with other observables, whereas a marble diagram visualizes the values emitted over time for every observable.

The reference implementation supports event processing for the (outdated) RxJS major versions 4 and 5 only and is available as an online application[5]. A proof-of-concept implementation working on a local computer is available through the projects Git repository[6].

1.5.3 RxViz. *RxViz* is a visualizer utility available online[7]. It is an "animated playground for Rx observables"[3] and allows the visualization of RxJS observables using marble diagrams. Engineers implement or copy-paste data-flows in an editor window using JavaScript. A diagram is generated based on this code over a configurable time interval. The diagrams are rendered immediately and are available as downloadable SVG files.

1.5.4 rxjs-playground. Building on the basic concept of marble diagrams, *rxjs-playground*[8] is a sophisticated sandbox to simulate and visualize RxJS observables interactively in the browser. Users define editable and computed observables, represented as vertical marble diagrams: Values emitted by an editable observable can be created and modified either by interacting with its marble diagram directly or using a simple JSON syntax. The behavior of a computed observable is controlled by implementing functionality with TypeScript in the provided editor.

rxjs-playground renders the values and life cycle events for all observables in real-time, allowing quick iterations on a specific piece of code.

2 Interviews and War Stories

On the way of finding our interview partners and war stories reporters, we noticed it to be a challenge to find people who understand themselves as users of RP and related technologies. E.g., even though Angular makes heavy use of RxJS, we will see that many engineers do not directly interact with its abstractions when building "basic" UIs. In the end, we were able to conduct interviews with five engineers and collect reports on hands-on experiences from another five.

2.1 Interviews

We organized informal interviews, which allowed us to gain insight into how software engineers work with RP in their daily jobs. We talked to five engineers (following identified using the codes *I1* through *I5*) and asked them about the

[4] RxJS' `tap` operator is used to execute a side effect whenever an observable emits a value. It cannot modify or influence the emitted value nor the observable in any way.

[5] https://rxfiddle.net/

[6] https://github.com/hermanbanken/RxFiddle

[7] https://rxviz.com/

[8] https://hediet.github.io/rxjs-playground

technologies they use, what their personal experience with RP was, what they most liked and most disliked about it. We used video chat to conduct all interviews remotely. The interview with I4 was done in English, all others in Swiss-German. Therefore, quotes by I1, I2, I3, and I5 are translated statements.

Our first three interview partners I1 to I3 stated to work currently or more recently have worked with RxJS in conjunction with Angular and ngrx[9] to develop frontend web applications. I4 was a proficient RxJS user. Our fifth interview partner I5 was a backend engineer who used akka-streams[10] in Scala to model data-flows for a WebSocket-based[11], reactive API layer serving a web frontend application.

All interview partners pointed out that they like RP because it provides them with "[...] a good way for composing multiple data sources" (I1) and, combined with "[...] a statically typed language, RP guarantees some kind of basic formal correctness of a program" (I5). Hence a significant strength of RP seems to be the ability to describe complex data-flow constructs using a specialized DSL. However, they also pointed to current challenges: The learning curve can be steep for a novice engineer: "Being challenged with new abstractions [of Angular and ngrx] already, I experienced RxJS concepts and operators to be hard to convey" stated I3, giving lectures in frontend web application development.

It was interesting to hear that, especially in the area of developing web applications using Angular, our partners seemed not to have to work with pure RxJS code often. E.g., when using ngrx for state management, "The framework hides observables from its main API surface carefully, so you do not have to interact with them directly" (I2). As soon as our interviewees had to extend built-in functionalities with own features, e.g., a new effect[12], I1 and I2 valued the possibility to interact with underlying observables though.

When asked explicitly about what they dislike the most about RP, all interview partners, with no exception, emphasized the debugging process of an RP program as unsatisfactory. The fact that our interviewees remembered the search for a bug as something negative did not surprise, hence a bug is commonly something negative afflicted. It was remarkable though that statements like "In 99% of all cases, I add console.log statements manually and run the program over and over again, trying to understand what is happening" (I1) were prevalent and showed *why* our partners dislike RP debugging in particular. I1 to I3 mentioned the Redux DevTools[13] as particular helpful when debugging Angular/ngrx applications nonetheless. Further, I1 noted marble diagrams as valuable in order to understand how an RxJS observable works, whether during development or debugging.

