Fiddlets: Contextualised Inline Code-Execution Environments

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

by
Dennis Lewandowski

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Bernhard Rumpe

Registration date: 21.05.2015
Submission date: 04.12.2015

Eidesstattliche Versicherung

Lewandowski, Dennis	317071
Name, Vorname	Matrikelnummer

Ich versichere hiermit an Eides Statt, dass ich die vorliegende ~~Arbeit/Bachelorarbeit/~~ Masterarbeit* mit dem Titel

Fiddlets: Contextualised Inline Code-Execution Environments

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum	Unterschrift

Contents

Abstract **xiii**

Überblick **xv**

Acknowledgements **xvii**

Conventions **xix**

1 Introduction **1**

1.1 Motivation 1

1.2 Introducing Fiddlets 4

1.3 Chapter Overview 5

2 Related work **7**

2.1 Debuggers 8

2.1.1 Back in Time Debugging 10

2.1.2 Reachability 11

2.2 Source Code Generation and Code Snippets . 15

2.2.1 Automatic Source Code Generation . 15
2.2.2 Code Snippet Libraries 16
2.3 Live Coding 19
2.4 Summary 22

3 Design 23

3.1 First Iteration: Live Autocomplete 24
3.1.1 Idea 24
3.1.2 Study: Trivia Implementation 25
3.1.3 Evaluation 25
3.2 Second Iteration: Runtime Data Widgets . . . 27
3.2.1 Idea 28
jQuery Selector Widget 29
jQuery Insert Widget 29
Function Chain Widget 31
3.2.2 Evaluation 33
3.3 Third Iteration: Adding Context 34
3.3.1 Idea 35
3.3.2 Evaluation 37
3.4 Summary 38

4 Fiddlets 39

4.1 Concept 39

4.2 Implementation 40

4.2.1 Context 42

4.2.2 Expanding the Context 45

4.2.3 Value Tracing 45

4.2.4 Visualisation 47

5 Study Design 49

5.1 Prototype 50

5.1.1 Limitations 50

5.1.2 Visualisations 51

5.2 Study 57

5.2.1 Setup 57

5.2.2 Procedure 59

5.3 Task Design 60

5.3.1 Task 1: Implementing a Regular Expression 61

5.3.2 Task 2: Implementing a Regular Expression 2 62

5.3.3 Task 3: Bug-fixing a Split Operation . 62

5.3.4 Task 4: Transforming an Array 63

5.3.5 Task 5: Changing Array Contents . . 63

6 Evaluation 65

6.1 Participants 65

6.2 Results 67

6.2.1 Usage 67

6.2.2 Programmer Performance 70

6.2.3 User Reception and Observations . . 76

6.3 Summary 78

7 Summary and future work 79

7.1 Summary and contributions 79

7.2 Future work 80

A Live Autocompletion Evaluation: Implementation Requirements 83

A.1 Trivia Game Requirements 83

A.2 Board Setup 83

A.3 Questions 83

A.4 Rules 84

A.5 Things to log 85

B Data Processing Tasks: Source Code 87

C User Study Task Descriptions 89

D Questionaires 95

D.1 Pre-Questionaire 96

D.2 SUS Scale 97

Bibliography **99**

Index **105**

List of Figures

2.1 Whyline user interface 13
2.2 Theseus user interface 14
2.3 SnipMatch 17
2.4 Codelets 18
2.5 Live Coding Prototype by Kurz 20
2.6 Xcode Playgrounds 21

3.1 Live Autocomplete Paper Prototype 24
3.2 Live Autocomplete Suggestions 25
3.3 Codelets Column Layout Widget 28
3.4 jQuery Selector Widget 30
3.5 jQuery Insert Widget 31
3.6 Function Chain Widget 32
3.7 Function Selection in the Function Chain Widget 33
3.8 High Fidelity Prototype 35
3.9 Value Selector 36

4.1 Fiddlets Architectural Overview 41

4.2 Simple Context Generation 43

4.3 Context with respect to scope. 44

4.4 Current Line Highlights 47

5.1 Study Prototype: Variable Assignment Visualisation 52

5.2 Study Prototype: String.prototype.replace Visualisation 53

5.3 Study Prototype: String.prototype.split Visualisation 54

5.4 Study Prototype: Array.prototype.map 55

5.5 Study Prototype: Array.prototype.splice . . . 56

6.1 Average relative usage times by task 68

6.2 Maximum relative usage per task 69

6.3 Number of Fiddlets invocations per task . . . 69

6.4 Task Completion Time Plot 72

D.1 Pre-Questionaire 96

D.2 Post-Questionaire: SUS 97

List of Tables

6.1 Age and programming experience (XP) of the participants 66

6.2 Reported preliminary knowledge about tools and concepts from the first questionnaire 66

6.3 Average usage time, maximum usage time and count of usage of all participants 68

6.4 "Task Completion Rate": number of successfully solved tasks by each participant 70

