

*Evaluating  
Developer Strategies  
in Live Coding  
Environments*

Master's Thesis at the  
Media Computing Group  
Prof. Dr. Jan Borchers  
Computer Science Department  
RWTH Aachen University



by  
*Joachim Kurz*

Thesis advisor:  
Prof. Dr. Jan Borchers

Second examiner:  
Prof. Dr. Bernhard Rumpe

Registration date: 15.02.2013  
Submission date: 09.08.2013



I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

*Aachen, August 2013*  
*Joachim Kurz*



# Contents

<b>Abstract</b>	<b>xvii</b>
<b>Überblick</b>	<b>xix</b>
<b>Acknowledgements</b>	<b>xxi</b>
<b>Conventions</b>	<b>xxiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Live Coding . . . . .	2
1.3 Chapter Overview . . . . .	5
<b>2 Background</b>	<b>7</b>
2.1 Different Levels of Liveness . . . . .	7
2.2 Errors, Failures, Faults and Bugs . . . . .	9
<b>3 Related work</b>	<b>11</b>
3.1 Live Coding . . . . .	11

---

3.1.1	Recent Developments . . . . .	13
3.2	Research into Developer Behavior and Errors	19
3.2.1	The Effect of Liveness . . . . .	20
3.2.2	Programming Errors . . . . .	22
	Summary . . . . .	29
<b>4</b>	<b>Prototype</b>	<b>31</b>
4.1	Motivation for a New and More High-Fidelity Prototype . . . . .	31
4.1.1	Limitations of Heinen's [2012] Prototype . . . . .	32
4.2	A New Backend . . . . .	33
4.3	The Prototype . . . . .	35
4.3.1	Implementing Victor's [2012a] Visualization . . . . .	36
	Problems with the Skewed Design . .	37
4.3.2	Final Version . . . . .	38
4.4	Changes to the Backend . . . . .	46
<b>5</b>	<b>Study Design</b>	<b>51</b>
5.1	Motivation for a New Exploratory Study . .	51
5.2	Designing the Tasks . . . . .	54
5.2.1	Task 1.1: Parsing an RSS-Feed using a SAX-parser . . . . .	56
	Refining the Task . . . . .	57

---

Expected Problems . . . . .	57
5.2.2 Task 1.2: Date & Time Conversion . . .	58
Refining the Task . . . . .	59
5.2.3 Expected Problems . . . . .	60
5.2.4 Task 3: Dijkstra's Algorithm . . . . .	60
5.2.5 Refining the Task . . . . .	61
5.2.6 Expected Problems . . . . .	63
5.3 Recruiting Participants . . . . .	63
5.4 Study Setup . . . . .	64
5.4.1 The Development Environment . . . .	65
Building a Continuous Compilation Plugin for Brackets . . . . .	65
Providing a Debugger . . . . .	68
5.4.2 Monitoring . . . . .	70
5.4.3 Procedure . . . . .	70
<b>6 Evaluation</b>	<b>73</b>
6.1 Participants . . . . .	73
6.2 Evaluating the Study . . . . .	75
6.2.1 Tasks . . . . .	75
6.2.2 Tool Quality . . . . .	76
6.3 Evaluating the Live Coding Tool . . . . .	78
6.3.1 Qualitative Results . . . . .	79

---

6.3.2	Task-Based Quantitative Evaluation . . . . .	84
	Task Correctness . . . . .	85
	Task Completion Times . . . . .	85
	Number of Changes . . . . .	87
6.3.3	Change-Based Quantitative Evaluation	88
	Clustering Changes into Change Clusters . . . . .	89
	Analyzing the Change Clusters . . . . .	94
6.4	Discussion . . . . .	108
6.5	Improvements to the Prototype . . . . .	110
<b>7</b>	<b>Summary and Future Work</b>	<b>113</b>
7.1	Summary and Contributions . . . . .	113
7.2	Future Work . . . . .	115
7.2.1	Evaluating the Gathered Data . . . . .	115
7.2.2	New Programmer Error Studies . . . . .	117
7.2.3	Improving the Live Coding Tool . . . . .	117
7.2.4	Research into Better Understanding of the Effect of Live Coding . . . . .	118
<b>A</b>	<b>Tasks Used in the User Study</b>	<b>119</b>
<b>B</b>	<b>Questionnaires</b>	<b>127</b>
B.1	Pre Task Questionnaires . . . . .	127
B.2	Post Task Questionnaires . . . . .	129



B.3 Post Session Questionnaires . . . . .	131
<b>C Change Cluster Graphs</b>	<b>133</b>
C.1 Task 1.1 . . . . .	133
C.2 Task 1.2 . . . . .	137
C.3 Task 3 . . . . .	141
 <b>Bibliography</b>	 <b>145</b>
 <b>Index</b>	 <b>151</b>



# List of Figures

1.1	Victor's [2012b] first version of a Live Coding editor, displaying an implementation of binary search. . . . .	3
3.1	Screenshot of the Eclipse program presented by Edwards [2004]. . . . .	14
3.2	A screenshot of Rehearse, showing how each line is evaluated directly. . . . .	15
3.3	A screenshot of Khan Academy's simple Live Coding editor. . . . .	17
3.4	A screenshot of Victor's [2012a] second prototype, showing how several values in a loop can be visualized. . . . .	17
3.5	A screenshot of Victor's [2012a] second prototype with a zoomed out view, showing how individual values are transformed into a graph view. . . . .	18
4.1	Screenshot of Heinen's [2012] prototype, displaying an implementation of Bubble Sort. . . . .	32
4.2	Screenshot of an early version of our prototype using a skewed display of variable values. . . . .	36

---

4.3	Screenshot of an early version of our prototype, showing how values quickly drift off-screen if long loops occur. . . . .	37
4.4	A screenshot of our final prototype, showing an implementation of Bubble Sort. . . . .	38
4.5	A list of pictures which illustrates how the iteration selector works. . . . .	39
4.6	A screenshot of the Column-Resize cursor hovering over an iteration selector. . . . .	40
4.7	A screenshot of our final prototype that depicts how nested functions are displayed. . .	41
4.8	A screenshot of our final prototype showing the different syntax colors used. . . . .	42
4.9	A screenshot of our final prototype illustrating the display of circular objects. . . . .	42
4.10	A screenshot of our final prototype explaining how errors are displayed. . . . .	43
4.11	A screenshot of the final prototype in which the execution indicator is visible. . . . .	47
4.12	A diagram depicting our changes to the backend process architecture. . . . .	49
5.1	The four level nested model by Munzner [2009]. . . . .	52
5.2	Screenshot of our Continuous Compilation Plugin showing a range of errors. . . . .	66
5.3	Screenshot of our Continuous Compilation plugin displaying an error message. . . . .	67

---

5.4	Screenshot of our Continuous Compilation plugin showing two nicely aligned error messages. . . . .	68
5.5	Screenshot showing the scripts used for debugging. . . . .	69
6.1	Histograms showing how difficult participants thought the tasks were. . . . .	76
6.2	Histograms showing the previous experience of participants in the task domain. . . . .	77
6.3	Error bars for the SUS Ratings of the 7 participants using our Live Coding plugin. . . . .	78
6.4	Histograms of participants answer to question 11 and 12. . . . .	79
6.5	Participants answers to Question 4: "I understand what the code I wrote does exactly and why it works (or doesn't)." . . . . .	80
6.6	Histograms showing participants' answer to questions 13/14 . . . . .	81
6.7	Histograms showing participants' answer to questions 15/16 . . . . .	81
6.8	Histograms showing participants' answer to question 6 . . . . .	82
6.9	Histograms showing participants' answer to question 5. . . . .	83
6.10	Histograms showing participants' answer to questions 17/18. . . . .	83
6.11	Histograms showing participants' answer to question 2. . . . .	83

---

6.12	Histograms showing participants' answer to question 3. . . . .	84
6.13	Error bar charts for the task completion times of the participants in different tasks and conditions. . . . .	85
6.14	Error bar charts for the number of changes of the participants in different tasks and conditions. . . . .	88
6.15	An example of a graph showing the changes of a participant working on Task 3. . . . .	94
6.16	Box plots for the gap duration between change clusters, by Live and Task. . . . .	98
6.17	Error bar charts for the duration of clusters grouped by tasks and whether the participant used Live Coding or not. . . . .	98
6.18	Error bar charts for the number of changes in clusters grouped by tasks and whether the participant used Live Coding or not. . . . .	103
6.19	Error bar charts for the gap duration of clusters grouped by tasks and whether the participant used Live Coding or not. . . . .	106

## List of Tables

6.1	Programming experience and age of the 13 participants of our study. . . . .	74
6.2	Occupations of of our subjects. . . . .	74
6.3	Summary of the task completion times of our participants . . . . .	86
6.4	Table showing a summary of the number of changes of our participants. . . . .	87
6.5	A number of attributes of the calculated change clusters for different threshold $y$ in the range of 1s–60s. . . . .	92
6.6	Critical values of the $\chi^2$ -distribution at the 5%-significance level . . . . .	97
6.7	Development of the Multilevel Linear Model for the duration of change clusters. . . . .	99
6.8	Development of the Multilevel Linear Model for the number of changes per change clusters. Part 1. . . . .	102
6.9	Development of the Multilevel Linear Model for the number of changes per change clusters. Part 2. . . . .	103

6.10 Development of the Multilevel Linear Model  
for the durations of the changeless gaps be-  
tween clusters. Part 1. . . . . 105

6.11 Development of the Multilevel Linear Model  
for the durations of the changeless gaps be-  
tween clusters. Part 2. . . . . 107



# Abstract

'To err is human', so programmers make a lot of errors when writing source code, especially since programming is a mentally taxing activity. Despite this, programmers are still mostly on their own when writing computer programs. The programmer has to "play computer" and mentally execute the code in front of him to understand what it does. At the same time a computer is idly sitting in front of him, which could perform this task much better, faster and more accurately.

Live Coding is the idea of letting a computer do this work. In a Live Coding Environment the computer executes the complete program whenever it is changed and shows the changes to the internal runtime state that occur in each line, freeing the programmer from doing so.

In this thesis we present a robust and fully-functional Live Coding prototype that has been tested in an extensive user study. We also identify a gap in current software engineering research that prevents a well grounded design of Live Coding environments and design a study to close this gap. We present the design and some results of this study, including a set of three tasks that can be used in similar studies.

In our preliminary analysis, we found that developers were convinced of our tool, but we did not find a significant difference in task completion time or task accuracy. However, we did find that the usage of the Live Coding environment did significantly predict how long the breaks between two change sequences of a developer are in a later phase of the implementation, although this influence depends on the task the developer is working on. These vague results indicate that the interaction between errors a programmer makes and the software development tool they use is very complex, and we only scratched the surface so far. Nevertheless, the data gathered in our study is quite rich and can likely be used to answer many of those questions. We point out several of the questions and analyses that are possible at the end of this thesis.



# Überblick

‘Irren ist menschlich’, so machen auch Programmierer oft Fehler wenn sie Quellcode schreiben, insbesondere da Programmieren eine mental anspruchsvolle Tätigkeit ist. Trotz alledem sind Entwickler oft auf sich allein gestellt während sie programmieren. Sie müssen “Computer spielen”, mental den Quellcode ausführen um zu verstehen, was er tut. Gleichzeitig steht vor ihnen ein Computer tatenlos herum, der die selbe Aufgabe viel besser, schneller und fehlerfreier ausführen könnte.

Live Coding ist die Idee, den Computer diese Aufgabe übernehmen zu lassen. In einer Live Coding Umgebung führt der Computer das Programm immer dann erneut aus, wenn es geändert worden ist. Die Live Coding Umgebung zeigt dann die Änderungen am internen Programmzustand nach jeder Zeile an und entlastet auf diese Weise den Programmierer.

In dieser Arbeit präsentieren wir einen robusten und voll funktionsfähigen Live Coding Prototypen, der in einer ausführlichen Nutzerstudie getestet wurde. Wir zeigen außerdem eine Lücke in der aktuellen Softwareentwicklungsliteratur auf, die das wohl fundierte Design eines Live Coding Prototypen unmöglich macht und entwerfen eine Studie um diese Lücke zu schließen. Wir beschreiben die Studie selbst und einige Ergebnisse der Studie, außerdem drei Aufgabenstellungen aus derselben, die in weiteren ähnlichen Studien wiederverwendet werden können.

Unsere vorläufige Analyse zeigt, dass die Entwickler von unserem Prototypen überzeugt waren. Wir konnten keine signifikanten Unterschiede in der Zeit, die pro Aufgabe benötigt wurde, oder der Korrektheit der Lösungen feststellen. Nur in der Länge der Pausen zwischen zwei Änderungen konnten wir Unterschiede feststellen, abhängig davon, ob Live Coding benutzt wurde oder nicht und welche Aufgabenstellung bearbeitet wurde. Dies galt jedoch nur in einer späten Phase der Implementierung. Diese unklaren Ergebnisse weisen darauf hin, wie komplex der Zusammenhang zwischen den Fehlern eines Programmierers und den Werkzeugen, die er benutzt, ist. Die gesammelten Daten sind jedoch sehr ergiebig und viele der offenen Fragen können vermutlich damit beantwortet werden. Wir zeigen mehrere dieser Fragen und mögliche Analysen zum Schluss auf.



# Acknowledgements

First of all, I would like to thank my advisor, Jan-Peter Krämer for letting me work on a very interesting project (again) and giving me a lot of freedom how to achieve our goals. Next, I would like to thank Chatchavan Wacharamanotham for again putting up with my annoying questions about statistics and what kind of test to use. While we are at it, I would also like to thank Andy Field for his book “Discovering Statistics Using SPSS”, because it is the only sensible way to understand and use those nasty statistical tests Chat recommends when you ask him. For helping me write a program to analyze the changes recorded during the study I would like to thank Moritz Wittenhagen. In advance, I would like to thank Kerstin Kreutz for handing in my Master thesis, so I can go on vacation.

In general, I would like to thank everyone at the Media Computing group for providing a very pleasurable working environment. And I would like to thank all of my participants of the user study, who spent many hours working on those mean tasks I invented.

I would like to thank Thomas Freese and Ewgenij Belzmann for providing me with a lot of feedback and correction on very short notice when I needed someone to proofread my thesis. I also would like to thank Ewgenij again for fruitful discussion about how to implement the Live Coding tool and implementing most of the features I requested.

Also, I would like to thank my mother, my father and my sister for all proofreading my thesis, also on short notice. I think I could not have completed it without so many people pointing out all the stupid mistakes I made.

Last, and most importantly I want to thank Leandra for all the things she did for me over the last few months. For painting our complete flat, while I was in Aachen, working on my master thesis. For organizing most of our vacation, while I was in Münster, working on my master thesis. For providing me with food and drinks and sweets and cuddles and everything a poor thesis-writing student needs and at the same time doing all the household chores alone... while I was at my desk, working at my master thesis. I am not sure who of us is happier that this thesis is over. ;-)



# Conventions

Throughout this thesis we use the following conventions.

## Text Conventions

Definitions of technical terms or short excursus are set off in coloured boxes.

**EXCURSUS:**

Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:  
*Excursus*

Source code and implementation symbols are written in typewriter-style text.

```
myClass
```

The whole thesis is written in American English.

## Statistical Conventions

Unless otherwise specified, when we speak of results being significant, we refer to a significance level of  $p < 0.05$ .





# Chapter 1

## Introduction

### 1.1 Motivation

‘To err is human’ and, as most programmers know, the first version of a program is seldom correct. Papert [1980], when talking about teaching children how to program, even argues that this is an important part of the software design and construction process and should be embraced:

“Learning to be a master programmer is learning to become highly skilled at isolating and correcting ‘bugs’, the parts that keep the program from working.” [Papert, 1980]

Still, debugging—correcting one’s errors in the code (or possibly the errors of other programmers)—is often seen as a separate phase and supported as such by modern Integrated Development Environments: [Visual Studio](http://www.microsoft.com/visualstudio/)<sup>1</sup>, [NetBeans](https://netbeans.org/)<sup>2</sup>, [Eclipse](http://eclipse.org/)<sup>3</sup>, [Xcode](https://developer.apple.com/tools/xcode/)<sup>4</sup>, just to name a few of the more popular ones, all provide a separate debugging mode.

Making errors is human and correcting them is an important part of the software construction.

Modern IDEs distinguish between code generation and debugging.

---

<sup>1</sup><http://www.microsoft.com/visualstudio/>

<sup>2</sup><https://netbeans.org/>

<sup>3</sup><http://eclipse.org/>

<sup>4</sup><https://developer.apple.com/tools/xcode/>

Today's debugging tools make it very difficult to find and correct errors or understand an unknown program in detail.

There has been a lot of research in program understanding and programming support tools, but programmers still have to use simple, one-directional, breakpoint-driven debuggers. While this is already an improvement over the command line debuggers of the past, the main difference is the addition of a nice user interface (UI), not a fundamentally different debugging experience. 16 years ago, Lieberman [1997] already criticized the lack of improvement in the area of debugging tools:

“Debugging is still, as it was 30 years ago, largely a matter of trial and error.” [Lieberman, 1997]

While using trial and error does not have to be bad in and of itself, identifying the errors in the trials should be a lot easier than hitting ‘compile’, waiting for the compilation to finish, setting a breakpoint at the correct location, hitting ‘run’, waiting for the program to start up, possibly interacting with the program, to make it hit the breakpoint, and then examining the program state at the breakpoint.

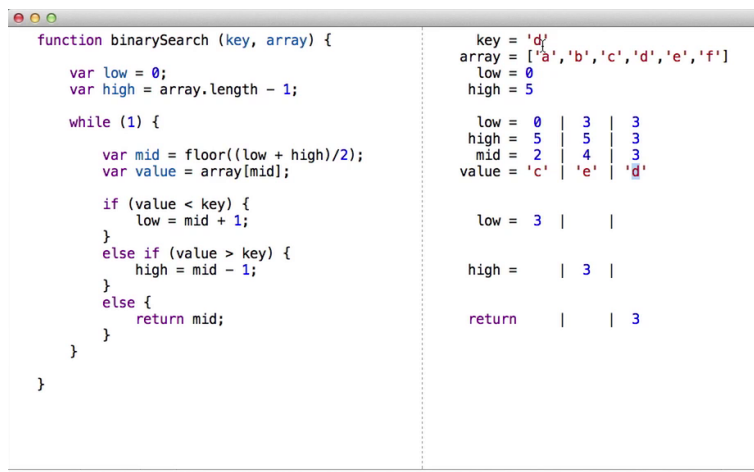
Programming is a mentally challenging process. Tools that reduce memory load should be welcome.

Instead of making fixing the bugs easier late in the development process, we can also look at the code generation and try to fix more bugs at an early stage. Programming is a mentally challenging process [Snell, 1997]. Writing a program statement cannot be done in isolation. The programmer has to know what the state of the program was at the beginning of the current block of code and how the lines of code between the beginning of the block and the position of his statement modify this state. Any mistakes the programmer makes, either in guessing the initial state or in simulating the preceding code in his head to determine the new state, likely affect whether the code he adds is correct. Any tool that can help him in this situation should be quite welcome.

## 1.2 Live Coding

We think it is time for a new tool that solves several of the previously mentioned problems and that Live Coding Environments can be such a tool. A Live Coding Environment provides an editor view and a preview that is synchronized with the editor view such that it shows a part of the program's state after each line in the editor. This could be done by simply displaying the result of the execution of the line as a string (see Figure 1.1) or in any other way that describes the state of the program. The preview updates automatically whenever the source code or the input data is changed.

A Live Coding Environment executes the code whenever it is changed and previews the internal program state after each line of code.



```
function binarySearch (key, array) {
  var low = 0;
  var high = array.length - 1;

  while (1) {
    var mid = floor((low + high)/2);
    var value = array[mid];

    if (value < key) {
      low = mid + 1;
    }
    else if (value > key) {
      high = mid - 1;
    }
    else {
      return mid;
    }
  }
}

key = 'd'
array = ['a', 'b', 'c', 'd', 'e', 'f']
low = 0
high = 5

low = 0 | 3 | 3
high = 5 | 5 | 3
mid = 2 | 4 | 3
value = 'c' | 'e' | 'd'

low = 3 | | 
high = | 3 | 
return | | 3
```

**Figure 1.1:** *The Live Coding editor presented by Victor [2012b]. Screenshot from the recording of the live demonstration during the talk. The source code is on the left, the live preview on the right. The parameters of the function, `key` and `array`, are set by the user to provide example data for the execution.*

The visualization is usually low-level, showing individual variables or objects referenced in the corresponding line, instead of abstracted modifications of data. It shows the state of the program by displaying the data being modified, not control structures. The preview is just an auxiliary view augmenting the source code, not replacing the editor view; it is usually not being used by itself.

Conventional IDEs have separate modes for editing, debugging and evaluation.	Conventional IDEs have a separate debug and edit mode. Programming environments for interpreted languages often have an additional evaluation mode in which program statements can be entered to get their result and better understand what they would do. Even for compiled languages this is often possible when using the debugger. However, in the evaluation mode, only one line of code at a time can be evaluated and the written code is not saved, so it is not well suited to writing the complete code in there. Also the evaluation has to be invoked explicitly.
A Live Coding tool combines and interleaves editing, debugging and evaluation.	A Live Coding tool overcomes all of these limitations by directly executing and thus instantly evaluating the code in the editor. This makes the evaluation mode redundant and likely reduces the needs for the debugging mode a lot, since a lot of information about the program's execution behavior can already be taken from the Live Coding preview. We expect it to have the following benefits: <ul data-bbox="651 1048 1375 1563" style="list-style-type: none"><li>• By directly executing a newly entered statement and displaying the result, the programmer can quickly check their assumptions about changes to the program state and more quickly recognize errors. This can also be used to experiment with unfamiliar statements or functions.</li><li>• By showing the program state after each statement, the programmer does not have to remember it, which reduces his memory load.</li><li>• Understanding a piece of code will become easier, because the programmer can avoid mentally simulating the code: The code is already executed and the state changes are shown.</li></ul>
A Live Coding tool helps finding errors earlier, reduces memory load and supports program understanding.	
Most Live Coding prototypes have not been evaluated so far.	Although several environments that more or less fit our definition of Live Coding have already been developed or at least been proposed (see Section 3.1—“Live Coding”), almost none of them has been evaluated to show which of the assumptions above are actually true. In addition, even basic research, providing knowledge about what kind of runtime state is particularly interesting or what kind of errors programmers make most often, is missing. This also

means that we do not know what kind of errors could more easily be found using such a tool. Out of the research that does look at the causes and kinds of errors in programming, most of it either focuses on high-level, very-hard-to-fix bugs or on bugs from novice programmers (see 3.2.2—“Programming Errors”), just as if experienced programmers would not make any errors, except for the really difficult ones.

Research to better understand the foundations of Live Coding is lacking.

In this thesis we present the following:

1. A plugin for an existing Open Source JavaScript IDE that adds Live Coding functionality to this IDE and is robust enough to be used for academic studies with real-world-like tasks.
2. An extensive user study in which we observed 13 programmers working a total of 40 hours on these tasks. We recorded the complete programming process on video (screen recording in all cases as well as face- and voice recording in most cases) and in addition to that logged the applications the programmers used in the process and the changes they made to the files individually. This thesis includes only a preliminary evaluation of all this data, a lot of future work is possible and needed to analyze this data in detail. We are confident that it makes it possible to learn a lot about how programmers solve code-generation based problems, what kind of minor and medium errors they make and how they fix them. This is a topic that is sadly underrepresented in today’s software development research.
3. Three different programming tasks that focus on code composition instead of asking the programmer to just find and possibly fix bugs and can be used to evaluate programmer ability. We used these tasks in our study with success and think they can be a good starting point for similar studies, because they provide a good balance between challenging the developer and being short enough to be implemented in a few hours.

We contribute a robust Live Coding prototype, a large amount of data about programmers composing programs, preliminary results about Live Coding and three programming tasks that can be reused for similar studies.

### 1.3 Chapter Overview

Background and Definitions	<b>Chapter 2</b> In the next chapter we define some terms we will use in this thesis. We will also describe some fundamental research that is useful for understanding the rest of the thesis.
Related Work	<b>Chapter 3</b> This chapter is split up into two parts due to the different parts of our contribution. We will first give an overview over the previous work related to and origins of Live Coding. After that we give an overview of previous research into programmer behavior and program understanding with a focus on programming errors.
Description of the Prototype	<b>Chapter 4</b> In chapter four, we describe the prototype we developed based on previous work. We will describe previous versions of the prototype, what we changed and how we changed it, including the design rationale behind it.
Study Design	<b>Chapter 5</b> In this chapter, we describe the study design and especially the tasks, describing problems we expected participants to encounter. In addition, we explain how we arrived at these tasks.
Evaluation	<b>Chapter 6</b> The results of the study are described in this chapter. We describe what data we gathered during the study, to help researchers understand what evaluations could be possible using this data. In the end, we present our preliminary evaluation of parts of the data. However, evaluating all the data of this study is beyond the scope of this thesis.
Summary and Future Work	<b>Chapter 7</b> The last chapter is used to summarize our work, and point out ways in which we think the gathered data could be used. In addition, we describe possible future work in the direction of Live Coding and programmer research.

## Chapter 2

# Background

In this thesis we will talk a lot about ‘liveness’ and other terms that can be very vague when used in an everyday setting. We explain several of these terms in this chapter.

### 2.1 Different Levels of Liveness

Tanimoto [1990] described four levels of *liveness* that are often referred to when talking about ‘live’ systems. He used the levels to describe liveness in visual programming systems, but they can be applied to programming systems in general. On the first level are descriptions of programs that cannot be executed, like documentation of software (whether in text form or a graphic showing how different components work together). On the second level these descriptions fully specify a program and can be executed. Examples can be source code files or a diagram written in a visual programming language. But to execute it, the programmer still has to explicitly request the execution (e.g., by hitting a “compile & run” button). This is no longer necessary for systems on the third level of liveness. Here, a change to the description of the software causes a (re)execution of the now changed program. But after executing it, the system waits for another change to the description. On the fourth level of liveness, the software con-

There are four basic levels of liveness.

Liveness level 1:  
Non-executable program description.  
Liveness level 2:  
Program description is only explicitly executable.

Liveness level 3:  
Program description will be executed automatically whenever it is changed.

Liveness level 4:  
Program can process events while being changed.

continues to run and can be changed while running. Events, streams of data, etc. are continually processed by the system and, by extension, by the software. So, changing the implementation of the software not only changes what it will compute next, but changes its behavior in the future when processing future events.

Liveness level 5 & 6  
add predictive abilities.

Recently, Tanimoto [2013] added levels 5 and 6 to his liveness scale. In both cases the programming system has some predictive abilities, it guesses what the programmer wants to do next and already evaluates these possible future version. On level 5 it only generates and evaluates nearby version (e.g. by auto-completing a method call the programmer started to type). On level 6 it makes more high-level predictions of the programmers intentions and can generate large program chunks from the current state of the software and other knowledge about the intended final product.

Liveness depends on which part of a programming system is considered.

This liveness can be evaluated for different parts of a programming system. For example, most of today's programming systems, when looking at a software's runtime behavior, only achieve a liveness level of 2. Programmers change the code but have to execute the program explicitly to check how it behaves. But when looking just at the syntactic correctness of a program most modern Integrated Development Environments (IDEs) reach liveness level 3. This is achieved by running an automatic syntax checker whenever the program code is changed and reporting errors by underlining the infringing parts of the source code. We call this kind of behavior Continuous Compilation.

#### **CONTINUOUS COMPILATION:**

Continuous Compilation means that the file currently edited is compiled after each change (without an explicit command by the programmer to save and compile it) and errors are displayed immediately after a syntactically-wrong expression has been entered, optimally at the location of the error. This is standard behavior in today's IDEs.

Definition:  
*Continuous  
Compilation*

A spreadsheet program achieves complete level 3 liveness.



The values displayed in the cells always reflect the current relation between all the cells and the formulas entered into them. When a formula is changed all cells depending on the changed formula's cell are updated directly and without the need for the user to do anything. However, in this example we can already see that the liveness of a system depends on what one sees as an atomic change. Spreadsheet programs are only live on level 3 if we consider the change of a cell's content to be atomic, so typing in a completely new formula or value is just one change. But if we consider each keystroke to be an individual change, most spreadsheets programs would not be considered live on level 3, since they enter a separate edit mode when changing the contents of a cell and only when the cell is left or the contents are confirmed e.g. by pressing the return key are the formulas of the spreadsheet (re-)evaluated. Since leaving the cell or pressing the return key is a separate user action, we would say they only achieve liveness level 2. However, this level 2 liveness would probably still feel a lot more live to a developer than the usual edit-run-cycle of today's IDEs.

Liveness depends on what is considered to be an atomic change.

If editing a cell is an atomic change, spreadsheets achieve liveness level 3.

If typing a character is an atomic change, spreadsheets achieve only liveness level 2.

In this work, we concentrate on systems that still use source code and ignore most Visual Programming systems. We also consider a system as a Live Coding system only if it achieves at least liveness level 3. We look at changes on a keystroke-basis, so spreadsheets would not be considered to be fully live on level 3.

## 2.2 Errors, Failures, Faults and Bugs

It is unclear what exactly constitutes an "error" in a software system without a more formal definition. We will use the terms defined in Ko and Myers [2005], which are the following:

- A *runtime failure* is a difference between the programmers expectation of the output (graphical, textual or otherwise) or, more generally, behavior of the program and the actual output/behavior of the program.

Runtime failure:  
Unexpected output.

- |   |   |
|---|---|
| Runtime fault:<br>Unexpected runtime<br>state.      | <ul style="list-style-type: none"> <li>• A <i>runtime fault</i> is a difference between the programmers expectation of the machine state (variable values, branches taken, etc.) and the actual machine state.</li> </ul> |
| Software error:<br>Incorrect source<br>code.        | <ul style="list-style-type: none"> <li>• A <i>software error</i> is an incorrect part of the code that may or may not cause a runtime fault.</li> </ul>   |
| Bug: Any of the three<br>above or a<br>combination. | <ul style="list-style-type: none"> <li>• A <i>bug</i> can be any of the three above or a combination of one or more errors, faults and failures.</li> </ul>   |

runtime failure  
 ⇒ runtime fault  
 ⇒ software error

Obviously, whenever there is a runtime failure there has to be an underlying runtime fault and whenever there is a runtime fault there has to be an underlying software error. The reverse is not true: There can be software errors that do not cause runtime faults and runtime faults that do not cause an observably different behavior of the software, which would be a runtime failure.

When searching for and fixing a bug, the *time to error* is the time from the introduction of a software error to realizing that a software error must exist (e.g. by noticing a resulting runtime failure), not necessarily knowing what exactly is wrong. The *time to fix* is the time from this point of realization to the point when the software error has been determined and fixed.

## Chapter 3

# Related work

In this chapter we describe some of the previous work related to this thesis. Since our contribution consists of two big parts, a Live Coding prototype and research into developers programming behavior, the related work can be split up in the same way. In the first section we will describe the origins of Live Coding and some of the more recent work in this area. After that, we describe previous research into program understanding and pay special attention to what kind of studies have been done and what part of a developer's behavior and program understanding the research focused on. Where applicable we will also briefly describe the study setup, to make it possible to compare it to our study.

The chapter is split up into Live coding research and programmer behavior research.

### 3.1 Live Coding

Live Coding is not actually a new idea. Henderson and Weiser [1985] already extended the idea of VisiCalc—the first spreadsheet—to programming, by proposing VisiProg. This environment provided views on the input, source code and output of a program and automatically updated the output view to keep all three synchronized when either the input or the source code changed. Furthermore, program variables could be selected to be displayed and would then

Live Coding is an extension of VisiProg.

be continuously updated as well.

VisiProg is an extension of VisiCalc, the first spreadsheet program.

Actually, the origins of VisiProg—VisiCalc—or spreadsheet programs in general, could also be seen as a predecessor of Live Coding, since they also evaluate all formulas continuously and display the results in the cells. This way the data that is modified is always visible, although this is not true for the code (the formulas in the cells). Shneiderman [1983] praises VisiCalc for being “programmed without a traditional procedural control structure” and making the “impact of changes [...] immediately apparent”. Since it was advertised as an “*instantly* calculating electronic worksheet” (emphasis ours) in the user manual [Shneiderman, 1983], we know that the Liveness level was an important feature.

Morphic was one of the first systems to achieve Liveness level 4

Liveness is also an important feature of Morphic [Maloney and Smith, 1995], a UI construction toolkit and environment based on the Self programming language which featured live editing. This means, all objects could be inspected and even changed at runtime and the system would just continue running with the changed objects. It thus reaches liveness level 4.

In Geo-Logo, previously issued commands could be edited to change the behavior of the program.

Clements and Sarama [1995] modified a Logo environment to make it possible to edit previously executed commands, which were then executed again. In their “Geo-Logo” environment, children were tasked with programming the behavior of a turtle that would then solve mathematical problems (e.g., drawing an equilateral triangle). They noticed that the possibility to change previous commands to the turtle enabled children to reflect a lot more about their code and understand more easily that they could combine statements such as `forward 20` and `forward 30` into `forward 50`. Sequences of commands could also be extracted into procedures to “teach” the turtle new behaviors. Updating the implementation of such a behavior would also update the drawing the turtle produced directly. In contrast to our Live Coding environment, the Logo environment only shows the graphical output, not the values of any variables in the program.

Geo-Logo only showed the resulting graphics output, not internal program state.

Snell [1997] proposed “Ahead-of-Time Debugging” (AOT)

in which each statement entered into the editor would be executed directly and the new state of program variables would be shown to the developer. He calls the debugging “ahead-of-time” to contrast it with the usual workflow of first writing a piece of code and then debugging it. For example, when programming a function, a programmer would enter some test values—several sets of values for different calls of the function were possible—and the programming environment would not only show how those sets of input values were transformed, but also show how the local variables defined later in the program are transformed in each of the different calls to the function. However, he did not describe how these values are visualized and noted that it is unclear whether his approach will be generalizable from primitive values and simple functions to more complex object-oriented programs.

Ahead-of-Time Debugging executes each statement in the code as soon as it is written and shows the resulting program state change.

In contrast to our approach, only the program state after the current line is shown, whereas we show the program state after every visible line, which should make it easier to understand how values change. Snell [1997] reports that informal testing indicates that users find and fix “a substantial number of bugs during initial code entry” [Snell, 1997], errors propagated less, and users were less stressed and tired after using the AOT tool. He also announced a formal user study, but we were not able to find any follow-up papers that would report results from this study.

Only the state in one line of code is shown, not different states in multiple lines.

Although there were many interesting early ideas going in a similar direction, many Live Coding environments have only been developed in recent years. This might be due to them only now becoming computationally feasible. For example, Belzmann [2013] reports that instrumenting the code to extract the appropriate data for Live Coding slows down the execution by a factor between 100 and 3000, heavily depending on the kind of code and the amount of data generated.