[9] https://ngrx.io/
[10] https://akka.io/
[11] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[12] https://ngrx.io/guide/effects
[13] https://chrome.google.com/webstore/detail/redux-devtools

2.2 War Stories

After we built an intuition for how software engineers work with RP by evaluating the interviews, we were interested in more RxJS-specific, hands-on experiences. We asked the engineering community via Twitter[14] about their personal, most recent RxJS debugging war story and sent out various emails with the same request. After reaching out, we were able to collect five responses in English: One by an RxJS core team member (R1), two from Angular Google Developer Experts (R2/3) and another set of reports by two software engineers (R4/5) building web and mobile applications using React and RxJS, which includes the author of this paper.

In their report, R3 focused on how they built code with improved testability because of recent changes in RxJS: "[...] in the beginning, it was very hard to write asynchronous tests [...]. I really disliked [...] you were forced to pass a TestScheduler". Allowing to pass the scheduler explicitly as parameter forced them to introduce code, which was only necessary for testing reasons they stated further. With its current major version, RxJS 6 improved profoundly on the `TestScheduler`. The runtime environment itself can be augmented with the scheduler now, which results in cleaner code.

Even though the share of non-productive, testing-related code necessary to build mature RxJS-based applications was mentioned to have decreased today, R2 as well as R4 and R5 commented on the common practice of manually modifying production code during debugging sessions, hence confirming earlier statements from our interview partners. R5 described a specific scenario where they suspected a problem within a complex observable composition: Having multiple asynchronous, remote data sources, they used observables to model the dependencies between them and implemented computations on their results as operators. On top, each data source could re-emit updated versions of previously requested information at any time. After a week in production, though tested thoroughly, the first of many bugs got reported: "Displayed numbers kept changing where we did not expected them to. In other places, they were not rerendered where they were supposed to, e.g. after we changed them in the system," they told us. The browser's debugger tools and its breakpoints did not help much since the operators were executed several times. Other parts of the stream were impossible to get a handle upon, even with conditional breakpoints. "I started to inspect the flow [...] with console.logs and later also using tags from rxjs-spy which exposed more detailed life cycle information." After a time-consuming log analysis, they finally were able to resolve all bugs. The log statements added were removed in the aftermath, though the rxjs-spy-related code was left in the observable stream in case they might be needed again in the future.

[14] https://twitter.com/swissmanu/status/1242429409208029185

In a related war story, R2 discloses similar invasive practices using an external tool when they implemented logic to request and cache batches of a remote resource: "It took quite some time to get it right and one of the most invaluable tools proved to be Stackblitz[15] which gave us the ability to quickly create smaller working examples and iterate on them." This sandboxed setting allowed them to run, debug, and iterate on selected pieces of a larger observable stream. Even though the final result had to be integrated back into the actual application, the extra effort was worth the result the engineer concluded in their report.

A final story describes yet another way of utilizing an external tool: Rather than using a dedicated sandbox to develop pieces of a more complex system, R5 used Rx Visualizer, an online visualization utility to generate marble diagrams from real code. Like in the report before, it was necessary to extract parts from the codebase of the actual application. Once done, the visualizations helped to understand when values got emitted, when subscriptions changed and when observables completed: "Marble diagrams were a huge help in order to understand detailed runtime and life cycle behaviors of the observable."

2.3 Insights

Software engineers value how they can describe data-flows using a RP DSL, even though the learning curve was perceived as steep. We heard some interesting reports on how RxJS is applied in a daily development environment: Marble diagrams help to understand an observables behavior and are useful to implement tests. Large frameworks like Angular hide some of the complexity of RxJS but allow engineers to make use of its full power once the pre-provided functionality needs to be extended.

Most participants considered a statically typed language like TypeScript, a fundamental necessity allowing them to implement data-flow graphs with minimal, formal correctness, as we heard in the interviews. When the engineers needed to interact with data-flow graphs at runtime, e.g. to debug the behavior of a specific part of a stream, we noticed throughout most reports that they were not 100% satisfied with the feature set they were provided by traditional debugging tools. Almost all of the reporters referred to the practice of modifying their source code manually and adding trace log statements where they assumed a problem to overcome the feature gaps in traditional debuggers. Listing 4 exemplifies two challenges when debugging a stream of observables with imperative-focused debugging tools.

