Vocal Reactive Programming: Enabling RxJava

Gustaf Eknefelt
Stefan Nilsson

Final Project
Main field of study: Computer Engineering BA (C)
Credits: 15
Semester/year: Spring 2022
Supervisor: Awais Ahmad
Examiner: Felix Dobslaw
Course code/registration number: DT133G
Programme: Software Engineering
Vocal Reactive Programming: Enabling RxJava

Gustaf Eknefelt and Stefan Nilsson

Department of Computers and Systems Sciences
Mid Sweden University
Östersund, Sweden
{gusek1900, stni1003}@student.miuun.se

Abstract—Previous research has shown that the reactive paradigm is suitable for programming by voice, due to its frequent use of expressions used in common English. However, the software used in the previous research (Talon) does not support reactive programming. To our knowledge, no other programming-by-voice software has built-in support for RxJava. The main focus of this study is the development and testing of a vocal programming environment that recognizes RxJava operators and commands. This is done by writing custom scripts that form a RxJava-adapter for a software called Serenade. The purpose of the adapter is to determine to what degree customization can improve the programming-by-voice experience, in terms of reduced cognitive and vocal load. The testing process also reveals challenges related to programming-by-voice in reactive Java, as well as challenges related to creating custom voice commands for Serenade. The results indicate that Serenade is superior to Talon, requiring on average 45% fewer words to produce the same code. When the default version of Serenade is compared to the custom version with custom voice commands, the improvement is 19%. The study also concludes that there are many challenges facing speech recognition when used in a programming environment. The custom script feature in Serenade, while still having some limitations, is advanced and allows the user to create complex custom voice commands.

Index Terms—Programming by voice, Reactive programming, RSI, Serenade, Speech recognition, Talon

I. INTRODUCTION

Voice user interfaces (VUI) [1] where you control devices through voice commands have grown to become a common part of many people’s lives with the advancement of smartphones and virtual assistants such as Amazon Alexa, Google Home, and Apple’s Siri. As the technology improves and people get used to the experience, it makes sense to utilize it in other areas as well. One area where studies have shown potential is within the programming industry.

Computer input in modern times has generally been through the keyboard but studies have shown that programming-by-voice can be both faster [2], [3] and less error-prone [4]. This is partly because the software can write large amounts of code instantly without errors, as long as the speech recognition is able to understand the command. Programming by voice can also take some workload away from hands and arms, giving relief to people suffering from a repetitive strain injury (RSI), such as mouse shoulder, carpal tunnel syndrome, or tendinitis [5]. Although carpal tunnel syndrome might not be directly caused by prolonged computer use, it seems to exacerbate the symptoms [6]. This is important for programmers since typists and people who use visual display units are among the professions with the highest risk of repetitive strain injury [7]. It also opens up new possibilities for people with physical disabilities that are not able to use a regular keyboard. Better accessibility would not only expand the industry but also allow programmers to continue their careers after a traumatic disabling injury which would be beneficial for not only the programmer but also the company and all of society.

Alongside the development of voice user interfaces, there has been a renewed interest in reactive programming which is a declarative programming paradigm focused on data streams. This push has partly come from the development of the ReactiveX library [8] that has been implemented in numerous languages such as RxJava, RxJS, and Rx.NET. The main advantages of an event-driven non-blocking application are found in the Reactive Manifesto [9], describing reactive programming as Responsive (respond in a timely matter), Resilient (stays responsive in the face of failure), Elastic (stays responsive under varying workload), and Message Driven (relying on asynchronous message passing).

In short, reactive programming is a programming paradigm and RxJava is a framework for Java that enables reactive programming. More details will follow in the Background section.

Research about programming-by-voice in the reactive paradigm is limited, however, one study from 2021 [10] compared programming-by-voice in the reactive and imperative paradigms. The results showed that reactive programming had advantages over imperative programming, but faced some major challenges, mainly because the software used for programming-by-voice had poor support for the reactive paradigm. The authors concluded that vocal programming could be further developed to work with the reactive paradigm by creating commands specifically for RxJava.

The software used in the previous study was Talon [11]. In our study, we will be using Serenade [12]. Serenade has, just like Talon, no dedicated support for RxJava. It does however provide the user with an API (Application Programming Interface) for creating custom commands.

In this study, we create a library of commands specifically customized for RxJava in order to make programming-by-voice in the reactive paradigm a smoother experience. There are currently (to the best of our knowledge) no studies that examine the efficiency of programming-by-voice in RxJava with custom commands. The previously mentioned study [10]
shows that programming-by-voice in the reactive paradigm has the potential for being useful, with some further improvements. In our study, we will determine how efficient programming-by-voice in RxJava can be, with the right customization.

II. PURPOSE AND CONTRIBUTIONS

There are many challenges that must be dealt with in order to make the programming-by-voice experience smooth and effortless. Based on previous research, we have identified that reactive programming appears to be suitable for programming-by-voice, but requires customization.

Serenade is a programming-by-voice software that enables the user to create custom scripts for voice commands. The purpose of this study is to determine to what degree the programming-by-voice experience in the reactive paradigm can be improved using Serenade’s custom-script feature.