6.5 "Task Completion Time" for conditions and tasks. 71

6.6 *t* test results for TCT, assuming a difference between conditions (Fiddlets vs. Control) . . 73

6.7 Classification of the participants according to their xp_score 74

6.8 Results of ANOVA against TCT, using the classification based on the median xp_score as factor 75

6.9 t-values for reported preliminary knowledge, assuming no difference 76

Abstract

Modern programming framework and libraries ease programmers lives by providing solutions to well known problems and hiding the complexities of the solution behind convenient APIs. Many of those frameworks and libraries are available as open source projects and each of these projects comes with its own programming API. Learning an unfamiliar API requires the programmer to invest time and effort into experimenting with its functions and types. Therefore reducing the feedback times of edit-compile-run loops due to changes in the program is vital.

With this thesis we introduce Fiddlets, a novel interaction technique to reduce feedback times about program changes. Fiddlets achieves this by copying code of interest into a separate context for execution. To help the programmer set up code experiments, Fiddlets includes values from previous program executions into the context. Fiddlets provides interactive visualisations the execution result of the code experiment to help the programmer understand the effects of the code.

We present the necessary requirements needed for a complete implementation of Fiddlets and an implementation of a proof-of-concept prototype of the system. We conducted a user study to analyse the usability and performance of our prototype. Although we could not find significant differences in task completion time and task completion rate between programmers using Fiddlets and programmers using established programming tools, our analysis showed that programmers used Fiddlets in manifold ways.

Überblick

Moderne Frameworks und Libraries vereinfachen das Leben von Programmierern indem sie Lösungen für wohlbekannte Programmier-Probleme bieten und die Komplexität dieser hinter komfortablen APIs verstecken. Viele dieser Frameworks und Libraries sind als Open Source verfügbar und jedes dieser Projekte bringt sein eigenes API mit. Um ein unbekanntest API zu lernen muss der Programmierer viel Zeit und Aufwand in das experimentieren mit dessen Funktionen und Typen investieren. Es ist daher wichtig dass die Feedback-Zeit der Edit-Compile-Run-Durchläufe auf ein Minimum reduziert wird.

Die vorliegende Arbeit stellt Fiddlets vor, eine neuartige Interaktions-Technik zur Reduktion der Feedback-Zeit von Programm-Änderungen. Fiddlets erreicht dies indem es aktuell interessante Code-Abschnitte in einen separaten Ausführungs-Kontext kopiert. Um den Programmierer beim Erstellen von Code-Experimenten zu unterstützen bezieht Fiddlets Werte aus vorhergegangenen Programm-Ausführungen in diesen Kontext. Fiddlets verfügt über interaktive Visualisierungen für das Ausführungs-Ergebnis des Code-Experiments und unterstützt den Programmierer somit dabei, die Auswirkungen seines Codes besser zu verstehen.

Wir präsentieren hier die Anforderungen die an eine vollständige Implementierung von Fiddlets gestellt werden sowie die Implementierung eines Proof-of-Concept Prototypen. Wir haben eine Benutzer-Studie zur Ermittlung der Benutzbarkeit und Leistung unseres Prototypen durchgeführt. Obwohl wir keine signifikanten Unterschiede in der "Task Completion Time" und "Task Completion Rate" zwischen Teilnehmern die Fiddlets benutzten und solchen die es nicht benutzen feststellen konnten, zeigen unsere Analysen dass Fiddlets gut von den Teilnehmern aufgenommen wurde und auf vielfältige Weise Einsatz fand.

Acknowledgements

I'd like to thank Prof. Dr. Jan Borchers and all people at i10 for 7 great years.

I thank my advisor Jan-Peter Krämer for helping me in keeping my focus on the HCI aspects of this thesis instead of writing code.

I thank Kerstin Kreutz for her efforts in proof-reading this thesis and for many hours of playing Frozen Bubble on the Fabarcade.

I thank my wife and family for helped me to stay motivated throughout my whole studies.