### 3.1.1 Recent Developments

Burnett et al. [2001] extended the idea of spreadsheets being

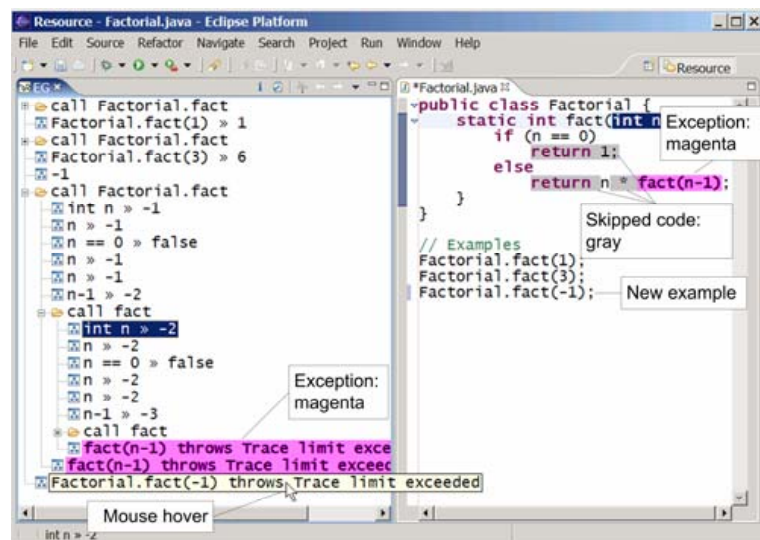
Forms/3 is a spreadsheet programming environment.

Forms/3 reaches  
liveness level 4.

Continuous Testing is  
similar to Live Coding  
by also continuously  
executing a program  
on some test data.

a kind of programming tool and implemented a programming language on the basis of the spreadsheet paradigm. However, while traditional spreadsheets only achieve liveness level 2-3, their spreadsheet programming environment Forms/3 actually reaches level 4. This is implemented by each cell not having just one constant value but instead a formula giving a value of the cell for each point in time. This way, there can be event-stream cells, having different events in them depending on the current time. Even debugging by traveling 'back in time' is possible by looking at values of cells at an earlier point in time.

Saff and Ernst [2004] introduced Continuous Testing by extending the idea of Continuous Compilation. Their tool runs test cases automatically when code changes and displays the errors immediately after a change. By doing this, they shortened the *time to error*. They suspected that this also shortens the *time to fix*.



**Figure 3.1:** The Eclipse plugin presented by Edwards [2004]. Source code is on the right, the functions is called with three different input values at the end of the source code and those executions are show on the left showing the values of the expressions statement by statement. Screenshot reproduced from Edwards [2004].

Edwards [2004] expanded on the idea of using input examples to help a developer write their program by modifying

a concrete example with abstract code. His plugin for the [Eclipse](https://www.eclipse.org/)<sup>1</sup> IDE shows values of expressions in a separate view next to the source code (see Figure 3.1). But the central idea is that the developer implements a piece of functionality by first using an empty template and then specifying example input and expected output. The plugin then provides support to write code to make sure that the code written actually fulfills these requirements and shows which requirements fail whenever the code is changed. Although the first part sounds like simple test-driven development, the plugin actually provides a lot more support in generalizing the examples with code by enforcing the requirements and continuously updating the example view showing which examples execute correctly. Thus, it provides a completely different experience.

Edwards provides a tool that makes it easy for programmers to go from concrete examples to abstract programs.

```

var s;
undefined
s = 'thin solid' + color;
thin solidblue
$('#p1').text();
Here is the first paragraph
$('#p1').css('border', s);
[object Object]
$('#p2').html();
Here is the second paragraph
$('#p2').html($('#p1').html());
[object Object]
$('#p2').css('color', color);
[object Object]

```

**Figure 3.2:** A screenshot of Rehearse. Source code is shown in non-italics, the result of a line is shown in italics below that line. Gray lines are lines that were entered and executed but have been undone. Reproduced from Choi et al. [2008].

Choi et al. [2008] showed Rehearse, which comes closest to the Live Coding approach later made popular by Victor [2012b]. Rehearse provides an environment for interactive JavaScript development. The editor not only displays the result of a line of code directly below that line after it has been entered completely. It also provides undo functionality by allowing the developer to select a line in the editor and undo the changes to the program state caused by this line. This enables a very exploratory style of programming.

Rehearse executes each statement as soon as it is typed and shows the result of that statement.

<sup>1</sup><https://www.eclipse.org/>

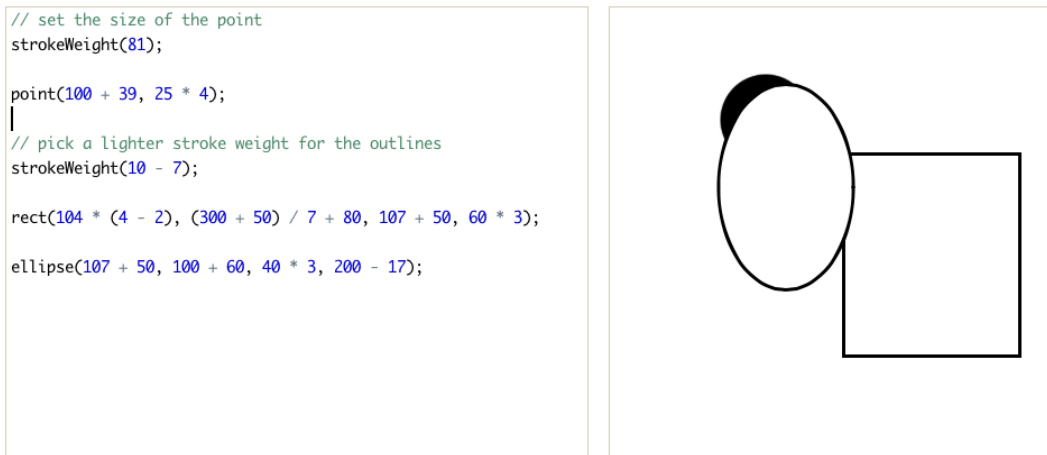
<p>Bret Victor's 'Inventing on Principle' gained a lot of attention. Different creation-focused live modification systems were shown.</p>	<p>In 2012, Bret Victor received a lot of attention, especially outside the research-community, for his talk 'Inventing on principle' [Victor, 2012b], in which he demonstrated several systems used for creative work with a high level of liveness. One of the big topics of this talk was Live Coding: He showed a graphics-output focused Live Coding editor, which executed drawing code live while editing it and was able to show which part of the code was responsible for which part of the drawing. Also presented was a video-game-specific Live Coding editor, which had time-travel functionality similar to the spreadsheet programming environment of Burnett et al.'s [2001] Forms/3. Lastly, he demonstrated a simple general-purpose Live Coding editor that visualized individual variable values by using their string representations (see Figure 1.1).</p>
<p>One of those was a general-purpose Live Coding editor.</p>	<p>Victor's talk inspired several developers to create their own Live Coding environments. Among others, a <a href="#">Kickstarter project</a><sup>2</sup> was created to build a new IDE featuring Live Coding, called <a href="#">Light Table</a><sup>3</sup>. Also, <a href="#">Khan Academy</a><sup>4</sup>, a non-profit organization with the goal of providing free education, created their own a browser-based Live Coding environment to teach programming (see Figure 3.3) [Resig, 2012].</p>
<p>Victor's talk inspired others to develop Live Coding environments.</p>	<p>Victor saw Khan Academy's approach to Live Coding and programmer education, decided it was not what he had in mind and wrote a long article explaining how it could be improved in his opinion [Victor, 2012a]. In this article, he built on his earlier simple Live Coding editor and extended it with a lot of nice variable visualizations and the ability to navigate through the execution history of the code (see Figure 3.4).</p>
<p>Victor proposed a new Live Coding editor with a much higher focus on descriptive visualizations for runtime states.</p>	<p>Figure 3.4 shows a view from his refined prototype. The source code is displayed on the left, the graphics output produced by the code is shown on the right and there is a view in the middle showing the state of the program af-</p>

<sup>2</sup><http://www.kickstarter.com/projects/306316578/light-table>

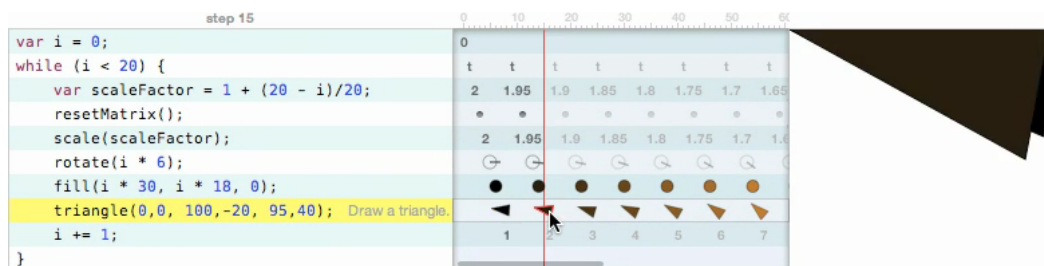
<sup>3</sup><http://www.lighttable.com/>

<sup>4</sup><http://www.khanacademy.org>



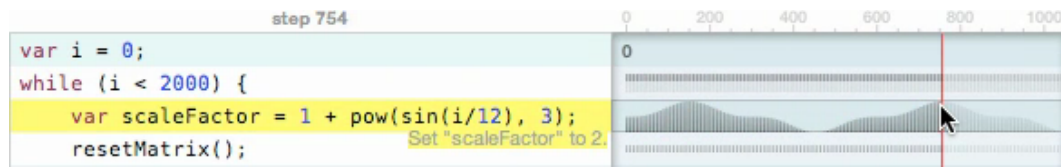


**Figure 3.3:** A screenshot from the Khan Academy Live Coding implementation for a simple example. Source code is on the left, the graphics output of the program is shown on the right. Only the graphics output is shown, no individual variables or program state is visualized. Animations will be correctly updated mid-animation when the source code changes.



**Figure 3.4:** Screenshot showing how several values in a loop can be visualized to facilitate comparison between them. Image is a frame from a video contained in Victor [2012a].

ter each individual line. That one is especially interesting, since most of the program is inside a loop; thus, each statement is executed multiple times. He solves this problem by displaying results of multiple executions of the same line on the x-axis. Results from two different lines are not positioned at the same x-position but slightly skewed, to show which of them executes first. This way, to follow the program flow, the user can follow the results from top to bottom in the first column and then, at the end of the column, start at the top again with the second column. They can also just view the, e.g., 40<sup>th</sup> statement by using the timeline at the top.



**Figure 3.5:** Zooming out gives an overview of the values the variable can have and how these change during the while-loop. In this example it is easy to see that the variable values describe a cubic sine wave. Image is a frame from a video contained in Victor [2012a].

Each semantic data type has its own visualization.

Depending on the type of the variable and some more semantic knowledge about the kind of data that should be displayed, he uses a whole slew of different visualizations. Boolean values are shown with ‘t’ or ‘f’, the double `scaleFactor` and integer `i` are simply printed as a string, the angle given to `rotate` is shown with a clock-like circle containing a line pointing in the correct direction, the chosen `fill` color in the graphics context is shown with a correctly colored circle and the command `triangle`, which paints a triangle, actually outputs a small version of the correctly colored and angled triangle.

Showing number values as rectangles enables graph drawing and thus comparisons.

But the interface also allows the user to “zoom out” to get a better overview of how values change in the loop (see Figure 3.5). If there is not enough space to use the string representation of a number value, a filled rectangle is used. These rectangles are set next to each other which leads to the group of rectangles for a statement creating a graph of values. This graph can easily be used to extract information about what minimum and maximum values of the variable are in this loop and how the values change during the loop.

Heinen implemented a prototypical Live Coding environment and evaluated in a small user study.

Also inspired by Victor’s talk, Heinen [2012] developed a Live Coding editor prototype that implemented some of the presented ideas. He came up with five different interface versions and evaluated them in a very small qualitative study (6 participants). After selecting, improving, and implementing one of those versions, a quantitative user study was done comparing the prototype to an IDE without Live Coding functionality. However, no significant differences in task completion times were found between participants with and without a Live Coding editor, although the mean task completion time in the Live Coding conditions was

slightly shorter. Although the prototype was evaluated, the task given to the participants was a very simple one not comparable to real programming tasks and the prototype had several severe limitations that would have prevented it from being useful for such real-world tasks.

Building on the prototype of Heinen [2012], Belzmann [2013] created a server-client architecture that can be used for continuous execution of JavaScript code. He did not provide a new interface, but laid important foundations to enable the creation of full-featured Live Coding environments. Our prototype is based on the foundations of Heinen [2012] and Belzmann [2013].

Belzmann [2013] built a server-client architecture as a foundation for Live Coding environments.

To conclude, there have been a lot of interesting ideas what kind of Live Coding could be useful. However, except for Forms/3 with its mixed results and Heinen's [2012] prototype with a very small and simple study, none of the proposed prototypes has been evaluated formally. So, we still do not know what kind of Live Coding editor works well and which of all the proposed features are actually helpful and which can be left out. One would hope that there is at least some fundamental research into developer behavior, how developers program, what mistakes they make etc. which might make it possible to judge whether these prototypes could solve some of the problems developers have. We will look at this direction of research next.

Almost none of the Live Coding prototypes have been formally evaluated.

## 3.2 Research into Developer Behavior and Errors

In this section we are giving an overview on the research into developers' code understanding and general programming behavior as far as it is related to Live Coding. First we present some of the research looking at the effects of Liveness on developers' behavior and problem solving in general. Second, we present some more research into developers' programming behavior and the kind of errors they make.

### 3.2.1 The Effect of Liveness

It is rather unclear whether increased liveness supports problem solving in general or programming specifically. Most results are mixed or even contradicting each other.

Progressive evaluation is important to novices and experts alike.

On the one hand, Green and Petre [1996] explain that “progressive evaluation”, the ability to evaluate their programs (and thus their progress) at frequent intervals is very important for novices and although experts can solve problems without it, they even use it more than novices, if they have the possibility to do so. Then again, basic research into problem solving seems to imply the opposite: Gilmore [1995] describes several studies showing that increased liveness in the interface actually interferes with the problem solving ability. He refers to a study in which programmers had to solve different instances of a tower of Hanoi puzzle in the optimal number of moves and another one where they had to solve an Eight Puzzle, one group using a command line tool, one using a direct manipulation interface. The subjects in the less-live group actually reached the optimal number of moves earlier than the ones in the more-live group. The question is now, whether these results can be applied to programming (and debugging) just because both activities, programming and puzzle solving, are a kind of problem solving.

Some research seems to imply that liveness inhibits problem solving abilities.

Results for helpfulness of liveness in Forms/3 were mixed.

Wilcox et al. [1997] built a non-live version of Forms/3 and compared it to the live one in a study with two different debugging tasks. 29 participants worked on these tasks in which 5 and 7 bugs respectively had been planted that should be fixed. Significant differences in debugging accuracy in favor of liveness could only be shown for one of the tasks (task 1). When looking at the time to fix each of the 12 errors in the tasks, Cook et al. [1997] found that 7 of them were fixed significantly faster in the live condition (4 of task 2, 3 of task 1). For 3 errors they could not find a significant difference and for 2 (both in task 2) they even found that the non-live participants fixed the errors significantly faster. These results imply that the advantage of an increased liveness level seems to be strongly dependent on the task and the kind of error to fix. They do not give an ex-

Some errors could be fixed faster live, some non-live.

planation as to why they think the helpfulness of liveness was different for the two tasks.

Saff and Ernst [2003] tried to confirm that a longer time to error leads to a longer time to fix. They looked at programmers using test cases to determine the correctness of the code and measured the time to error, as the time when the developers ran test cases and the test failed (so the developers noticed the software error in form of a failure), and the time to fix, as the time from the failing test case to the test passing again. The time of the introduction of the error was determined by recording all changes to the source code and checking all intermediate versions of the source code against the test suite to determine when a test would have failed for the first time. The gathered times to error and times to fix were then compared for each case and a correlation between the two was determined. However, with an  $R^2 = 0.14$  for a Perl program and  $R^2 = 0.34$  for a Java program the correlation is rather small. Also, all of this data comes from a single developer, so the results cannot be generalized.

The link between time to error and time to fix is not conclusively shown.

The Continuous Testing tool developed by Saff and Ernst [2004] was mentioned before. To test whether it had a positive effect they tested it with 22 student programmers. Half of those used the Continuous Testing tool, a quarter used just Continuous Compilation and a quarter used neither. The participants worked on two different tasks as part of their regular coursework. Their changes to the code, whether they manually ran tests, how long it took them to reach a solution was recorded and the correctness of the solution was measured. Saff and Ernst [2004] found that the kind of tool used predicted whether participants were able to successfully solve the task. Continuous Testing participants were three times as likely to complete the task and Continuous Compilation participants were twice as likely.

Using a Continuous Testing Tool improves task success rate significantly compared to no such tool.

However, it is not completely clear from the paper and a corresponding Master thesis by Saff [2004] whether the difference between Continuous Compilation and Continuous Testing is actually statistically significant, since no results of the posthoc test are given. At least, more direct feedback seemed to have a positive effect in that case. Saff [2004]

The relationship between Continuous Testing and Continuous Compilation remains unclear.

The tool used had no significant effect time to error and time to fix.

A more detailed analysis of the errors occurring in the study was not possible due to the study setup.

We are interested in simple, low-level errors, their causes and ways to prevent or quickly fix them.

An example for a simple error is a syntax error. Those already have good support in IDEs.

notes that for one of the tasks the tool used also predicted the average duration in which the code was compilable and the average time when it was not compilable.

Several other data points have been recorded and tested, including the time to error and the time to fix for each error. However, no significant differences in these times for any of the predictors have been reported [Saff and Ernst, 2004]. Also, since the time to error was simply determined by a failing test case, this ignored the possibility of the programmer becoming aware of the error any other way; thus, ignoring errors that were not caught by the provided test suite. These more detailed analyses were not possible in the study setting, since the only monitoring possible was monitoring the changes of the participants to the source code, but identifying more errors and determining the time to error more correctly would probably only have been possible by using video recordings of the participants. Again, a breakdown of what types of errors occurred is missing.

### 3.2.2 Programming Errors

We will now look at the more general developer behavior and programming error research. The problem with most of this research is that it focuses on program understanding and; therefore, on higher level cognitive problems. However, we are looking for the kind of everyday errors programmers make, that happen even to experienced programmers but could be prevented or easily fixed with a tool. One example would be syntax errors: Every programmer makes them often, they are simply typing errors, sometimes they result from having forgotten the exact syntax of a seldom used construct (e.g., a switch-case statement, which is slightly different in many languages). However, these kinds of problems are already well-mitigated in many modern IDEs: Syntax errors are usually easy to discover using today's tools and are often found shortly after they have been introduced. We are also not looking at domain specific problems and we are not looking for high-level errors like an overall faulty software architecture or misconception of the overall workings of a part of the system.

There is a famous saying among programmers: “There are only two hard problems in computer science: (Implementing) caching correctly, naming things and off-by-one-errors”. Out of the three errors in this saying, the one we are most interested in, is the one of the punch-line, usually forgotten and the least abstract one: The off-by-one error. Almost every programmer probably made such a mistake, probably hundreds of them, e.g., stopping the iteration at `array.length` instead of `array.length-1` or similar. The errors we are interested in (and the accompanying research) are errors that are like the off-by-one errors. We will now review some of the research looking at programmers’ errors in this context, mostly in chronological order.

We are looking for errors like ‘off-by-one’ errors.

Gould [1975] looked at 10 experienced FORTRAN programmers fixing different kinds of software errors. 4 programs extracted from the IBM scientific subroutines package were used to create 3 modified versions for each of the 4 programs, such that each version had one of three different kinds of software errors: An assignment error, an array error, or and iteration error. The times needed to identify an error, the number of errors not found and the number of incorrectly identified errors were all measured for each participant.

Participants had to find an error in 12 simple FORTRAN programs.

Johnson et al. [1983] looked at the software errors 206 novice Pascal programmers introduced working on a simple programming task by recording the first syntactically correct version of the program and analyzing the errors in these versions. 783 errors were categorized as “MISSING” (a required part of the correct implementation is missing), “SPURIOUS” (a part of the implementation should not be there), “MISPLACED” (a part of the implementation was correct but inserted at the wrong position in the program), and “MALFORMED” (a part of the implementation was at the correct position but incorrectly implemented). They also classified them by the component of the implementation the errors belonged to, i.e. whether they were related to input, output, initialization, updates, guards (if- and case-statements), syntactic, complex plans (requiring more than one component), or declarations.

Students errors in working on programming tasks were analyzed by looking at the compilable version and the errors it contained.

For each of the combined error classes (e.g. “MISPLACED

INITIALIZATION”) the number of errors of that type was reported. The majority of the errors were of the “MISSING GUARD” class. This is a problem that should be much easier to find using a Live Coding system. The errors were then classified in even more subcategories, 99 in total.

While the breakdown of different errors is quite interesting there are several problems with this study. First, it only considers novice programmers. Second, they looked at the first syntactically correct version of a program, thus ignoring bugs that were introduced later on. Additionally, some of the bugs could simply be incomplete implementations the programmer was aware of. Also, while frequencies of the bug types were reported, it is unknown which of those bugs were difficult to fix for the programmer, since no time to error or time to fix was recorded.

Students errors in working on programming tasks were analyzed by looking at the compilable version and the errors it contained.

Spohrer and Soloway [1986] looked at the software bugs novice non-science student programmers created and classified them. They conclude that the majority of bugs were not caused by misunderstanding language constructs but result from other problems. While they sorted the 284 bugs found into 101 categories to then decide for each category whether the bug was caused by misunderstanding a programming language construct, they do not report the frequency of different types of bugs. They also just looked at the first syntactically correct version of a novice’s program to find as many bugs as possible before they have been fixed, leading to the same problems discussed above. Lastly, by only looking at novice programmers it is unclear which of the bugs are only due to the programmers’ inexperience and which also happen to experienced programmers.

“Bug War Stories” by expert developers were analyzed and the bugs categorized.

Eisenstadt [1993] asked programmers on an online bulletin board for their “tales of debugging” and got 78 responses. He then classified these bugs in different categories and looked at some other attributes of the bugs. However, he specifically asked for hard-to-find bugs in large projects, but we are more interested in the simpler bugs in any kind of project.

Ko and Myers [2003] created a new model of programming



errors and tested its application with 7 programmers working on different tasks using the Alice programming language and environment. In addition to the errors the programmers made they also looked at the breakdowns that caused these errors. A breakdown, in this model, consists of a problem (attentional, knowledge or strategic), an action (e.g. "Implementing" or "Understanding") and an artifact (e.g. a specific piece of documentation, an algorithm or even an error). While they do report the frequencies of the different types of breakdowns in [Ko and Myers, 2003] and report which breakdown-chains most often occurred in Ko and Myers [2005], they did not classify the different errors and thus do not report their frequencies. However, Ko and Myers [2003] report the distribution of errors over different artifacts, e.g. noting that most errors are related to algorithms (33.3%) and language constructs (30.4%).

The causes of 7 Alice programmers' errors are analyzed in detail.

Errors are not categorized and not reported in detail.

It is unknown whether the underlying breakdowns causing errors or the, as Ko and Myers [2005] call it, "surface qualities" of an error are more useful to predict whether a Live Coding tool would help prevent or more easily fix these errors. At least the data presented is not detailed enough to be able to judge which of those errors could have been prevented by Live Coding. Also, Alice is a rather simple language [Ko and Myers, 2008], targeted at non-programmers [Conway et al., 2000] and Ko and Myers [2005] themselves suspect that the results from their method might be hard to replicate since "even the smallest interactive details of a programming system can cause cognitive breakdowns" [Ko and Myers, 2005], which might differ a lot between different systems.

This kind of detailed analysis is likely highly sensitive to the programming environment and other external factors.

They later developed a tool, the Whyline. A first version was developed for Alice [Ko and Myers, 2004] and tested by repeating the study from Ko and Myers [2003] with 5 participants and comparing it to the previous results. But this time, they only looked at how the debugging time changed and how many tasks were completed, not what kind of errors could more easily be fixed using the new tool or whether the distribution of errors that did occur changed. Later a version for Java was presented in [Ko and Myers, 2008]. It was tested with 20 participants (10 with the tool, 10 without) that worked on two simple debugging

5 more participants were tested, but only their task completion times and task correctness was reported.

20 Java developers were asked to find two bugs in a real application, but could not change code.

task [Ko and Myers, 2009]: The subjects had to find a bug in an existing program and could not change the code, since the prototype did not contain an editor. So no code-editing was observed.

<p>10 Java developers working on 5 maintenance tasks are observed, but errors are not reported in detail.</p>	<p>Ko et al. [2005] observed 10 Java programmers working on five different maintenance-tasks during a total time of 70 minutes. Three of those required only a single-line change, but two required creating and modifying several bits of code. Although Ko et al. [2005] state that “any errors introduced by the programmer” are recorded, those are not reported in the paper. Instead only high-level conclusions from the gathered data are given.</p>
<p>Participants were observed working in a known and an unknown code base. Only questions they ask are reported.</p>	<p>Sillito et al. [2006] looked at “Questions Programmers Ask During Software Evolution Tasks” and at the information a programmer has to gather to successfully perform a change in a medium or large codebase. To do so they let 9 participants work in pairs on a total of 12 software change task sessions à 45 minutes in an unknown codebase and let 16 programmers work (mostly) alone on 15 software change task sessions in a known codebase. They found several questions for which we expect Live Coding to be useful when trying to answer them, including “What are the arguments to this function?”, “What are the values of these arguments at runtime?”, “How does this data structure look at runtime?”, or “Why isn’t control reaching this point in the code?”.</p>
<p>The reported questions are interesting to our cause, but they likely differ from questions a programmer asks when implementing a new piece of software.</p>	<p>The questions were grouped into categories and a frequency distribution of the questions, showing how often each type of question was asked, was given in Sillito et al. [2008]. After that a list of tools was qualitatively evaluated from literature research to judge whether a given tool could answer any of the identified questions. However, it was only checked whether a tool provided “full” or “partial” support. Also, the tasks were clearly maintenance tasks, for the first study even in unknown source code. This affects the kind of questions asked. The questions asked during a fresh software construction task would likely be different. Even the number of questions asked for each type of question varied quite a lot between the two studies, most likely because one of the studies was done in unknown source</p>

code and the other in known source code.

The study by LaToza et al. [2007] had 13 participants working on two different improvement tasks on jEdit. Each of the tasks described a problem with the architectural design of jEdit and asked the participants to fix it. When analyzing the data from the study, LaToza et al. [2007] mainly looked at how participants navigated through the code and what information they were gathering. Although they also examined the changes introduced by participants, they only checked the final changes of the participants and did so on a rather abstract level (e.g. “Six novices and one expert extracted an update method from `getFoldLevel` and had `doDelayedUpdate` update folds by calling this method” [LaToza et al., 2007]). They did not consider low-level errors programmers made while implementing these changes, but only looked at the conceptual error.

13 participants worked on a software change task in Java. Only the navigation of participants was analyzed.

Three years later, they reanalyzed the data in [LaToza and Myers, 2010]. This time, they clustered the edits into changes and looked at each such change to determine whether it contained a bug. If so, they tried to determine what the root cause of this bug was (e.g. missing information or a false assumption). Half of all changes contained a bug and of those, again half were determined to be related to reachability questions. Sadly, they do not report the cause of the other half of bugs. Although not explicitly stated, we assume from the description in [LaToza and Myers, 2010] that they concentrated on conceptual errors again, not typos or similar smaller errors.

This time, the errors participants made were analyzed, but only reported focusing on one specific cause of errors.

Another study is described in the same paper, this time actually observing real developers working on real tasks. But LaToza and Myers [2010] were only able to take notes about the developers actions and record audio from the session, which did not enable them to look at smaller and higher-frequency errors, but only analyze how developers spend their time on a coarser level.

Fenwick et al. [2009] looked at the compilation errors occurring in beginning computer science students’ programs, confirming an earlier study by Jadud [2004]. The top 5 errors of Fenwick et al. [2009] appear in the top 6 errors of

<p>Compilation Errors of programming novices were analyzed and classified, repeating and confirming an earlier similar study.</p>	<p>Jadud [2004], and both studies agree that these 6 make up more than 60% of novice programmers compilation errors but they do not agree on the exact ranking. They are “Missing semicolon”, “Unknown variable”, “Bracket expected”, “Illegal Start of Expression”, “Unknown class”, and “Unknown method”. However, they only looked at compilation errors and only gathered them, when the students explicitly compiled the programs. While the abstractness of the errors matches the level that is interesting to us, we are interested in semantic errors which are not diagnosable by a compiler. Obviously, what errors the compiler does find depends on the programming language (and the compiler) used. For example, in JavaScript semicolons are often optional, and methods can be assigned to objects at runtime, so the compiler cannot know whether a specific method will exist at runtime.</p>
<p>Errors of novice programmers were analyzed, categorized and counted, but only those that novices could not solve themselves.</p>	<p>Bryce et al. [2010] also examined novice computer science students’ errors, but they actually concentrated on the semantic errors. Whenever a student had a problem that they could not solve themselves, they would come to the tutor’s lab and ask for help. Before and after fixing the problem the students were asked to describe their problem and this description was saved. This data was then analyzed to identify common errors in students’ programs. This was done for both an introductory computer science course (CS1) and the more advanced following course (CS2) to be able to see how the error distribution changes for more experienced programmers.</p>
<p>Software Errors similar to the MISSING GUARD category by Johnson et al. [1983] were quite common, again.</p>	<p>The authors identified 20 different software error types, one of them called “Loop and switch statements”, which includes the aforementioned common off-by-one errors. After the general “problem solving” category this is the most common error type for CS1 students with 28 of the 210 bugs belonging to that category, but only 11 of the 234 bugs reported by CS2 students belonged to that category. Interestingly, the most common category of Johnson et al. [1983], MISSING GUARD, also included all incorrect guards for ending a loop, which might make up a large portion of the observed “Loop and switch statements” error. So although these two studies are almost 30 years apart, there seems to be some continuity in the kinds of errors programmer</p>

make.

While some of the bug categories are a bit too abstract or broad to be helpful for our research (like “Problem Solving”) and some of them are rather domain specific (e.g. bugs related to “File I/O”), which might differ a lot between different programming languages, several of the categories identified are quite interesting. But there are also some problems. The first problem with the study is that it, like many before, only looks at novice programmers. The bigger problem, however, is that it only looks at bugs the students could not fix without help. It could be the case that CS2 students made less “Loop and switch statements” bugs, but another possibility is that they simply learned to debug and fix those errors themselves and still make as many of them as before. Therefore, they might still waste a lot of time trying to fix those bugs but no longer need the help of a more experienced programmer to do so. Nevertheless, a tool that makes fixing these bugs easier could be helpful.

Errors students to find and fix themselves could not be captured.

### Summary

Many of the studies we found focus only on novices’ errors, but do not analyze which of those errors still happen to experienced programmers. The studies examining experienced developers’ behavior often just look at debugging, often just letting programmers find a bug, without ever changing the code. In those few cases in which experienced programmers are actually asked to write more than a few lines of code, researchers focus on their high-level strategies or errors most of the time. Finally, in the rare cases that researchers closely observed experienced programmers implementing a non-marginal piece of code and even recorded the kinds of errors those programmers made, no attempt is made to classify them and most are not even reported. Therefore, none of the studies mentioned above answer our question what kind of errors experienced programmers make everyday and which could be prevented by a tool like Live Coding (or whether there are no such errors). Some provide important hints, though,

Most studies did not provided the information we needed.

like Ko and Myers [2005], Johnson et al. [1983] and Sillito et al. [2006].

## Chapter 4

# Prototype

In this chapter we will describe the prototype we developed and why we chose to design it this way. We first explain our motivation to extend and modify the prototype of Heinen [2012]. Next we briefly describe the backend by Belzmann [2013] we decided to use. We then outline the development of our prototype, explaining the design decisions where appropriate and highlighting the problems we found with earlier versions. In the end, we will highlight the changes that were necessary to the client-server architecture for Continuous Execution by Belzmann [2013] to enable the features of our prototype.

### 4.1 Motivation for a New and More High-Fidelity Prototype

Since we wanted to conduct a user study using real-world or close-to-real-world tasks (see Section 5.1) to better understand what errors programmers make in the real-world and which of those could be mitigated by Live Coding, we also needed a Live Coding environment that could support these real-world tasks. As a starting point we decided to use the plugin for [Adobe Brackets](http://download.brackets.io)<sup>1</sup> by Heinen [2012].

We wanted our subjects to solve challenging tasks, which the original prototype did not support.

---

<sup>1</sup><http://download.brackets.io>

```

1 function mySort(theArray) {
2   var temp;
3   for (var j = 0; j < theArray.length-1; j++) {
4     for (var i = j; i < theArray.length-1; i++) {
5       if (theArray[i+1] < theArray[i]) {
6         temp = theArray[i];
7         theArray[i] = theArray[i+1];
8         theArray[i+1] = temp;
9         theArray;
10      }
11    }
12  }
13  return theArray;
14 };

```

```

function mySort([70,30,5,2,10,5,11,12,20]) {
  var temp;
  for (var j = 0; 0 < [70,30,5,2,10,5,11,12,20].length-1; 0++) {
    for (var i = 0; 0 < [70,30,5,2,10,5,11,12,20].length-1; 0++) {
      if ([70,30,5,2,10,5,11,12,20][0+1] < [70,30,5,2,10,5,11,12,20][0]) {
        temp = [70,30,5,2,10,5,11,12,20][0];
        theArray[0] = [70,30,5,2,10,5,11,12,20][0+1];
        theArray[0+1] = 70;
        [30,70,5,2,10,5,11,12,20];
      }
    }
  }
  return [30,2,5,5,10,11,12,20,70];
};

```

**Figure 4.1:** Heinen [2012] version of the Live coding plugin for Adobe Brackets IDE! Adobe Brackets displaying code (left) and the evaluated results (right) of simple bubble-sort function.

Adobe Brackets is an Open-Source IDE for Web Development, focused on HTML, CSS and JavaScript and has a nice Plugin API that Heinen [2012] already used successfully.

#### 4.1.1 Limitations of Heinen’s [2012] Prototype

Code outside of a function could not be evaluated and only on function’s execution could be shown at any time.

The prototype by Heinen [2012] had several limitations that were appropriate for the study conducted in [Heinen, 2012] but prevent it from being used for real-world tasks. It could evaluate the code of an individual function called with user-selected parameters and show the runtime state of the referenced variables (see Figure 4.1). However, this only worked inside of functions. Code outside of a function could not be evaluated and only the currently selected function was evaluated. Thus, it was not possible to split up the implementation into several functions, provide the outer function with some test-parameter and see how it affects the inner function.

It did not support more than 9999 iterations of a loop and did not respect outer scope of functions correctly.