By default, Serenade does not recognize RxJava. It does not know how reactive operators can be used to create a chain of operations. This makes the coding rather cumbersome. If we can teach Serenade to recognize these operators and in what context they are used, it may improve the experience tremendously. To our knowledge, no research has been done previously that includes Serenade’s custom-script feature.

The motivation for this study is to improve programming-by-voice, specifically in the reactive paradigm. By determining the challenges and limitations, future studies can focus on dealing with those. We believe that programming-by-voice can become smooth and easy, but it requires research into, and elimination of, the main difficulties.

RQ1 What are the current limitations for vocal Reactive Programming?

In order to fully understand how to improve the programming-by-voice experience in reactive Java, we need to understand the limitations and challenges facing speech recognition in a programming environment, in particular when it comes to RxJava-specific code.

Since the language used when programming differs from natural speech, we hypothesize that we will encounter problems when using words and phrases that are non-existent or uncommon in regular English. Also, programming uses abbreviations and certain naming conventions that may be problematic for a speech recognition engine to understand.

RQ2 To what effect can vocal programming tools be customized to better accommodate Reactive Programming and improve developer experience?

The most important research question of the study is to determine how well Serenade can be customized to improve the programming-by-voice experience in RxJava. How flexible and advanced is the custom-script feature in Serenade and how much of an impact do custom scripts have on the experience and the efficiency?

We do not doubt that there will be some improvement. How large the improvement will be is hard to estimate beforehand. It is reasonable to think that the examples with more advanced RxJava-features will see a greater improvement compared to the more simple examples. Even if there is little improvement in terms of words required to produce the code, it is likely that the custom programming environment feels more natural and intuitive, since the user will have chosen keywords and commands themselves. This is hard to measure and is highly subjective.

RQ3 What are lessons learned regarding challenges for customizing vocal programming tools in general and Reactive Programming in particular?

Challenges, problems and limitations with the custom script feature in Serenade will be documented so that these problems can be addressed by developers looking to improve programming-by-voice software.

Scientifically, this study aims to contribute with details regarding the main difficulties when programming-by-voice in RxJava, how those difficulties can be dealt with, and what difficulties still remain. Furthermore, we will research the challenges in terms of customizing RxJava in Serenade, but also what areas of RxJava can be dramatically improved by customization. Finally, we will discuss our conclusions and propose future research.

Technically, we will provide an introduction to Serenade, its features and limitations. We will mainly focus on the custom-script feature with its advantages and limitations. Our main technical contribution is the scripts written in Javascript that make up the custom voice commands for programming in reactive Java using Serenade. This adapter enables anyone programming vocally in RxJava using Serenade to increase their productivity by reducing vocal and cognitive load.

III. BACKGROUND

Programming-by-voice is performed using software that utilizes speech recognition in order to convert speech to text. Previous research have studied Talon [11] and VoiceCode [13], but recently Serenade [12] emerged as an interesting alternative. Serenade has several features designed to help the user code more efficiently. It supports adding, navigating, and editing code, as well as system commands such as raw keyboard and mouse input and application control.

Several studies have shown that speech recognition is more difficult for programming tasks than natural language [5], [14]. This is due to the fact that when programming, only certain keywords are recognizable by regular speech recognition engines. Many words are names of variables and functions, which often need to be spelled out. Additionally, code syntax usually consists of many special characters, complicating the coding process even more. Tools such as Serenade need to be carefully designed to recognize the syntax of common programming languages.
Voice User Interfaces (VUIs) [1] are interfaces that allow the user to interact with a system using their voice. Programming-by-voice is a VUI, as well as Google Home and Siri. One of the difficulties with VUIs is that not only does the VUI need to understand spoken language, but the user also needs to know what voice commands the VUI can and cannot understand.

Serenade is a free software that uses speech recognition to convert speech to code, and it is the software we decided to use in this study. Serenade has a feature that allows the user to create custom scripts, which is a requirement for this study. It supports many IDEs (Integrated Development Environment) and its protocol is open source, allowing anyone to write a plugin for an IDE or for another application. It also supports a wide range of programming languages and it recognizes the language automatically. The custom script feature means the user can teach Serenade to understand voice commands it otherwise would not. This study will mainly focus on this feature and determine if it can be used to successfully, and to a large degree, improve Serenade’s understanding of RxJava-code.

Reactive programming has many definitions, and the concept can be rather abstract. It revolves around streams of data and emitters of events and it provides an efficient means to handle data updates whenever a certain entity requests it. It can be thought of as multiple concurrent streams of data that can be accessed in real-time by the entities that request it. RxJava deals with this using observables, observers, and publish/subscribe. Reactive programming also enables data streams to be easily sorted, filtered, and combined [15].

We chose the reactive paradigm for this study because it has advantages over the imperative paradigm in terms of complexity, concurrency, event handling and exception handling [15], [16]. RxJava is also closer to the natural English language, with more full words rather than having to spell words out letter by letter [10], making it more suitable for programming-by-voice.

RxJava is a framework that enables reactive programming in Java. It is a Java-based extension of Reactive Extensions, also called ReactiveX. ReactiveX was originally created as a reactive framework for .NET, but was quickly ported to many other languages, including Java. The first version of RxJava, the Java-based extension of ReactiveX, was released in 2014, followed by version 2.0 in 2016 [15].

In the study we will mention vocal and cognitive load. Vocal load is measured as the number of words required to complete a certain task, whereas cognitive load is the mental workload required to complete a certain task.