Finally, I thank Nintendo for delaying the release of "Star Fox Zero" until I finished this thesis.

Thank you!

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 British English and uses the generic feminine.

Download links are set off in coloured boxes.

File: myFile[a]

[a]http://hci.rwth-aachen.de/public/folder/file_number.file

Chapter 1

Introduction

1.1 Motivation

iOS, Android SDK, Microsoft .net, Ruby on Rails, Angular.js — modern application ecosystems come with a rich set of features already implemented. By leveraging this, developers can produce powerful programs in very short time. However, to take full advantage of these functionalities, developers have to be experienced with the systems frameworks and APIs, in order to not accidentally "reinvent the wheel" by wasting time on building their own implementation of an already existing feature.

Software developers can choose from a large selection of already implemented functionality

With the rising popularity of open source communities such as GitHub, there has grown a plethora of open source libraries and APIs that build up new software ecosystems (Ember.js, Angular.js), provide easier-to-use APIs for existing functionality (AFNetworking, ThoMo Networking), or fill gaps in the existing APIs (underscore.js). As many of them have documentation available as written API documentation, example code and tutorials, it is yet a difficult task for a developer to learn and keep track of the many functionalities offered, especially for fast evolving software ecosystems like the JavaScript community[1]. Besides reading all the documentation and example code, developers

More and more open source frameworks and libraries are being developed.

[1]http://bitworking.org/news/2014/05/zero_framework_manifesto

eventually have to write code in order to get familiar with a new API.

Getting familiar with a new API requires experimentation.

Writing code against an unfamiliar API goes along with a set of short edit-compile-run cycles: the developer will write a small section of code that calls out to the API, compile and run the code and verify whether the result of her code matches her expectation of what the API is supposed to do. If it does not, she will change her API calls, recompile and re-run the program again. As soon as the expected result is met, she will add more API calls, exercising the edit-compile-run loop until the goal which she wanted to achieve with this API is reached.

The following techniques for exploring unfamiliar APIs can often be observed among experienced as well as novel developers:

Developers come up with different techniques for experimentation.

1. *Trial and Error Programming*
   The programmer changes the code of an existing program and tests the desired API directly in the program, using the edit-compile-run cycle as mentioned before. To make this technique efficient, she needs fast compile and execution times. Most of the times, the program itself has to be put into a state, where the new code is executed, e.g. by pressing a button or providing special input.

2. *Example programs (Spikes)*
   Spike-Solutions (`http://www.c2.com/cgi/wiki?SpikeSolution`) are related to "Trial and Error Programming". When writing a "spike" the programmer writes a small piece of software, independent of the program in which the solution will later be applied, which exercises the unfamiliar API. By reducing the executed code to the bare minimum that is necessary to run the API code the compile time is shortened and the need to put the program in a specific state is removed.

3. *Copy and Paste Programming*
   For "Spike solutions" as well as "Trial and Error Programming" developers often use code copied from

tutorials and examples found on the internet. This copied code is then modified in ways to match the behaviour intended by the programmer.

4. *REPLs/Fiddles*
   Many programming languages and development environments such as Node.js, Ruby on Rails or Python offer interactive console applications performing Read-Evaluate-Print-Loops (REPL). In these the programmer interactively writes line after line of code, with the runtime environment executing these lines, giving rapid feedback about the correct use of syntax and API. There are also less interactive REPLs where the programmer has to provide a bunch of code first, which is then executed, such as Eclipse Scrapbooks or JSFiddle.net.

Feedback time is vital for experimentation.

Considering the previously mentioned API exploration and development techniques, it seems that fast feedback time is a vital aspect to efficiently gain insight into the usage of an API or framework. In recent research, there have been interesting approaches of new interactions that build upon these techniques and aim to help developers. One of the more promising interactions is *Live Coding* [Kurz, 2013, Belzmann, 2013, Heinen, 2012]. In Live Coding the program is continuously executed while being edited. Intermediate runtime results (valuations of variables, log output, return values of functions) are shown directly in the editor, providing fast feedback about program changes and behaviour.

Live Coding does not scale for large programs and user interaction.