Also, outer scope of the function would not be respected, i.e., if a function was declared inside another function and referenced some of the outer function’s variables, these would not be defined (see Figure 4.7 for an example). Lastly, the prototype was severely limited regarding loop constructs: It did not support more than two levels of nested loops and it did not support more than 9999 iterations of a loop. The number of iterations limitation was introduced to prevent infinite loops, which would also stall the plugin. This was not a problem in the user study since none of the tasks needed that many iterations, but we cannot make that assumption about real-world source-code.



In addition to the functional limitations, there were also a number of minor and not-so-minor usability flaws in this version of the prototype. For example, the iterations in a loop can be stepped through using arrow buttons next to the iteration index. However, for a large number of iterations this can become quite cumbersome, as mentioned by several users in the first qualitative study of Heinen [2012]. To make it easier to go to a specific iteration, Heinen [2012] added a button to go to the first and the last iteration and made the iteration index editable, so a specific iteration index could be typed in. However, it is still not possible to “skim” through several iterations by, e.g., scrolling through them.

Another problem was that, whenever the code was reexecuted, a lot of data was lost. Firstly, the data used as test-input to the function was not saved when the code was changed. This means, whenever the user changed the code, the input data had to be reentered. Secondly, whenever the input data was changed or the code was changed, the selected iteration of a loop was reset to the first iteration of said loop. Both of these limitations made it a lot harder to quickly test different versions of the code or different inputs on the same code.

Lastly, the code to evaluate was simply executed on the main thread, forcing the user interface to wait for the execution to finish, potentially making the user-interface unresponsive. Again, this was not a problem for the simple problems tested during the study of Heinen [2012], but would be a problem in real-world usage.

## 4.2 A New Backend

Belzmann [2013] improved the execution backend of Heinen’s prototype a lot by splitting up the visualization and the code execution part into a server-client-architecture and building a robust server that got around several of the limitations in Heinen’s version. He made the execution in the background possible for arbitrary code including arbitrarily many nested functions and more complicated con-

Skimming through a large number of iterations is hard using the UI of the old prototype.

Both specified input data to a function as well as the selected iteration of a loop were lost when the code was reexecuted.

structs such as functions declared inside functions. Also, all variable values in the complete source file, no matter what function they belong to, will be evaluated and can be reviewed by the user, given a suitable user interface (UI).

If the executed code has an infinite loop, the front-end will no longer freeze.

Splitting up the implementation in a client and a server part not only increases reusability of the code (e.g. new UIs can more easily be built by simply implementing the specified communication protocol and using the data provided by the server), it also has the advantage of preventing the UI from stalling when the backend executes non-terminating code. JavaScript, by default, is single-threaded; thus, a simple implementation of such a Live Coding plugin would just block the complete Live Coding UI and in Adobe Brackets' case even block the complete editor. Separating the UI and the backend into two separate processes also has the advantage that the client can continue to run even if the backend encounters an unrecoverable exception and crashes. The continuous execution server implemented by Belzmann [2013] even spawns separate child-processes for each execution to keep the server responsive while executing and prevents it from crashing. This enables a very smooth recovery from non-terminating code, since the server will just kill the obsolete child-process and start a new one when it receives new code to execute.

A complete JavaScript file is instrumented and executed, not just a single function.

In contrast to Heinen [2012], who simply extracted the currently selected function from the code, gathered the specified input data and called the function with this input data, Belzmann's [2013] version simply instruments and executes a complete JavaScript file. This way, the possibility to specify arbitrary input data to a function is lost for now, but it can simply be augmented by adding a call to the function in the code directly. This also solves a problem of Heinen's version that did not become apparent with small code examples: When specifying input data for a function in the Live Coding view, it is unclear in what context this function call should be executed. Consider the following code example:

```
var a = 2;
function bar(b) {
    var c = b + a;
}
```

```
a = 3;
```

When now specifying input 5 for function `bar` what should the value of `c` be? In Heinen’s visualization of the input fields it is unclear to the user whether `a` in that case has the value 2, 3, or `undefined`. Heinen chose the third possibility for simplicity. In Belzmann’s [2013] version the user specifies the location of the call explicitly, e.g.

```
var a = 2;
function bar(b) {
  var c = b + a;
}
a = 3;
bar(5);
```

This way, it is always clear what the outer scope of the function call should be. Of course, this is somewhat more laborious than just inputting the parameters. Also, these calls pollute the code, but we still decided to keep this solution in our version, since we considered it to be an acceptable limitation. In a later version, this feature can be added again, but implementing it correctly (respecting all outer scope) is not trivial in the current implementation of Belzmann’s [2013] continuous execution server and also requires some careful UI design to clearly show in what context this test-call to the function is executed. Thus, we decided to concentrate on other features of our improved prototype.

### 4.3 The Prototype

Heinen’s [2012] plugin for Adobe Brackets had some limitations that prevented it from being used for the tasks we wanted developers to solve during our study (see 5.2—“Designing the Tasks”). To overcome these, we used Belzmann’s [2013] backend for executing the code, since it already fixes many of the functional limitations of the first prototype. However, we also listed a few more usability issues in Section 4.1.1 that we tried to fix with our prototype. We will now describe how we changed the prototype

Due to outer scope, the result of a function can be ambiguous even if the input is fully specified.

In the new version, functions have to be called explicitly to test them.

Our prototype uses Belzmann’s [2013] back end, so we only have to provide the front end.

```

1  var x = 123, c = 1;
2  var f = 0,
3      g = 0,
4      h;
5  var y = 36;
6  var z = y - x + 30;
7  var a = z * z;
8  var b = Math.sqrt(a);
9  b = c = 2;
10 var i;
11 for (i = 0; i < 10; i += 1) {
12     b *= c;
13 }

```

123 1  
0 0  
36  
-57  
3249  
57  
2  
0 1 2 3 4 5 6 7 8 9 10  
4 8 16 32 64 128 256 512 1024 2048

**Figure 4.2:** First version of the prototype, inspired by Victor [2012a]. Each value is displayed slightly right of the preceding value in execution order. This way it is easy to see in what order they were executed. Notice the values in the for loop jumping between the two lines nicely visualizing how the execution progresses in the for-loop.

of Heinen [2012], what changes were necessary to the backend of Belzmann [2013] and how our prototype evolved.

### 4.3.1 Implementing Victor’s [2012a] Visualization

Although Heinen [2012] already tested different visualizations of the Live Coding preview (the right view in Figure 4.1), including ones that interleaved the preview with the code, we still found it to be rather cluttered and tried to simplify it. While the version chosen by Heinen [2012] is the one that was preferred out of 5 different versions in a qualitative user study, one of the six users was first confused when seeing the Live Coding preview because he did not recognize it as the evaluated part, but thought it was just code. Another participant was worried about losing too much screen space.

Heinen did not test a version similar to Victor’s [2012b].

We considered Victor’s [2012a] version to be less cluttered and started from there.

Heinen did not evaluate a version similar to the one shown by Victor [2012b,a], which shows just the resulting values on the right without the code around it. We expect this version to solve the concerns of the users mentioned before, because it uses less screen space and is clearly different from the source code view, while still avoiding the problems of other versions tested, which interleaved the values with the code in the editor view. Therefore, we implemented a version resembling Victor’s [2012a] example. For each line, it just shows a value representing the execution of a line, usu-

```

19 for (j = 0; j < theArray.length - 1; j += 1) {
20   for (i = j; i < theArray.length - 1; i += 1) {
21     if (theArray[i] > theArray[i + 1]) {
22       var temp = theArray[i];
23       theArray[i] = theArray[i + 1];
24       theArray[i + 1] = temp;
25     }
26   }
27 }
28 var a = 2;

```

	0	1	2	3	4	5	6	7
	70	70	70	70	70	70	70	70
	30	5	2	10	5	11	12	
	70	70	70	70	70	70	70	

**Figure 4.3:** The skewed values become a problem if many values have to be displayed. E.g. after a for-loop with many iterations the values resulting from code after the loop cannot be seen on the screen anymore, since they have to be off-screen to the right to follow the layout logic. In this example the result of the declaration `var a = 2` is off-screen because of the preceding for-loop’s many iterations. Some of these problems can be mitigated by automatic and manual scrolling but it is still a problem to get all the interesting values on screen at the same time.

ally the result of an assignment operation, instead of the complete code around it (see Figure 4.2). In addition, there is a slight offset of later runtime state values compared to preceding runtime state values, which makes it easier to understand in what order the code is executed.

### Problems with the Skewed Design

When looking at larger examples, we noticed that the values would quickly drift off-screen, because many values were displayed and each value had to be slightly right of the preceding values (see Figure 4.3). We tried to fix this by introducing an automatic scrolling functionality that would scroll the leftmost value of the currently selected line to the left border of the preview, showing as much of the selected and the following lines as possible. This mitigates the problem, but does not solve it completely. In the example of Figure 4.3, the result of the last declaration is still off-screen, although this is still a rather small example.

The skewed design makes it difficult to show all the values of interest if no zoomable interface is implemented.

One of the problems—not enough values being displayed in the preview to be usable—could likely be solved by making the interface zoomable, as proposed by Victor [2012a]. However, doing so raises other questions and we specifically did *not* want to focus on the visualization of the Live Coding values and were afraid that our results would depend too much on the visualization. Therefore, we decided

```

43 var theArray = [70, 30, 5, 2, 10, 5, 11, 12, 80];
44 var testString = "hallo";
45 for (var j = 0, c = 2; j < theArray.length - 1; j += 1) {
46     for (i = j; i < theArray.length - 1; i += 1) {
47         if (theArray[i] > theArray[i + 1]) {
48             var temp = theArray[i];
49             theArray[i] = theArray[i + 1];
50             theArray[i + 1] = temp;
51         }
52     }
53 }

```

Preview:

```

[70, 30, 5, 2, 10, 5, 11, 12, 80]
"hallo"
< 1/8 > 2 truthy(true)
< 1/8 > 0 truthy(true)
truthy(true)
70
30
70

```

**Figure 4.4:** A screenshot of the final prototype. Only one iteration is shown at a time and assignment results are shown. In line 45, two values are shown since two variables are initialized in the initialization part of the for-loop. The cursor hovers over the first value in line 45, causing a tooltip with the variable name to appear below it and the statement in the source code that produced this value to be highlighted. Line 49 is highlighted because it is currently selected and the highlight extends into the preview to make it easier to find a corresponding line in the preview. The `truthy(true)` values are the values of the `for`-loop- and `if`-conditionals on the left. The value in brackets gives the actual result of the conditional, the value in front shows whether this value will be interpreted as true (truthy) or false (falsy).

to abandon this visualization in favor of a much simpler one, hoping that this simple visualization would still allow our subjects to get enough of an advantage out of Live Coding.

### 4.3.2 Final Version

In the final version, we only show one values per statement in a line of code and keep the visualization simple.

For our final version we reverted our decision to show as many values as possible to get rid of the iteration selectors of Heinen [2012]. Instead, we reintroduced the selectors and are now showing at most one value for each variable reference in the code on the left (see Figure 4.4). One important difference to Heinen [2012], in addition to not showing the code, is that we always show complete results of assignments, whereas Heinen showed the individual values of a calculation (as shown in Figure 4.1).

**New Iteration Selectors** But the iteration selectors have been changed in comparison to Heinen [2012]. Instead of adding start and end buttons and making the iteration index editable, we decided to make it scrubbable. This means a user can click and drag on the iteration selector to change

1	function fibonacci(n) {	< 1/21891 >	20	
1	function fibonacci(n) {	< 2/21891 >	19	
1	function fibonacci(n) {	< 3/21891 >	18	
1	function fibonacci(n) {	< 4/21891 >	17	
1	function fibonacci(n) {	< 6/21891 >	15	
1	function fibonacci(n) {	< 10/21891 >	11	
1	function fibonacci(n) {	< 20/21891 >	1	I
1	function fibonacci(n) {	< 40/21891 >	5	I
1	function fibonacci(n) {	< 80/21891 >	7	I
1	function fibonacci(n) {	< 162/21891 >	1	I
1	function fibonacci(n) {	< 5789/21891 >	2	I

**Figure 4.5:** Illustration of the exponential growth of the iteration selector dragging. By clicking and dragging on the iteration selector the selected iteration can be changed. The more the cursor is dragged to the right the more the selected iteration index increases. Growth is linear in the beginning and then changes to exponential growth after a few iterations to reach the last iteration index when the cursor reaches the right border of the editor. This case is not shown here since it would need too much space, but it is indicated by the last one already reaching iteration 5789 of 21891. Dragging a few more centimeters to the right would reach iteration 21891.

the selected iteration. Dragging to the left will decrease the selection index, dragging to the right will increase it. This enables users to quickly skim through a few neighboring iterations and glance at the values in these iterations or scrub through a high number of iterations effortlessly.

The important part is that this dragging does not linearly increase/decrease the selection index. Instead, the growth is exponential and the growth function depends on the number of iterations. It is calculated in a way that dragging to the left/right border of the editor view will always reach the minimum/maximum iteration index. This way, a user can ‘throw’ their mouse pointer to the left/right edge of the screen to reach the minimum/maximum edge of the screen making it much easier to reach this minimum/maximum iteration according to Fitts’ Law [Fitts, 1954]. But since an exponential growth with a small number of iterations would mean that the user would have to drag quite far to reach the next iteration, the first few iterations are reached via a linear growth function, which smoothly changes into an exponential one later. An illustration of this can be found in Figure 4.5.

We implemented new iteration selectors to enable skimming through iterations.

The growth function of the iteration selector is exponential to always make it possible to reach the maximum value with just one movement of the mouse.

We consider the scrubbing functionality to be an expert shortcut.



**Figure 4.6:** The Column-Resize cursor is shown, when hovering over the iteration selector, indicating that the user can drag horizontally.

Even without additional UI elements users can estimate how far they can drag.

We left out additional UI elements to avoid conflicts with other parts of the UI.

On first glance, the possibility for this interaction is not indicated in the UI in any way, which makes this feature hard to detect. We do not think this is a problem, since we see this feature as an expert shortcut. It is not necessary to use the tool, but it will make life easier if it is known.

Intentionally, two independent UI elements have partially been left out: (1) An indication that the user can click and drag on the iteration selector and, after the drag started, (2) an indicator how far the user has dragged and can continue to drag.

The UI element is partially there: When the mouse pointer is over the iteration selector, the mouse pointer changes to a column-resize pointer (see Figure 4.6), which indicates that a horizontal dragging action is possible. If the user tries it, they will likely understand what it does.

A second UI element, showing the user's current drag position and how far they can still drag is not there. Still, the user can determine how far they have dragged by simply comparing the distance of the mouse pointer to their origin. Also, they can look at the iteration index, which changes when they drag. The number of iterations also tell them how far they can still drag. Additionally, dragging to the edge of the Brackets window will always reach the maximum value and we expect frequent users of this functionality to learn this over time, which provides another fixed reference point to determine how far they can drag.

The decision to leave out this UI element was made, because the surrounding UI is very dynamic and it is difficult to determine what kind of values are displayed next to the iteration selector. Since the scrubbing functionality of the iteration selector is used to skim through runtime state in different iterations and this runtime state is next to the iteration selector, a position indicator displayed next to the iteration selector could hide exactly this runtime state that was the reason to use the iteration selector in the first place. Thus, we decided not to display any additional UI elements.

Still, this functionality is somewhat hidden and although it



is discoverable, it is not easy to do so. Therefore, we explained this feature to all our participants during the user study, giving them the kind of knowledge an expert user of the system would have.

**Functions and Nested Functions** Another change visible in Figure 4.5 is the iteration selector now being used for functions as well. In our version of the prototype not only loops have iteration selectors to select a specific iteration, but functions have one as well to select a specific call to the function to display. For example, the function `fibonacci` in Figure 4.5 was called 21 891 times. Next to the function, the arguments given to it in the selected call are shown.

Functions have iteration selectors as well.

```

1  function createFunction(a)
2      return function() {
3          return a;
4      };
5  }
6
7  f1 = createFunction(1);
8  f2 = createFunction(2);
9
10 var b = f1();
11 var c = f2();
12 var d = f1();

```

Call stack details:

- Iteration 1: `< 1/2 >` (selected), `1`, `function`
- Iteration 1: `< 1/2 >`, `1`, `function`
- Iteration 2: `2`, `function`
- Iteration 1: `1`, `function`

**Figure 4.7:** The function `createFunction` is called twice and creates two different functions. The first of those is called twice the second version is called just once. Since the currently selected call to the outer function is the first one, only the two calls to the first version of the returned function are shown.

Nested functions are handled the same way nested loops are handled: Inner functions have their own iteration selector which shows just the calls to the version of the function declared in the selected call of the outer function (see Figure 4.7). This way it should always be clear what the current outer scope of the function is, because it is shown in the selected iteration of the outer function call.

Inner functions are handled in the same way inner loops are handled.

**Improved Value Display** Considerable effort was invested in making the values in the Live Coding preview

```

5 var mo = {};
6 mo.type = "bla";
7 mo.so = {2: 3, "foo": "bar"};
8 mo.date = new Date("2013-01-01");
9 var a = mo;
10
11 function parse($body) {
12     var domBody = $body[0];
13 }

```

```

{}
"bla"
{2: 3, foo: "bar"}
UTC: 2013-01-01T00:00:00.000Z
{type: "bla", so: {2: 3, foo: "bar"}, date: UTC: 2013-01-01T00:00:00.000Z}
$[<body> <form action="wo_en_mensa_ahorn.std.php" n...]
<body> <form action="wo_en_mensa_ahorn.std.php" n...

```

**Figure 4.8:** Values in the preview are colored according to the syntax highlighting in the code editor where applicable. Special cases exist for `Date` objects (line 8), jQuery objects (line 11) and HTML DOM objects (line 12). Object representations that would make the interface slow when being displayed in full are abbreviated using an ellipsis (...) as shown in line 11 and 12. Clicking on the ellipsis expands the complete object representation.

By default, a JSON stringification is used. This is augmented by colors and special cases where necessary.

Circular references are recognized and displayed appropriately.

discriminable and easily recognizable as well. By default, we simply use a string or JSON representation of the values, but we added correct syntax coloring to it. In addition to the default syntax coloring as used by the Adobe Brackets editor, which differentiates objects, numbers and strings, we also included individual syntax coloring for Date objects (green, slightly darker than numbers), HTML DOM objects (blue), and jQuery objects (violet) (see Figure 4.8). JQuery objects are often used as collections; thus, we display them like an array using square brackets. But, to make them easily distinguishable from actual arrays, even for color-blind users, we added a `$`-symbol in front, which is often used to represent the jQuery function. Circular references in objects are properly indicated, as can be seen in Figure 4.9.

```

1 var a = {};
2 a.b = {parent: a};

```

```

{}
{parent: {b: circular(root)}}

```

**Figure 4.9:** Circular references are detected and displayed in the Live Coding preview. The path to the first occurrence of the referenced object is given in parentheses. In this case the value of `b` is the complete object itself).

**Abbreviating Long Objects Representations** When testing our Live Coding plugin to download and parse an RSS feed (see the tasks of our user study in Section 5.2) we noticed that the interface became quite unresponsive. This was due to a function receiving the XML of the feed as an array of bytes, which would then be displayed in the inter-

face. If the XML file of the RSS feed had a size of just 10kB this meant, that an array with over 10 000 entries would be displayed, each entry in its own `<span>`-element and correctly colored according to the syntax color specification. In such situations the modification of the DOM using jQuery in our plugin's UI and even the rendering of that HTML becomes so slow that it is not only noticeable by the users, but hindering them. We therefore abbreviate long strings and long arrays to a fixed length and display an abbreviation indicator ('...') at the end to indicate something was left out. Clicking on this abbreviation indicator expands the value display and shows it completely.

Long values are abbreviated, both to fit on the screen and for performance reasons.



```

1 for (var i = 0; i < 2; i++) {
2   try {
3     var a = foo;
4   } catch (e) {
5     if (i)
6       a.b();
7   }
8 }

```

< 2/2 > 1 truthy(true)  
 foo is not defined  
 caught: foo is not defined  
 truthy(1)  
 Cannot call method 'b' of undefined

**Figure 4.10:** Exceptions occurring in the code are also reported. Exceptions that are later caught appear in yellow, uncaught exceptions are red. If a loop or function or otherwise repeatable block of code produced an exception its iteration selector will have a red border to indicate that some of the iterations had an exception.

**Display of Errors and Exceptions** In addition to values of assignments, iteration variables of loops/functions, conditions of loops/conditionals, and parameters of functions, we also display exceptions that occur. The backend catches exceptions occurring in the evaluated code and reports them back to the client. Our client then looks into the information about the exception and, if a line can be determined where it occurred (usually from the stack trace), it will be displayed there similar to a normal value, but in red. Otherwise the exception will be displayed as a global exception overlaying other data in the preview. Exceptions that are caught in the evaluated code are shown in yellow (see Figure 4.10).

Exceptions occurring in the evaluated code are caught by the backend and displayed in the appropriate location by the our UI.

Since not all values are displayed all the time, but only some selected iterations of loops and calls to functions are displayed, not all exceptions are always visible. If an ex-

If an inner function has an exception, the outer function's iteration selector will be red.

ception occurred in a function call that is currently not displayed, the error itself will not be displayed either. But the iteration selector of the corresponding function will have a red and thicker border to indicate that something went wrong in one of the iterations (see Figure 4.10). This is also propagated through outer functions and loops, ensuring that at least one iteration selector always shows a red border if an uncaught exception occurred, which makes it easier to find that exception.

No value is being shown for simple expression statements, that do not assign a value.

**Using `console.log` to Overcome Limitations** In addition to the missing possibility to provide custom input values to functions, there is another limitation of our prototype compared to Heinen's [2012]. In his version, a line with just an expression (not an assignment) like `theArray;` would print the result of the expression. This is something the backend by Belzmann [2013] does not support, yet. The problem here is that it is again ambiguous what should be displayed for such an expression in general. Consider the following code:

```
myObject.array.push(2);
```

As a workaround, users can use `console.log` to display any value they like in the preview.

What should be displayed here? Simply the return value of the call to `push`? Or the changed value of `myObject`? Most of the time the programmer would likely want to see the new value of `myObject.array`. However, determining that would need a good heuristic and likely extensive testing. Thus we decided to leave this feature out and provide a work-around: Whenever the user wants the preview to show a value that is not shown by default he can use `console.log()` (JavaScript's equivalent of `printf` or `System.out.println`) and give the requested values as an argument. It will then be evaluated and the result will be shown in the preview in the line of the `console.log`-statement. Using `console.log` to print a value in the preview should be very natural to most programmers since most of them use logging statements for debugging anyway. Another workaround would of course be to just assign a variable to itself, e.g. `theArray = theArray;`, which would also prompt the Live Coding plugin to show this value.

Again, this makes using a previously existing feature a bit more laborious. But we are confident that the extra work required is small enough not to prevent developers from using it and we hope that in both cases—having to make function calls explicit instead of just providing test input and having to use `console.log` for outputting special values—our version will have the advantage of forcing the programmer to make their intent explicit.

### Saving the Selected Iteration Through Code Changes

When discussing the limitations of Heinen’s [2012] prototype we mentioned that the selected iteration is reset when the code is re-executed. In our version the plugin tries to remember what iteration was selected in a loop or function. We identify functions by their location, that is, the starting line and column and the ending line and column. Since only one function can be defined at any point in the source code, this is a unique identifier. This unique identifier is used as key in a map that saves the selected iteration for each function across executions.

The selected iteration is stored by using the location of a function.

However, by changing the code, the location of a function can change as well. Thus, when a previously selected iteration for a function cannot be found we look for neighboring function identifiers, i.e., functions that start or end in the next or preceding line or column to the original function. This way, our storing of selected iterations is robust against most code changes except for ones where several lines are changed at once. But, since the code is reevaluated on every change to the editor, this should not happen too often.

Storage of the selected iteration is robust against minor location changes of the function.

We also differentiate between the displayed selected iteration and the actually selected iteration. The actually selected iteration can be much higher than the current number of iterations. When the selected iteration should be displayed, the actually selected iteration index is coerced into the range of the current iterations. This also makes the saving of the iteration selector robust against code changes. Imagine the following code:

```
for(var i = 0; i < 12; i++) {  
    // do something interesting  
}
```

To achieve additional robustness against code changes the selected iteration is saved independently from the displayed iteration.

The developer now selects iteration 10, because something interesting happens there. He then decides to change the number of iterations to 22. To do so, he first deletes the 1, which leaves him with the following code:

```
for(var i = 0; i < 2; i++) {  
    // do something interesting  
}
```

Now the number of iterations is 2, so the displayed selected iteration will be iteration 2 (with  $i=1$ ), but internally the selected iteration is still 10. Now he adds a second 2 in front of the first, leaving him with the following code.

```
for(var i = 0; i < 22; i++) {  
    // do something interesting  
}
```

The number of iterations is now 22, so the selected displayed iteration will be 10 again. This way, the loss of the selected iteration index due to an intermediate code change with a low number of iterations was avoided and the developer can continue checking their code.

We added an execution indicator to show whether there is still code running.

**Adding an Execution Indicator.** Since our code evaluation is now happening in the background and can potentially take a long time or even never terminate, we decided to add an execution indicator (see Figure 4.11). First, whenever code is executed in the background a busy indicator is shown in the lower right corner of the IDE. In addition to that, a similar busy indicator is shown in the line that was last executed. This way, the developer can not only see when a program runs longer than it should, in case of an infinite loop he can also detect which statements are executed over and over again.

## 4.4 Changes to the Backend

To implement some of the functionality described above we had to make some changes to the backend, since it did not



**Figure 4.11:** Long-running code that is currently being executed. A spinning indicator in the lower right corner of the window shows the current execution and another spinning indicator in line 78 shows that this is the last line that was executed so far.

support all the functionality we needed.

**Nested Functions** The backend was already capable of handling nested functions, but it would only tell the client which function was entered, identified by the function’s location. However, this is not always sufficient, especially in a functional programming language like JavaScript. Consider the following example:

```

1 function foo(a) {
2     return function bar() {
3         return a;
4     }
5 }
6 var f = foo(2);
7 var g = foo(3);
8 f();
9 g();
10 g();

```

In this case the function in line 2–4 is executed three times, but in two different versions. First it is executed in version 1, then twice in version 2. Obviously both versions have the same location in the code but different semantics, so we need to differentiate them. But it is impossible to know how many different versions there will be at compile time, so a static analysis is not possible. Therefore, we extended the instrumentation of Belzmann’s [2013] backend to create a declaration index for each function declaration and function statement and increase it whenever the corresponding

The location of a function is not a sufficient identifier in the case of calls to nested functions.

We extended the backend to register every defined function and assign a declaration index to it, to make it uniquely identifiable.

function declaration/statement is encountered. This way, we can now identify each version of the function uniquely via its location combined with the declaration index.

We had to calculate a source map to be able to map the locations of uncaught exceptions to the original code.

**Exception Display** Although the backend already caught exceptions occurring in the executed code, the location in the stack trace of the error was given in the context of the instrumented code. Since the backend adds quite a lot of instrumentation code to the original code, the reported location is quite different from the actual location. We solved this problem by configuring the [Escoregen](#)<sup>2</sup> code generator used in the backend to also create a source map when generating the code. The source map provides a mapping from the location of a statement in the original code to the location of the same statement in the instrumented code and vice versa. It is then used to map locations in the instrumented code back to the original code; thus, translating the location of the errors.

The source map generation takes a long time, so we do it in an additional background process.

Since the creation of the source map takes a lot of time (roughly as much time as the complete instrumentation and code generation itself), we are spawning yet another process for the source map generation. The process architecture of the backend before and after our change is described in Figure 4.12.

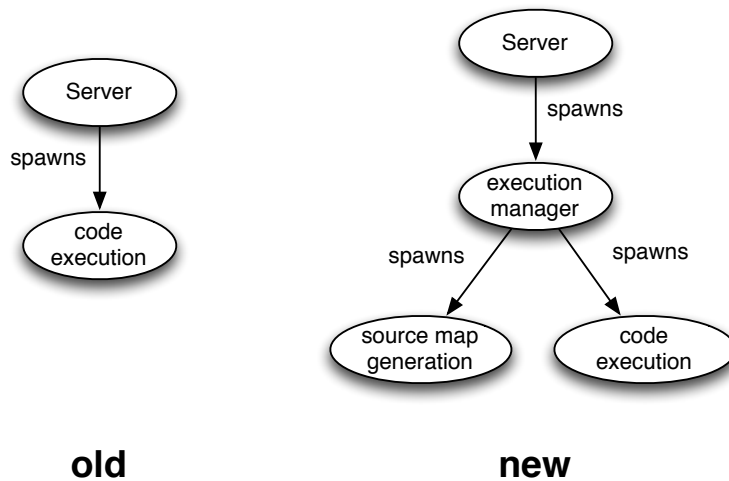
Also, the backend would only catch exceptions in the main execution, not any exceptions occurring in timeout callbacks or otherwise asynchronous code. We used Node.js<sup>3</sup> error domains and supplied our own implementations of `setTimeout`, `setInterval` and `process.nextTick` to catch and report exceptions in all these cases.

**JSON stringification** By default, the backend uses the `JSON.stringify` function to create string representations of all objects [Belzmann, 2013]. In addition, a de-

<sup>2</sup><https://github.com/Constellation/escoregen>

<sup>3</sup>Node.js is a platform to build server-side applications with JavaScript: <http://nodejs.org/>. The complete backend is written in Node.js.





**Figure 4.12:** Our changes to the process architecture of the backend. In the original version the server would spawn a new execution process whenever it received code to execute. It then received data about the execution from the execution process and relayed it to the client. In the new version the server spawns an execution manager process which simply spawns two more processes. One to execute the code and one to calculate a source map from the instrumented code to the original code. When it receives data from the execution process, it checks whether this data has to be corrected for incorrect locations (e.g. in the case of uncaught exceptions) and if so it uses the generated source map to correct the location. If the source map generation is not done yet, it will buffer the messages from the execution process and wait for the source map generation to finish.

cycling library is used which replaces cycles in the object tree by string references to the first occurrence of an object. But we noticed two problems with this stringification. Firstly, `JSON.stringify` ignores values such as `null`, `undefined`, `NaN`, `Infinity`, and function objects which could all be interesting to our Live Coding environments. We therefore extended the stringification to include special cases for these values. Secondly, the decycling of objects can take a really long time. We tested our plugin by trying to write a parser for the [UIST 2012 conference program](http://www.acm.org/uist/uist2012/program.html)<sup>4</sup>. It is a simple 200kB HTML page. But after parsing, the corresponding jQuery object of the document object is really

JSON stringification ignores several important values like `undefined`. We corrected that.

<sup>4</sup><http://www.acm.org/uist/uist2012/program.html>

Stringification of large objects is really slow, so we added special cases for some common large objects, like jQuery-objects.

large. The decycling of this one object alone took 20 *seconds* on a state-of-the-art laptop and the resulting string representation was 7MB long, which added a considerable time just for transferring the data to the client. Thus, we decided to add a special case to the backend for jQuery and DOM objects: In these cases we simply use the html-string as a string representation, which is often orders of magnitude shorter and much easier to calculate.

We made sure a “finished” message is only sent when the program has actually quit.

**Execution management** In the original version of the backend, it would send a “finished” message when the main part of the code was executed, even if a timeout was registered that would only fire minutes later, so the application actually was still running. We made use of the fact that a Node.js process exits by itself when no timeouts or other callbacks are registered anymore. We thus observed the execution process and waited for it to quit. Only when it did, did we send a “finished” message. This way, we were able to reliably show to the user whether the code they wrote was still executing.

## Chapter 5

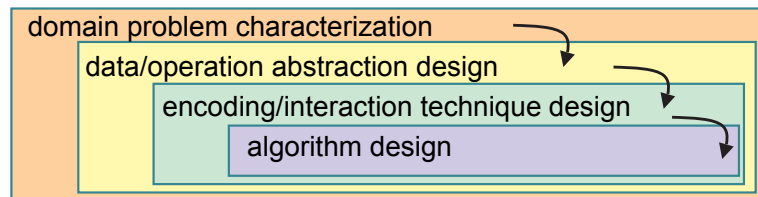
# Study Design

In this chapter we describe the study we designed to evaluate the effects of a Live Coding tool on developers and look at the errors developers make in general, whether using a Live Coding tool or not. We first highlight, why we think a new study is necessary and why we developed a high-fidelity prototype for this study. Next we describe the tasks our participants had to solve and how we designed those tasks. After that, we describe the IDE we used and a custom extension we built to make it more comparable to state-of-the-art IDEs in use today. In closing, our process to advertise our study and find participants is explained, followed by the procedure we used when conducting the study.

### 5.1 Motivation for a New Exploratory Study

Munzner [2009] describes four nested levels for evaluating (software) visualization design (see Figure 5.1). The outermost level is the domain problem characterization. Here, the designer must learn about the target domain, the tasks of the target users and the data they produce. The next level is the level of data and operation abstraction design. On this level a decision has to be made what data needs to be

Munzner [2009] describe 4 levels on which a software visualization tool can and should be evaluated.



**Figure 5.1:** *The four levels of the nested model for designing and evaluating evaluations by Munzner [2009]. Reproduced from Munzner [2009].*

visualized. On the third level, the visual encoding of this data and the interaction of the user with the visualization is designed and evaluated. The last level is the algorithm design level, which is concerned with the algorithm to use for implementing this visualization and evaluating the (computing) performance of the resulting system.

Design decisions on inner levels depend on outer levels.

In our case, we have to determine the data to visualize, since the target domain is clear.

Munzner [2009] explains that the design decisions of the inner levels depend on the outer levels and failures in the outer levels make great designs in the inner levels useless. For example, by choosing the wrong data to visualize (level 2), even if the chosen visualization is great (level 3), it won't matter and the resulting system will still be useless to the target users. In our case, the target domain is mostly clear, our target user group are programmers implementing some program functionality, especially while they are writing source code. The problems they face, and our prototype should solve, are the problems producing correct programs that do what they are designed to do. However, the data that should be shown is not that clear. We assume that it would be helpful to show the runtime state of the program to the programmer, hoping that faults are more easily discovered that way, which should make fixing the underlying software errors easier as well. However, we do not know what runtime data should be shown to the programmers, so determining that should be the next step.

As we saw in Section 3.2, most programmer research looked into program understanding, and if they did look at programmer's errors they either looked at difficult to fix bugs or only novice errors. Even if simpler errors were observed, at most the frequency distribution was described,

but information about how difficult to fix these actually were or what information could have helped discovering the errors, is missing.

Therefore, we think a new study looking at programmers' errors is necessary. This study will try to achieve both an exploratory approach, and a validating approach. On the one hand, we will try to gather data that helps us understand what errors programmers make, how long it takes to fix them and how they fix different errors. On the other hand, we are interested in measurable effects of Live Coding tools, so we will try to enable some quantitative analyses on the data as well, allowing us to compare Live Coding users with a control group.

We will try to integrate both exploratory and validating approaches into the study.