Where a breakpoint can easily be added to Line 2 within the arrow function, this is impossible for `take` on Line 3. One would need to place the breakpoint within the operator's internal implementation instead, which can be cumbersome

[15] Stackblitz is a full JavaScript development environment available online https://stackblitz.com/

```
1 interval (1000). pipe(
2   map(i => i * 2),
3   take (4),
4   tap(console.log)
5 ). subscribe(showValue); // Emits: 0, 2, 4, 6
```

Listing 4. An observable emitting a sequence of increasing integers every second. Traditional breakpoints are possible inside the arrow function on Line 2. Though a breakpoint can be added on Line 3, it will never be hit during the actual execution of the `take` operator. Line 4 shows a manually introduced trace log statement using the `tap` side effect operator.

in case the operator is used in a different stream as well. Once the breakpoint on Line 2 interrupts the execution of the program, we will notice another shortcoming related to this circumstance: Rather than representing the logical flow implemented using the DSL, the call stack as shown in Listing 5 points deep into RxJS' internal implementation. That is why a traditional debugger's step controls cannot operate on the data-flow graph; It just can not interpret this level of abstraction.

```
1 <anonymous> RxJS
2     rxjs  6.5.2/ internal/operators/map.js:49
3     rxjs  6.5.2/ internal/Subscriber.js:66
4     rxjs  6.5.2/ internal/observable/interval.js:23
5     // ...
```

Listing 5. A call stack showing the internal RxJS execution stack for a breakpoint in the arrow function on Line 2 in Listing 4.

We learned that simple trace augmentation, like on Line 4 in Listing 4, logs emitted values only. Such trace logs might be helpful when debugging simple graph compositions. Though, they lack life cycle information of the underlying observables completely. The importance of such information was emphasized by R5 describing their usage of rxjs-spy: To know, when an observable gets subscribed and unsubscribed, when it completes or fails, helped them to solve complex problems multiple times. When dealing with higher-order observables, they value this information even as indispensable.

Finally, we understood that if a problem is hard to replicate within the actual application, the engineers use external sandbox development environments to isolate specific parts of an observable composition. These allow them to iterate on it faster than it would be possible otherwise.

After the evaluation of all reports and interviews, we speculate that software engineers truly lack, but also seldomly use debugging tools that can handle RP concepts provided by RxJS. Even though traditional debuggers might help to

some extent, they do not provide all the information an engineer requires in a particular situation. Instead, they turn to manual trace log augmentation and extraction of source code as we saw repeatedly.

3 Validation

Almost all participants from our interviews and war story reports showed a tendency to manually modify source code with trace logs during the hypothesis instrumentation phase when debugging RxJS code. This practice is often not perceived as efficient since the evaluation and interpretation of trace information tends to be cumbersome and very time-consuming. Also, removing log statements after a successful debugging process might leave new bugs in production code if not done carefully. Like Banken et al. [6] before, we identified this technique as one of the primary debugging practices when software engineers work with RxJS-based code.

That is why we saw demand in validating this statement and previous findings about manual code modification for debugging reasons with an observational study. Our study sought to validate the following hypothesis:

- *Hypothesis: If software engineers must solve an RxJS-based problem, then they will instrument the code manually in order to understand its behavior.*

3.1 Study Design

The subjects for our study were required to have experience in developing applications with RxJS. We recruited four subjects willing to participate in our experiment. We were interested in seeing how the subjects apply debugging techniques they would use in everyday situations in their jobs. Hence we decided to conduct the experiment in a somewhat uncontrolled environment where the subjects used their own devices with their development environments of personal preference. Our objective for the experiment was communicated as broad as possible to prevent bias: "We are interested in how *you* debug a problem" did not mention our hypothesis by intention.

We planned to have a one-hour session for the actual experiment with each subject, followed-up by an unattended after-action survey. We executed the experiment in two consecutive blocks of 25 minutes each. We provided a ZIP file[16] containing the source code for two frontend web applications implemented using TypeScript and RxJS along with a Jest test suite at the start of a session. Each of these applications was rigged with two to three bugs, which we asked the subjects to identify and fix using whatever debugging techniques they prefer and commonly use. Where the first application required less complicated intervention to resolve the contained bugs, the second application demanded substantial modifications in the data-flow as it made heavy use of higher-order observables. The provided test suite allowed the subjects to understand the functional requirements of each application as well as to quickly verify their changes to be successful (or not).

A block was considered as complete once the test suite signaled all bugs as resolved, or the 25 minutes expired. We asked our subjects to act like in a pair programming situation where they "think out loud" their thought process. Though we refrained from answering any question related to the "where" a bug has to be expected.

We sent out the participant briefing document to all of our subjects a week before the experiment. We outlined the course of action and provided them with an example ZIP file. This file contained the same setup as the file provided at the experiment and allowed the subjects to get accustomed to things like starting the web applications or running the test suites.