However, Live Coding requires complete program runs and will only show runtime results for paths of the program, that have been executed in a program run. As an example, if the programmer wanted to see the runtime results of a function that performs financial transactions in a banking program, she needs to manually trigger the transaction in the UI. She also has to provide data that is required by the UI to validate the transaction. Whenever she changes the transaction code and wants to re-evaluate it using live coding, she will have to do the same setup again. Another often cited drawback of Live Coding is performance. Live Coding requires instrumentation of the complete program, thus reducing the program execution speed. Big applications that take up significant amounts of processing power will perform slower in Live Coding environments than they would if a standard debugger was used. If the program performs computations that are time-critical computations, the program might not work properly in a Live Coding environment, thereby negating the positive effects of Live Coding.

Running small parts of the program instead may provide faster and focused feedback.

We believe that a variant of Live Coding that instruments and runs only small portions of a program in a Live Coding environment will solve these issues. In this kind of *local Live Coding*, the environment lets the programmer decide which part of the code should be executed. For this it finds any code that is associated to the part of the program that the programmer is interested in and initialise the data that is required to execute this code. We believe that such a system will provide faster and more focused feedback about the parts of the program that the programmer is interested in.

1.2 Introducing Fiddlets

Fiddlets dynamically creates a context to execute the current line of code, without running the complete program.

This thesis introduces "Fiddlets", a new interaction technique for rapid code exploration and experimentation that allows the programmer to execute small pieces of a program without running the whole program. Fiddlets focuses on executing the current line of code that a programmer is working on by assembling code lines that are related to the current line into an example program. Whenever Fiddlets finds a variable whose value could not be determined statically in the original program, it tries to fill the variable with values from previous program runs. The example is then executed and the result of execution is visualised. Visualisations can be interactive and are allowed to change values in the example code. Offering the possibility to provide different visualisations depending on the code that is executed in the current line, Fiddlets allows the programmer to interact with the current line of code in a way that is unmatched by today's debugging tools.

Contributions of this thesis are the following:

1. A prototype-driven evaluation of on-demand-visualisation of the execution and results of a specific line of code, as well as the interaction with properties of said line of code.

2. A prototypical implementation of Fiddlets that applies the insights gained from the evaluation of the prototypes.

3. A comparative user study that examines the difference in programmers performance between programmers using Fiddlets and programmers using standard debugging tools.

4. An architectural overview of the components necessary for an advanced implementation of Fiddlets.

1.3 Chapter Overview

This chapter provided an overview over the motivation behind this thesis and listed the contributions for the field of HCI.

In Chapter 2 "Related work" we will survey existing research regarding programmer performance and programmer tools. We will locate our ideas in the corpus of existing research and identify the gaps that we intend to fill with the work of this thesis.

Chapter 3 "Design" we will present the design process we used to iteratively work towards an interaction concept that suites our vision of a rapid code exploration and experimentation interaction.

In Chapter 4 "Fiddlets" we will discuss the architectural design of a software that would be able to provide the features of the final prototype presented in Chapter 3 "Design".

To evaluate the our interaction context we conducted a user study. The setup of the user study as well as the implementation and limitations of the software prototype we built for the study will be the subjects of Chapter 5 "Study Design".

Chapter 6 "Evaluation" will take a look at the data we collected with our user study and will evaluate the data with respect to programmer performance and usefulness of the prototype.

Finally, we will wrap up this thesis by summarising the findings and insights made in the previous chapters and by pointing out areas for future work in Chapter 7 "Summary and future work".

Chapter 2

Related work

This chapter provides an overview about research in the fields of programmer performance and programming tools.

Complex programs contain errors that are often hard to find.

Writing programs that perform correctly is a hard task. When the complexity of requirements imposed on a computer system grows, the likelihood of programming errors as well as the variety of said programming errors grow. The abstract nature of program code makes it hard for programmers to create a mapping from the syntactic representation of a program (the source code itself) to the runtime behaviour of the program.

Hanson and Rosinski surveyed programmers about the tools they perceive the most useful.

There have been a lot of tools that aim to help creating this mapping and thus improve the understanding of the dynamic behaviour of a program. Hanson and Rosinski [1985] surveyed 29 COBOL programmers to find out what kind of tools they perceive as useful for performing their daily programming tasks. The tools that were perceived the most useful in terms of increasing productivity were the following: *Interactive debuggers*, *Screen editors*, *Subnetwork checkers* (a tool for checking whether a network has been set up consistently) and *Process (and resource) meters* (tools that instrument source code in order to generate performance metrics about program execution). Except for the sub-network checkers, these tools belong to the default tool chain that programmers use today.