IV. RELATED WORK

There are numerous studies looking at the performance of programming by voice [2], [3], [17]–[19] however only one has looked at programming by voice related to the reactive paradigm [10].

A. Speech recognition

Back in 2006 Begel and Graham [17] conducted a study to make an assessment of speech-based programming environments. In the study, they conducted two experiments with their speech-enabled Java editor to find out how programmers use Java voice commands when programming. The first experiment made use of Dragon NaturallySpeaking as a speech recognizer to see what could be done with state-of-the-art voice recognition with minimal training. The second experiment replaced the speech recognizing software with a human non-programmer to assess what was possible with near-perfect voice recognition. They found that accuracy problems and low interpretation speed in the voice recognition software caused participants to use a stop-and-go pattern where they introduced pauses for the software to catch up. This was in stark contrast to the second experiment with human transcribers which allowed participants to speak at a normal pace and even stop in the middle of a command to think, something that was not possible with the software. Participants also suggested better commands such as Jump To for navigation and being able to insert templates in the correct location without having to navigate there first.

B. Performance of vocal programming

With speech recognition identified as the main challenge for programming by voice, researchers continued to study the subject as the technology improved. In 2016 Maloku and Plana [19] set out to analyze the output performance of vocal programming. This was done through three separate tests comparing input from keyboard and mouse against only voice input, as well as a combination of the two methods. Their code examples were written in Java within their framework named HyperCode which was built to be used with IntelliJ IDEA. They found that the total time for all the tasks with keyboard/mouse took 65 seconds, pure voice input took 84 seconds, and a combination took a total of 46 seconds. The results are compiled in Table I. Pure voice input compared equally well in several of their examples but struggled when navigating suggested autocomplete from the IDE and when adding multiple single characters such as plus sign and colon.

<table>
<thead>
<tr>
<th>Input method</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard/Mouse</td>
<td>65 seconds</td>
</tr>
<tr>
<td>Voice input</td>
<td>84 seconds</td>
</tr>
<tr>
<td>Combination</td>
<td>46 seconds</td>
</tr>
</tbody>
</table>

Table I: The aggregated time for each input method to complete the tasks in the study by Maloku and Plana [19].

Ionescu and Schlund [3] also focused on performance when they 2021 published a study comparing input methods for robotics through an API offered by the manufacturer. In their experiment, four participants with no experience in programming the robot were instructed to write a simple pick-and-place application with the purpose of moving four screws. In the first part of the experiment, the participants wrote the application using the robot’s interface with a mouse and keyboard. In the second part, they replicated their application but were only allowed to use voice commands through the
commands written by the researchers. The result showed that programming by voice was around 46% faster than using the robot’s native interface. They concluded that most of the time was saved by not having to navigate around the software. The researchers also noted that voice commands allowed the user to free their hands and use them for other things such as guiding the robotic arm.

Hossain et al. [2] contributed to the field by developing a system named Voice Command Language (VCL) that utilizes Google speech recognition API to generate code in multiple languages (C, Java, Python) based on generic voice commands. It somewhat resembles reactive programming in the sense that it is a functional programming approach where the user describes what to do and not how to do it. The researchers had five programmers complete a code example in all three languages and the result available in Table II suggests that voice commands outperform traditional hand input. A combination of both input methods performed even better and the researchers concluded that this is because it is quicker to fix errors with the keyboard than through voice commands.

<table>
<thead>
<tr>
<th></th>
<th>Voice</th>
<th>Hand</th>
<th>Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>72</td>
<td>134</td>
<td>59</td>
</tr>
<tr>
<td>Java</td>
<td>73</td>
<td>229</td>
<td>63</td>
</tr>
<tr>
<td>Python</td>
<td>29</td>
<td>48</td>
<td>27</td>
</tr>
</tbody>
</table>

Table II: The average time, categorised by user input type, for five programmers to write the code example in the study by Hossain et al. [2].

C. Vocal programming in the reactive paradigm

The study published by Lagergren and Soneryd [10] is the only study found to compare programming by voice in different paradigms. To accomplish this the authors identified five code examples from Rosetta Code [20], a wiki-based platform with code implementations for common algorithms and programming solutions in many different languages. These five code examples were then rewritten by the authors from imperative to the reactive paradigm. With the code examples at hand, they used a voice recognition software named Talon to write the code through voice commands. The commands were recorded, transcribed, and then analyzed. The results consisted of, among other things, the number of words uttered to produce the code, the number of forced pauses, and the number of words that produced a single character. The authors found that reactive programming reduced both the vocal and cognitive load since more utterances led to words rather than single characters. However, they also noted that Talon is not tailored for the reactive paradigm because of how cumbersome the switch between natural English and programming keywords was. The authors concluded that accuracy in speech recognition is still one of the main challenges and one limitation of their study is that they themselves wrote the reactive examples based on imperative code instead of writing reactive examples from scratch which might result in code better suited for the reactive paradigm.