We decided to monitor our subjects' progress remotely using voice chat and screen sharing due to the COVID-19 situation at the time of our study. Furthermore, this allowed us to record the sessions with relatively low technical effort for later evaluation.

The after-action survey[17] was provided within 24 hours after a subject's participation in the main part of the study. We asked the subjects about (Q1) if they currently use RxJS on or off their jobs, the (Q2) number of years they have experience with RxJS, in (Q3) which field (like frontend, backend or others) they use RxJS and finally (Q4) which tools and techniques they use to debug RxJS-based code. The respective answers allowed us to put the observed actions into perspective and detect potential irregularities in case a subject acted differently as they would have in a "real" situation.

3.2 Study Execution and Results

After the subjects got themselves accustomed to the application provided and understood its purpose, all of them used the test suite to gather context about what features do not work as expected initially. Further, all of them tried to recreate the failing behavior in the UI manually. We could not observe any of the subjects using external tools, e.g. RxViz, to inspect specific code parts in isolation in later iterations of the debugging process. Though, S4 noted that they would have usually started to decompose the problem into smaller pieces and observe their behavior in specific after the 25 minutes of the second block expired.

While all subjects added manual trace log statements to existing arrow functions or by adding `tap` operators in the instrument hypothesis phase, none of them used additional libraries like rxjs-spy for doing so. S2 and S4 used the traditional debugging tools provided by their browser or IDE to

[16] https://github.com/swissmanu/mse-pa1-experiment/archive/v1.0.2.zip

[17] https://github.com/swissmanu/mse-pa1-experiment/blob/f70102885be86fb2323b9516005e1d6dfeb9795b/after-action-survey-questions.md

Table 1. Observed practices and tool usage per subject.

Subject	Trace Logs	Debugger	Add. Tools
S1	X		
S2	X	X	
S3	X		
S4	X	X	Next step

Table 2. Results per subject for each presented problem.

Subject	Problem 1	Problem 2
S1	Time expired	Time expired
S2	Time expired	Time expired
S3	Time expired	Time expired
S4	Solved	Time expired

add breakpoints. Both of them commented on the inability of stack traces to interpret RP abstractions as unsatisfying. We could further observe a "trail-and-error" approach in later iterations of the debugging process. The subjects started to introduce modifications to the system, which they immediately tested against their latest hypothesis. Table 1 provides an overview on the complete collected data regarding used techniques and tools.

Only S4 was able to solve the first problem given, as shown in Table 2. None of the subjects was able to successfully identify and fix the bugs hidden in the second problem within time.

The survey responses available in Table 3 showed that S2, S3 and S4 had two or more years of experience with RxJS. Where all of them use RxJS to develop frontend applications, S4 declared having used RxJS for backend development as well. When asked what tools they usually use for debugging, S2, S3 and S4 stated to use the traditional debugger of their IDE. S1 and S3 leverage additional tracing functionality of rxjs-spy, and all four of our subjects use manual log statements.

3.3 Interpretation

We were able to observe how all subjects predominantly used manual source code augmentation by adding trace logs. Two of the subjects used traditional debugging utilities in order to inspect the program's state at runtime in addition. All subjects used the new information gained to refine their hypothesis about underlying problems before starting a new iteration in the debugging process. We could not observe the extraction to and reintegration from an external tool. All subjects exhibited the debugging behavior described in our hypothesis. Further, we could verify previous results by Banken et al. [6] successfully as well.

Table 3. After-action survey responses per subject.

Subject	Q1	Q2	Q3	Q4
S1	Yes	1 year	Frontend	Trace Logs, rxjs-spy
S2	No	> 3 years	Frontend	Debugger, Trace Logs
S3	Yes	2 years	Frontend	Debugger, Trace Logs, rxjs-spy
S4	No	2 years	Frontend, Backend	Debugger, Trace Logs

Even though S1 and S3 stated in the after-action survey to regularly use rxjs-spy for debugging RxJS programs, neither of them made use of this library during the experiment part of the study.

Interviewing professionals, consolidating RxJS hands-on experiences from the war stories, and evaluating the results from our observational study showed us that software engineers use a variety of practices, tools and utilities to debug RP programs. Beside the habit of adding trace logs manually, we saw them evidently trying to answer their debugging hypotheses using traditional, imperative-focused debugger utilities. The later way of debugging was repeatedly commented as unsatisfying as these utilities cannot handle RP constructs, and with this, cannot help to detect problems located within these at all. The former way, the introduction of manual log statements, was both described as the prevalent way of debugging RxJS or as "the last resort" when no other debugging technique helped before.