In order to understand how programming tools can support programmer performance, it is vital to know about the questions that programmers try to answer with these tools. By surveying professional programmers about hard-to-answer questions about code, LaToza and Myers [2010b] identified 94 distinct questions that programmers ask themselves during coding tasks. The questions they received covered a broad spectrum of programmer activities. The spectrum covers questions about common programming tasks like implementation, refactoring, testing and debugging, as well as higher level problems like program architecture, contracts or dependencies between software modules, and even questions that do not directly relate to the code itself, like team communication and code history.

LaToza and Myers asked developers for hard-to-answer questions about code.

Out of the most frequently submitted questions, we identified the following to be the most interesting ones in the context of this thesis:

1. Is this code correct? (Testing)
2. How can I test this code or functionality? (Testing)
3. How did this runtime state occur? (Debugging)
4. What does it do in this case? (Intent and Implementation)

In the remainder of this chapter we present previous research that addressed these questions. This will cover enhancements of existing technologies like debuggers, systems that try to improve programmer performance by recommending source code from various sources and continuous feedback tools like Continuous Testing and Live Coding.

2.1 Debuggers

Interactive Debuggers have been around since the 1960's [Evans and Darley, 1965], but despise the addition of user interfaces to show runtime state and enable stepping through the code by pressing buttons, today's debuggers have not evolved very far from their historic precursors. The standard work-flow that programmers are going through when inspecting a bug with an interactive debugger is as follows:

Debuggers have not significantly evolved during the last 50 years.

- Locate the line of code that is suspected to cause the bug.
- Set a break-point for the line or some lines before.
- Start the program and provide input that triggers the break-point.
- Program execution will pause in the line where the break-point was set.
- Inspect the program state and compare it to the expected program state.
- Step through the code and watch where out the program state becomes inconsistent with the expected program state.

Using a debugger requires several distinct steps.

This interaction assumes that the programmer somehow knows or has an idea about the location of the code that causes the bug that she wants to fix. It also implies that she knows about the kind of input that is required to reproduce the bug and to trigger the breakpoint. Another aspect that makes debugging a difficult task is that the programmer has to keep the program state in mind that she expects to find in a correct implementation of her program as well as the lines of code that are expected to create this program state. Adding to this the observation of the current program state and the difference between current and expected program state illustrates the mental efforts that are imposed on programmers during debugging.

Debugging requires knowledge about the location of a bug and requires the developer to restore program state.

2.1.1 Back in Time Debugging

Modern debuggers only allow stepping forward in a program.

A critical aspect in debugging is that even modern debuggers only allow stepping forward in a program. When a programmer accidentally steps over the line of code that she wants to inspect, she must restart the whole debugging process in order to restore the program state that she was interested in before accidentally stepping over the line of code in question.

The Omniscient debugger allows to step back in time.

With the *ODB* (Omniscient Debugger), Lewis [2003] presents a concept for a debugger that allows the programmer to step backwards in program execution. In order to preserve previous program states, ODB creates a complete program trace history over the course of the execution of the program. This program trace is then used to restore previous program states while debugging. This eliminates the need to manually restore program states for debugging purposes as explained previously. Furthermore, ODB allows to inspect the values of a variable over time.

ZStep 94 allows stepping backwards in graphical applications.

Before ODB, *ZStep 94* by Lieberman and Fry [1995] approached reversible debugging for graphics related programs. ZStep 94 also enabled back in time debugging by keeping a history program execution and output. The user interface offered graphic visualisation of program state and connected the line that caused a certain program state to the visualisation of that program state. This allows navigation to a line of code that caused a variable to change by selecting the change of the variable in the code. ZStep 94 also offered *Graphical Step Forward* and *Graphical Step Backward* functionality that only stepped forward and backward between lines of code that are associated with drawing code.

Back-in-Time debugging is expensive.

As back in time debugging needs to track the complete program state over an indefinite execution period, it tends to be very memory consuming and slows down program and debugging performance when used in non-trivial programs. Optimising Runtime and memory space performance of Back in Time Debuggers has been subject of current research [Lienhard et al., 2008] and with the sinking costs of computer memory, Back in Time Debugging might be usable in practice within the coming years [Barr and Marron, 2014].

2.1.2 Reachability

Reachability analysis is crucial for bug fixing.

Other vital aspects in debugging are the localisation of the source of a bug and the question of how the code that is suspect to be the source of the bug can be reached in order to debug it. Both aspects depend on each other — before the programmer can navigate the debugger to code that may contain a bug, she has to know which code is suspect to contain bugs, but to know this, she has to know about the parts of the code that are responsible for a certain program behaviour. LaToza and Myers [2010a] refer to these aspects as *Reachability Question*.