D. Summary

To summarise it seems programming by voice is a well-researched area and the software has improved over the years. Available research indicates that there is a potential performance gain for programming by voice-over-hand input but the results are inconclusive. One reason for this might be that programming by voice and mouse/keyboard input excel in different areas. Voice commands can quickly generate large amounts of code but it can be time-consuming to navigate the user interface and manually fix errors caused by speech recognition mishearing the intention. In addition to this performance gain, Lagergren and Soneryd [10] show that there might be room for improvement by using the reactive paradigm. Despite using software not tailored for the paradigm, their data indicate that reactive programming requires fewer words and increases the number of characters in code generated per second. In turn, fewer words indicate lower vocal load which would be beneficial for the programmer’s voice.

The related work also highlights gaps in the field. Since we only found one study related to programming by voice in the reactive paradigm, there is no comparison with alternative software so it is difficult to get an overview of the current state. The inconclusive data regarding time gain also highlights the importance of distinguishing between different types of use. For example, it is not enough to have good voice recognition, the software also needs to support anonymous functions and reactive chaining.

V. Research Methodology

This section goes through the research method of learning Serenade, writing the adapter, selecting code examples, transcribing the voice commands, gathering data, and finally analyzing it. See Fig. 1 for a visual overview.

In the study by Lagergren and Soneryd [10], they concluded that reactive programming by voice might be beneficial but more research is needed on software dedicated to RxJava and its impact on vocal load. Since their study, Serenade completed its major components in late 2021 and presented itself as an interesting alternative. It is free software available as a plugin in many popular programming environments such as Atom, Visual Studio Code, and JetBrains. Serenade also support other types of software such as Google Chrome, GitHub, Gmail, and Bash which allows the user to control more of their programming environment. This made the software, not only a suitable candidate for researching the current state of reactive programming by voice, but also to compare the results with the previously mentioned study made in Talon.

```java
Observable.just("Hello World")
    .map(String::length)
    .observeOn(Schedulers.computation())
    .map(length -> 2 * length)
    .subscribe(System.out::println);
```

Listing 1. Code example of chaining in RxJava

To achieve this, we first had to learn how Serenade works and get a feel for what it is capable of. This was accomplished by installing the plugin for IntellJ, reading the documentation,
going through their interactive tutorial as well as practicing dictating example code.

Once we had a decent understanding of Serenade, we set out to extend Serenade by writing an adapter supporting RxJava. Serenade allows this by offering an API that supports custom commands in JavaScript files that get automatically imported when launching the software. They also offer a Github repository [21] where users can share and download these custom files. The programming was an iterative process where we dictated code, saw gaps in the software and implemented improvements.

```java
@serenade
.language("java")
.snippet(
    "add argument print line",
    "System.out::println",
    {},
    "argument"
);
```

Listing 2. Custom command for adding `System.out::println`.

The next step was to put together suitable code examples to test the software on. We opted for two different categories. The first type was written by us to give an accurate representation of what is commonly used in RxJava based on the available documentation, course literature, and our previous knowledge. This includes operators for creating, transforming, filtering, and combining. The second type of code example was code snippets from the previous study. They were originally imperative code fetched from Rosetta Code [20] covering traditional programming examples such as "Guess the number" and then rewritten into the reactive paradigm by the authors. We selected the first two examples from the previous study to enable a comparison between Serenade and Talon.

Thereafter we transcribed the voice commands needed to produce the example code. We came to the conclusion that a verbal measurement might impact the data and therefore decided to write down an optimal transcript and do the analysis on it instead of recording it ourselves.

```java
add observable dot just of string hello world
chain map of string colon length
chain observe on of computation
chain map of length with two times length
chain subscribe of print line
```

Listing 3. Custom voice command for the code in listing 1.

Finally, the transcripts were analyzed and the data collected. One concern of programming by voice is vocal load so the main purpose of data gathering was to measure the change. We chose to measure vocal load by the number of uttered words needed to generate the code examples. We also wanted to measure cognitive load and this was done by measuring the number of uttered words that generated a single character in code. The measurement is based on similar methods in previous studies [10], [18].

The data collected were:
- **Characters in code**: The number of Java characters in the code example.
- **Commands**: The number of voice commands required to produce the code example.
- **Words that produce single characters**: The number of words in the voice transcript that only produce a single character (dot, semi-colon, etc).
- **Words**: Number of words uttered to produce the code.

VI. RESULTS

This section describes the result. Using two code examples from the study by Lagergren and Soneryd [10] allowed us to not only compare the two versions of Serenade but also compare Serenade with Talon to get a better understanding of the current state of reactive programming by voice. A longer analysis of the results is available in Section VII. Discussion.

Table III shows an overview of the collected Serenade data comparing the default Serenade voice support with the custom RxJava adapter written by us. The results indicate that the custom adapter performs better than default Serenade in every measured way, which highlights the room for improvement in its current support of RxJava. Half of the examples show no reduction in the number of commands needed to write the code but these examples also require the fewest amount of commands which might indicate that there are not enough commands to see a noticeable difference. Four code examples stand out by having a word reduction of 25-29% when produced with the custom adapter. These are do, map, flatmap, and Sum & Product. They have in common that they all make use of reactive chaining and anonymous lambda functions, which highlights the room for improvement.
TABLE III
Compilation of the data from using Serenade in its default and extended mode.