Usually, when developing a new tool, a small low-fidelity prototype would be built, it would be evaluated, refined, reevaluated and so on. The problem, in our case, is twofold: Firstly, it is difficult to build a low-fidelity prototype for a Live Coding tool. Of course, different visualizations can be tested with users, e.g. by paper prototypes. But the essential part, the liveness is hard to reproduce, since human experimenters are too slow to properly replace a computer in that case. For a system to appear live it should have a response time less than 100ms [Nielsen, 1993]. We would lose the feeling of complete liveness, but could likely keep a feeling of direct interaction and mostly-liveness by filling the waiting time with incremental updates and progress indicators if we could keep the total response time on the order of seconds. However, even that is most likely impossible to achieve for a human experimenter tasked with looking at the code a participant produced and producing an appropriate result. It would only be possible with highly constrained tasks, which Munzner [2009] says are inapt for learning about the data the target users are interested in, since real tasks could possibly make use of completely different data.

Liveness is hard to test using paper prototypes.

Secondly, Munzner [2009] argues that the validity of a tool on the data/abstraction level can only be properly evaluated downstream, with a complete tool used by real users on real tasks. Of course, that creates a chicken/egg dilemma, since a poorly designed visualization could foil

A decision on the data abstraction level can only be properly evaluated using a complete system from the lower levels. But designing a good system on the lower levels requires proper data from the upper level.

We tried to design a simple prototype that has all necessary characteristics of a Live Coding environment.

While not able to use true real-world tasks, we tried to find tasks that came very close to being real-world tasks.

the positive effects of a Live Coding tool with the correct data chosen, but we cannot test different visualizations before we do not know what data to visualize and we cannot test what data to visualize before we have a finished tool to test with. Therefore, we decided to extend an already existing Live Coding prototype developed by Heinen [2012], to make it usable for real-world tasks and be able to analyze its usage in real-world-like settings. To avoid designing a complicated visualization, which we could not properly test due to time constraints and thus possibly designing a bad visualization, we tried to design a very simple visualization following a ‘good-enough’ approach. We also decided to show a lot of different data, on a high level of detail, avoiding abstraction and provided the possibility to show more data on request using `console.log` (see Section 4.3.2—“Using `console.log` to Overcome Limitations”). While this possibly shows more data than necessary, we hope that this helps to show us what data is actually needed and used, and does not hide any data that would be needed. Basically, we tried to design a simple Live Coding prototype that has no major flaws and that has enough of the characteristics of a Live Coding environment to enable us to learn what effects on developers Live Coding has.

## 5.2 Designing the Tasks

Munzner [2009] also notes that for determining the necessary data to visualize, target users should be observed working on their own tasks, not on specifically designed ones. We were not able to fulfill this requirement completely, since we wanted to achieve some comparability between participants, but we tried to choose tasks that could have been given to programmers in the real world and give them just a description about what has to be achieved and then let them work on their own to implement the requirements. We hoped this would give enough freedom to the programmers to make the same errors they would make when working on their own tasks, enabling us to see what errors are common, how programmers go about fixing them and which errors are especially difficult to detect.

Since the previous work by Heinen [2012] and Belzmann [2013], on which we built our prototype, implemented Live Coding for JavaScript, we did so as well. Therefore, we also wanted to create tasks that used JavaScript as a programming language, optimally ones that were familiar to JavaScript programmers and representative of their normal programming work. Then again, we did not want to design specific JavaScript tasks, but ones that could also be implemented in other languages without a lot of difficulties to enable them to be reused for evaluating other Live Coding tools designed for other programming languages. Following these design goals, we chose two areas which seemed (a) relevant to JavaScript programmers and (b) general enough to be relevant in other programming languages as well: XML-parsing and date/time conversion. Out of these two areas we created two tasks. We added a third one that was clearly more academic, but which we hoped could single out more programming-specific issues (in contrast to understanding a protocol, a library or simply finding the right function to use): Implementing a basic algorithm. This way we arrived at three different tasks.

Our tasks had to use JavaScript as a programming language.

Other than that, we tried to keep them as generally applicable as possible.

We tested each of the tasks with an experienced programmer that did *not* know JavaScript (we explained it to them before the task) to get an upper bound on how long participants would need to solve the tasks and how difficult it was. During the first test, the Live Coding plugin was used, to rule out any major flaws. After that we refined each of the tasks and tested each of the tasks with two further participants, one using the Live Coding plugin and one without the Live Coding plugin, again to help us estimate the time needed to solve the tasks and further test our study setup. The tasks were slightly refined after these two preliminary tests again. The descriptions of the tasks, which were given to the participants, can be found in Appendix A.

Before using the tasks in the study we tested them on experienced programmers to rule out major flaws and measure the task duration.

### 5.2.1 Task 1.1: Parsing an RSS-Feed using a SAX-parser

Parsing data and handling XML documents is likely a common task, especially in web development, but also common in client-server-based programming. We wanted to have an aspect of asynchronicity and tried to model the task to be as believably a real-world task as possible. Thus, we decided the XML file to be parsed should be downloaded first from an HTTP-webserver.

In the first task, out subjects have to download and parse an RSS-feed.

An application where it is common to download XML files from web servers is RSS, so we decided to make the first task about RSS-parsing. We looked for an RSS-feed that had a sufficient number of different cases that required slightly different handling on the one hand but was still clean and structured XML to not make it unnecessarily difficult for developers on the other hand. A feed that fits this description is the RSS-feed of the weblog [Daring Fireball](http://daringfireball.net)<sup>1</sup>. It only uses 13 different tags, which makes it simple enough to understand it, but also uses CDATA-tags, data in attributes, and tags that have a different meaning depending on what tags they are nested in to make the task interesting enough to not be trivial after one understood how to use the parser.

Participants used an XML parsing library.

We did not want participants to write a complete XML parser themselves, but instead they should use a parsing library. We expect Live Coding to also have benefits working with unknown APIs, because they can simply be tested to understand their behavior possibly sparing a developer from having to read the complete documentation. Therefore, designing a task that required making sense of a library seemed sensible as well.

There are two ways of XML parsing: DOM-parsing and SAX-parsing.

There are basically two big approaches to XML parsing: DOM-Parsing and SAX-parsing. A DOM-parser reads in the complete XML document and constructs the XML-tree from it. After that, queries can be used to access the data in the XML tree, often using specific query languages such as XPath. SAX-parser do not read in the complete document

---

<sup>1</sup><http://daringfireball.net>



to save memory. Instead, they post events whenever they encounter an interesting element (e.g. a tag, or an attribute) and the client application has to listen to these events and decide what information to keep and what to ignore. This is usually done by building a simple state-machine that checks where in the tree the reported data element is and decides whether it is worth keeping.

We were afraid that, had we used a DOM-based parser, participants would simply spend most of their development time constructing the correct query string to extract the data they needed. While Live Coding could have some benefits there as well, this could also be done using other means, like a specific XPath query testing tool. We therefore decided to ask participants to use a SAX-Parser. We selected a popular Node.js library called [sax-js](https://github.com/isaacs/sax-js)<sup>2</sup>, to make sure that all participants were using the same libraries to decrease the variability of results from different participants.

We did not want participants to simply choose an optimal query, so we used a SAX parsing library.

### Refining the Task

In a preliminary version of the tasks we asked participants to also implement the HTTP client to download the XML file from the server. We noticed that participants already had several problems doing so and decided to remove this part from the task to save time during the study. Therefore, the participants receive a partial implementation that downloads the XML file from the web server and already initializes the parser with it. All the participants have to do in the refined version is to register callbacks for the different events of the parser, extract the requested data, and report it to the caller using a given callback function.

After noticing problems with it, we did no longer ask participants to download the XML file themselves, but provided the code to do so.

### Expected Problems

We expect participants to make the following errors and have the following problems during the task:

---

<sup>2</sup><https://github.com/isaacs/sax-js>

We predict several problems we expect participants to encounter.

- Problems with asynchronous callbacks and correct ordering of actions.
- Not initializing or not correctly resetting state-saving variables.
- Incorrect filtering of tags of interest, since some of the XML-tags have different meanings depending on the context. E.g. the `title`-tag is being used for the title of an entry but also to specify the title of the complete feed.
- Finding out what events to listen to when trying to receive a specific bit of data.
- Correctly interpreting and using the parser-API. E.g., correctly understanding which arguments each of the event-callbacks has.

### 5.2.2 Task 1.2: Date & Time Conversion

Date/Time programming is often complicated, partially because of many different time zones.

Most programmers have to deal with dates and times at some point. Not only do many different time zones exist which have a different amount of hours as offset to each other, some of them even have half-hour offsets. In addition, the amount of offset depends on the time of the year, because of Daylight Saving Time in many countries. To make matters worse, the Daylight Saving Time switch does not happen at the same time all over the world but on many different days, again depending on the individual country. Therefore, it is no wonder that programming using dates and times is often very complex and even the Date/Time libraries that are available are often complex to use.

Another complication comes from inconsequential implementations in many frameworks.

In addition, with minutes and seconds usually being zero-indexed but days and month being 1-indexed there is a lot of potential for do-I-start-at-1-or-0-errors and with all the offsets due to time zones there are many sign-errors possible ('should it be +1 or -1?'). The JavaScript Date API makes it especially easy to make errors, since days start at 1, but months start at 0, a fact that most programmers probably would not expect. In addition, when creating a new `Date`-object by specifying a date and a specific time of day these

values are interpreted in the local time zone, meaning the resulting date depends on the system setting for the time zone of the computer the code is executed on. And lastly, the getter methods of `Date`-objects sometimes have confusing names. E.g. `getDay` does not return the day of the month, but the day of the week. To get the day of the month `getDate` has to be used. In short, there is a lot of potential for small and ‘stupid’ errors when using such an API, so we decided this would be an ideal task to see what kind of errors developers make in this context and how they fix them.

Since some developers might already know the JavaScript `Date` API, but we expected several errors to originate from a wrong mental model about how the API should work (e.g. months starting at 0), we decided to provoke these errors by providing our own mental model of how the API should work that developers had to use. To do so, we specified a date object that a naïve but reasonable programmer might construct, but which is considerably different from the JavaScript `Date` object.

- It let months start at 1, as is done in normal use.
- It specified all times in UTC.
- It specified a property called `day` that held the day of the month.
- In addition, it referred to the time properties in singular (`hour`, `minute`, `second`), whereas JavaScript refers to them in plural (`getHours`, `getMinutes`, `getSeconds`).

The task now is to convert this kind of date object into a correct JavaScript `Date` object representing the same date and time.

### Refining the Task

This task was originally part of Task 1.1, we split it up to save time on task 1 and to make it possible for people to

Even some of the getter methods in JavaScript's `Date` object have confusing names.

To ensure a more common starting point for developers we chose a date format that differs in almost any possible way from the JavaScript `Date` object.

This task was originally part of Task 1.1, this is the reason for it being named Task 1.2.

work on the date time conversion even though they were not able to complete Task 1.1. The general idea and motivation for the participants is that they want to extend their project from Task 1.1 by adding filtering by dates. Since our design of Task 1.2 is a continuation of Task 1.1, we developed a complete solution to Task 1.1 up to the point where Task 1.2 would start. Participants receive this example solution to Task 1.1 as a skeleton of code for working on Task 1.2.

### 5.2.3 Expected Problems

We predict several problems we expect participants to encounter.

We expected participants to make the following errors and have the following problems during the task:

- Conversion from input months to `Date` months, since the first starts at 1, the second starts at 0. Both noticing that the conversion is necessary and then applying it correctly ('is it +1 or -1 month?').
- Using the `Date`-constructor with UTC-values, not knowing that it will interpret those in local time.
- Using `getDay` instead of `getDate`.
- Using singular versions of time getter-functions, instead of plural or the other way round.

### 5.2.4 Task 3: Dijkstra's Algorithm

During the third task, participants had to implement Dijkstra's algorithm.

As a third task we decided to ask participants to do a rather pure computer science task: Implement an algorithm. We did not want to use a sort algorithm, since this was already done by Heinen [2012] and seemed too simple. In addition, most programming languages provide built-in sort functionality or at least have libraries to do so; thus, it is rare for a programmer to actually have to implement his own sorting algorithm. Instead we decided to use Dijkstra's [1959] shortest path algorithm as an example.

Dijkstra's algorithm is taught to each student at our university in one of the basic computer science lectures and we expect every experienced programmer to know it or be at least able to understand it in a reasonable amount of time. It is one of the simpler graph-based algorithms, yet we expect it to be complicated enough to provoke errors when implementing it.

### 5.2.5 Refining the Task

The main challenge in designing this task was to come up with a suitable data structure for the graph. We did not want participants to decide for themselves what the data structure should be like, since we were afraid that would take too much time and we expected the difficulty of the task to be strongly dependent on the data structure selected.

The main problem was finding a suitable data structure.

One possibility to represent a graph is to represent each node as an object that has a set of predecessors and possibly a set of successors, with each predecessor/successor also being a node object. This way, each node object knows the whole graph and can reach each other node, which is a strongly object-oriented approach. But since each node is connected to each other node, when our plugin tries to visualize such a node, it will always display the whole graph, just starting at different nodes. Although the plugin will recognize and show the circular references, such a visualization is still likely very cluttered, since our prototype is not designed to handle such situations. For this reason, we decided not to use this representation, but one that encodes the edges between the nodes more indirectly and affords a less cluttered visualization.

An object-oriented approach would lead to display problems with our plugin.

Our first approach was to encode the graph with two lists of objects (see Listing 5.1). One list of nodes, with each node just having a `name` property, and one list of edges, with each edge having a `weight`, a `from`, and a `to` property, with the `from` and `to` properties just being strings that gave the name of the corresponding node. This way, there was no direct connection between nodes or between nodes

```

var graph = {
  nodes: [{name: "a"}, {name: "b"}, {name: "c"},
          {name: "d"}, {name: "e"}],
  edges: [
    {from: "a", to: "b", weight: 1},
    {from: "a", to: "c", weight: 2},
    {from: "c", to: "a", weight: 3},
    {from: "c", to: "b", weight: 9},
    {from: "d", to: "a", weight: 10},
    {from: "d", to: "c", weight: 5},
    {from: "c", to: "e", weight: 2},
    {from: "e", to: "d", weight: 7},
    {from: "e", to: "b", weight: 6},
    {from: "b", to: "e", weight: 4}
  ]
};

```

**Listing 5.1:** Example of the first selected encoding of the graph in the Dijkstra's algorithm task.

Our first approach was to encode all nodes and edges in two independent lists.

On of our prestudy subjects complained that using this data structure was cumbersome, so we changed it.

and edge. To find all the successors of a node, a programmer had to search through the list of edges and look at those edges that had the searched node's name as a `from` property. This is obviously more cumbersome than a direct connection, but also not unrealistically complex, since graphs are often encoded using adjacency matrices which are even more abstract. We also hoped that this additional level of indirection would provoke some more interesting errors.

When testing the task with one of our prestudy subjects, they complained about always having to search through the whole list of edges to find the corresponding successor node and proposed using key-value-maps for the encoding. Since every object on JavaScript is basically a key-value-map of property names and values, which can be arbitrarily added and removed at runtime, this seemed like a sensible suggestion and we decided to change our task appropriately. Therefore, the graph now has a `nodes` object, which has a property for each node, with the name of the node being both the property name and value (see the Task description for Task 3 in Appendix A). The edges are represented by a similar object. Again, it has a property key for each node's name, but the value is another key-value map. This map has a property for each name of a successor of the selected node. The value of the map is the weight of

the edges between the two nodes. This way, the weight of the edges between node "a" and "b" can be accessed via `graph.edges.a.b` or if the nodes are saved in variables `x` and `y` via `graph.edges[x.name][y.name]`.

We expected this change to make the code written easier to understand, by making the connection between nodes and edges a bit more direct, although not as direct as in the object-oriented case. It also avoided the need to search through the list of edges to find the correct one for the current node. None of our additional two prestudy participants had problems with this task, but several of our actual subjects had serious problems understanding the key-value-maps and correctly iterating over them using `for-in`-loops (see Chapter 6). This was something we did not expect to happen.

We now use a map-based approach, which requires less code to write but does not necessarily make the code more understandable.

### 5.2.6 Expected Problems

We expected participants to make the following errors and have the following problems during the task:

- Incorrectly calculating the new distance between nodes.
- Incorrectly sorting the nodes by their temporary distance/selecting the wrong next node, due to incorrect comparisons.
- Incorrectly iterating through all the nodes/edges, e.g. stopping too early or producing infinite loops.

We predict several problems we expect participants to encounter.

## 5.3 Recruiting Participants

We advertised our study in different ways. To recruit more participants, we decided to pay participants €25 for their participation, since we expected them to have to invest around 3 hours on average to take part in the study. This

We offered participants €25 as reimbursement when they took part in our study.

is still a very low price for a skilled programmer, so we did not expect participants to take part ‘just for the money’, but hoped that we could motivate more people to invest this much time.

We created a website to advertise our study and enable subjects to register.

We then created a website with a short introduction to the tool and contact details. The website also included a short list of self-test tasks to help potential participants determine whether they had sufficient JavaScript knowledge to take part in the study. However, passing this self-test was not a requirement to take part, we simply wanted to prevent inexperienced programmers from taking part in the study and then stay there for many hours desperately trying to solve the tasks. We did want the tasks to be somewhat challenging, but we did not want to overtax a participants abilities.

The website was advertised in several CS lectures, in the CS building and via a local developer meeting.

The website’s URL with an accompanying short description of the study was advertised by e-mail to students of our chair’s lectures and another chair’s lectures, which focused on web development. In addition, we distributed flyers advertising the study in the computer science building of the university and presented the study at a local developer’s meeting, to also recruit some professional developers. The participants that actually took part are described in 6.1.

## 5.4 Study Setup

We used a high-level MacBook Pro from 2010 running either Mac OS X 10.8 or Windows 7.

All participants used a MacBook Pro from 2010 with an 2.66GHz Intel Core i7 processor, 4GB of RAM and an SSD. A 22" screen was connected to this MacBook Pro, which was used as the main screen, but participants were free to use the screen of the laptop to gain additional screen space. Participants could choose whether they would like to use Windows 7 or Mac OS X 10.8 as their operating system. We enabled this choice since we expect programmers to be highly accustomed to their respective operating system, its specific shortcuts and keyboard layouts which differ in small but important details. To prevent participants from having to learn shortcuts and other practices in an unfamil-



iar operating system we provided both possibilities, which was not a problem, since the Brackets runs on both Mac OS X and Windows 7. It does not run on Linux, so we could not offer that choice.

We also offered participants to tell us specific editor short-cuts they usually used and we would then try to configure those in Brackets. We provided a Mac hardware keyboard with a US layout and a Microsoft hardware keyboard with a German layout, which participants could choose from. Both could be used with either operating system. We only provided a simple two-button mouse with a scroll-wheel but participants were free to bring their own mouse or keyboard and choose a virtual keyboard layout they liked (e.g. Mac DE, or Win RU).

Participants could bring their own input devices.

### 5.4.1 The Development Environment

The plugin we developed is a plugin for the [Adobe Brackets](#)<sup>3</sup> IDE. It is a web development environment for HTML, CSS and JavaScript, but we only made use of the JavaScript editor. Brackets is still in development and currently released as a beta version. We used Sprint 24, which provides many features expected of today's IDEs, like syntax highlighting, auto completion, jump to definition, auto-closing of braces and project-wide as well as file-based search. One notable exception is the missing syntax checker.

We used the web development environment Adobe Brackets (Sprint 24).

#### Building a Continuous Compilation Plugin for Brackets

Brackets does have a JSLint plugin that checks the syntax of the JavaScript source code. [JSLint](#)<sup>4</sup> is a syntax and style checker developed by Douglas Crockford. It not only finds syntax errors but also undefined references, bad style and other problems in JavaScript source code. But the JSLint plugin of Brackets only runs when the code is saved and only displays a simple list of errors below the document.

The syntax checking done by Brackets is sub-standard and not live.

<sup>3</sup><http://download.brackets.io>

<sup>4</sup><https://github.com/douglascrockford/JSLint>

Although it is possible to click on individual error messages to jump to the appropriate location in the code, the error messages are not displayed next to the errors and the code is not checked live.

But Saff and Ernst [2004] already reported that Continuous Compilation improves programmer performance and Continuous Compilation with error reporting inline is a standard feature in today's IDEs. We feared that we would not be able to distinguish the effect of Continuous Compilation, which is a requirement for Live Coding, from the effect of Live Coding itself, had we compared Brackets with our Live Coding plugin to a plain Brackets installation. Also, we considered Brackets without Continuous Compilation to be an unrealistic setting compared to other IDEs and were afraid that our results might not be generalizable to other IDEs if it did not support Continuous Compilation. Therefore, we decided to develop a Continuous Compilation plugin for Brackets.

Comparing a Live Coding environment to an environment without Continuous Compilation would be unfair and unrealistic.

We therefore developed our own Continuous Compilation plugin for Brackets.

```

3  var a, b;
2 4  a = 2( b = 3)
5
! 6  "foo"();
7
! 8  a = c;
9
! 10 if (a === NaN) { a = b; }
11
! 12 function foo(alreadyDefined) {
13     var alreadyDefined;
14 }
15
! 16 a = {}: 22};

```

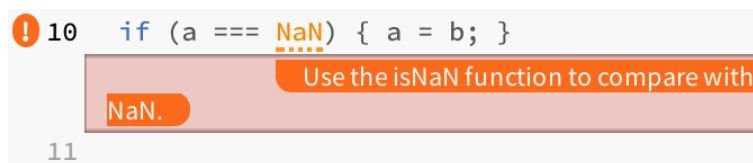
**Figure 5.2:** Our Continuous Compilation plugin shows errors found by JSLint inline in the editor. It displays an error indicator in the line number column, underlines infringing code or displays an insertion marker when something is missing. It distinguishes different levels of seriousness of errors and indicates these by using different colors.

We use JSLint to find the JavaScript errors.

Our plugin makes use of the same JSLint backend already used in Brackets, so it does not report other errors. However, it does improve the location information of the errors. JSLint only provides a start location and even that is sometimes not completely correct. For example, a missing semi-

colon will usually be reported at the location of the next token, which is normally in the next line or even further away from the actual error location. For undefined reference errors, e.g. because of mistyped variable names, we calculate the length of the variable name to then give a location range of the error. These ranges are then used to underline the infringing parts of the code (see Figure 5.2). The plugin also knows that some errors mean that something is missing and will display an insertion marker in these cases instead.

In contrast to JSLint, which reports everything as an error, even style warnings, our plugin distinguishes three levels of errors and ignores style warnings, like how many spaces there should be between an operator and a variable. Yellow errors are simply warnings, things like variables that are declared twice. Orange errors are more serious warnings, for which the syntax checker is pretty sure that something is wrong with the code, but the code will not crash. Dark red errors are serious errors that either prevent the code from compiling, the syntax checker from continuing the checking process or will lead to a crash when the program is run (e.g. an undefined reference). We also classified missing semicolons as such serious errors although they are often optional in JavaScript. We did so, since we expect programmers to usually want to have the semicolons, because not having them sometimes does lead to an error. Also, this way the error reporting is consistent with many compiled languages in which semicolons are mandatory.



**Figure 5.3:** An orange error message. This code will still run and not crash due to the error, but it most likely will not do what the programmer expects it to do. In this case the reason is that `NaN === NaN` is false.

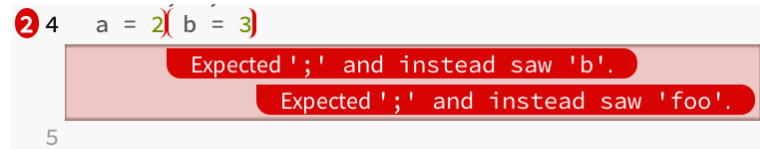
Clicking on the error indicators in the line number gutter will toggle the display of the corresponding error message(s) (see Figure 5.3). The error messages are aligned

We improve the location reporting done by JSLint and use the new locations to highlight the infringing regions in the code.

We filter out style warnings and classify the other errors into warnings, serious warning and errors.

Clicking on the error indicator reveals the error message.

with the error location columns to make it easy to see which error messages belong to which error (see Figure 5.4).



**Figure 5.4:** Two errors in the same line. The offset of the error message is aligned with the error's location to make the message bubble point at the error and make it easy to see which message belongs to which error.

Our Continuous Compilation plugin reports a lot more errors than a simple syntax checker.

Our Continuous Compilation plugin finds a lot more errors and problems with source code than a simple syntax checker. This will make it more difficult for the Live Coding plugin to have an advantage since many errors could theoretically be found by our Continuous Compilation plugin (e.g. mistyped variable names). We see this as an advantage of our study since it should make it easier to identify those kinds of errors Live Coding can help to fix that existing tools like static analyzers are not useful for.

### Providing a Debugger

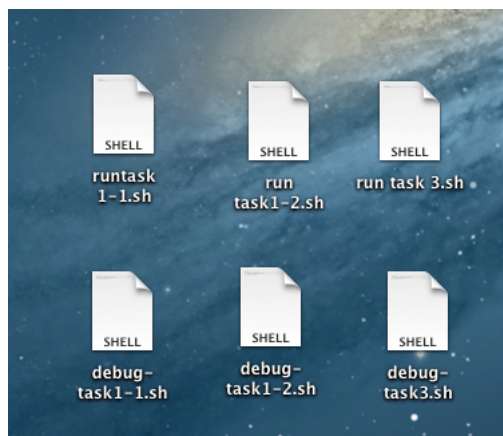
Brackets does not have a built-in debugger.

Thus, we built a workaround to be able to execute a piece of source code with a click of a button.

Brackets does not have a debugger built in and does not have a button to execute and debug the currently displayed code in an external debugger. Node.js code is usually executed using the `node` command line tool. But this meant that participants would have to start a command line interface and execute the `node` tool with the path of the file every time they wanted to run the code, which is quite cumbersome. We therefore wrote a shell-script for each of the tasks that executes the source code of that task and displays the results and any errors. These shell-scripts were put on the desktop, so they could be executed with one click by the participants (see Figure 5.5).

Another important part that is missing from Brackets, which other IDEs usually have, is a graphical debugger. While the `node` command line tool provides command line

debugging functionality, it does not have a graphical interface that allows the user to see the whole code and step through it, set breakpoints in interesting lines and quickly glance at current variables on the stack. However, the `node-inspector`<sup>5</sup> command line tool can attach to the default `node` debugger and provide a debugging interface that can then be used by any graphical debugger adhering to a specific remote debugging protocol. One debugger that implements this protocol is the built-in Chrome debugger, which can then be used to debug Node.js programs using a graphical interface. To prevent participants from having to individually start `node`, `node-inspector`, and the Chrome debugger, we wrote a second set of shell scripts to combine all of these actions. Now participants can start a graphical debugger or execute the code with the click of a button, as it should be for a modern IDE (see Figure 5.5).



**Figure 5.5:** *Since Brackets does not have functionality to execute or debug the displayed code we provided two command line programs for each task. One (top row) to just execute the corresponding code and one (bottom row) to debug the code. The debugging shell script would start the program to debug and attach a debugger to it to then start the Chrome debugger interface which would be attached to the running debugger to provide a graphical user interface for the debugging session.*

Both of these additional tools, the debugging shell scripts as well as the Continuous Compilation tool were available to both groups of participants. So there was no difference

We used `node-inspector` to provide a graphical debugger to our participants in the same way.

Participants can execute shell scripts with the click of a button to debug or execute their code.

<sup>5</sup><https://github.com/node-inspector/node-inspector>

between Live Coding participants and non-live coding participants, except for our Live Coding plugin. Also, participants could completely ignore the plugin and program and debug the old-fashioned way, if they so wished.

### 5.4.2 Monitoring

We recorded the the screen of all participants and video and voice of participants who opted in.

In addition, we saved every change participants made to the source code.

To have multiple ways of analyzing how developers worked on our tasks we recorded their behavior in different ways. Firstly, we recorded the screen using a screen capture software. On Mac OS X we used [Silverback](http://silverbackapp.com/)<sup>6</sup>, on Windows we used [Camtasia Studio](http://www.techsmith.de/camtasia.html)<sup>7</sup>. Both tools were run with a high frame-rate since we wanted to be able to follow the movement of mouse pointers and see typing errors. Also, if participants consented, we recorded the voice and face of participants to be able to interpret their reaction when working on a problem. To make the analysis of the coding behavior itself easier, we also saved a version of the source code whenever the content of the editor changed and recorded the timestamp. This way, we can later look at questions like how many changes did developers make on average, how many versions of the produced source code actually compiled, for how many versions did a particular error persist, etc.

### 5.4.3 Procedure

First, we explained the study process and goal to participants.

When participants arrived we explained to them the aim of the study and asked them to fill out a consent form. We explained to them that we would record the code they wrote and record the screen during the study. We also asked them for permission to record their face and upper body and their voice to make it easier for us to understand what they were doing or why they were having problems, but explained that this was optional and they were free to deny either or both of those requests.

---

<sup>6</sup><http://silverbackapp.com/>

<sup>7</sup><http://www.techsmith.de/camtasia.html>

Next, we gathered some statistical data about them, like their age, their programming experience and how many hours they program per week, on average. We then explained the Brackets IDE to them including the extra tools we provided, like the Continuous Compilation plugin, the debugging scripts, and, if appropriate, the Live Coding plugin and asked them to try out the different tools on a test project. This was also used to check whether the keyboard layout was set up correctly and everything else worked as they expected.

We then explained the tools to the participants.

After this setup was done, participants were given the first task. Since Task 1.2 depends on Task 1.1 it was always given to them after Task 1.1. But other than that, the task order was counterbalanced with half of participants starting with Task 1.1 and Task 1.2 and half of participants starting with Task 3.

There was no time limit for the tasks and participants were informed of that fact and it was explained to them that they could just quit a task without any consequences and continue on the next one, at any point in time. If participants took considerably longer than expected, the experimenter would ask whether they would like to quit, but made it clear that it was entirely their decision and they were free to go on. This was done to make sure that participants used the possibility to give up on a task if they were really stuck and did not continue just because they were afraid to ask for the next task. Also, participants were allowed to take breaks at any point in time, but were asked to do so between the tasks, if possible.

There was not time limit for the tasks, participants were asked to decide for themselves when they wanted to stop.

Participants were asked to work on the task as they would normally work on their own programming tasks. They were allowed to use the internet and even copy snippets of code, if they liked. The experimenter explained to the participants that they were welcome to think aloud and explain what they were doing or what they were thinking about, but they were not required to do so.

Participants were asked to work on the task as they normally would work on a task of their own.

Before each task, participants were asked to fill out a questionnaire to assess their knowledge of topics required in the tasks (see Appendix B.1) and after each task another ques-

After the study and before and after each task participants had to fill out a short questionnaire.

tionnaire was given to them to measure the subjective difficulty of the task and ask for any problems they encountered (see Appendix B.2). After all three tasks were completed an additional questionnaire was handed to the participants which asked for feedback about the tool. This questionnaire included the 10 questions of the SUS by Brooke [1996] and an additional 8 of our questions (see Appendix B.3).



## Chapter 6

# Evaluation

In this chapter we will describe preliminary results from our study. We were not able to look at all the data we gathered, due to time constraints, but will report what we found so far. First, an overview of our participants is given. Next, we look at the study itself and show that it fulfilled its design goals as far as we can tell, but will also highlight some problems. Last, we look at differences between Live Coding participants and participants in the control group.

### 6.1 Participants

We were able to recruit 13 mostly experienced developers between the age of 19 and 51 (median: 26 years) of which only 2 were female. 7 of them used our Live Coding plugin during the study, for all tasks. A short overview of the programming experience of our participants is given in Table 6.1. All participants had more than 4 years of programming experience, except for one, who only had about half a year programming experience, but also worked with JavaScript during that time (median: 9.0 years). 3 participants stated they had no prior JavaScript experience. Out of those, one only heard a lecture including JavaScript, so they just knew the theory; one used it for a short time and then decided to use CoffeeScript instead, a modified version of JavaScript

We were able to recruit 13 participants with a median age of 26 and all except one having more than 4 years of programming experience.

Some of our participants did not have JavaScript experience, but they had many years of programming experience, so we did not expect this to be a problem.

that compiles to JavaScript code; and one simply never used JavaScript before. Only 4 of the participants had used Node.js before, but since there are no important differences to web-based JavaScript development in our tasks, we did not see this as a problem. All of them had sufficient programming experience (5, 13 and 13 years), so we did not expect them to have problems because of a slightly unfamiliar language. Our participants programmed between 3 and 40 hours per week, except for one who said he did not currently spend time programming (median: 10 hours).

	Median	Mean	Standard Deviation
Age	26	28.1	7.9
Programming Exp. (years)	9	11.6	8.8
JavaScript Experience (years)	2	3.5	4.8
Programming per Week (hours)	10	13.6	11.5

**Table 6.1:** *Programming experience and age of the 13 participants of our study.*

Almost a third of our participants were students, but most of them being in senior years or close

All 13 subjects had studied Computer Science, a related subject (such as software engineering), or were doing so at the time. 8 of them were still studying computer science or a related subject for a Master or Bachelor degree, 2 were doing their PhD, one was working as a post-doctoral researcher and 2 were professional software developers (see Table 6.2).

	Count	Percent
Student	8	61.5%
PhD Student	2	15.4%
PostDoc	1	7.7%
Software Developer	2	15.4%

**Table 6.2:** *Occupations of our subjects.*

Half of participants used Mac OS X and half used Windows. Some brought their own input device.

7 of the participants did the study with Mac OS X 10.8, 6 with Window 7. A US keyboard layout was used by 9 participants, 4 used a German one. 1 participant mainly used the US keyboard, but sometimes switched to other, which we counted as simply using the US keyboard lay-

out. All but one participant used the two-button mouse we provided, one brought his own Magic Trackpad and one participant brought his own keyboard.

## 6.2 Evaluating the Study

We asked participants several questions to determine how well the study achieved our design goals. After each task, participants were asked how difficult the task was for them, the results are shown in Figure 6.1. After the session we gave a SUS [Brooke, 1996] questionnaire to participants using the Live Coding tool, to check whether it was good enough to be usable.

### 6.2.1 Tasks

Participants thought that Task 1.1 was not difficult. Although two participants were not able to complete it, only one participant thought it was difficult. But since 5 participants also were neutral, we think the Task 1.1 was not too easy, either.

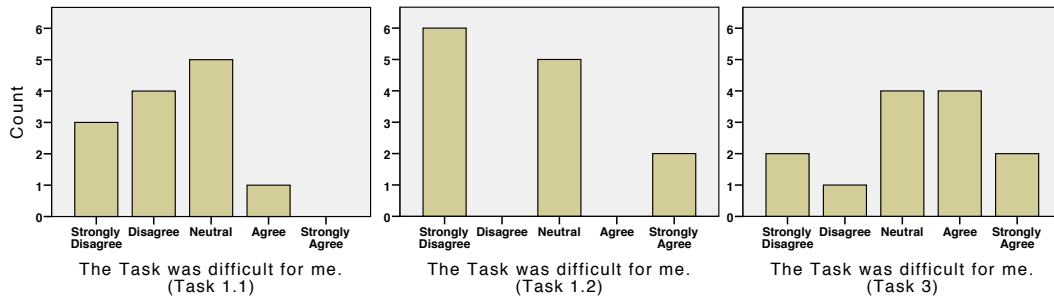
Task 1.1 was not considered difficult, but neither easy.