We heard further how engineers isolate specific observables from bigger data-flows and how they inspect those in sandboxed environments and visualizers. This helps them understanding the observable life cycle and value emitting behaviors better and iterate faster in order to resolve problems.

More than 50% of our 14 peers throughout the interviews, war story reports and the experiments after-action survey stated to know about specific RxJS RP debugging tools. It was apparent that all subjects during the observational study refrained from using any of them, though. It is our speculation that the subjects knowing about specific tools held themselves back from using them because they perceived the effort of setting them up (e.g., installing and configuring rxjs-spy) as too time-consuming. Not having the "right" tool available without significant additional effort is also what we interpret from the statement by S4: Though they would have started to extract parts of the data-flow and inspect it with other tools, they would have done so only after the 25

minutes of the block expired; Hence a more accessible way allowing such analyses would have influenced the behavior of the subject.

The best RP debugging tools are useless if either the hurdle to use them is too high, or engineers do not understand which particular part of the debugging process they can benefit from them. Salvaneschi et al. [18] provided in their previous study on the *Reactive Inspector* for *REScala* evidence on the effectiveness of a fully integrated RP debugging solution, which supports developers in their daily work using the Eclipse IDE. Hence, we can postulate an answer to our first research question RQ1: The most significant challenge software engineers face when debugging RxJS-based programs is to know *when* they should apply *what* tool to resolve their current problem in the *most efficient* way.

4 Future Work

We see the biggest shortcoming of current RxJS-oriented debugging solutions like rxjs-spy, RxFiddle, or RxViz in fact that they are not integrated in established development environments (e.g., IDEs or internet browser developer tools). This leads to the practice of manually augmenting code itself rather then working with it in a less obtrusive, fully integrated way as we were able to proof in our observational study. Using specialized utilities is an extra effort an engineer has to invest every time they want to debug a data-flow: Either tagging an observable for rxjs-spy or extracting parts of it to an external environment, all of these practices require engineers to "go the extra mile" in order to inspect the runtime behavior of an RxJS-based application. The additional effort might be neglectable when treating a rather complex data-flow composition. However, it holds back engineers from applying the tools to simple observables like in the first block of the experiment we conducted.

The observation that two out of four of our study subjects tried to debug an RP application with traditional, imperative-centric debugger utilities, as well as related statements from the interviews and war stories, strengthened our assumption regarding tool integration. Engineers expect the debugging tools they know and rely on to give correct insight on every program, no matter the paradigm (imperative or declarative) with which it got implemented.

This leads us to the answer to our second research question RQ2: We want to improve the experience of debugging RxJS-based applications by providing RP specific debugging utilities where software engineers expect them the most: Fully integrated with the traditional debugger they know from their IDE or browser developer tools[18].

[18] A positive example of such a seamless integration is the debugger of the Google Chrome developer tools: It combines call stack frames of asynchronously executed functions[9] seamlessly with those of synchronously executed code. This provides software engineers with a better understanding about which part of the program triggered the statement they currently inspect.

The answer on which RP debugging tool exactly (e.g., a full reactive debugger or a visualizer using marble diagrams) we are going to integrate, how such integration will look like in detail, how it will support engineers in a particular step of the debugging process, as well as the answer on RQ3 will be part of our future work on the topic of "Debugging RxJS-based Applications".

5 Threats to Validity

This study is subject to the following threats and limitations:

5.1 External Validity

The data we collected from interviews, war story reports, and the observational study is based on a sample population with 14 individuals. Hence, the results we conveyed in this paper are not representative and are not transferable to the entire software engineering population.

5.2 Internal Validity

The observational study was executed in an uncontrolled environment. All subjects used their personal computers, running their own software development environments. We have no comparative data to measure how this design influenced the observed outcome, assuming that this setup diminishes the reproducibility of the experiment.

We noticed that the subjects needed time to understand the intention of the applications they were provided with before they were able to start with the actual debugging process. Since the amount of required time was different from subject to subject, we suspect it influenced the result of the experiment.

5.3 Construct Validity

The time limit of 25 minutes per experiment block bears the potential to put the subjects under time pressure. This risk might explain why we could not observe any more time-consuming debugging techniques (e.g., installing additional utilities like rxjs-spy) during the study.

6 Conclusion

In this paper, we have explored how software engineers debug data-flow-oriented programs implemented using RxJS. We presented an observational study to validate a hypothesis based on the outcome of ten individual interviews and hands-on experience reports from software engineering professionals. More than 50% of the 14 engineers we worked with during our research told us that they know of the existence of specific debugging tools for RP with RxJS. Nonetheless, the experiment conducted with four participants allowed us to prove that engineers augment source code manually with trace logs instead of using such specialized utilities.