<table>
<thead>
<tr>
<th>Commands</th>
<th>Default</th>
<th>Custom</th>
<th>Change</th>
<th>Words</th>
<th>Default</th>
<th>Custom</th>
<th>Change</th>
<th>Words resulting in single chars</th>
<th>Default</th>
<th>Custom</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observable example</td>
<td>8</td>
<td>7</td>
<td>-12.50 %</td>
<td>97</td>
<td>80</td>
<td>-17.53 %</td>
<td>-</td>
<td>26</td>
<td>15</td>
<td>-42.31 %</td>
<td>-</td>
</tr>
<tr>
<td>combineLatest example</td>
<td>6</td>
<td>6</td>
<td>— %</td>
<td>67</td>
<td>63</td>
<td>-5.97 %</td>
<td>-</td>
<td>15</td>
<td>11</td>
<td>-26.67 %</td>
<td>-</td>
</tr>
<tr>
<td>merge example</td>
<td>4</td>
<td>4</td>
<td>— %</td>
<td>54</td>
<td>47</td>
<td>-12.96 %</td>
<td>-</td>
<td>15</td>
<td>11</td>
<td>-26.67 %</td>
<td>-</td>
</tr>
<tr>
<td>do example</td>
<td>5</td>
<td>5</td>
<td>— %</td>
<td>61</td>
<td>44</td>
<td>-27.87 %</td>
<td>-</td>
<td>15</td>
<td>4</td>
<td>-73.33 %</td>
<td>-</td>
</tr>
<tr>
<td>map example</td>
<td>5</td>
<td>5</td>
<td>— %</td>
<td>45</td>
<td>32</td>
<td>-28.89 %</td>
<td>-</td>
<td>12</td>
<td>2</td>
<td>-83.33 %</td>
<td>-</td>
</tr>
<tr>
<td>flatmap example</td>
<td>14</td>
<td>13</td>
<td>-7.14 %</td>
<td>107</td>
<td>76</td>
<td>-28.97 %</td>
<td>-</td>
<td>21</td>
<td>6</td>
<td>-71.43 %</td>
<td>-</td>
</tr>
<tr>
<td>Read Files</td>
<td>14</td>
<td>12</td>
<td>-14.29 %</td>
<td>122</td>
<td>113</td>
<td>-7.38 %</td>
<td>-</td>
<td>20</td>
<td>16</td>
<td>-20.00 %</td>
<td>-</td>
</tr>
<tr>
<td>Sum and Product</td>
<td>15</td>
<td>10</td>
<td>-33.33 %</td>
<td>91</td>
<td>68</td>
<td>-25.27 %</td>
<td>-</td>
<td>16</td>
<td>16</td>
<td>— %</td>
<td>-</td>
</tr>
</tbody>
</table>

![Graph](image1)

**Fig. 2:** Average data for default and custom Serenade. Commands and words are the number of commands and words uttered to generate the code. Code represents the code characters generated by each uttered word. Single chars represents the number of words resulting in a single character of code, such as "dot" and "semi-colon".

When aggregating the eight code examples in Fig. 2, we see a similar trend. On average, the custom adapter makes use of 1.1 fewer commands (despite short code examples), 15.1 fewer words, and outputs an additional 0.9 characters of code for every uttered word. Although modest in low quantities, these results equal roughly 19% reduction in uttered words needed to produce the same code. The benefit of this would be a reduction in vocal load.

<table>
<thead>
<tr>
<th>Read files</th>
<th>Talon</th>
<th>Serenade (D)</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>212</td>
<td>122</td>
<td>-42.45 %</td>
<td></td>
</tr>
<tr>
<td>Sum and Product</td>
<td>174</td>
<td>91</td>
<td>-47.70 %</td>
</tr>
</tbody>
</table>

Table IV: Number of uttered words required to produce the code in default Serenade and Talon.

Thanks to the published transcripts [22] from the previous study we were also able to compare the performance between default Serenade and Talon in Table IV. The difference is significant with Serenade requiring about 45% fewer words to write the two code examples. Serenade seems to be more adept at handling spoken commands that mix English and programming keywords while also filling the gaps by itself. For example, the Serenade command "add import java dot io dot buffered reader" seems to be equal to the more verbose "phrase import space word java dot sit odd dot hammer Buffered reader semicolon slap" in Talon.

![Graph](image2)

**Fig. 3:** The average number of characters of code generated per uttered word in Serenade and Talon.

The performance difference becomes more prominent when comparing the code output for each word in Figure 3. Every uttered word in Talon produces on average 1.9 characters of Java code. The default version of Serenade produces 3.4 characters per uttered word which is about 79% more efficient. The custom adapter for Serenade has an even higher output at 4.1 which corresponds to roughly a 116% difference in Java output per uttered word.
Fig. 4: Average number of uttered words resulting in a single character of Java code in Serenade and Talon. Examples are words such as dot and semi-colon.

Another reason for the difference between Serenade and Talon might be found in Figure 4. It shows the average number of uttered words resulting in a single character of code.

All the code and transcripts are available on Github1.

VII. DISCUSSION

In this section, we will discuss the challenges we faced, our overall reflections, and the results.

A. Research Questions

RQ1: Current limitations of reactive voice programming?

During the testing process, we discovered many limitations of both vocal programming and speech recognition in general, and RxJava in particular.

Having the programming-by-voice software (in our case Serenade) as a separate application in a separate window is a disadvantage. Having the tool integrated into the IDE would have been a major advantage. The Serenade window is “always on top”, meaning that it is always covering part of the IDE window if the IDE is running in full-screen mode.