The results of Task 1.2 are particularly interesting. No participant simply agreed or disagreed when asked whether the task was difficult. They were either neutral, strongly disagreed or strongly agreed. This could indicate that there were a few big problems in the tasks some participants could solve easily and others could not. With almost half of participants strongly disagreeing with the task being difficult, we are confident that it was not too difficult for most.

Task 1.2 had mixed results, but almost half of students found it to be easy. Two found the task very difficult.

Task 3 seems to be the most difficult one, which is also supported by the fact that 4 participants were not able to solve the task. But even more agreed that the task was difficult. Still, 3 participants thought the task was not difficult, and 4 were neutral, which we interpret as the task not being too difficult. While we did not look at the errors participants made in detail, yet, we noticed during the study that

Task 3 was judged the most difficult one.



**Figure 6.1:** Histograms showing how difficult participants thought the tasks were. Task 1.1 seems to be the easiest, Task 1.2 produced mixed results and Task 3 was considered to be the most difficult by participants.

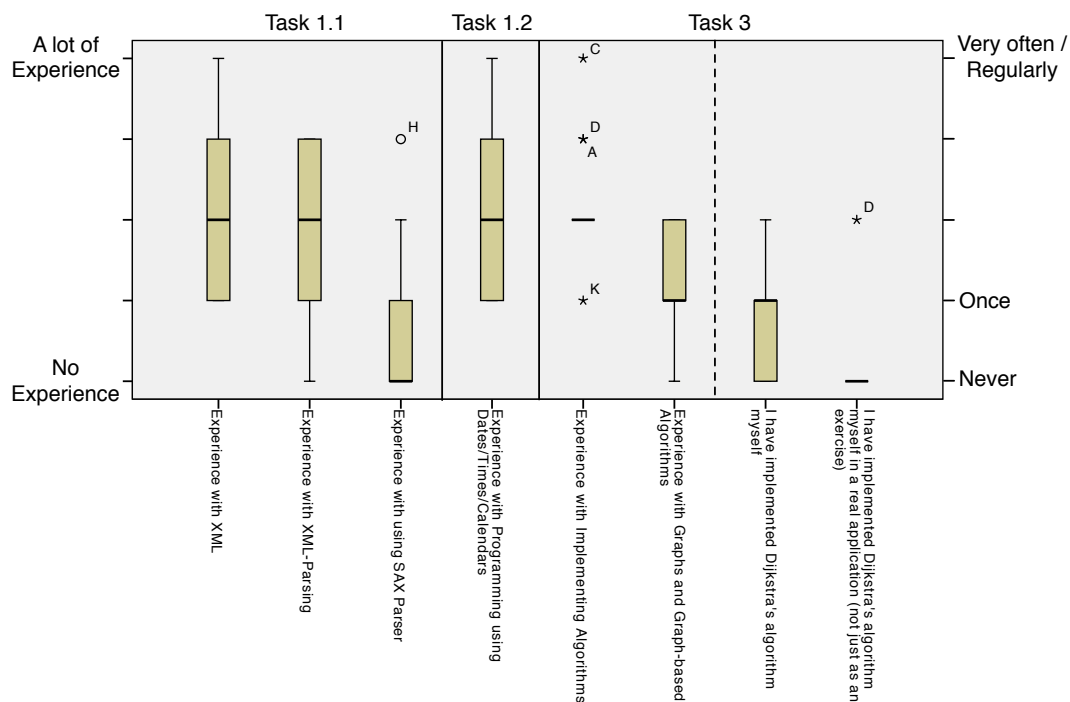
several participants had problems understanding the algorithm correctly and many participants had problems understanding how to iterate over the properties of the graph object correctly.

We also recorded how much previous knowledge participants had of the task domains. The results are shown in Figure 6.2. As we expected most participants had previous knowledge in the domains, e.g. XML-parsing or programming using dates and times. However, while 7 participants had implemented Dijkstra’s algorithm before at least once as an exercise, 6 stated they never implemented it so far. This was a higher percentage than we expected and might have been part of the problem participants had with Task 3. Originally, we expected that participants would just implement an algorithm they had implemented before and would only needed a refreshment how it worked.

## 6.2.2 Tool Quality

We used the System Usability Scale to check whether our plugin had any major flaws.

The System Usability Scale is a “quick and dirty” questionnaire to get a usability rating for any kind of interactive system [Brooke, 1996]. It can be used to compare two similar versions of a system [Bangor et al., 2008] but it is questionable whether it can be used to compare two very different systems. It certainly cannot be used to judge whether a system is good for the problem it was designed to solve, since none of the questions in it address whether it actually

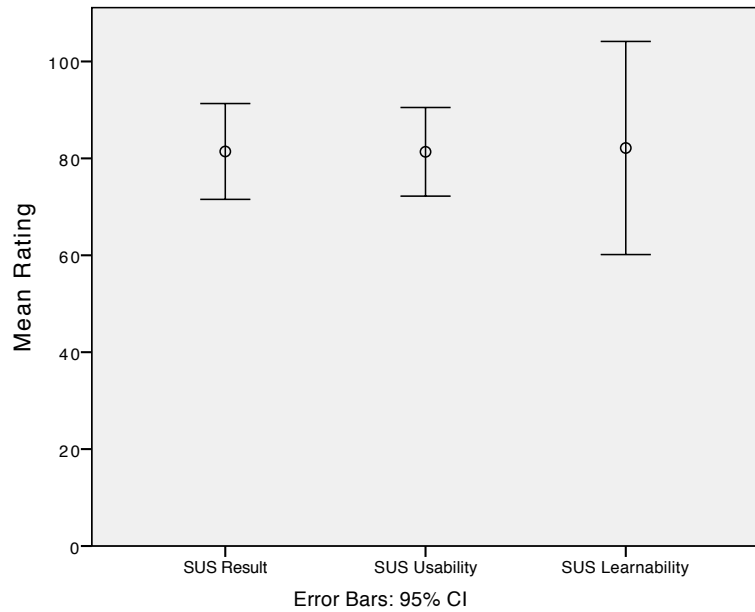


**Figure 6.2:** The previous experience of participants in the task domain. Most participants had experience in using and parsing XML, but only a few had experience with SAX parser. All participants stated they had at least some experience programming with Dates/Times/Calendars. Regarding Task 3, all participants stated they had experience implementing algorithms and most also said they had experience implementing graph-based algorithms. More than half of participants said they had implemented Dijkstra algorithm before, but for most it was just an exercise since only one participant actually implemented Dijkstra's algorithm in a real application.

helped the user solve their problems or supported them in their task. But if a system achieves a sufficiently high SUS score we can rule out any major usability flaws and this is what we use the SUS for in our case.

After participants that used our Live Coding tool were done with all three tasks, we gave another questionnaire to them (see Appendix B.3), containing the 10 questions of the System Usability scale. We explained to them that “the system” meant just the Live Coding tool, not the complete Brackets IDE and they should answer the questions accordingly. The results are shown in Figure 6.3.

The mean SUS rating of our participants for the Live Co-



**Figure 6.3:** Error bars for the SUS Ratings of the 7 participants using our Live Coding plugin. On the left the mean rating for the complete SUS rating is shown. We also show the Usability score and Learnability score postulated by Lewis and Sauro [2009].

Our plugin achieved a SUS score of 81.4 which is good.

ding tool is 81.4 with a minimum rating of 70. Bangor et al. [2008] analyzed the SUS in detail with many different products and participants. They describe acceptable systems as reaching a score above 70 (our lowest rating) with “better products scoring in the high 70s and upper 80s” [Bangor et al., 2008], a range our mean score falls into. We therefore conclude that our implementation of the Live Coding plugin has no major usability flaws. We would like to point out again that this still does not mean that it solves any meaningful problems, but whatever it does, it does in a good and usable way.

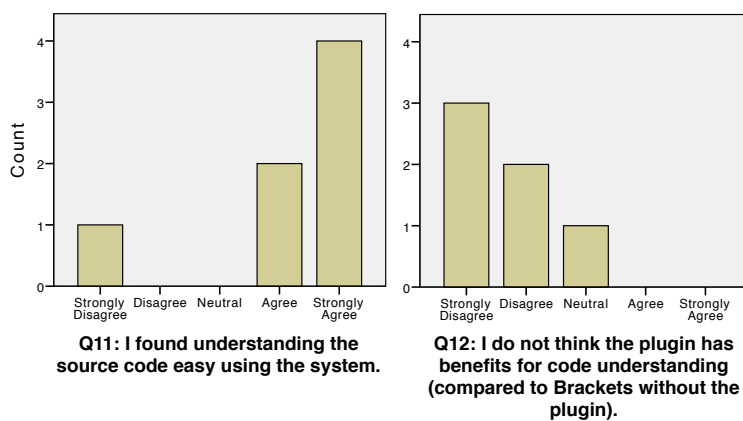
### 6.3 Evaluating the Live Coding Tool

After checking that our study did not have any major flaws, such as an ill-designed tool or tasks that were strictly too easy or too difficult, we will now look at the remaining re-

sults of our study to see whether we can find any advantage of a Live Coding tool. To do so, we will first look at some qualitative results and identify some trends there. After that, we check whether we can find any kind of support for these trends in our quantitative results.

### 6.3.1 Qualitative Results

We included eight questions in the post-session questionnaire designed to find out whether the tool was considered to be helpful by participants. They can be found as questions 11-18 in the post-session questionnaire in Appendix B.3.

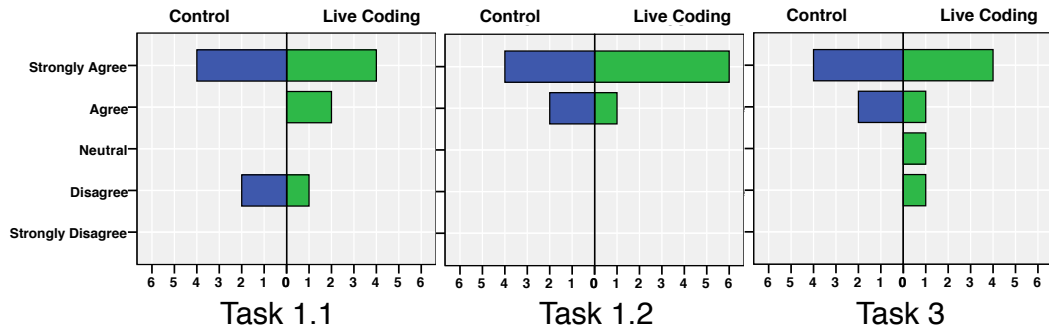


**Figure 6.4:** All participants except one were convinced that the Live Coding plugin improved their understanding of their source code. The one participant, who did not find understanding his source code easy still strongly disagreed with Question 12.

Regarding benefits to understanding what their own code does, all but one participants agreed that they found understanding source code easy when using the system (see Figure 6.4). The one participant that strongly disagreed also strongly disagreed when we suggested that the plugin had no benefits for code understanding. Thus, he thought that the tool improved his understanding of the source code, but it was still difficult to understand. All other participants, except for one in Q12, also disagreed or strongly disagreed

Almost all participants were convinced that the tool helped them understand their code.

**Q4: I understand what the code I wrote does exactly and why it works (or doesn't).**



**Figure 6.5:** Participants answers to Question 4: “I understand what the code I wrote does exactly and why it works (or doesn’t).” The results are quite similar between the two groups.

when we suggested the tool had no benefits for code understanding.

But there are no significant differences in code understanding ratings between participants using the Live Coding tool and our control group.

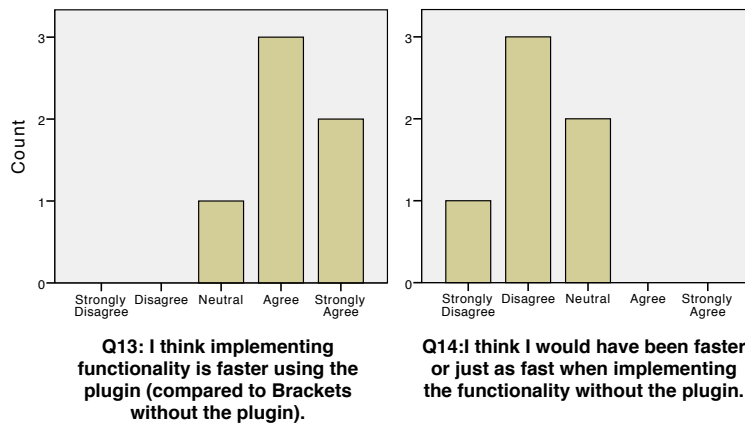
To further understand the relationship of the tool to understanding their own code, we also asked *all* participants whether they thought they understood their code exactly (see Figure 6.5). Overall the answers are quite similar between the conditions. While the Live Coding participants were slightly more confident of their code understanding in Task 1.1 and 1.2 than the participants of the control group, they were actually slightly less confident in Task 3. None of those differences were significant by a Mann-Whitney’s U test.

When asked whether they could complete their tasks faster using the plugin, results are similarly positive (see Figure 6.6). All but one were convinced that the plugin made them faster and not a single one doubted the speed improvement.

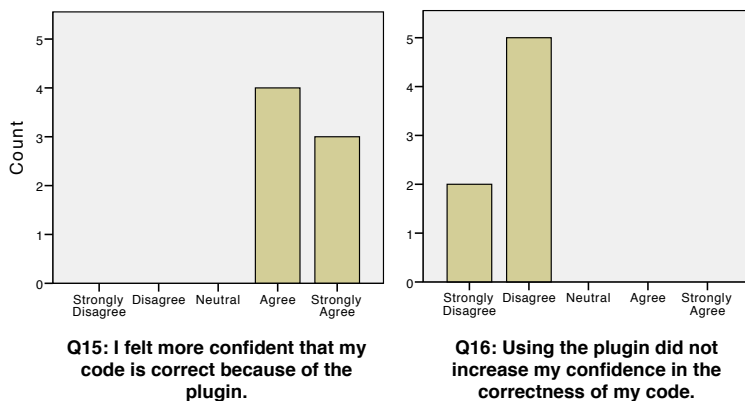
Every single Live Coding participant thought the tool improved their confidence in their code.

Results for improved confidence are even better: Every single participant stated that the plugin increased their confidence in the correctness of their code (see Figure 6.7). Of course, that does not mean, that their code was actually better, it could even be possible that it was worse and they were still more confident that it was good. We also asked all participants after each task how confident they were that





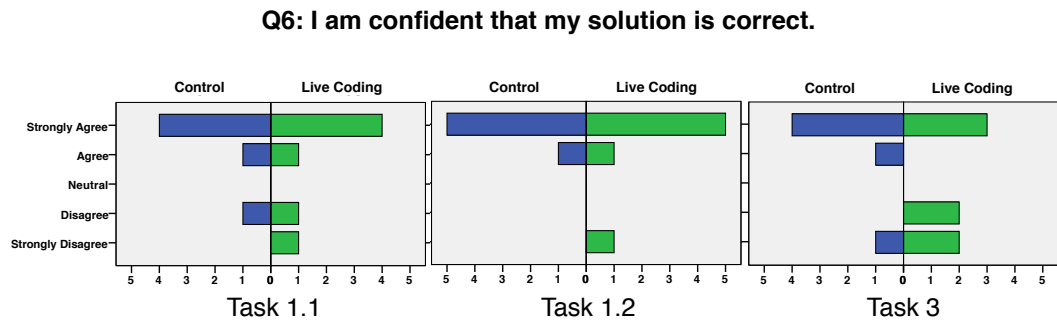
**Figure 6.6:** A majority of the participants was convinced that the Live Coding plugin helped them solve their tasks faster.



**Figure 6.7:** Every single participant said that the tool improved their confidence in their own code.

their solution is correct (see Figure 6.8). The distribution for Live Coding and control group participants for Task 1.1 and Task 1.2 seem to be almost identical.

However, for Task 3, suddenly a lot more participants were not confident that their solution is correct. As described in Section 6.3.2, 4 participants were not able to solve Task 3; 2 with Live Coding and 2 without Live Coding. But 4 of the Live Coding participants were not confident that their solution is correct and only one of the control group participants was not confident. That means, one control



**Figure 6.8:** Most participants were confident that their solution is correct. The differences between Live Coding participants and the control group are minor for Task 1.1 and 1.2. However, in Task 3 several Live Coding participants were not confident that their solution was correct. Still, the difference between the Live Coding group and the control group is not significant.

However, the difference in confidence between Live Coding and control group could not be confirmed by the post-task questionnaires.

If there was a difference, even more Live Coding participants were not confident of their code., contradicting their answers to question 15/16

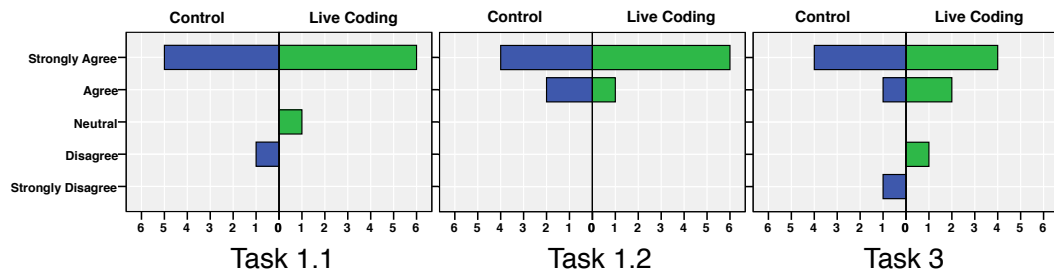
group participant, who did not complete the task, was still confident their solution was correct. This might be due to the fact that they gave up on the task for time reasons but thought their solution up to now was correct. Also, two Live Coding participants were not confident that their solution was correct although they did successfully complete the tasks and made at most minor mistakes.

The fact that more Live Coding participants were not confident of the correctness of their solution than participants in the control group seems to contradict the statement of Live Coding participants that the tool increased their confidence in their code. To assess whether the difference for Task 3 is significant we ran a one-tailed Mann-Whitney's U test, which revealed no significant difference ( $U = 15$ ,  $z = -0.939$ ,  $p = 0.2$ ).

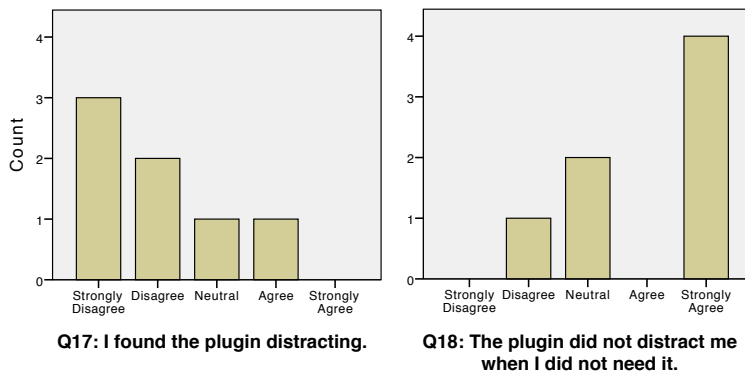
We also looked at a similar, slightly weaker correctness statement: "I was able to implement most of the requirements". The results are shown in Figure 6.9. They are largely similar to the correctness question, but do not have the anomaly in Task 3. Generally the distributions of answers are quite similar for both conditions in all three tasks.

Of course, showing more data to developers could also be distracting, so we asked for that as well. A few participants thought the tool was distracting, but the majority did not find it distracting.

**Q5: I was able to implement most of the requirements.**

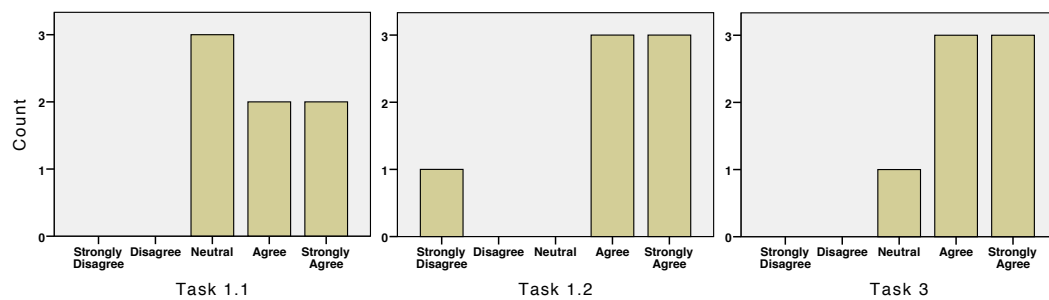


**Figure 6.9:** Most participants thought they were able to implement most of the requirements. The differences between tasks and conditions are only small.

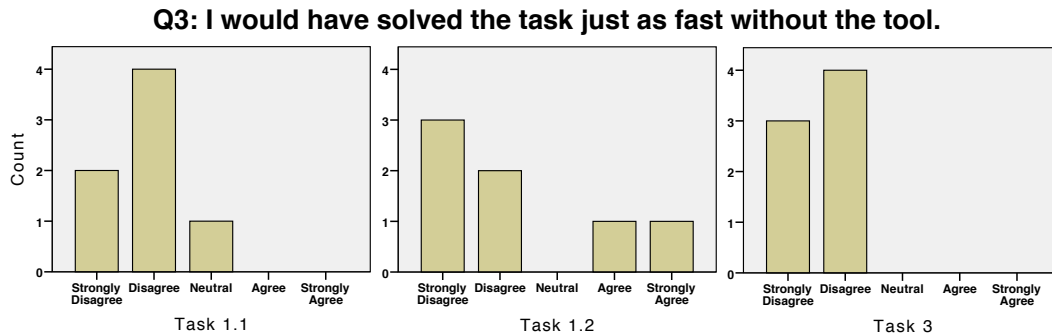


**Figure 6.10:** One participant found the tool distracting and 1-2 participants were not sure, but the majority did not find the tool distracting.

**Q2: The tool helped me solve the task.**



**Figure 6.11:** Only one participant said that for one task the Live Coding plugin did not help him. All others were convinced it helped them solve the task or were unsure. There are only small differences between the tasks.



**Figure 6.12:** Most participants agreed that the Live Coding plugin helped them solve the task faster. Only for Task 1.2, some disagreed.

Most participants found the tool helped them solve the task.

We also wanted to see whether the helpfulness of the tool depended on the task, so we asked some question in that direction after each task. The first question we asked was whether the tool was helpful at all (in whatever way). Almost all participants agreed with that statement, although some were unsure, especially for Task 1.1 and one participant said it did not help him in Task 1.2 (see Figure 6.11). Overall, the differences between the tasks are rather small. Question 3 gives similar results (see Figure 6.12). Two participants said they would have solved Task 1.2 just as fast without the tool, but everybody was convinced they would not have solved Task 3 as fast without the tool. Only one participant is unsure for Task 1.1, everybody else thinks they would not have solved Task 1.1 as fast, had they not had the plugin.

### 6.3.2 Task-Based Quantitative Evaluation

As mentioned in Chapter 5—“Study Design”, we monitored the changes a participant made to the file and of course also the time they took. We will now look at some of this data to see whether we find a difference between participants who used a Live Coding plugin and those who did not.

### Task Correctness

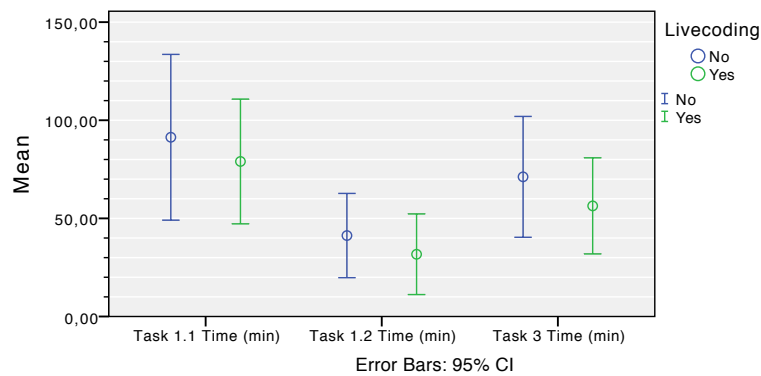
Most participants were able to solve the tasks, but some gave up. 4 participants gave up on Task 3, half of those were participants using our Live Coding plugin. For Task 1.1, two participants gave up, one of them a Live Coding user and for Task 1.2 only one participant of the control group was not able to complete the task. So overall, there were no differences in the ability to solve the tasks between the two groups.

There was almost no difference in task correctness between the two groups.

### Task Completion Times

We measured the task completion time as the time from the participant deciding to start working on the task (after having read and understood the task description, usually when they started to write code) to the time when they decided they were finished or gave up. The mean times and standard deviations can be found in Table 6.3 and Figure 6.13 shows error bar charts for the different tasks and conditions.

Task completion time is measured from the participant having understood the task to them completing the task or giving up.



**Figure 6.13:** Error bar charts for the task completion times of the participants in different tasks and conditions. Participants in the Live Coding condition were slightly faster in each task, but the differences between participants are very big.

Looking at the values and the error bar chart, we directly see that the difference between participants is very large.

	Live Coding	Task 1.1 Time (min)	Task 1.2 Time (min)	Task 3 Time (min)
Mean	No	91.33	41.25	71.17
	Yes	79.00	31.71	56.36
Median	No	75.5	47.0	58.0
	Yes	82.0	25.0	50.0
Standard Deviation	No	40.24	20.46	29.34
	Yes	34.39	22.21	26.48
Min	No	50.0	13.5	41.0
	Yes	23.0	12.0	17.5
Max	No	152.0	63.0	110.0
	Yes	122.0	78.0	103.0

**Table 6.3:** Table showing a summary of the task completion times of our participants. For almost all cases (mean, median, min, max) the times for the Live Coding participants are shorter than those of the control group. However, they are often close, such for Task 1.1 although the mean task completion time is shorter for the Live Coding condition, the median task completion time is actually higher. Also the standard deviations are very high, often almost reaching 50% of the mean with the differences between the means being much less than that.

While Live Coding participants seem to be slightly faster in solving the tasks, this difference is not significant.

While the mean task completion times between the conditions differ by up to 15 minutes or 25% and is always in favor of the Live Coding condition, the standard deviation in all cases is much higher than that, reaching up to 50% of the mean in some cases. We checked whether any of the differences were significant using a mixed-design ANOVA with the task as a repeated measure and Live Coding as a between-groups condition. We found no significant main effect for Live Coding on task completion times ( $F(1, 11) = 0.86$ ), but found a significant main effect of the task ( $F(2, 22) = 16.48, p < 0.001$ ). Not even the interaction between task and Live Coding had a significant effect on task completion times ( $F(2, 22) = 0.048$ ). We also looked at just those participants that were able to complete all tasks successfully, but the results were similar with only the task having a significant main effect.

### Number of Changes

As mentioned before we also recorded all the changes a participant did to the source code. We will now check whether using the Live Coding plugin had a significant effect on how many changes a participant made. The distribution of the number of changes of the participants is described in Table 6.4 with an accompanying error bar chart in Figure 6.14.

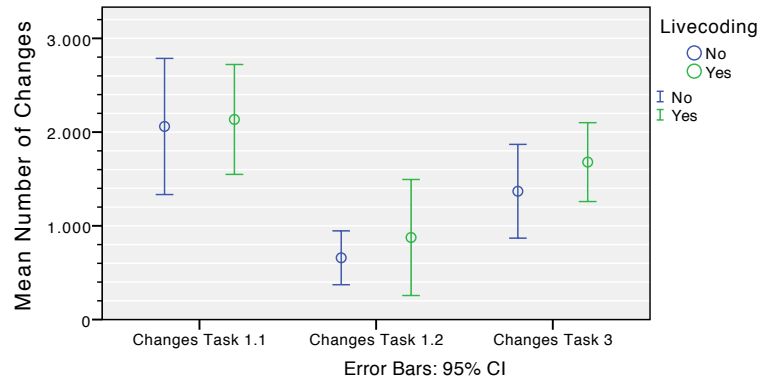
We also recorded how many changes a participant made to the code.

	Live Coding	Task 1.1 Number of Changes	Task 1.2 Number of Changes	Task 3 Number of Changes
Mean	No	2060	659	1369
	Yes	2135	876	1680
Median	No	2030	626	1392
	Yes	2012	718	1757
Standard Deviation	No	692	274	477
	Yes	634	669	455
Min	No	1322	328	741
	Yes	1273	238	850
Max	No	3134	1100	2072
	Yes	3218	2143	2185

**Table 6.4:** Table showing a summary of the number of changes of our participants. The number of changes is mostly similar within one task and does not change much between the conditions. A notable anomaly is the large standard deviation in Task 1.2 of the Live Coding condition. It is caused by a single participant, who did more than 2000 changes while the participant with the second-most changes (of all participants) only did 1185 changes.

Overall the number of changes are rather similar and do not differ much between the conditions, although there seems to be a very small trend to more changes in the Live coding condition. However, the number of changes does differ a lot between the different tasks, like the task completion times did. Testing the differences with an ANOVA again does not reveal a significant main effect of Live Coding ( $F(1, 11) = 0.908$ ), does reveal a significant main effect of task ( $F(2, 22) = 22.982, p < 0.001$ ), and does not reveal a significant interaction effect of Live Coding and task

The difference in the number of changes was not significant either.



**Figure 6.14:** Error bar charts for the number of changes of the participants in different tasks and conditions. The number of changes are mostly similar between the condition in each task, but differ between the tasks. The large confidence interval for Task 1.2 in the Live Coding condition is caused by one participant, who did more than 2000 changes compared to all other participants who did less than 1200 changes.

$(F(2, 22) = 0.182)$ .

The difference in the number of changes per minute was not significant either.

We also looked at how often a participant changed the code, the changes per minute. The results are similar to the observations for the task completion time and the number of changes: Live Coding does not have a significant effect. An important detail is, that the changes per minute do *not* differ significantly in different tasks.

### 6.3.3 Change-Based Quantitative Evaluation

We now look at the changes in more detail.

As we saw in the previous section, we could not find any significant effect of the Live Coding tool on the overall attributes of participants working on the tasks. This might be due to the fact, that there is no difference or because we did not look at the correct part of the data. Optimistically, we suspect the second case. As explained in Chapter 1—“Introduction”, we expect Live Coding tools to have an effect on how long errors stay in the code and how much time developers need to fix errors they introduced while writing code. Thus, it might make sense to look at the individual changes a developer made.



But developers likely differ a lot in their coding style depending on their programming experience, typing skill and familiarity with the development environment. Therefore, we think it does not make sense to look at individual changes to the editor content, because they will depend too much on the developers coding style.

### Clustering Changes into Change Clusters

To normalize these kinds of changes, we decided to try to cluster the changes into change blocks. For example, typing a variable name completely could be a change block or even typing out a complete if-statement including its body could be one change to the code. To classify changes as belonging to the same or different change block, we used the following attributes of the changes: The time the change was made and the location of a change.

We make the following two assumptions: Changes that happen closely after another have a higher probability to belong to the same change block. And changes that happen next to each other (e.g. in the same line) have a higher probability to belong to the same change block. This lead to the following algorithm:

**Clustering Algorithm** Given two changes, the timestamp they happened and the lines that were changed in these changes. Let *timeDiff* be the time difference between two changes. Then sort the changes, by the line number of the first changed line, such that the first changed line of the first change has a lower or equal line number than the first changed line of the second change. Then let the *lineDiff* be the difference between the last changed line of the first change and the first changed line of the second change. If  $lineDiff \leq 0$  the two changes overlap.

The algorithm then works in the following way: Depending on two thresholds  $x$  and  $y$  with  $x < y$ , the following decision is made:

To normalize the changes we cluster changes into change blocks.

We make two basic assumptions as to what changes belong together.

We devise a clustering algorithm that depends on the line distance between to changes and the amount of time between to changes.

$timeDiff < x$ : The two changes belong to the same change block.

$x \leq timeDiff < y$ : Depending on the  $lineDiff$  the following decision is made:

$lineDiff \leq 1$ : The changes belong to the same change block.

$lineDiff > 1$ : The changes belong to different change blocks.

$timeDiff \geq y$ : The changes belong to different change blocks.

There is a lower threshold  $x$ , because there is a lower limit in human cognition limiting how fast a new decision can be made.

There is an upper threshold  $y$ , because there is an upper limit how long a person can concentrate on one single thing.

The reasoning for the algorithm is the following: In terms of Norman's [1988] Seven Stages of Action, we suspect that there is a threshold  $x$  for the time difference below which we can be sure that participants made the two changes in the same *action sequence* because the time difference is too small to form a new *goal* and a new *action sequence*. And there is a threshold  $y$  above which we can be reasonably sure that the changes do not belong to the same change block because the participant most likely started working on a new (sub)problem or restarted working on the same problem. If the time difference between the changes falls between  $x$  and  $y$  we cannot be sure what the mental model of the participant about these changes is. We therefore look at the location of the changes. If they happen in the same line, we assume that the participant is just continuing his previous work, e.g. finishing typing a variable name or the head of a for-loop. Similarly we make this assumption if the changes go over several lines and overlap ( $lineDifference \leq 0$ ). Of course, it might be that the participant finished one line and continues on the next, therefore we also include adjacent changes in lines ( $lineDifference = 1$ ). If the difference in the location of the changed lines is greater than 1 we assume that the two changes belong to semantically unrelated code and thus to two different change blocks.

**Determining the Thresholds** For threshold  $x$  we decided to choose  $x = 1s$ . This has two reasons: First, according to

We choose  $x = 1s$ , based on Nielsen's [1993] response time heuristic.

Nielsen [1993], 1 second is “about the limit for the user’s flow of thought to stay uninterrupted”, which is exactly what we are looking for. Also, our Live Coding plugins delay to display results is about 1 second, so even the Live Coding users could not get feedback on their changes faster than 1s and thus decided to make another change.

For threshold  $y$  we thought about simply choosing  $y = 10s$ , because according to Nielsen [1993], this is “about the limit for keeping the user’s attention focused” and if users have to wait for the computer to respond longer than 10s they “will want to perform other tasks while waiting for the computer”. Of course, developers do not have to wait for the computer in our case, so it is not completely applicable, but from this heuristic, we assume that if developers do not interact with the computer for more than 10s, it is because they are working on a different (sub)task, even if this task is ‘being stuck and trying to find out what to do next’. But in this case we wanted to double-check our guess, so we ran our clustering threshold with  $x = 1s$  and  $y$  ranging between 1s and 60s in one second steps and looked at a range of attributes of the resulting clusters, including the cluster length, the length of the gap without changes between two clusters and the number of changes per cluster. We assumed that all these attributes would asymptotically approach a threshold and wanted to choose a threshold such that the difference between this threshold and our results is not too big.