We identified the fact that RxJS specific debugging tools are not tightly integrated with existing, traditional debuggers in IDEs and the developer tools of internet browsers as

the main reason why software engineers do not use them more often. In order to lower the effort necessary to use specialized RP debugging tools for engineers, we declared the integration of such as the matter for our own future research.

Acknowledgments

We want to thank the engineers who participated in our study for their time.

References

[1] 1990. *IEEE Standard Glossary of Software Engineering Terminology.* https://doi.org/10.1109/IEEESTD.1990.101064

[2] 2019. *An example using the console API | rxjs-spy.* Retrieved 17-May-2020 from https://cartant.github.io/rxjs-spy/ Versioned as https://github.com/cartant/rxjs-spy/tree/2bffdee2d5f712d70583ef48297446bd31a9a6f4.

[3] 2020. *RxViz - Animated playground for Rx Observables.* Retrieved 16-May-2020 from https://rxviz.com/ Versioned as https://github.com/moroshko/rxviz/tree/51a737717a27f15b68f907b2329f7b0b6b11cb2b.

[4] Manuel Alabor. 2019. Reactive Applications in Frontend Engineering Today. (2019). https://github.com/swissmanu/mse-seminar-reactive-applications-in-frontend-engineering-today/releases/tag/v1.0.1.

[5] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666

[6] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging Data Flows in Reactive Programs. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE âĂŹ18)*. Association for Computing Machinery, New York, NY, USA, 752âĂŞ763. https://doi.org/10.1145/3180155.3180156

[7] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software.* Pearson Education India.

[8] Google. 2020. *Angular - Observables in Angular.* Retrieved 24-Apr-2020 from https://angular.io/guide/observables-in-angular Versioned as https://github.com/angular/angular/blob/64ac1062489bbc97a0d4b95af5ce9566091fe044/aio/content/guide/observables-in-angular.md.

[9] Google. 2020. *JavaScript Debugging Reference.* Retrieved 16-Aug-2020 from https://developers.google.com/web/tools/chrome-devtools/javascript/reference#call-stack Versioned as https://web.archive.org/web/20200713153259/https://developers.google.com/web/tools/chrome-devtools/javascript/reference.

[10] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. 2013. Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 383âĂŞ392. https://doi.org/10.1109/ESEM.2013.43

[11] Shixia Liu, Yingcai Wu, Enxun Wei, Mengchen Liu, and Yang Liu. 2013. StoryFlow: Tracking the Evolution of Stories. *IEEE Transactions on Visualization and Computer Graphics (Proceedings of IEEE InfoVis 2013)* 19, 12 (2013), 2436–2445.

[12] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record And Replay For Deployability: Extended Technical Report. *CoRR* abs/1705.05937 (2017). arXiv:1705.05937 http://arxiv.org/abs/1705.05937

[13] G. Pothier and E. Tanter. 2009. Back to the Future: Omniscient Debugging. *IEEE Software* 26, 6 (2009), 78–85.

[14] ReactiveX. 2020. *ReactiveX.* Retrieved 24-Apr-2020 from http://reactivex.io/ Versioned as https://web.archive.org/web/20200419004415/http://reactivex.io/.

[15] ReactiveX. 2020. *RxJS - Introduction.* Retrieved 15-Aug-2020 from https://rxjs.dev/guide/overview Versioned as https://github.com/ReactiveX/rxjs/blob/46e35f71b02d02c5a7d7f426e78eadd625c1a67a/docs_app/content/guide/overview.md.

[16] ReactiveX. 2020. *RxJS - Testing RxJS Code with Marble Diagrams.* Retrieved 16-May-2020 from https://rxjs-dev.firebaseapp.com/guide/testing/marble-testing Version 6.5.5-local+sha.7e4589a1.

[17] 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* (Lugano, Switzerland) *(MODULARITY âĂŹ14)*. Association for Computing Machinery, New York, NY, USA, 25âĂŞ36. https://doi.org/10.1145/2577080.2577083

[18] Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE âĂŹ16)*. Association for Computing Machinery, New York, NY, USA, 796âĂŞ807. https://doi.org/10.1145/2884781.2884815

[19] Andre Staltz and Contributors. 2019. *RxJS Marbles.* Retrieved 16-May-2020 from https://rxmarbles.com