While the speech recognition engine makes rather few mistakes, the ones that it does make can be annoying and it is time-consuming to undo and try to fix them. Many mistakes occur when a variable or function name is similar to an actual word, for example, the variable name “arr” (commonly used as short for “array”). It is very similar to the word “are” and can be difficult for the speech recognition engine to figure out.

When it comes to reactive code, a very common feature is the “chaining” of operators, such as:

```java
Observable.just("Hello World")
  .map(String::length)
  .observeOn(Schedulers.computation())
  .map(length -> 2 * length)
  .subscribe(System.out::println);
```

Listing 4. Code example of chaining in RxJava

Serenade does not have built-in support for this, which makes the coding rather cumbersome.

RxJava operators usually utilize lambdas, like so:

```java
numbers.subscribe(e -> {
  sum.addAndGet(e);
  prod.updateAndGet(v -> v * e);
});
```

Listing 5. Code example of lambda in RxJava

This was very difficult to produce in the default version of Serenade. It required commands such as “arrow” and “curly brackets”.

A very common word in RxJava is “observable”. Most of the time, this word should be capitalized. However, sometimes it is used as the name of a variable, and should therefore be lowercase. Serenade gets this right most of the time, but not always, which can be annoying. The user can specify in each case, for example by using the word “capital”, but this obviously increases the complexity of the coding.

In Java in general, a very common expression is “System.out.println”. In RxJava, it is also common to use the version “System.out::println”. Serenade does not have an easy way to produce this code. Saying “system dot out colon colon print l n” is not ideal, and even this does not always succeed since “println” is not really a word that can be easily pronounced. This caused major struggles each time either of these versions was used.

RQ2: To what effect can we improve the experience?

First of all, it is worth noting the major difference between Talon and Serenade, without any customizations. Serenade required an average of 45% fewer words than Talon to produce the same code. This indicates that Serenade is either much better in general, or just better when it comes to RxJava. In addition to this, with Talon, the average number of words that only produce a single character was 111. In the default version of Serenade, this number was 18. This means that Talon requires the user to specifically say words like “space”, “semi-colon”, etc. very frequently. In Serenade, these characters are almost always added automatically. When words like these are needed, they disrupt the flow significantly.

When comparing the default version of Serenade with the custom version, the average number of words that produce a single character is reduced from 17.5 to 10.1, a 42% improvement.

The default version of Serenade required an average of 80.5 words per code example. With customization, that number drops to 65.4, which is a 19% improvement. In the more...
simple code examples, the improvement was as low as 6%. However, in the more advanced code examples, the improvement was as high as 29%. In a large project with both simple and complex code, the average number of 19% is likely close to the truth.

Four code examples, in particular, stand out. The merge example, do example, map example and sum & product all saw a very large improvement through customization. When analyzing these examples, we can see that all of them contain chaining of operators, lambdas and "println", all of which were heavily simplified with custom commands.

We implemented a solution to the chaining of operators by creating a new keyword, “chain”. This is what the Serenade script code looks like:

```java
.language("java")
.command("chain <%operator%> of <%var%> with <%contents%>",
async(api, matches) => {
  await api.runCommand("before semi insert dot camel " + matches.operator + " of");
  await api.runCommand("add argument " + matches.var + " arrow");
  if (matches.contents === "body") {
    await api.runCommand("insert curly brackets");
  } else {
    await api.runCommand("left enter");
  }
},
  {autoExecute: true});
```

Listing 6. Serenade chain script

The command has three variables: operator, var and contents. A voice command such as "chain map of length with two times length" will produce:

```java
.map(length -> 2 * length)
```

Listing 7. Chain map

The command "chain flat map of s with body" will produce:

```java
.flatMap(s -> {
})
```

Listing 8. Chain flatMap

This allows the user to very easily chain operators together and include a lambda, either with or without a function body (curly brackets). Lambdas were one of the most difficult tasks to perform in default Serenade, both in terms of cognitive and vocal load, so this made a big difference.

We added support for commonly used code snippets such as "System.out.println". For example, the command "insert print line of myVariable" now produces:

```java
System.out.println(myVariable)
```

Listing 9. Standard print line

Simply saying "insert print line" produces the version of print line called "method reference" that is more common in RxJava:

```java
System.out::println
```

Listing 10. Print line method reference

We also simplified some commands, such as allowing the user to say "save" in order to both format the code and save the project. We introduced the keyword "options" in order to bring up the suggestions by the IDE.

We simplified the creation of lists. A list such as:

```java
List<Integer> myList = Arrays.asList(1, 2, 3, 4);
```

Listing 11. List example

... can now easily be created with the command "add integer list my list of 1 comma 2 comma 3 comma 4", leaving out Arrays.asList().

A user working on a large software project would likely come across many more possible customizations. In fact, the Serenade customization feature is advanced enough to create shortcuts and simplifications for most scenarios.

**RQ3: Lessons learned and challenges in customizing vocal reactive programming**

The most apparent challenge during this project has been the time it takes to write the customization scripts for Serenade. It requires a lot of trial and error in order to get it right. First, the situation needs to be analyzed and the user needs to come up with an idea of how to simplify a certain command or series of commands. Then, write a prototype and test it. It will almost certainly not work as intended on the first try. The script then needs to be rewritten and re-tested multiple times until the desired result is achieved. It is time-consuming and can be frustrating.