The results of these calculations can be found in Table 6.5. Of course a change block or cluster can have more than one change or consist just of a single change. Since a large percentage of clusters (more than 19%, even for  $y = 60s$ ) consists of only one change we filtered these clusters out in our analysis of the cluster attributes so they would not hide the changes in the attributes, due to the large percentage of the total. We looked at the following attributes:

**Mean Number of Changes** : The average of the number of changes in each cluster that had more than one change.

**Mean Duration** : The number of seconds between the first

For  $y = 10s$  we partially base our decision on Nielsen [1993] response time heuristic.

But we also look at our data and try choose it based on the data.

We filter out the single-change clusters.

y	Mean Number of Changes	Mean Duration	Mean Time Since Last Change	Number of Clusters > 1	Number of Gaps	Percentage of 1-Change Clusters	Perc. Diff. Mean Number of Changes	Perc. Diff. Mean Duration	Perc. Diff. Mean Time Since Last Change	Perc. Diff. Clusters > 1 Number Change	Perc. Diff. Change of Number of Gaps
1	5,95	1,36	11,62	7247	11585	37,65 %					
2	8,58	2,95	17,21	5267	7474	29,89 %	30,70 %	53,83 %	32,46 %	27,32 %	35,49 %
3	10,25	4,28	20,79	4468	6008	26,11 %	16,30 %	30,95 %	17,23 %	15,17 %	19,61 %
4	11,52	5,44	23,33	4001	5241	24,22 %	11,01 %	21,34 %	10,86 %	10,45 %	12,77 %
5	12,71	6,66	25,58	3639	4676	22,82 %	9,36 %	18,38 %	8,82 %	9,05 %	10,78 %
6	13,64	7,71	27,24	3398	4311	21,89 %	6,79 %	13,60 %	6,08 %	6,62 %	7,81 %
7	14,55	8,78	28,67	3191	4032	21,62 %	6,23 %	12,16 %	4,99 %	6,09 %	6,47 %
8	15,21	9,63	29,71	3055	3843	21,30 %	4,37 %	8,88 %	3,50 %	4,26 %	4,69 %
9	15,77	10,39	30,56	2948	3690	20,94 %	3,54 %	7,32 %	2,80 %	3,50 %	3,98 %
10	16,26	11,11	31,27	2861	3570	20,73 %	3,02 %	6,43 %	2,26 %	2,95 %	3,25 %
11	16,80	11,89	31,88	2768	3456	20,80 %	3,24 %	6,58 %	1,90 %	3,25 %	3,19 %
12	17,18	12,52	32,40	2709	3369	20,51 %	2,19 %	5,01 %	1,63 %	2,13 %	2,52 %
13	17,54	13,11	32,85	2655	3295	20,37 %	2,02 %	4,53 %	1,35 %	1,99 %	2,20 %
14	17,87	13,67	33,20	2606	3235	20,40 %	1,87 %	4,08 %	1,08 %	1,85 %	1,82 %
15	18,21	14,30	33,59	2558	3169	20,26 %	1,88 %	4,41 %	1,16 %	1,84 %	2,04 %
...	...	...	...	...	...	...	...	...	...	...	...
60	22,20	25,57	34,68	2103	2566	19,27 %	0,14 %	0,58 %	-0,11 %	0,14 %	0,16 %

**Table 6.5:** A number of attributes of the calculated change clusters for different threshold  $y$  in the range of 1s–60s. The values between 16s and 59s are left out for brevity, but the last row shows the values reached at  $y = 60$ s. Displayed are the means of following attributes of change cluster (column index in parentheses) that contain more than one change: Number of changes in the cluster (2), Duration of the cluster (3), length of gap with no changes (4). In addition, we display the number of cluster with more than one change and the number of gaps without changes between all cluster (this is roughly equal to the number of all clusters, including the ones with only one change). These are then used to calculate derived values, such as the percentage of single-change clusters of all change cluster or the percentage the value changed from the previous threshold calculation. Up to a certain point these values are monotonically decreasing and we highlighted (in yellow) in each column the first threshold where they increase, because we think that indicates a point where the change in these values is only minor from then on, so it would be a good threshold.

change of a cluster and the last change of a cluster for clusters with more than one change.

**Mean Gap Duration** : A ‘gap’ is a time frame in which no change happened. We calculate it as the number of seconds between the last change of the previous cluster and the first change of the current cluster. In this case, it does not matter how many changes the cluster has.

**Number of Clusters** : This gives the number of clusters with more than one change, not the total number of change clusters.

**Number of Gaps** : This is the number of the aforementioned gaps.

**Percentage of Single-Change Clusters** : The percentage of clusters with just one change of the number of all Clusters.

For each of these values we then calculated the percentage change between each two consecutive threshold values, because we were interested in the threshold from which on these values only changed marginally. All of these percentage change values are quite large in the beginning, meaning the attributes change a lot for the first few thresholds, so it would likely make quite a difference. At threshold  $y = 6$ , all of them are below 8%, except for the change in the mean duration, which only falls below 10% at  $y = 8$ . Since all of these values fluctuate a bit, so do the percentage changes, so it is not a nice logarithmic decrease, but there is some variability to it. Since we were looking for the threshold at which the change in these attributes becomes marginal, but we did not know what 'marginal' was, we decided to define a marginal change as one that is relatively smaller as one that happens for a higher threshold. One example for that would be difference in the mean number of changes between threshold 9 and 10 (shown in Table 6.5 in row 10): The change is 3.02% of the larger value (the value of  $y = 10$ ). But the change for  $y = 11$  is 3.24%, so it is actually a larger relative change, than the one of  $y = 10$ , which makes the change of  $y = 10$  a marginal change.

For each of the attributes we now determined the first marginal change. For 3 out of 5 values we looked at, the first marginal change happens at  $y = 10$ s. Also, the percentage of 1-Change Cluster actually increases for the first time after  $y = 10$ s. The remaining two first marginal changes happen at  $y = 14$ s. We therefore think that  $y = 10$ s is a reasonable choice for the threshold, above which two changes will not be considered as belonging to the same change clusters. We will use this from now on in the following analysis.

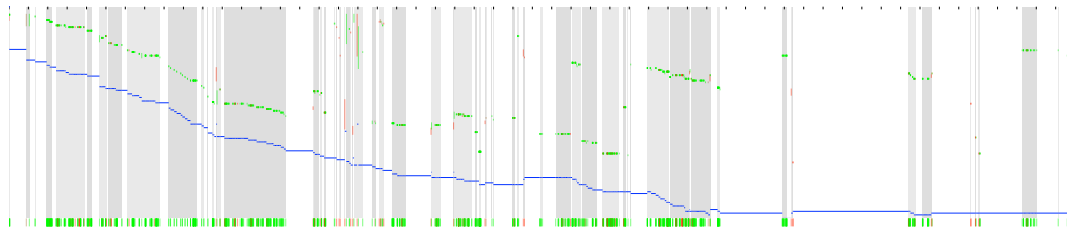
We wrote a small application that processes the changes and draws a graph from the changes, including the clusters. An example is given in Figure 6.15. We produced such a chart for each participant and each task to be able to look at the data and generate hypothesis about how developers

We calculate the percentage change of different attributes between two threshold levels.

A marginal change is one that is smaller than a succeeding one.

For 3 out of 5 attributes the first marginal change is at  $y = 10$ s, so we choose it.

We have created change-graphs including the clustering.



**Figure 6.15:** An example of a graph showing the changes of a participant working on Task 3. The *y*-axis position gives the line of the change (multiline changes are represented by longer lines), on the *x*-axis is the time. The distance between the ticks is always 60s. Green lines indicate additions, red lines indicate deletions. The clusters are shown by the gray boxes in the background. The blue lines indicate the current length of the file (in lines) so it can be used to judge the relative position of a change and shows how the source code grows over time. In the bottom row there is a second view showing the changes simply grouped together, ignoring the line that was changed. This is useful to find busy times and changeless times

change the code and in what way they could have been affected by our Live Coding tool. All the graphs can be found in C—“Change Cluster Graphs”.

### Analyzing the Change Clusters

We repeated the analysis we did for individual changes, now for change clusters, but did not find a significant difference.

We repeated the analysis we did for the number of changes for the attributes of the change clusters: The number of changes in each cluster, the duration of each cluster and the duration of the changeless gaps. We calculated the mean of these values for each task-participant combination and then checked whether we found any differences between the conditions using a mixed-design ANOVA as before. No significant effects, except for the influence of the task again, were found, so we will not go into detail here. Instead, we decided to use another statistical test that is more suited to the problem at hand.

Naïvely, one might assume that we could do an ANOVA on the change clusters directly, since we know for each change cluster whether it was done in a Live Coding condition or in the control group and from which task it originated. We would then compare the distributions of e.g. the durations of the change clusters with a mixed-design ANOVA and use task as repeated-measure and Live Co-

ding as a between-groups factor as before. However, doing that would not be correct and violate one of the assumptions of ANOVA: The independence of the observations [Field, 2009]. To use ANOVA we have to assume that the cases are independent, but this is not the case since some of the change clusters originate from the same participant. Change Clusters from the same participant will likely vary less than change clusters from different participants, which violates this assumption. So, our problem is, that our cases have a hierarchical structure: Change clusters belong to specific participants, and we have several different participants. A kind of statistical model that can handle this kind of data is a Multilevel Linear Model [Field, 2009].

We cannot use an ANOVA to compare the change clusters to each other, because they are not independent.

**Multilevel Linear Models** The explanation in the following few paragraphs is based on Field [2009] and we refer the interested reader to him for a more detailed explanation. Multilevel Linear Models are designed to handle a hierarchical data structure. As almost all other statistical tests they fit a linear model of the form

$$Y_i = a + b \cdot X_i + \epsilon_i$$

to the data, with  $Y$  being the dependent variable (the outcome),  $X_i$  being the value of the independent variable  $X$  for case  $i$ ,  $\epsilon$  being an error term and  $a$  and  $b$  being the actual coefficients that vary between different models.

Instead, we have to use Multilevel Linear Models

After finding the best-fitting model by varying the coefficients, this new model is compared to a basic model, usually the overall mean. If our new-found model can explain significantly more of the variance between the cases (i.e., it fits much better), we assume that this model reflects the reality better and since it usually predicts that the outcome varies with our independent variable, we assume that the independent variable has a significant effect. Of course, more generally, for  $n$  independent variables, the linear model looks like this:

We try to find a good model and compare it to our basic model.

$$Y_{i,j} = b_0 + \sum_{k=1}^n b_k \cdot X_{k,i} + \epsilon_i$$

In this case, we say that the outcome  $Y$  can be predicted

Usually, we only have fixed coefficients.

by the formula above and all we need to know is the coefficients  $b_k$ . We only determine them once, and they hold for all different cases, so we could say they are *fixed*. But because our clusters originate from different participants, we could assume that the coefficients actually depend on the participant. Or more generally, the coefficients vary with a higher-level variable, a level 2 variable (compared to the cases, which are a level 1 variable). To reflect this in our model, we introduce a variable part of the coefficients, called  $u_{k,j}$ . It varies with the different levels of the level 2 variable reflected by  $j$ . Our formula then looks like this:

$$Y_{i,j} = (b_0 + u_{0,j}) + \sum_{k=1}^n (b_k + u_{k,j}) \cdot X_{k,i,j} + \epsilon_{i,j}$$

Now, we introduce random coefficients.

Field [2009] calls these varying parts of the coefficients *random coefficients*. Similarly,  $u_{0,j}$  is called a *random intercept* and  $u_{k,j}$ ,  $k \geq 1$  is called a *random slope*. Now, let us assume that we guessed that the duration of a change cluster depends on whether a Live Coding plugin was used, what task it originated from, and additionally, that the intercept as well as the slope of the task coefficient were both dependent on the participant, but not the slope of the Live Coding coefficient. We could then propose the following model and test how well it fits the model:

$$\begin{aligned} Duration_{i,j} = & (b_0 + u_{0,j}) \\ & + b_1 \cdot live_{i,j} \\ & + (b_2 + u_{2,j}) \cdot task_{i,j} \\ & + \epsilon_{i,j} \end{aligned}$$

with  $j = participant(i) \in participants$  and  $i \in clusters$ .

Instead of just coming up with some complex model, we should start with a simple one and build our model up gradually.

However, according to Field [2009], a better way is to start with a simple model (e.g.  $Duration(i, j) = b_0 + b_2 \cdot task_{i,j}$ ) and assess the fit of this model and then add a coefficient to get a more complex model and assess its fit. If the more complex model fits the data significantly better we use it and add more coefficients, slowly building up a model. If it did not fit the data better, we continue to use the simpler model and try adding other coefficients to it until we end up with a model we cannot improve by adding more coefficients. We then assess how much of the variance it explains



compared to the basic model. Thusly, we determine which of the coefficient's predictions is actually significant.

To assess the fit of the model, Field [2009] recommends using the -2 log likelihood value. So, to compare two models, we calculate the difference in the -2 log likelihood. To determine whether the difference is significant, we also calculate the difference in the degrees of freedoms of the model and compare the difference in the -2 log likelihood with the corresponding 5%-significance values of the corresponding  $\chi^2$ -statistic for the difference in the degrees of freedom (see Table 6.6).

$df$	$p = 0.05$
1	3.84
2	5.99
3	7.81

**Table 6.6:** Critical values of the  $\chi^2$ -distribution at the 5%-significance level. Taken from Field [2009].

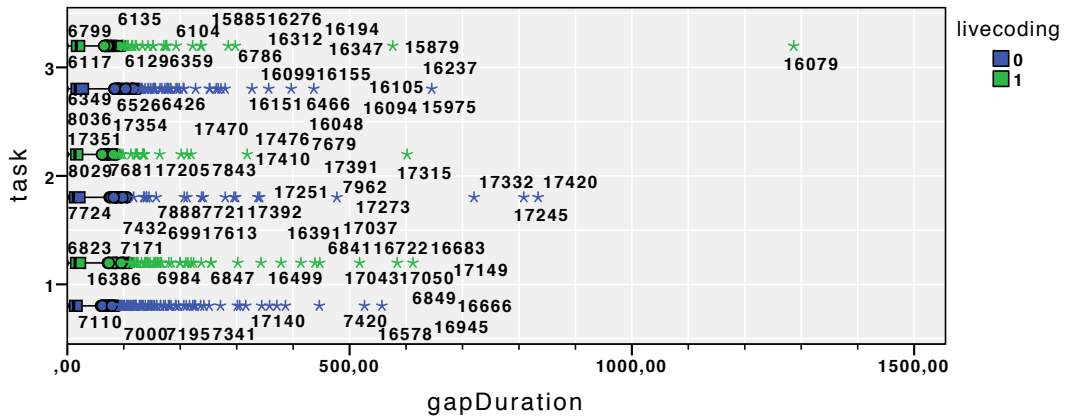
**Filtering the Data** Before we can do a multilevel linear analysis we need to clean up the data a bit. We removed several clusters from the analysis due to considering them outliers or not interesting for our analysis. Firstly, as explained above, for the number of changes per cluster and the duration of clusters is only considered for clusters with more than one change. The clusters with only one change obviously always have one change and duration 0, so they are not interesting.

However, we look at all the gaps between the clusters, when looking at the gap duration, not just those in before a multi-change cluster. But we consider any gap duration that is longer than 10 minutes (600s) to be an outlier, since this is likely due to the participant being severely stuck on the task or spending a long time researching something, which are activities the tool will likely not influence. This is still a very conservative estimate as can be seen by the boxplots in Figure 6.16.

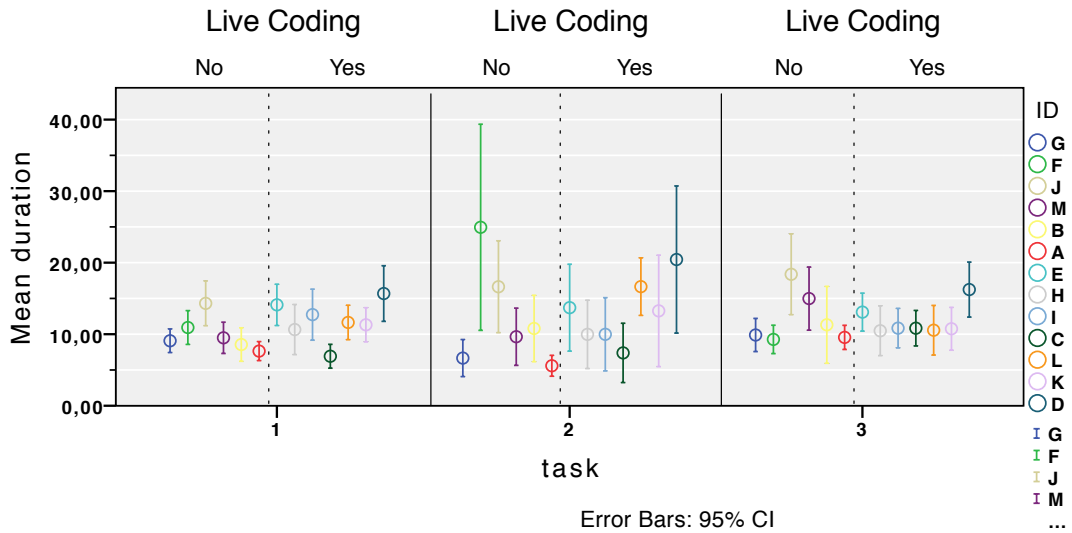
We use the -2LL to assess the fit of our model.

We ignore 1-change clusters.

We ignore gaps that are longer than 300ms.



**Figure 6.16:** Box plots for the gap duration between change clusters, by Live and Task. Using SPSS’ standard outlier classification for boxplots there are a lot of outliers. We only consider anything above 5 minutes / 300 seconds to be an outlier.



**Figure 6.17:** Error bar charts for the duration of clusters grouped by tasks and whether the participant used Live Coding or not. The letters represent individual participants.

**Change Cluster Duration Multilevel Linear Model** We will now build up a multilevel linear model as explained above. The summary of the build-up process can be found in Table 6.7. Figure 6.17 shows the error bar charts of the gap duration for each of the participants for each of the tasks.

We begin with a simple model that has just fixed effects.

Model Index	Model	$-2LL$	Diff $-2LL$	df	compare to model index	significant change
1	fixed live, fixed task, fixed task*live	23 069.024	-	7	-	-
2	fixed live, fixed task, fixed task*live, r. intercept (participant)	23 018.435	15.546	8	1	<b>yes</b>
3	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope live	23 018.435	0	9	2	no
4	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task	23 018.117	0.318	9	2	no
5	<b>fixed live, fixed task, fixed task*live, r. intercept (participant), fixed whichhalf</b>	23 003.154	15.281	9	2	<b>yes</b>
6	fixed live, fixed task, fixed task*live, r. intercept (participant), fixed whichhalf, fixed whichhalf*live	23 003.138	0.016	10	5	no
7	fixed live, fixed task, fixed task*live, r. intercept (participant), fixed whichhalf, fixed whichhalf*task	22 997.508	5.646	11	5	no
8	fixed live, fixed task, fixed task*live, r. intercept (participant), fixed whichhalf, random slope whichhalf	23 003.154	0	10	5	no

**Table 6.7:** Development of the Multilevel Linear Model for the duration of change clusters.

We begin with just a fixed effect for Live Coding, task and the interaction of Live Coding and task.

Adding the random intercept makes the model significantly better.

A random slope for Live Coding does not make the model better.

Whichhalf is the information whether the cluster originated in the first or the second part of the task duration.

Adding a fixed effect for whichhalf improves the model significantly.

The effects we consider are the Live Coding Tool (live), the task participants worked on (task), and the interaction of the two (live\*task). This model has 7 degrees of freedom and the  $-2LL$  was calculated as 23 069.024. Now we add a random intercept that is dependent on the participant. The new model has 8 degrees of freedom and a  $-2LL$  of 23 018.435, so the difference between the two models is 15.546. Since the difference in the degrees of freedom is just 1, our critical value is 3.84 (see Table 6.6). The difference between the models is much higher than that, so it is a significant change.

Next, we try to add a random slope for the Live Coding effect. The  $-2LL$  for this model is no different from the previous one, so the model did not improve and we remove the random slope again. Next we try a random slope for the task. This only improves the model a little bit ( $-2LL$  difference 0.318), which is not a significant difference, so we remove the effect again.

Clusters also have another attribute that was not mentioned before: We know when the changes occurred. Looking at the change cluster diagrams (e.g. Figure 6.15), we notice that there seem to be less clusters near to the end. So we suspect that the cluster's attributes might actually depend on when in the task the cluster was created. This also makes sense intuitively: First the developer writes the code they think they need and produces some long change clusters. Later in the task he might just change minor bits in the code, producing much shorter change clusters. To check this hypothesis, we saved for each cluster whether it was started in the first or the second half of the task (we call this factor *whichhalf*).

Thus, we now added the whichhalf factor to the model as a fixed effect. This improves the model a lot, the change in the  $-2LL$  is 15.281, the change in the degrees of freedom is just 1, so this is a significant change.

We then tried to add some interaction with whichhalf. First we added the whichhalf\*livecoding interaction as fixed effect. This only changed the model very slightly, so we discarded the interaction again. When looking at the inter-

action of whichhalf\*task, the difference in the  $-2LL$  was 5.646. But since the difference in the degree of freedoms was 2, the difference would have to be larger than 5.99 (see Table 6.6). This is not the case, so the difference is not significant. Lastly we checked whether adding a random slope for the whichhalf factor made a difference and conclude that it does not since it does not improve the model.

So the resulting model is the one with fixed effects for Live Coding, the task, the interaction of Live Coding and the task, the whichhalf factor and a random intercept depending on the participant. We then evaluated each of these effects individually to see whether they are significant.

Only the whichhalf effect is significant ( $F(1, 2853) = 15.33, p < 0.01$ ). Neither task ( $F(2, 2859) = 1.38, p = 0.252$ ), nor Live Coding ( $F(1, 13.21) = 1.58, p = 0.231$ ) nor the interaction of task and Live Coding ( $F(2, 2858) = 4.65, p = 0.1$ ) was significant.

**Multilevel Linear Model for Number of Changes per Cluster** Next, we will look at the number of changes in each cluster. We will not describe the model build-up in detail this time, a summary is given in Table 6.8 and Table 6.9. Also, an overview of the distribution of the number of changes for the different participants and different tasks can be obtained from the error bar charts in Figure 6.18.

By following the same process as before we build up our model step by step. The model we end up with is shown in Table 6.9. It has fixed effects for Live Coding, task, the interaction of Live Coding and task, whichhalf, the interaction of the task and whichhalf, and finally a fixed effect for the interaction of Live Coding, the task and whichhalf. In addition to that, there is a random intercept dependent on participant and a random slope for task dependent on participant. When looking at which of these effects are significant, we find the following: Live Coding did not have a significant effect on the number of changes ( $F(1, 12.80) = 1.915, p = 0.19$ ), neither did task ( $F(14.91) = 0.15, p = 0.86$ ) or the interaction of task and Live Coding ( $F(1, 14.91) = 0.482, p = 0.63$ ). But whichhalf

We found a final model and can analyze it now.

Only whichhalf is significant.

We now look at the model for Number of Changes per Cluster.

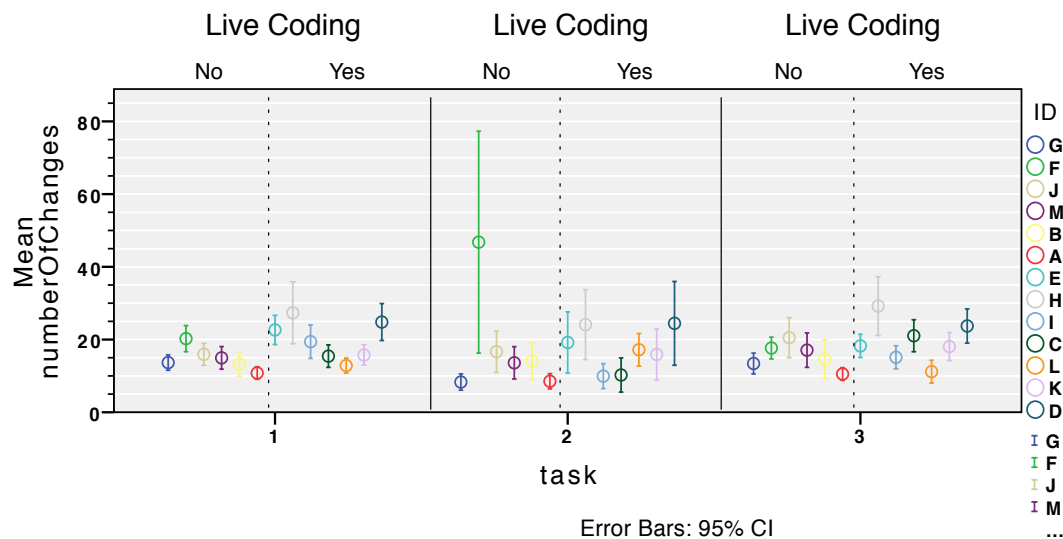
Our final model has the following effects: fixed live, fixed task, fixed task\*live, r. intercept (participant), random slope task, fixed whichhalf, fixed whichhalf\*task, fixed whichhalf\*task\*live

Model Index	Model	$-2LL$	Diff $-2LL$	df	compare to model index	significant change
1	fixed live, fixed task, fixed task*live	24 654.126	-	7	-	-
2	fixed live, fixed task, fixed task*live, r. intercept (participant)	24 560.632	93.494	8	1	<b>yes</b>
3	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope live	24 560.632	0	9	2	no
4	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task	24 555.205	5.427	9	2	<b>yes</b>
5	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task, fixed whichhalf	24 533.804	21.401	10	4	<b>yes</b>
6	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task, fixed whichhalf, fixed whichhalf*live	24 532.240	1.564	11	5	no
7	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task, fixed whichhalf, fixed whichhalf*task	24 523.993	8.247	12	5	<b>yes</b>

**Table 6.8:** Development of the Multilevel Linear Model for the number of changes per change clusters. Continued in Table 6.9.

Model Index	Model	$-2LL$	Diff $-2LL$	df	compare to model index	significant change
8	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task, fixed whichhalf, fixed whichhalf*task, fixed whichhalf*task*live	24 503.386	20.607	15	7	yes
9	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task, fixed whichhalf, fixed whichhalf*task, fixed whichhalf*task*live, random slope whichhalf	24 503.386	0	16	8	no

**Table 6.9:** Development of the Multilevel Linear Model for the number of changes per change clusters.



**Figure 6.18:** Error bar charts for the number of changes in clusters grouped by tasks and whether the participant used Live Coding or not.

whichhalf,  
whichhalf\*task and  
whichhalf\*task\*live  
had a significant  
effect.

For clusters in the  
first half of the task,  
Live Coding  
significantly predicts  
the number of  
changes.

We repeat the  
analysis for the gap  
duration.

did have a significant effect on the number of changes per cluster ( $F(1, 2841) = 23.955, p < 0.001$ ), as did the interaction of task and whichhalf ( $F(2, 2841) = 3.964, p = 0.019$ ) and the interaction of whichhalf, task and Live Coding ( $F(1, 2840) = 6.898, p < 0.001$ ). To break down the interaction, we ran a similar model again on the data for the two halves of the task duration independently. The modified model had the same main effects and interaction terms as the original model but with the main effects for whichhalf removed and the interaction effects without the whichhalf effect. This leaves us with the following 3 fixed effects: live, task, live\*task. For clusters in the first half of the task duration, we found that Live Coding significantly predicted the number of changes ( $F(1, 12.305) = 6.03, p = 0.03$ ), as did the interaction of Live Coding and task ( $F(2, 8.92) = 4.40, p = 0.047$ ). However, the task did not predict the number of changes significantly ( $F(2, 8.92) = 1.03, p = 0.395$ ). In the case of clusters from the second half of the task duration, neither Live Coding ( $F(1, 12.18) = 0.099, p = 0.76$ ), nor the task ( $F(2, 16.83) = 0.735, p = 0.49$ ) nor the interaction of the two ( $F(2, 16.83) = 3.058, p < 0.74$ ) had a significant effect, although the interaction is just barely non-significant. To further understand the interaction effect of Live Coding and task on the number of changes in clusters in the first half of the task duration, this interaction was broken down by the task and an appropriately modified model was run on each of the tasks. For clusters in the first half of Task 1 we do not find a significant effect on the number of changes per cluster ( $F(1, 13.28) = 2.88, p = 0.11$ ). Neither do we find one for clusters of the first half of Task 2 ( $F(1, 9.55) = 3.07, p = 0.11$ ) or Task 3 ( $F(1, 11.437) = 3.00, p = 0.110$ ).

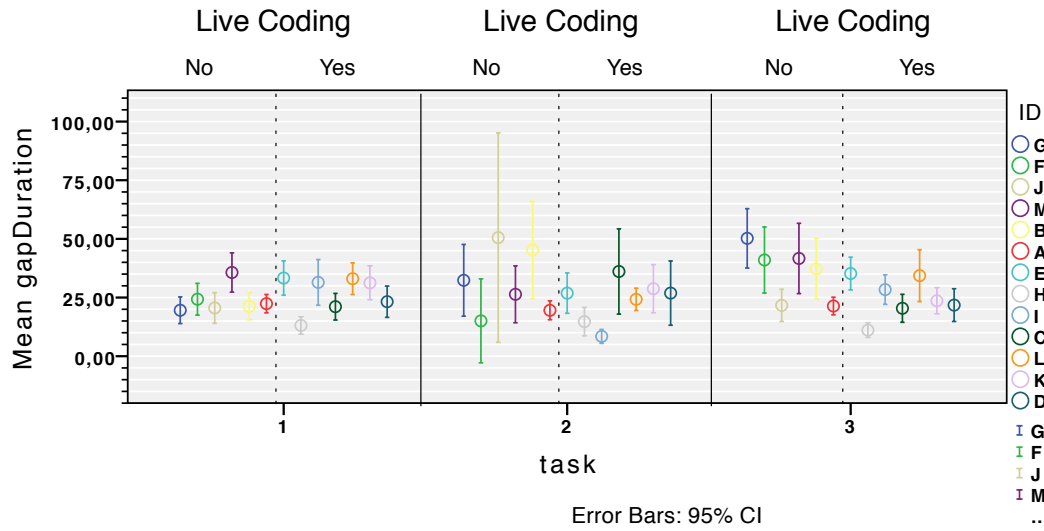
**Gap Duration Multilevel Linear Model** We repeat the previous analysis for the duration of the gaps between clusters. An error bar chart for the gap duration can be found in Figure 6.19, the model build-up is described in Table 6.10 and Table 6.11.

The following effects of our models are not significant: The Live Coding effect ( $F(1, 14.78) = 3.837, p = 0.069$ ) and the task ( $F(2, 21.62) = 2.36, p = 0.119$ ). However,



Model Index	Model	$-2LL$	Diff $-2LL$	df	compare to model index	significant change
1	fixed live, fixed task, fixed task*live	35 985.739	-	7	-	-
2	fixed live, fixed task, fixed task*live, r. intercept (participant)	35 948.197	37.56	8	1	<b>yes</b>
3	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope live	35 948.197	0	9	2	no
4	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task	35 930.953	17.244	9	2	<b>yes</b>
5	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task, fixed whichhalf	35 901.826	29.127	10	4	<b>yes</b>
6	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task, fixed whichhalf, fixed whichhalf*live	35 900.601	1.225	11	5	no
7	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task, fixed whichhalf, fixed whichhalf*task	35 878.264	23.562	12	5	<b>yes</b>

**Table 6.10:** Development of the Multilevel Linear Model for the durations of the changeless gaps between clusters. Continued in Table 6.11.



**Figure 6.19:** Error bar charts for the gap duration of clusters grouped by tasks and whether the participant used Live Coding or not.

Our final model has the following effects: fixed live, fixed task, fixed task\*live, r. intercept (participant), random slope task, fixed whichhalf, fixed whichhalf\*task, fixed whichhalf\*task\*live, random slope whichhalf

For clusters from the second half the task-Live-Coding interaction significantly predicts the gap duration.

the following effects are significant: The Live Coding and task interaction ( $F(2, 21.62) = 3.71, p = 0.041$ ), whichhalf ( $F(1, 16.204) = 24.775, p < 0.001$ ), the interaction of whichhalf and task ( $F(2, 3441) = 12.43, p < 0.001$ ) and the interaction of Live Coding task and whichhalf ( $F(3, 71.91) = 4.59, p = 0.005$ ). As before, to further analyze the interaction, we break it down by conducting analyses on the clusters from the first and second half of the task duration independently. To do so we use a similar model that has been modified to accommodate the missing whichhalf effect. For the cluster from the first half of the task duration, none of the three effects is significant, neither Live Coding ( $F(1, 12.52) = 1.45, p = 0.251$ ) nor the task ( $F(2, 20.46) = 1.35, p = 0.0.282$ ) nor the interaction of the two ( $F(2, 20.46) = 0.296, p = 0.0.747$ ). When looking at clusters originating from the second half, however, we do find that the task ( $F(2, 15.17) = 4.43, p = 0.0.31$ ) and the task Live Coding interaction ( $F(2, 15.17) = 4.52, p = 0.29$ ) significantly predict the gap duration. But Live Coding alone did not have a significant effect ( $F(1, 15.372) = 3.95, p = 0.065$ ).

To analyze the interaction between task and Live Coding further we ran the analysis again for the cluster of the sec-

Model Index	Model	$-2LL$	Diff $-2LL$	df	compare to model index	significant change
8	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task, fixed whichhalf, fixed whichhalf*task, fixed whichhalf*task*live	35 865.544	12.72	15	7	yes
9	<b>fixed live,</b> <b>fixed task,</b> <b>fixed task*live,</b> <b>r. intercept (participant),</b> <b>random slope task,</b> <b>fixed whichhalf,</b> <b>fixed whichhalf*task,</b> <b>fixed whichhalf*task*live,</b> <b>random slope whichhalf</b>	35 859.696	5.848	16	8	yes
10	fixed live, fixed task, fixed task*live, r. intercept (participant), random slope task, fixed whichhalf, fixed whichhalf*task, fixed whichhalf*task*live, random slope whichhalf, random slope whichhalf*task	35 858.781	0.915	17	9	no

**Table 6.11:** Development of the Multilevel Linear Model for the durations of the changeless gaps between clusters.

ond half of the task duration, this time for each task, modifying our model as needed. For none of the tasks did the Live Coding condition significantly predict the gap duration for gaps originating in the second half of the task duration (Task 1:  $F(1, 13.00) = 3.93, p = 0.069$ ; Task 2:  $F(1, 6.815) = 2.70, p = 0.145$ ; Task 3:  $F(1, 12.84) = 3.93, p = 0.069$ ).

## 6.4 Discussion

As we saw in the quantitative evaluation, participants using the tool were quite convinced that it helped them. Despite this subjective impression, we were not able to find convincing arguments why or how it helped them. Although participants said they felt more confident using the tool, participants not using the tool were just as confident when asked whether their solution was correct. Neither in the time to solve a task nor in the number of changes per task could we show significant differences between participants using the Live Coding plugin and those who did not.