When programming in a regular IDE environment, there are usually error messages and compilation errors when there are errors in the code. Coding the Serenade scripts is much more challenging as there is a lack of information when there are errors in the code. Sometimes during the coding of the scripts for this project, Serenade simply stopped working without any information about what the problem was. It was up to the user to try to find the error in the code.

Another challenge is the limitations of the customization feature in Serenade. One limitation, in particular, is the fact that command variables cannot be written directly after one another. There must be at least one word between them. An example is the following:

```java
"add <%type%> list <%name%> of <%contents%>"
```

Listing 12. Actual command for creating a list

It would have made more sense to construct the command like so:

```java
"add list <%type%> <%name%> of <%contents%>"
```

Listing 13. Invalid command for creating a list
This is not possible because there is no word between "type" and "name".

Serenade is run as a cloud service. There is always a risk of downtime due to technical problems or maintenance. Additionally, the service requires a quick internet connection, which could definitely be a challenge for many users.

The speech recognition engine used by Serenade makes surprisingly few mistakes, but after a few hours of coding, it becomes obvious that there is still plenty of room for improvement.

### B. Overall Reflections

Our first impression of reactive programming with Serenade was very positive. The speech recognition is surprisingly accurate. Serenade's understanding of context is impressive. Overall, it appeared perfectly possible to code entirely with voice. However, we quickly encountered several issues related to RxJava-specific code, such as the chaining of operators and very frequent use of lambdas.

We encountered problems related to speech recognition, such as the difficulty to distinguish between similar-sounding words. Serenade struggled with understanding when a word should be capitalized, and when not. Some frequently used code snippets or words were very cumbersome to code, such as the word "println", which can not really be pronounced.

Many problems could be eliminated with custom scripts, but the process is very time-consuming. Mainly because it requires a lot of trial and error, and Serenade provides poor information about errors in the code.

There are two major findings in this study. The first is the fact that selecting a good programming-by-voice tool is crucial. Serenade required 45% fewer words than Talon, and there may be other tools that are even better. The second major finding is that the effectiveness of the vocal coding experience in reactive Java can be increased by approximately 19% by implementing custom scripts. Whether that 19% is worth the time and effort it takes is up for debate, but we believe that adding support for very common features such as chaining of operators and lambdas is a good idea.

It is worth noting that there is only a small difference in the number of commands required before and after customization. This means that customization does not reduce the number of commands, but it makes the commands shorter. We discovered during testing that long commands are generally bad because it can be cognitively difficult to get the entire command right, without any mistakes.

The number 19% may not sound that impressive, but creating custom commands does more than reducing the number of words required. It also allows the user to choose keywords that feel natural. For example, the word "body" can be used to create an empty function body and place the cursor inside, ready to continue coding. Without customization, the user would have to say "curly brackets" followed by "left enter", which is not only four times as many words, but also not as intuitive.

In order to put the results into perspective, we can make a few assumptions about a typical Java class. According to Martin Lippert's and Stephen Roock's "Rule of 30" [23], a Java class should not contain more than 30 methods and a method should not contain more than 30 lines of code. A large class would then contain a maximum of 900 lines of code. For this example, we use a class with 15 methods and 15 lines in each method, in order to get close to the size of an average class. This gives us 225 lines. The traditional line length is 80 characters, but this is hardly the case in 2022 with high resolutions. A more reasonable line length is 125 characters since that is the maximum number of characters in Github diff view [24]. If we assume that each line is an average 50% of the max line length, this gives us roughly 14,000 characters.

The default version of Serenade produces 3.9 characters per word, so in order to produce the example class of 14,000 characters, 3,590 words are required. For the custom version of Serenade, this number drops to 2,917 words. With an average speaking rate of 150 words per minute [25], default Serenade would require 24 minutes to produce the code. The same code would theoretically take 19.5 minutes to produce with the custom version of Serenade, a time saving of 4.5 minutes.

A developer using custom Serenade would complete three classes in 58.5 minutes. A developer using default Serenade would require an hour and 12 minutes for the same task. This means that for every hour the developer using custom Serenade works, the developer using default Serenade requires an additional 12-13 minutes.

This example is theoretical and is only an estimation. Developers do not code constantly. There is planning and thinking involved. It is simply an attempt put the results of this study into a perspective that is easier to comprehend. The example only takes into account the reduced vocal load in custom Serenade. Due to the reduction in words that produce a single character and the more intuitive keywords used in custom Serenade, there's likely a reduction in cognitive load as well. This would reduce the time developers have to think about what to say.

Lastly, there are ethical considerations of vocal programming. In the introduction, we mentioned the benefits of creating a more inclusive profession for the programmer, companies, and the entire society. However, there are also privacy concerns. Both between the employee and employer, and between the company and the vocal recognition cloud service. The microphone might pick up discussions that are of private character or even company secrets. Serenade supposedly manages this concern by offering a fully local alternative (Serenade Pro). The underlying issue is still there though and it might be difficult to verify closed proprietary software.

### C. Results in relation to related work

Our findings are in line with previous research [17] and suggest that speech recognition is still a challenge in vocal programming. The accuracy has improved over the years but instances of misinterpretation are demanding since it requires the user to pay attention to the output and fix mistakes. This
is associated with both a high vocal and cognitive load. In their study, the participants also suggested better commands for navigating the code and inserting templates in the correct location without having to navigate there first. Both of these features are now supported in Serenade.

Related to reactive vocal programming, the study by Lagergren and Soneryd [10] indicated that the reactive paradigm was associated with less cognitive and vocal load than the imperative paradigm. This was despite Talon not being adapted for the reactive paradigm and having a high count of words resulting in single characters of code. Our comparison of Serenade and Talon might be of importance here since the data suggests that Serenade performs significantly better. It may have implications for the data comparing the imperative and reactive paradigms.

These findings might also have had implications for any study looking at the performance of vocal programming when comparing it to other input methods [2], [19]. If one software requires significantly fewer words and time, then the result of the study will be strongly dependent on software selection.

D. Limitations & validity threats

We have limited experience with Serenade, both in terms of using the software for programming-by-voice and using the customization feature.

Neither of the authors is a native English speaker. This may or may not have an effect on the results, but certain accents could potentially cause the speech recognition engine to misinterpret some words it otherwise would not.

We have a fairly limited amount of example code. We have tried to include all common aspects of RxJava, but in order to get even more accurate results, one would have to measure the code for an entire reactive software project.

In this study, we have implemented solutions to the many common issues that we discovered while using default Serenade to program by voice in reactive Java. There may be other ways to make these improvements. We can not be sure that our solutions are the most optimal solutions.

It should be mentioned that this study was conducted in reactive Java (RxJava). One cannot assume that the results would be the same when coding in the reactive paradigm in another language.

VIII. CONCLUSIONS

We can conclude that developing custom Serenade scripts specifically for RxJava reduces the number of words required to produce the example code by 19%. It also reduces the number of words that only produce a single character by 42%. We can also conclude that Serenade required 45% fewer words than Talon to produce the same example code. Even more significant is the fact that Talon required more than six times as many words that only produce a single character, compared to the default version of Serenade.

Speech recognition in a programming environment is challenging. Programming contains words and phrases that would not otherwise be used in regular speech. Serenade is able to guess correctly most of the time but does make mistakes occasionally. The mistakes cause an increased vocal load because the user has to undo and make a new attempt. It also increases the cognitive load since the user has to be aware of potential mistakes and check for these while thinking about the logic and next command.

We would have preferred to have the programming-by-voice interface to be integrated into the IDE, instead of in a separate window that is always blocking part of the IDE.

customizing voice commands allows the user to not only reduce the number of words required but also choose keywords that make sense, are easy to remember, and are closer to natural speech, thus reducing both vocal and cognitive load and improving the flow of the coding.

The research questions were answered. The study explains the challenges facing vocal programming in reactive Java, as well as the advantages of implementing custom commands.

Future research would be to test the adapter on a large RxJava software project in order to get even more accurate results. Additionally, since an RxJava-project almost certainly contains a lot of regular Java code, the adapter could also include custom voice commands for non-RxJava code, such as easy ways to produce for-loops, switch-statement, etc. One of the conclusions of this study is that creating custom scripts for reactive Java only reduces the vocal load by 19%. Future researchers can re-use our methodology, but convert the code examples into the imperative paradigm and determine if the improvement is greater or less than 19% in that case.

One other area worth studying would be an IDE developed specifically for voice commands. This study was conducted in IntelliJ IDEA and focused on the code, but programming requires a complete environment and a traditional IDE with hidden settings is not specifically suited for voice commands.

To summarize, these are the main takeaways:

- Writing custom scripts for RxJava-functionality in Serenade resulted in 19% fewer words required to produce the example code.
- The custom scripts reduced the number of words that produce a single character by 42%.
- Serenade required 45% fewer words to produce the example code compared to Talon.
- Talon required more than six times as many words that produce a single character compared to the default version of Serenade.
- Speech recognition in a programming environment is not perfect and results in multiple misinterpretations.
- Having the programming-by-voice application as a separate window is a disadvantage.
- Customization allows the user to select keywords, reducing cognitive load.
- Writing custom scripts for Serenade is time-consuming.
- Serenade has poor error information and can stop working when there is an error. Use version control and make frequent commits.
REFERENCES


APPENDIX 1: TIME PLAN

<table>
<thead>
<tr>
<th>Week</th>
<th>Planning</th>
<th>Research</th>
<th>Code adapter</th>
<th>Code snippets</th>
<th>Transcribing</th>
<th>Analysing results</th>
<th>Writing report</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5: Proposed Time Plan.

<table>
<thead>
<tr>
<th>Week</th>
<th>Planning</th>
<th>Research</th>
<th>Code adapter</th>
<th>Code snippets</th>
<th>Transcribing</th>
<th>Analysing results</th>
<th>Writing report</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 6: Actual Time Plan.

APPENDIX 2: CONTRIBUTIONS

Gustaf:
RxJava adapter, code examples, testing and transcribing custom Serenade, abstract, introduction, purpose and contributions, background, discussion, and conclusions.

Stefan:
RxJava adapter, testing and transcribing default Serenade, introduction, related work, research methodology, results, and results in relation to related work.