The difference between the tasks were quite large.

We suspect that one reason is the huge differences between participants. For example both the slowest and the fastest participant for Task 1.2 used the Live Coding tool but the slowest needed 6.5 times as much time as the fastest participant. For the other two tasks, this factor is also larger than 5.5, so there are really big differences and we suspect that any differences between the two tools were swamped by the changes between participants.

Some participants acted quite inexperienced.

Another problem is that although all but one participant claimed they had several years of programming experience, several clearly had difficulties with basic programming problems. For example, one participant would spend a lot of time searching for code snippets online and then using them without even roughly understanding what they did. This led to one case where they were looking for a way to test whether an element is contained in an array. But because of a bad search term they found code to test whether an element *is* an array (using the `instanceof` operator) and used that without questioning its correctness. As mentioned before, 4 participants were not able to complete the task in which they had to implement Dijkstra's algorithm, a fairly simple graph-based algorithm that is probably part of at least one basic programming lecture. Judging from our observations during the study, we suspect that the failure to complete the task successfully was due to a combination of inexperience with JavaScript and problems understanding the algorithm. An experienced JavaScript developer would likely have had fewer problems with the syntax

and could have concentrated on understanding the algorithm. And had the algorithm been crystal-clear to the participants, they probably could have figured out JavaScript idiosyncrasies. These problems might have been prevented by a stricter preselection of participants. While we did post a self-assessment test on the website, and heard from some potential participants that they took it and decided they did not know enough JavaScript to take part in the study, we did not enforce any entry requirements. It might make sense to do a pretest with potential developers to filter out inexperienced developers resulting in a more homogenous group. Of course, this would also limit the generalizability of the results.

Another reason could be that developers need to get accustomed to such a tool to really use it. For example, one of our participants did not look at the tool once during the first 30 minutes working on their first task, but then had a possibility to use the tool, remembered it, used it successfully and from then on used it a lot. It may well be, that to actually change a developer's behavior, it takes more than a few hours and effects of the tool would only be seen in a long-term study or at least with experienced tool users compared to non-tool-users.

It might be, that the tool has to be learned and is only effective after a longer period of use.

When looking at the differences on the level of change clusters we had some more success in identifying differences, but they were still hard to find. For the number of changes we found a significant effect of Live Coding, but only for clusters in the first half of the changes and depending on the task. Interestingly, for the gap duration, we only found a significant effect of Live Coding for clusters in the second half of the task, again, depending on the task. Of course, this first-half second-half notion is rather arbitrary, we could have just as well split the task into thirds, or calculate a correlation between the start time of a cluster and its attributes. However, we wanted to be able to do an easy posthoc test and were just interested to show that there is some difference. Future research should look at this relationship in more detail, to determine the exact relationship of the time of a change cluster, its attributes and the influence of Live Coding on those.

```

30 function obscureFunction(x, y, z) {
31   console.log(x);
32   console.log(y);
33   console.log(z);
34 }

```



**Figure 6.20:** A situation that we often encountered: A participant use `console.log` to print out the values of the arguments to the function, although they are displayed just above.

## 6.5 Improvements to the Prototype

Several participants have made comments about the prototype while using it, especially mentioning features they would like to see. Other problems were not specifically mentioned by the participants, but observed by the experimenter. We will give a short summary of those comments and observations now.

Apparently, the arguments of function display is difficult to see or understand.

As mentioned before, users could use `console.log` to print any values, they would like to see, to the preview right of the `console.log` statement itself. Several users made use of that functionality. One kind of value that was printed using this functionality particularly often was arguments to functions. This is particularly remarkable, since this is a functionality that was already provided by the plugin. When a function is called the arguments it is called with are displayed next to the function header (see Figure 6.20), so the output of the `console.log` would be just below the values that they were interested in.

This indicates that participants did not see our preview of the function values for whatever reason and there is a problem with the kind of display we chose. In some cases, we even pointed out that the arguments were displayed at the function head, because we thought the participant did not get the explanation in the beginning containing this information. But even after that, some forgot again and used `console.log` instead.

Another indicator of the UI that participants sometimes missed was the exception indicator. If an exception occurred inside of an iteration, the associated iteration selector's border would turn red and become thicker (see Fig-

ure 4.10). Still, several participants missed that. Thus, they missed an uncaught exception that was probably telling them about an important error.

One issue some participants encountered, especially in Task 1.1, where they had to react to a lot of events from the SAX parser in different ways, was the following: They were inside a function that was called several hundred times and handling one special case. To handle this special case they used an if-statement. But since most of the calls to the function did not enter this if-statement, the code in the function was usually not shown to be executed (since only one iteration or call is shown at any one time). So, they did not have the Live Coding feedback for this part of their code, and missed it. Participants could scrub through the iterations to find one that executed the specified line, but they often felt this was too cumbersome.

3 participants of the 7 Live Coding ones thus requested a feature to jump to an iteration in which a currently selected line is executed. This might have also revealed problems with some of the corresponding if-statements since some of them were written in a way that their body was never executed and the tool might have told them that there is no iteration, in which this code is executed. A similar feature request is, to be able to filter iterations to only show iterations matching a specific criterion.

Participants would like to be able to jump to an iteration in which a specific line is executed.

Another issue some participants encountered, especially in task 1 with a lot of asynchronous calls, was that they wanted to know in what order specific functions were called. This can be useful to, e.g., check their assumption that function *a* is called before function *b*. This is not possible with the current implementation of our prototype, but is just a visualization problem. For example, the visualization in Figure 4.2 does provide this information and internally each displayed value has an execution index providing a total order of all the values. So it is not a technical problem to get this information, but a visualization problem of how to show this information without cluttering the display. And it being a visualization problem is exactly the reason why we did not solve it: We did not want to concentrate on the visualization itself.

Participants would like to know in which order functions were called.





## Chapter 7

# Summary and Future Work

In this chapter we will briefly summarize our results and contributions. After that we will highlight different directions of future work, including improving our Live Coding prototype, doing studies on developer behavior and evaluating the data gathered during the study we did further.

### 7.1 Summary and Contributions

Our contribution in this thesis is threefold: We developed a Live Coding prototype, we designed and used an interesting set of tasks for developers to work on that might be reused in other studies and provide some feedback on these tasks, and we evaluated our Live Coding prototype using these tasks.

In Chapter 4—“Prototype”, we described how we combined the prototype developed by Heinen [2012] with the Live Coding backend developed by Belzmann [2013] and fixed several usability issues and implemented some missing functionality that prevented it from being used for productive work. Although there are still some limitations, we made sure it supported all common JavaScript language

We have built a robust prototype.

Our prototype achieved an SUS rating of 81.4.

constructs appropriately, was robust against infinite loops and any kind of exceptions, reported uncaught exceptions and other errors appropriately, and could also handle asynchronous code and event callbacks without any problems. The prototype achieved a SUS rating of 81.4, which can be interpreted as “good” [Bangor et al., 2008]. Also, none of our participants had major problems using it, so we are quite confident to have built a great prototype and a reasonable overall product.

We conducted a study to help close a gap in the HCI and software engineering research body.

Since we realized that the HCI and software engineering research body lacks research about minor and medium programming errors that happen to experienced programmers while writing code and we expect Live Coding to be particularly helpful in preventing and correcting these kinds of error, we decided to conduct our own study, looking at errors programmers make during implementing a piece of functionality. We combined this with our interest in Live Coding tools and looked at how Live Coding tools change the developer’s behavior and whether we could find evidence to support the notion that they help programmers writing code in any way.

We could only find a few statistically significant difference between the control group and the Live Coding group.

While our subjects were quite fond of the Live Coding tool and were convinced that the tool improved their confidence in the correctness of their code, we could not find evidence for that. We also could not find evidence for the Live Coding tool improving the task correctness or shortening the task completion time or changing the number of changes a participant did per task. This was likely due to the large differences between individual participants and the fact that we only had a small sample size, but of course it could also be that the Live Coding tool did not have an effect on these variables.

We then looked at individual changes and, to compensate individual coding styles, clustered them into change groups and analyzed these change groups. We found that the number of changes in a change group significantly depends on the interaction of the task and whether a Live Coding plugin was used, only for change clusters originating in the first half of the task duration. We think the fact that Live Coding has a different effect depending on the time

frame in the task could mean that it depends on the programming phase the developer is in. For example it might be more helpful when writing code than when fixing minor errors in previously written code or the other way around. For the duration of the breaks between change groups, only the task and Live Coding interaction effect for clusters from the second half of a task had a significant effect.

As a side effect of our research, we developed three tasks, that are mostly independent from the development environment and programming language. One task is about building an RSS-parser, one about date/time conversion and the last one about implementing Dijkstra's [1959] algorithm. They fared well in the tasks and participants did not find them too easy or too hard. All but one participant could solve Task 1.2, all but two participants could solve Task 1.1 and all but 4 participants could solve Task 3.

We propose three tasks to be used in similar studies.

## 7.2 Future Work

Although our tool is already the second iteration of a Live Coding prototype and our study is certainly not the first on programmer errors there is still a lot of potential for future work. We will highlight some interesting directions in this section.

### 7.2.1 Evaluating the Gathered Data

We already looked at some of the data collected during the study, but there is a lot more potential in the gathered data. It was simply beyond the scope of this thesis to look at all aspect of the gathered data.

**Look at Time to Error and Time to Fix** One assumption we have about Live Coding tools is that they reduce the time to error by making many errors more obvious because the resulting faults are directly visible. Another assumption is that a shorter time to error also leads to a shorter time

---

Try to pin down the relationship between time to error and time to fix.	to fix, as was already suspected by Saff and Ernst [2003]. However, they were not able to conclusively show that this relationship exists and is strong. Both assumptions could likely be tested with our data. To do so the screen recordings of the participants working on the task have to be annotated to record the errors that occurred and for each error annotate the time to error and time to fix. We recorded 50 hours of video, 40 hours of those are spent working on the tasks. We estimate that it would take around one to two months to annotate the videos properly. To prevent subjective ratings, optimally such an annotation would be done by two researchers independently and they would compare the annotations afterwards.
Classify the errors that occurred.	<b>Classifying Errors</b> Our study is, to our knowledge, the first one to look at experienced programmers all implementing the same small to medium sized bit of functionality with the possibility to record every single error they made. The list of errors programmers made during these tasks, accompanied by attributes about each error such as how often it occurred, how long it took to fix it on average etc., could be quite interesting and should be a good inspiration for new programming tools.
Analyze the Error Distribution.	<b>Look at Error Distribution</b> If Live Coding enables developers to more easily find errors they just made while writing code, one would expect less errors to build up during programming. Following this theory, a developer without a Live Coding tool would write his code, make several errors and then, when they are done implementing, start to fix the errors and reduce the number of errors again. This means, the total number of errors active in a piece of source code should, on average, be lower for Live Coding developers, even if they do not make less errors than their peers.
	<b>Analyzing the Raw Change Data</b> Since we recorded every single change made by a developer we have a lot of raw data about how developers change their source code. This makes several potentially interesting analysis possible. For

example, one could look at how often code is copied either from other parts of the code or from outside the code base, since these changes should be rather easy to distinguish from the developer simply typing in code (which would mean an individual change per character). It could also be used to analyze how much time tools such as Auto Completion actually save, by looking at how often it was used and how often the suggestion was correct. Many more analyses of this kind of change data could be possible.

There are likely more analyses possible.

### 7.2.2 New Programmer Error Studies

While we hope our study provides a good start into learning more about experienced programmers everyday errors, there are still numerous other kinds of tasks developers are working on, which might differ in important details from our task selection. Also, when looking at, e.g., different kinds of errors, it might be interesting to see in what way these differ between different programming languages and programming paradigms. For example, how many of the common everyday errors can type-safe languages prevent or detect earlier?

### 7.2.3 Improving the Live Coding Tool

In Section 6.5—“Improvements to the Prototype” we listed several feature requests participants made during our study and highlighted a few other problems with our prototype. These observations could be used to improve our prototype further. But there are also other interesting directions. For example, one could think about integrating current code-proposal tools such as Auto-Completion with Live Coding. This way, when looking for a specific function to use one would not only see the different possibilities of functions and possibly the accompanying documentation, one would also see how the previewed function would affect the code without actually adding it to the code. Such a Live Coding plugin would approach Tanimoto’s Liveness level 5 (see Section 2.1—“Different Levels of Liveness”).

There is still a lot of room for improvement for the Live Coding tool.

#### **7.2.4 Research into Better Understanding of the Effect of Live Coding**

We suspect that a Live Coding tool, when used in everyday programming strongly affects the way programmers program. Some of our participants, when using the Live Coding plugin, made statements like “it tempts me into experimenting”. Also, the author himself, when implementing and testing the prototype suddenly felt an urge to “just try” things instead of reading up on how to correctly use that function or API. For example, it was possible to implement the RSS parser correctly without ever looking at the underlying XML feed once, just by looking at the data that got passed around in the functions and then incrementally modifying the functions to take the data apart and convert it to the desired format. Therefore, a long-term study evaluating the effects of a Live Coding plugin could be very interesting. Will Live Coding users do a lot more trial-and-error programming? And will that lead to better or worse code quality? Might that be actually faster?

## Appendix A

# Tasks Used in the User Study

Task1-1

# Task 1.1: RSS-Feed Parsing

You want to build a simple RSS-Reader to read and search through articles from [daringfireball.net](http://daringfireball.net) using JavaScript/Node.js. To do so, you plan to implement a function `getArticles()`, which takes a callback and asynchronously calls it as soon as the RSS-XML-file was successfully parsed. The articles extracted from the feed should be given as an argument to the callback. For performance reasons, you want to use a stream-based SAX XML parser (see hints for more information about SAX parsers).

The method you implement should have the following signature:

```
getArticles(callback)
```

- `callback`: a function that should be called when the requested entries have been retrieved. It takes one argument, an array of articles. Each article object should have the following form:
  - {
    - `title`: The title of the article. (`String`)
    - `published`: The date the article was published, in UTC/GMT. (`Date`)
    - `link`: A url to the article. (`String`)
    - `shortURL`: A short url to the article. (`String`)
    - `author`: The name of the author. (`String`)
    - `content`: The content (the complete html) of the article. (`String`)

Example input:

```
getArticles(function (articles) {  
    // array of articles should have 47 articles  
    // with the required properties  
});
```

A web server for testing purposes is installed on your machine. The URL of the RSS-Feed for this task is <http://localhost/daringfireball/index.xml>. It is already used in the provided implementation. The `published`-dates in the XML are in UTC/GMT. You may use a browser to look at the file, if you like.

## Hints

- A SAX parser does not construct a complete XML-Tree but instead posts events when it encounters nodes. You can register for the events you are interested in, e.g., when the parser encounters an opening-tag or when it encounters a text-node. A link to the documentation of the SAX parser you should use is given below. An excerpt of the documentation can be found at the end.
- A SAX parser does not tell you what the parent of the current node is or what its children are, so you have to write your own code to remember that.
- The `Date` constructor accepts the date-strings in the RSS-Feed-XML.
- You do not have to implement a true RSS-parser, it is fine if it just works for the Daring Fireball feed. You also do not have to retrieve other data than specified in the article object description above.
- You can assume that the XML is valid and always has the format given in the example file.



Useful documentation links:

- Date: [https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Date](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Date)
- Array: [https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Array/](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/)
- SAX-Parser: <https://github.com/isaacs/sax-js#usage>

Possibly useful events the SAX parser emits:

- `error`: Indication that something bad happened. The error will be hanging out on `parser.error`, and must be deleted before parsing can continue.
- `text`: Text node. Argument: String of text.
- `opentag`: An opening tag. Argument: object with `name` and `attributes`.
- `closetag`: A closing tag. Note that self-closing tags will have `closeTag` emitted immediately after `openTag`. Argument: tag name.
- `attribute`: An attribute node. Argument: object with `name` and `value`.
- `opencdata`: The opening tag of a `<![CDATA[` block.
- `cdata`: The text of a `<![CDATA[` block. Since `<![CDATA[` blocks can get quite large, this event may fire multiple times for a single block, if it is broken up into multiple `write()`s. Argument: the String of random character data.
- `closecdata`: The closing tag `[]]>` of a `<![CDATA[` block.
- `end`: Indication that the closed stream has ended.

## Example Code:

### Parse XML:

```
var sax = require("sax");
var parser = sax.parser(true); //strict

parser.onerror = function (error) {
  console.log(error.message);
};

parser.onopentag = function (node) {
  console.log("Opened tag " + node.name);
};

parser.onend = function () {
  console.log("Reached end of document");
};

parser.write("<myDoc>foo</myDoc>");
parser.close();
```

# Task 1.2: Filtering by Dates

You want enhance your RSS-Reeder for [daringfireball.net](http://daringfireball.net) by adding functionality to filter by dates. To do so you changed your function `getArticles()` to `getArticlesBetweenDates()` and implemented a date comparison. Now you want to make the input more flexible and not require all the properties of a date to be set. E.g. if just a day is given it should be interpreted as the given day of the current month of the current year at midnight. That is, your method should accept partial objects. All that is left to change now is to convert these partial objects into actual and complete `Date` objects.

To do so you have to implement a method with the following signature:

```
createDateFromDateInfo(dateInfo)
```

- `dateInfo`: an object representing the date that should be constructed:
  - {
    - `year`: the year of the date (e.g. 2013). Use current year if not given.
    - `month`: the month of the year (1-12). Use current month if not given.
    - `day`: the day of the month (1-31). Use current day if not given.
    - `hour`: the hour of the day (0-23). Use 0 if not given.
    - `minute`: the minutes of the hour (0 - 59). Use 0 if not given.
    - `second`: the seconds of the minute (0-59). Use 0 if not given.
  - This parameter will always be defined, but it might be an empty object.
  - The properties are all in UTC/GMT.
- **Return Value**: a `Date` object representing the time and date specified by the input object, in UTC.

Example input for the complete function:

```
getEntriesBetweenDates({month: 5, day: 1, hour: 0}, {}, function (articles) {  
    // array of articles should have 19 articles  
    // with the required properties  
});
```

## Hints

- You don't have to check the input dates for the correct format (e.g. `day = 32`, or `month = 13`, or `second = 100`). You can assume that only valid values will be used.
- You can ignore leap seconds.
- Time zones are important, though. Remember that the input date dictionaries as well as the output `Date` objects of the article's objects should be in UTC.

Useful documentation links:

- `Date`: [https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Date](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Date)
- `Array`: [https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Array/](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/)

# Task 3: Dijkstra's Algorithm

You want to implement Dijkstra's algorithm using JavaScript/Node.js to solve the single-source shortest-path problem on a given graph, i.e., for a given start node in the graph find the shortest paths to all other nodes.

The method you implement should have the following signature:

```
calculateShortestPathFromNodeToOtherNodesInGraph(startName, graph)
```

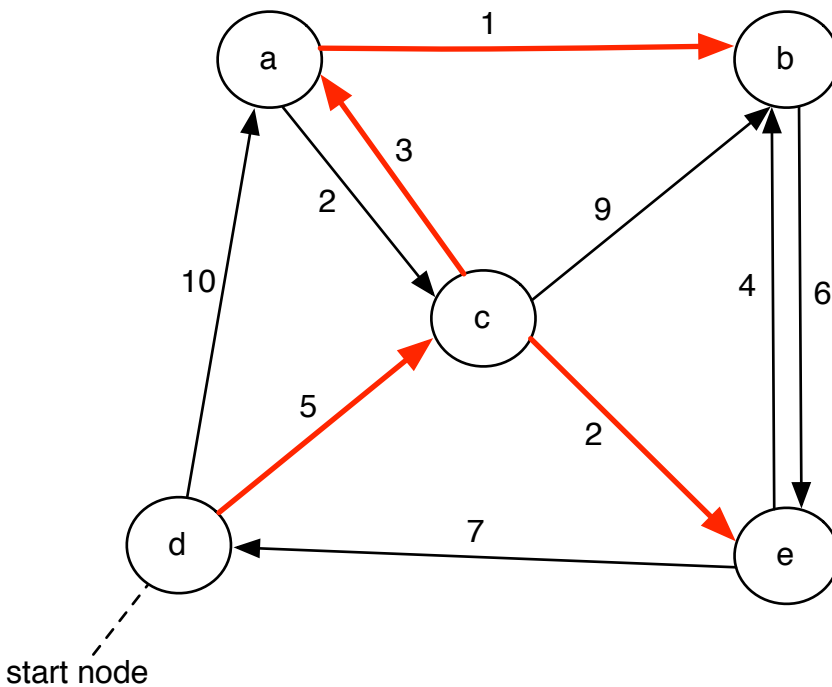
- `startName`: a String providing the name of the start node, for which to calculate the shortest paths
  - `graph`: a graph in the following format
    - {
      - `nodes`: An object/dictionary with the node names as keys and the nodes as value. The nodes have the following form. {
        - `name`: the name (unique identifier) of the node
    - `edges`: Object/dictionary with the `from`-node names as keys and the values being arrays objects/dictionaries again with the `to`-node name as key and the `weight` as value (see example).
- **Return Value**: return the original graph, but modify the nodes so each has two additional properties:
  - `predecessor`: name of the predecessor in the shortest path from the start node to this node.
  - `distance`: The distance of this node from the start node. (number)
- You do not have to copy the graph or the nodes, it is fine if you modify the input.

Example input:

```
var graph = {
  nodes: { a: {name: "a"}, b: {name: "b"}, c:{name: "c"}, d:
{name: "d"}, e:{name: "e"} }
};
graph.edges = {
  a: {b: 1, c: 2},
  b: {e: 6},
  c: {a: 3, b: 9, e: 2},
  d: {a: 10, c: 5},
  e: {d: 7, b: 4}
};

var result = calculateShortestPathFromNodeToOtherNodesInGraph("d",
graph);
console.log(result.nodes.a.predecessor); // prints "c"
```

## Graphical Representation of the graph in the example input and its shortest paths.



## Dijkstra's algorithm (as explained on Wikipedia)

Let the node at which we are starting be called the **initial node**. Let the **distance of node Y** be the distance from the **initial node** to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the *unvisited set* consisting of all the nodes except the initial node.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. For example, if the current node A is marked with a distance of 8, and the edge connecting it with a neighbor B has length 1, then the distance to B (through A) will be  $8 + 1 = 9$ . If this distance is less than the previously recorded tentative distance of B, then overwrite that distance. Even though a neighbor has been examined, it is not marked as "visited" at this time, and it remains in the *unvisited set*.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the *unvisited set* is empty, stop. The algorithm has finished.
6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

## Useful documentation links:

- Array: [https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Array/](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/)
- Array forEach(): Takes a function and applies it to each element of the array
  - [https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Array/forEach](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/forEach)
- Array sort(): Takes a function with two arguments and uses it to compare elements of the array to each other and then sort the array.
  - [https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Array/sort](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/sort)
- Array shift(): Removes the first element of an array and returns it.
  - [https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Array/shift](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/shift)

## Hints

- Please do not search for (pseudo) code and just copy the implementation but instead implement the algorithm yourself using the explanations given above.
- For performance reasons, Dijkstra's algorithm is usually implemented using a priority queue for the set of unvisited nodes. This is not necessary in this task. You can simply use a normal JavaScript array and sort it to find the node with the smallest tentative distance to the start node or simply iterate through it to find the smallest one. We are only interested in a working implementation, for now, not the most efficient one.
- Dijkstra's algorithm requires a graph in which all edges have a positive weight. You can assume that this requirement is fulfilled by input provided to the function and do not have to check it yourself.
- You can assume that the graph is fully connected.

## Example Code

```
// Iterate over the properties of an object
var myObj = {firstName: "Paul", middleName: "Bar", lastName: "Pi"},
    names = [];
var property;
for (property in myObj) {
    names.push(myObj[property]);
}
console.log(names) // will print: ["Paul", "Bar", "Pi"]
```



---

## Appendix B

# Questionnaires

### B.1 Pre Task Questionnaires

#### Live Coding Pre Task 1 Questionnaire

Participant ID:

Experience with	No Experience				A lot of Experience
1. XML					
2. XML-Parsing					
3. Using SAX-Parsers					
4. Programming using Dates/ Times/Calendars					

**Figure B.1:** *The questionnaire participants received before Task 1.1 and Task 1.2.*

## Live Coding Pre Task 3 Questionnaire

Participant ID:

	Never	Once			Very often/ Regularly
1. I have implemented Dijkstra's algorithm myself					
2. I have implemented Dijkstra's algorithm myself in a real application (not just as an exercise)					
Experience with	No Experience				A lot of Experience
3. Implementing Algorithms					
4. Implementing Graphs and Graph-based Algorithms					

**Figure B.2:** The questionnaire participants received before Task 3.



## **B.2 Post Task Questionnaires**

Participant ID:

Task:

	Strongly Disagree				Strongly Agree
1. The task was difficult for me.					
2. The tool helped me solve the task.					
3. I would have solved the task just as fast without the tool.					
4. I understand what the code I wrote does exactly and why it works (or doesn't).					
5. I was able to implement most of the requirements.					
6. I am confident that my solution is correct.					

What made the task difficult?

What problems did you encounter (minor ones as well)?

**Figure B.3:** *The questionnaire participants received after each task.*

## **B.3 Post Session Questionnaires**

Participant ID:

	Strongly Disagree				Strongly Agree
1. I think that I would like to use this system frequently					
2. I found the system unnecessarily complex					
3. I thought the system was easy to use					
4. I think that I would need the support of a technical person to be able to use this system					
5. I found the various functions in this system were well integrated					
6. I thought there was too much inconsistency in this system					
7. I would imagine that most people would learn to use this system very quickly					
8. I found the system very awkward to use					
9. I felt very confident using the system					
10. I needed to learn a lot of things before I could get going with this system					
11. I found understanding my source code easy using the system					
12. I do not think the plugin has benefits for code understanding (compared to Brackets without the plugin)					
13. I think implementing functionality is faster using the plugin (compared to Brackets without the plugin)					
14. I think I would have been faster or just as fast when implementing the functionality without the plugin.					
15. I felt more confident that my code is correct because of the plugin.					
16. Using the plugin did not increase my confidence in the correctness of my code.					
17. I found the plugin distracting.					
18. The plugin did not distract me when I did not need it.					

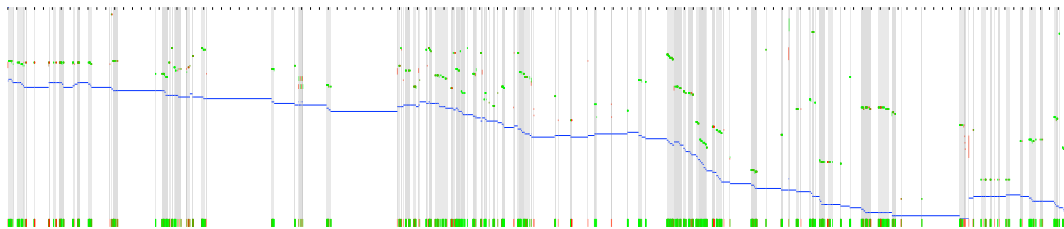
**Figure B.4:** *The questionnaire the Live Coding participants received after they completed all tasks.*

## Appendix C

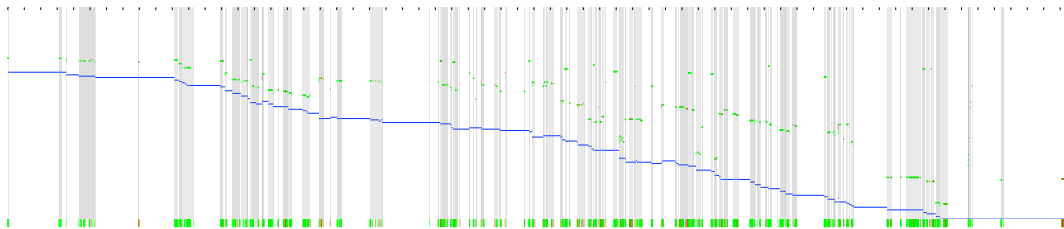
# Change Cluster Graphs

In this appendix we show all the change cluster graphs of all participants and all tasks.

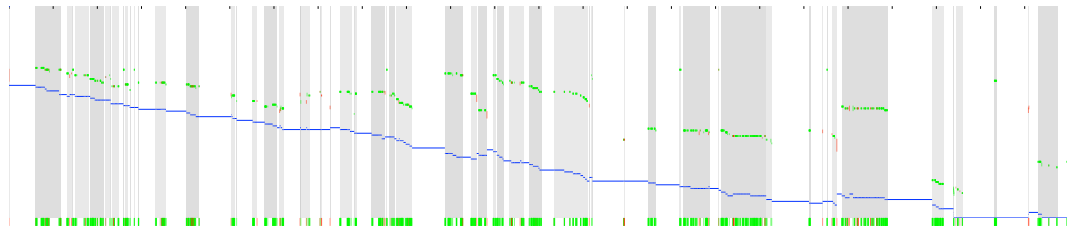
### C.1 Task 1.1



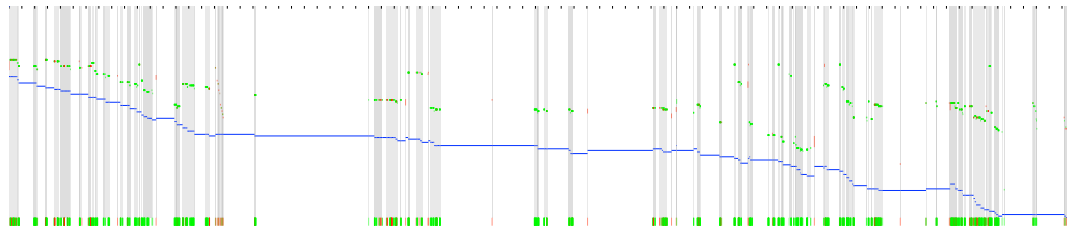
**Figure C.1:** Change Cluster Graph for Participant E. For an explanation of the graph please refer to Figure 6.15.



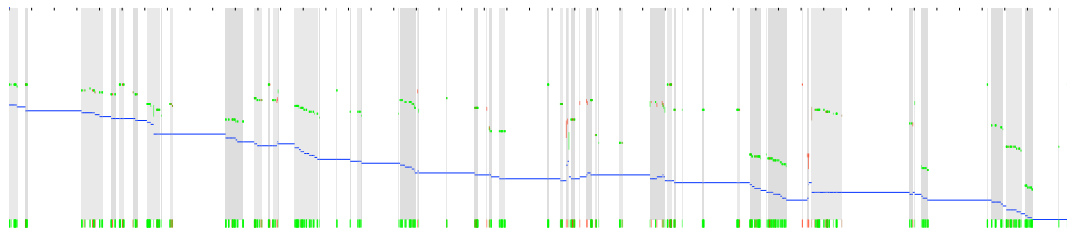
**Figure C.2:** Change Cluster Graph for Participant G. For an explanation of the graph please refer to Figure 6.15.



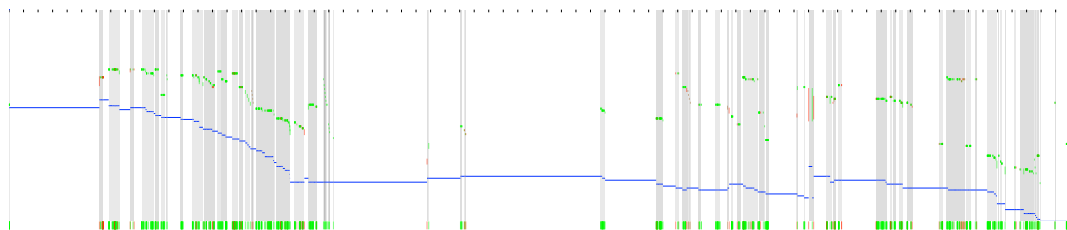
**Figure C.3:** Change Cluster Graph for Participant H. For an explanation of the graph please refer to Figure 6.15.



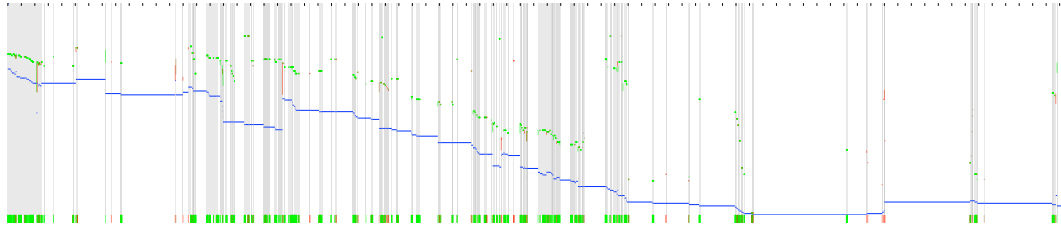
**Figure C.4:** Change Cluster Graph for Participant E. For an explanation of the graph please refer to Figure 6.15.



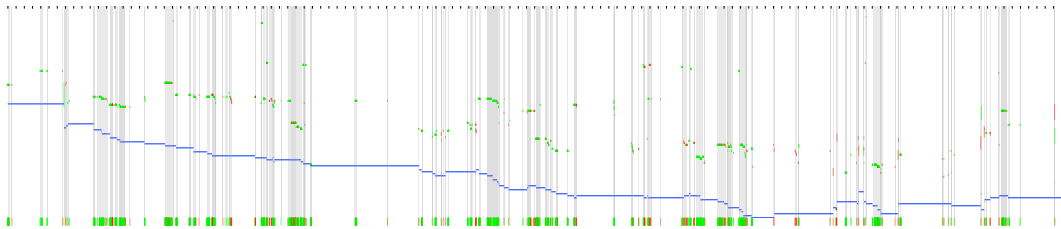
**Figure C.5:** Change Cluster Graph for Participant I. For an explanation of the graph please refer to Figure 6.15.



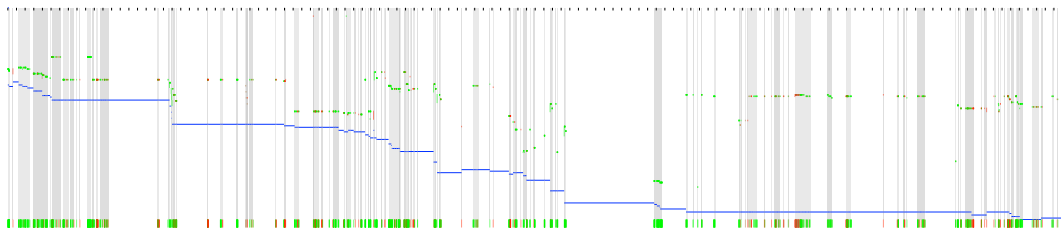
**Figure C.6:** Change Cluster Graph for Participant J. For an explanation of the graph please refer to Figure 6.15.



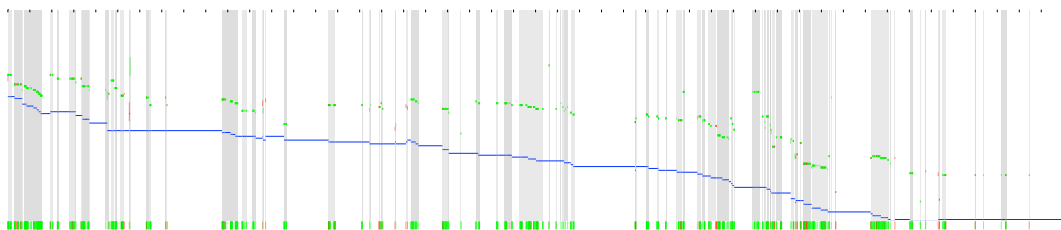
**Figure C.7:** Change Cluster Graph for Participant C. For an explanation of the graph please refer to Figure 6.15.



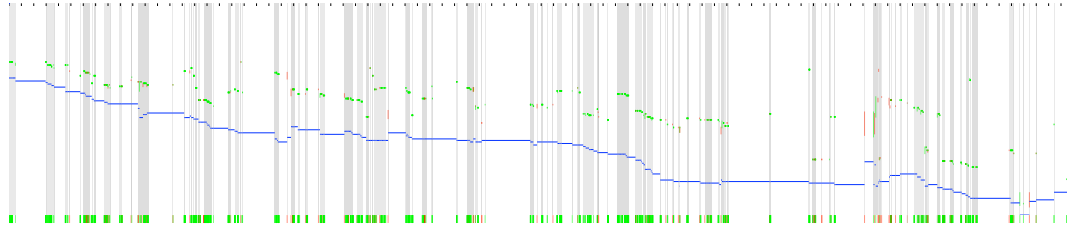
**Figure C.8:** Change Cluster Graph for Participant M. For an explanation of the graph please refer to Figure 6.15.



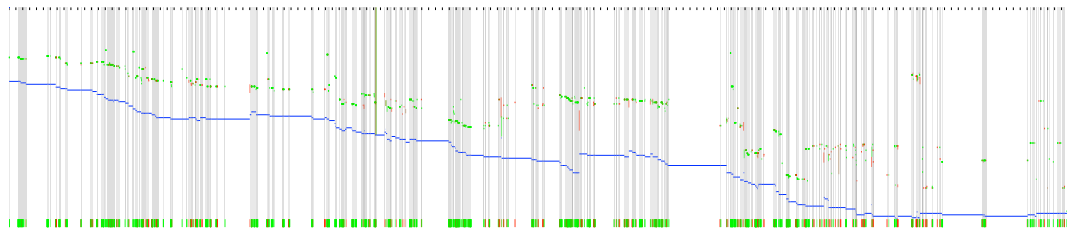
**Figure C.9:** Change Cluster Graph for Participant L. For an explanation of the graph please refer to Figure 6.15.



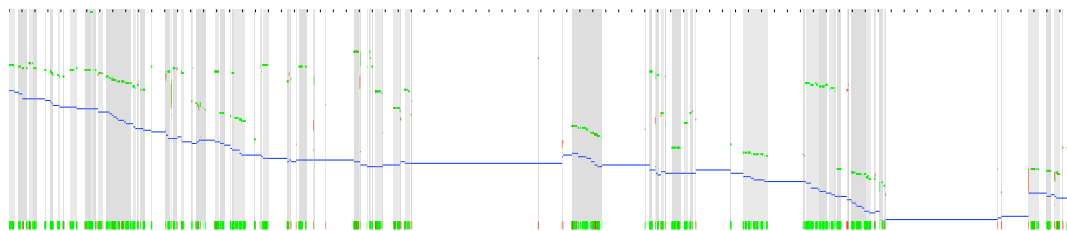
**Figure C.10:** Change Cluster Graph for Participant B. For an explanation of the graph please refer to Figure 6.15.



**Figure C.11:** Change Cluster Graph for Participant K. For an explanation of the graph please refer to Figure 6.15.



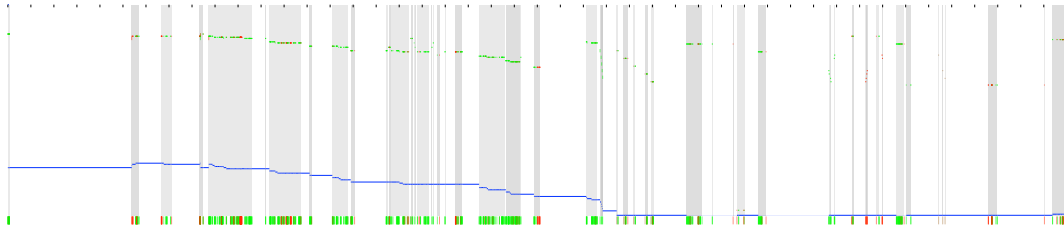
**Figure C.12:** Change Cluster Graph for Participant A. For an explanation of the graph please refer to Figure 6.15.



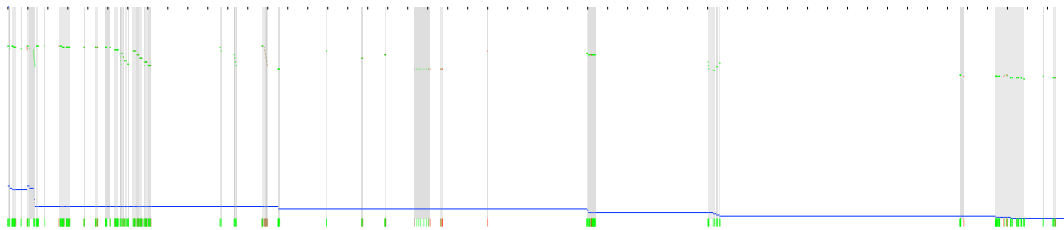
**Figure C.13:** Change Cluster Graph for Participant D. For an explanation of the graph please refer to Figure 6.15.



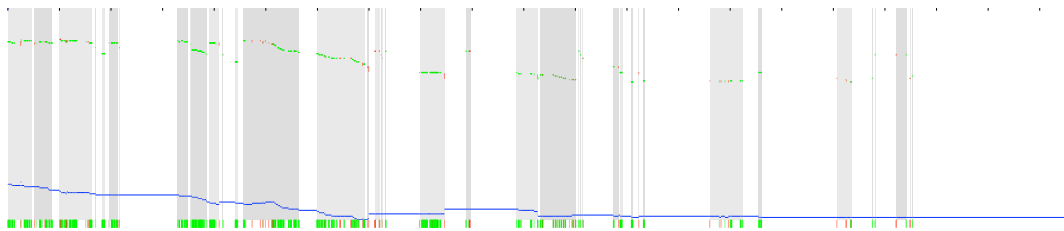
## C.2 Task 1.2



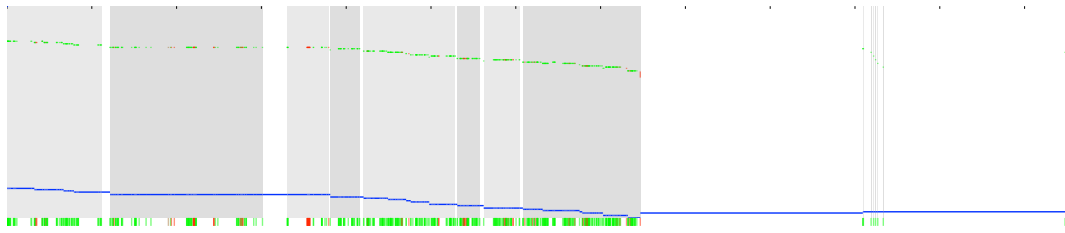
**Figure C.14:** Change Cluster Graph for Participant E. For an explanation of the graph please refer to Figure 6.15.



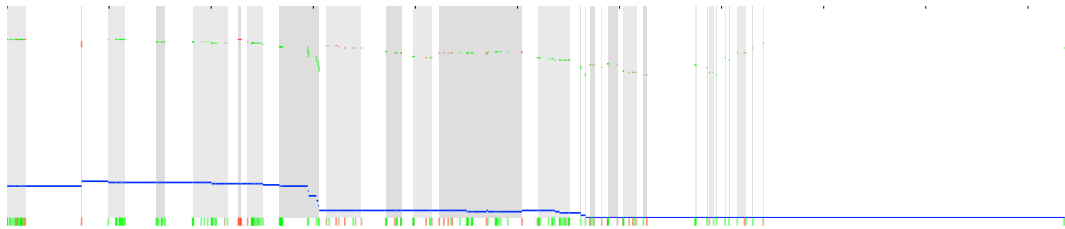
**Figure C.15:** Change Cluster Graph for Participant G. For an explanation of the graph please refer to Figure 6.15.



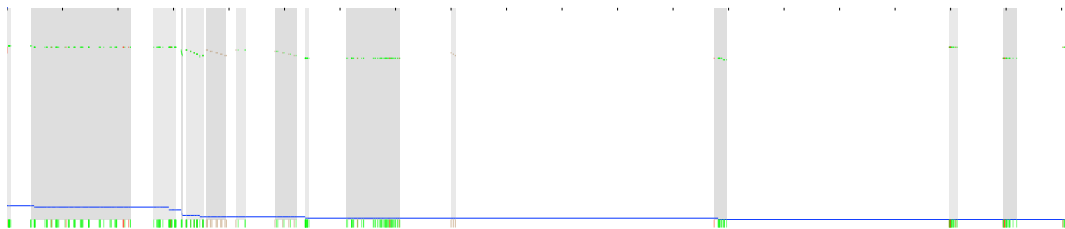
**Figure C.16:** Change Cluster Graph for Participant H. For an explanation of the graph please refer to Figure 6.15.



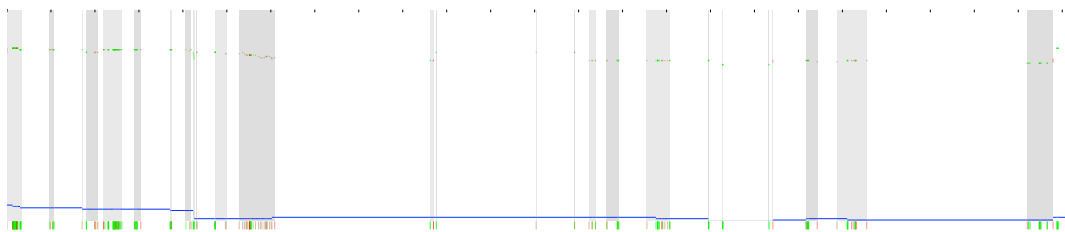
**Figure C.17:** Change Cluster Graph for Participant F. For an explanation of the graph please refer to Figure 6.15.



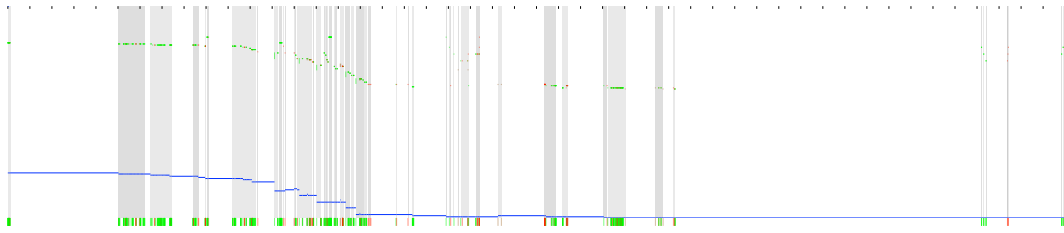
**Figure C.18:** Change Cluster Graph for Participant I. For an explanation of the graph please refer to Figure 6.15.



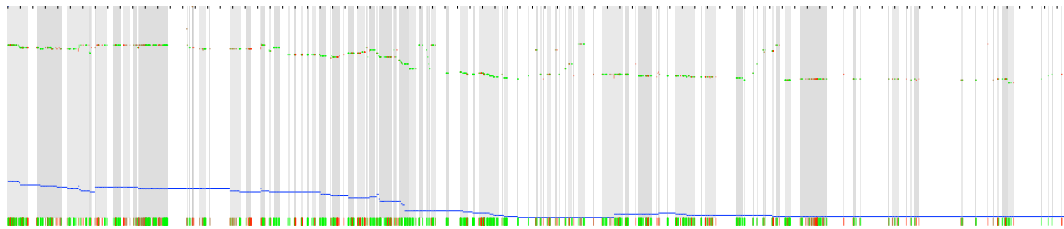
**Figure C.19:** Change Cluster Graph for Participant J. For an explanation of the graph please refer to Figure 6.15.



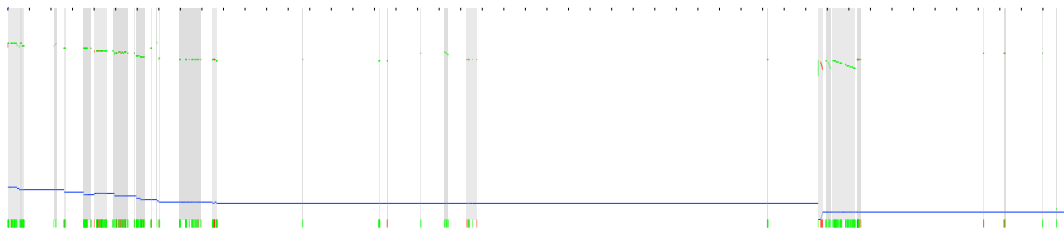
**Figure C.20:** Change Cluster Graph for Participant C. For an explanation of the graph please refer to Figure 6.15.



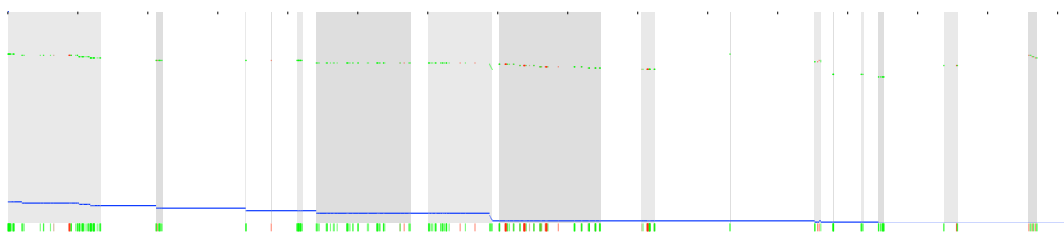
**Figure C.21:** Change Cluster Graph for Participant M. For an explanation of the graph please refer to Figure 6.15.



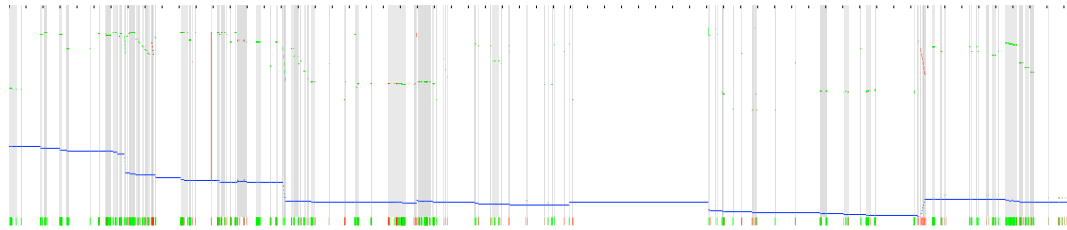
**Figure C.22:** Change Cluster Graph for Participant L. For an explanation of the graph please refer to Figure 6.15.



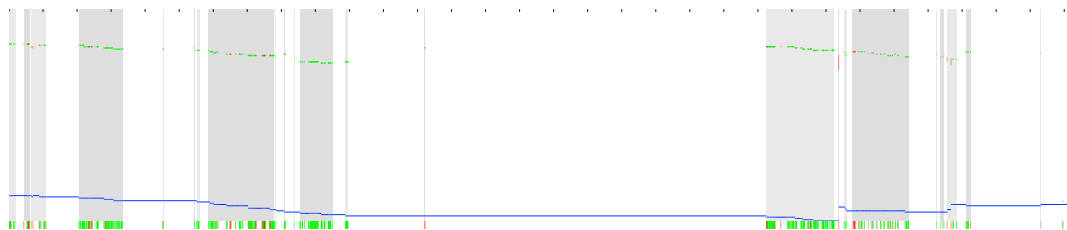
**Figure C.23:** Change Cluster Graph for Participant B. For an explanation of the graph please refer to Figure 6.15.



**Figure C.24:** Change Cluster Graph for Participant K. For an explanation of the graph please refer to Figure 6.15.

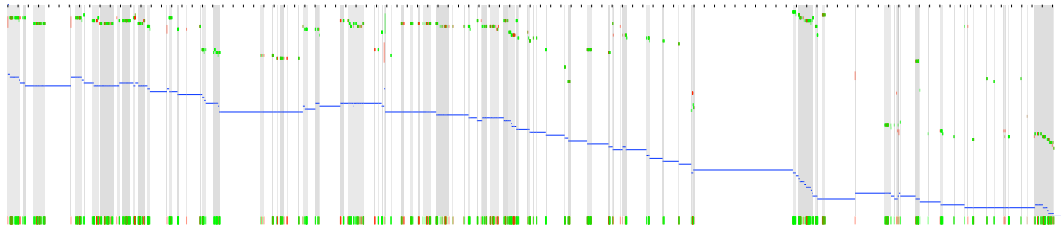


**Figure C.25:** Change Cluster Graph for Participant A. For an explanation of the graph please refer to Figure 6.15.

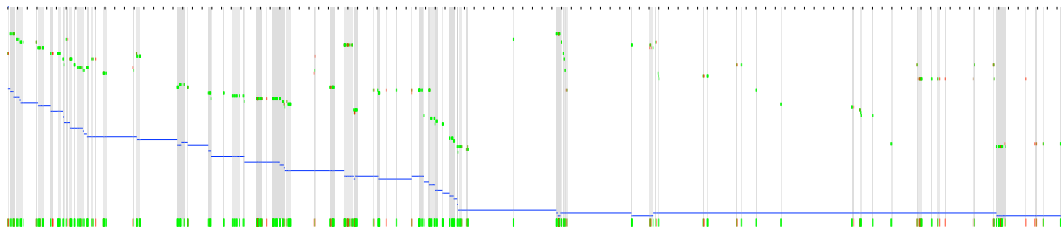


**Figure C.26:** Change Cluster Graph for Participant D. For an explanation of the graph please refer to Figure 6.15.

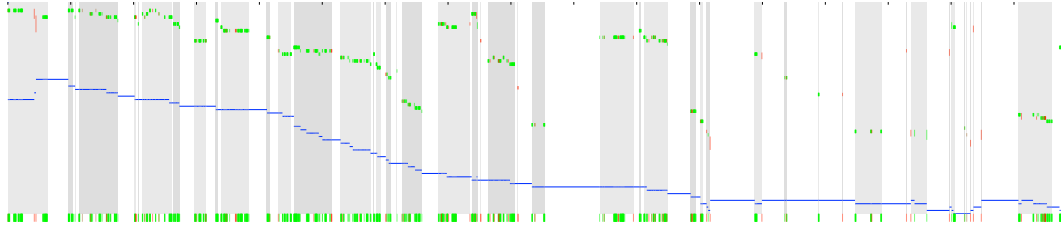
### C.3 Task 3



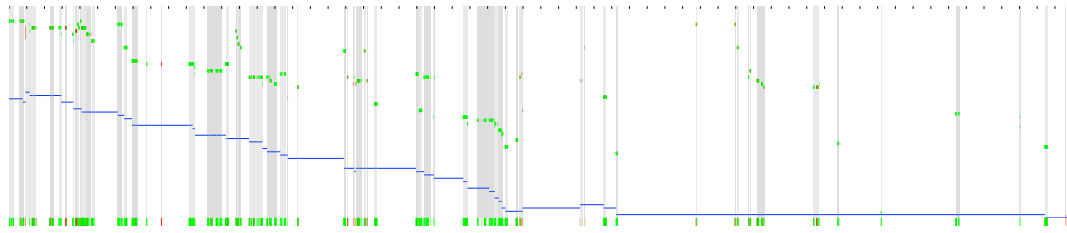
**Figure C.27:** Change Cluster Graph for Participant E. For an explanation of the graph please refer to Figure 6.15.



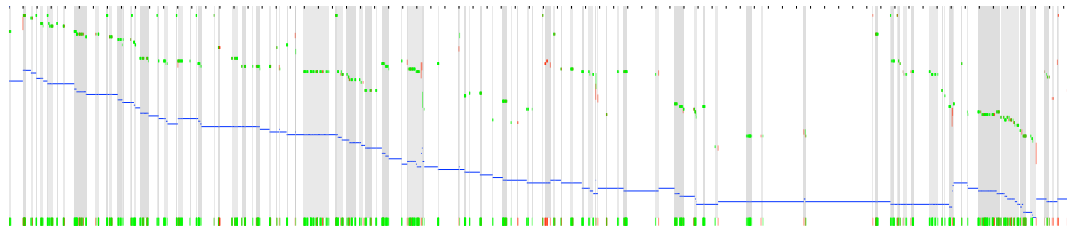
**Figure C.28:** Change Cluster Graph for Participant G. For an explanation of the graph please refer to Figure 6.15.



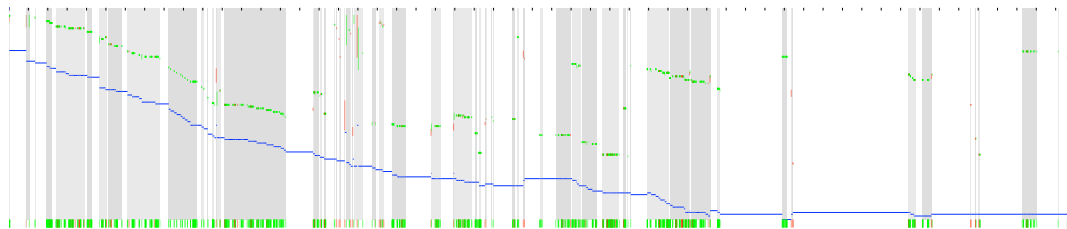
**Figure C.29:** Change Cluster Graph for Participant H. For an explanation of the graph please refer to Figure 6.15.



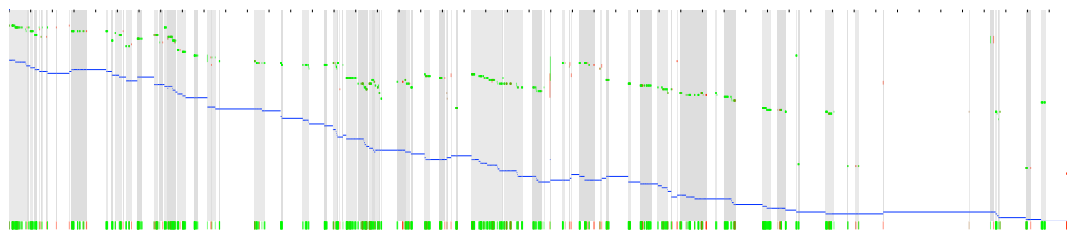
**Figure C.30:** Change Cluster Graph for Participant E. For an explanation of the graph please refer to Figure 6.15.



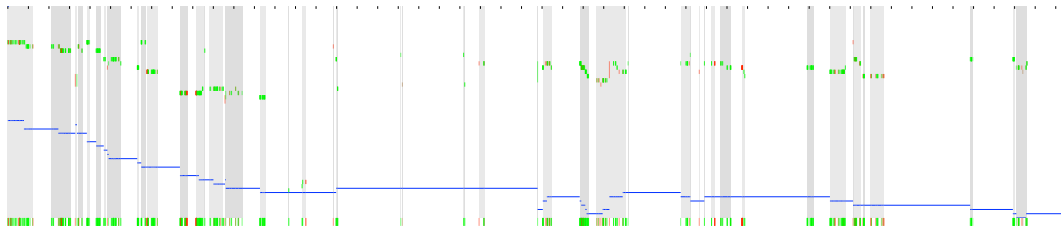
**Figure C.31:** Change Cluster Graph for Participant I. For an explanation of the graph please refer to Figure 6.15.



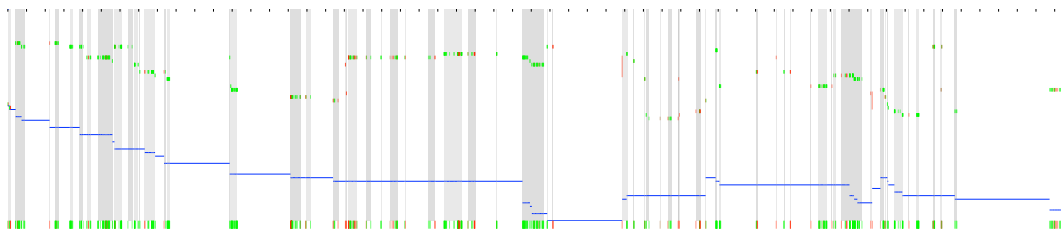
**Figure C.32:** Change Cluster Graph for Participant J. For an explanation of the graph please refer to Figure 6.15.



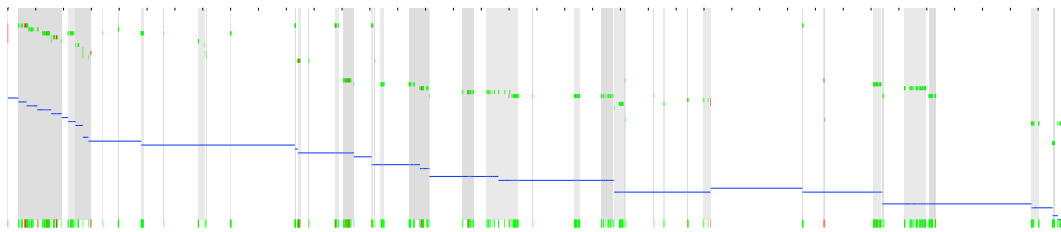
**Figure C.33:** Change Cluster Graph for Participant C. For an explanation of the graph please refer to Figure 6.15.



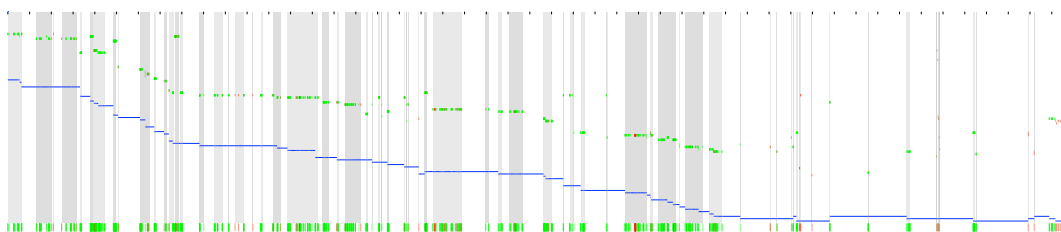
**Figure C.34:** Change Cluster Graph for Participant M. For an explanation of the graph please refer to Figure 6.15.



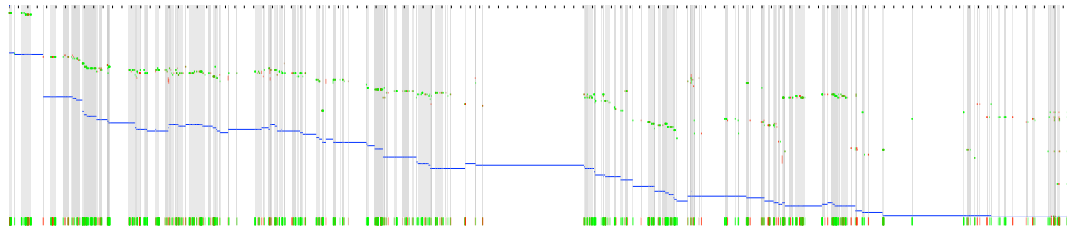
**Figure C.35:** Change Cluster Graph for Participant L. For an explanation of the graph please refer to Figure 6.15.



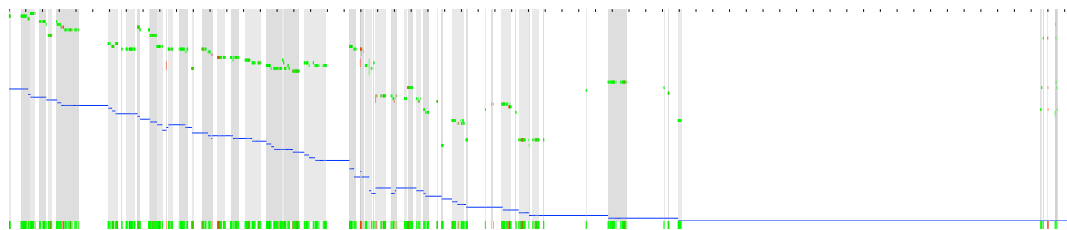
**Figure C.36:** Change Cluster Graph for Participant B. For an explanation of the graph please refer to Figure 6.15.



**Figure C.37:** Change Cluster Graph for Participant K. For an explanation of the graph please refer to Figure 6.15.



**Figure C.38:** Change Cluster Graph for Participant A. For an explanation of the graph please refer to Figure 6.15.



**Figure C.39:** Change Cluster Graph for Participant D. For an explanation of the graph please refer to Figure 6.15.



# Bibliography

- Aaron Bangor, Philip T Kortum, and James T Miller. An Empirical Evaluation of the System Usability Scale. *International Journal of Human-Computer Interaction*, 24(6):574–594, July 2008.
- Ewgenij Belzmann. Utilization and Visualization of Program State as Input Data in a Live Coding Environment. Master's thesis, RWTH Aachen University, Aachen, April 2013.
- John Brooke. SUS-A quick and dirty usability scale. *Usability evaluation in industry*, 189:194, 1996.
- Renée C Bryce, Alison Cooley, Amy Hansen, and Nare Hayrapetyan. A one year empirical study of student programming bugs. In *Frontiers in Education Conference (FIE), 2010 IEEE*, October 2010.
- Margaret M Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(2), March 2001.
- W Choi, J Brandt, and S R Klemmer. Rehearse: Coding Interactively while Prototyping. 2008.
- Douglas H Clements and Julie Sarama. Design of a logo environment for elementary geometry. *The Journal of Mathematical Behavior*, 14(4):381–398, December 1995.
- Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. Alice. In *the SIGCHI conference*, pages 486–493, New York, New York, USA, 2000. ACM Press.

- Curtis Cook, Margaret M Burnett, and Derrick Boom. A bug's eye view of immediate visual feedback in direct-manipulation programming systems. *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, October 1997.
- E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.
- Jonathan Edwards. Example centric programming. *SIGPLAN Notices*, 39(12), December 2004.
- M Eisenstadt. Tales of Debugging from the Front Lines. *Empirical Studies of Programmers: Fifth Workshop*, 1993.
- James B Fenwick, Jr, Cindy Norris, Frank E Barry, Josh Rountree, Cole J Spicer, and Scott D Cheek. Another look at the behaviors of novice programmers. In *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*. ACM Request Permissions, March 2009.
- Andy Field. Multilevel Linear Models. In *Discovering statistics using SPSS*. Sage Publications Ltd, London, 2009.
- P M Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47(6):381–391, May 1954.
- David J Gilmore. Interface Design: Have we got it wrong. *INTERACT*, 1995.
- John D Gould. Some Psychological Evidence on How People Debug Computer Programs. 1975.
- T. R. G. Green and M Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.
- Björn Heinen. *A Live Coding Editor*. PhD thesis, RWTH Aachen University, September 2012.
- Peter Henderson and Mark Weiser. Continuous execution: the VisiProg environment. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*. IEEE Computer Society Press, August 1985.

- Matthew C Jadud. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education*, 15(1): 25–40, December 2004.
- W L Johnson, E Soloway, B Cutler, and S Draper. Bug catalogue: I., 1983.
- Andrew J Ko and Brad A Myers. Development and evaluation of a model of programming errors. In *IEEE Symposium on Human Centric Computing Languages and Environments, 2003*. 2003, pages 7–14. IEEE, 2003.
- Andrew J Ko and Brad A Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *CHI '04: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Request Permissions, April 2004.
- Andrew J Ko and Brad A Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, 16(1-2), February 2005.
- Andrew J Ko and Brad A Myers. Debugging reinvented. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, 2008.
- Andrew J Ko and Brad A Myers. Finding causes of program output with the Java Whyline. In *CHI '09: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Request Permissions, April 2009.
- Andrew J Ko, Htet Htet Aung, and Brad A Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 126–135, 2005.
- Thomas D. LaToza and Brad A Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 185–194, May 2010.
- Thomas D. LaToza, David Garlan, James D Herbsleb, and Brad A Myers. Program comprehension as fact finding. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of*

*the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM Request Permissions, September 2007.

James Lewis and Jeff Sauro. The Factor Structure of the System Usability Scale. In Masaaki Kurosu, editor, *Lecture Notes in Computer Science*, pages 94–103. Springer Berlin / Heidelberg, 2009.

Henry Lieberman. The Debugging Scandal and What to Do About It. *Communications of the ACM*, 40(4):26–29, April 1997.

John H Maloney and Randall B Smith. Directness and liveness in the morphic user interface construction environment. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*. ACM Request Permissions, December 1995.

T Munzner. A Nested Model for Visualization Design and Validation. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):921–928, 2009.

Jakob Nielsen. Chapter 5. In *Usability Engineering*. Academic Press, Inc, Cambridge, 1993.

Donald A Norman. *The Design of Everyday Things*. Basic Books, basic books edition, April 1988.

Seymour Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., January 1980.

John Resig. Redefining the Introduction to Computer Science, August 2012. URL <http://ejohn.org/blog/introducing-khan-cs/>.

David Saff. Automated continuous testing to speed software development . Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, February 2004.

David Saff and Michael D Ernst. Reducing wasted development time via continuous testing. *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 281–292, 2003.

- David Saff and Michael D Ernst. An experimental evaluation of continuous testing during development. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. ACM Request Permissions, July 2004.
- B Computer Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, 16(8), 1983.
- J Sillito, G.C Murphy, and K De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.
- Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Request Permissions, November 2006.
- J L Snell. Ahead-of-time debugging, or programming not in the dark. In *Software Technology and Engineering Practice, 1997. Proceedings., Eighth IEEE International Workshop on [incorporating Computer Aided Software Engineering]*, pages 288–293. IEEE Computer Society, 1997.
- J C Spohrer and E Soloway. Alternatives to construct-based programming misconceptions. In *CHI '86: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Request Permissions, April 1986.
- Steven L Tanimoto. VIVA: A visual language for image processing. *Electronic Notes in Theoretical Computer Science*, 1(2):127–139, June 1990.
- Steven L Tanimoto. A Perspective on the Evolution of Live Programming. *Workshop on Live Programming (LIVE)*, 2013.
- Bret Victor. Learnable Programming, September 2012a. URL <http://worrydream.com/LearnableProgramming/>.
- Bret Victor. Inventing on Principle, January 2012b. URL <http://worrydream.com/InventingOnPrinciple/>.

E M Wilcox, John Atwood, Margaret M Burnett, J J Cadiz, and Curtis R Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *CHI '97: Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. ACM Request Permissions, March 1997.