Definition: *Reachability Question*

REACHABILITY QUESTION:
A reachability question is a search across all feasible paths through a program for statements matching search criteria. Reachability questions capture much of how we observed programmers reasoning about causality among behaviours in a program. [LaToza and Myers, 2010a]

Developers ask reachability questions around 9 times a day.

Reachability questions often arise in complex programs, where program behaviour is distributed over several classes and files. By surveying 460 professional software programmers, they found that programmers deal with reachability questions more than 9 times a day. In a follow-up study where they observed 17 programmers in the field they found that in 9 out of 10 longer lasting tasks were associated with reachability questions. They therefore conclude that tools that support programmers in answering reachability questions would improve programmer performance [LaToza and Myers, 2010a].

Automatic debuggers can locate bugs without user interaction.

Automatic debuggers, e.g. Jiang and Su [2007], use statistical analysis and stack trace comparison to locate bugs. The tool by Jiang and Su also locates faulty control flow paths that lead to the bugs that were found. These control paths provide a bigger context for the bug by showing which execution paths are responsible for the faulty behaviour. As the source of a bug does not always correspond with the location where the bug can be observed in the program, this extended context might make fixing the bug easier.

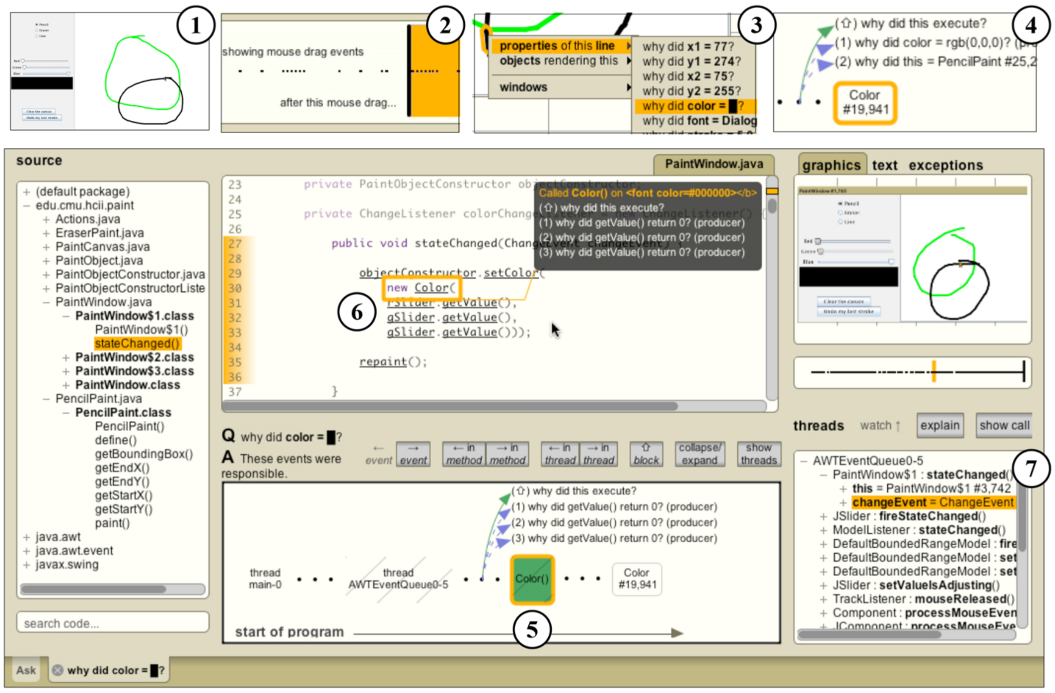
Oscilloscope finds bug-fixes by looking at bugs with related behaviour.

"Oscilloscope" by Gu et al. [2012] uses stack trace analysis and a database of stack traces of known bugs, to suggest bug reports whose stack traces are similar to a program behaviour that the programmer suspects to be faulty. Based on the premise that most of the bugs introduced by programmers are similar, Oscilloscope harvests information of known bugs in order to point the programmer to similar bugs and the solution of those bugs. Gu et al. suggest that the information provided by Oscilloscope will help the programmer fix a bug faster, by offering solutions to similar bugs that she can then employ in her program. However, their approach requires certain application behaviour to already be classified as faulty.

The previously mentioned automatic debugging tools assumed that programmers are only interested in the causes of obvious faulty program behaviour like program crashes or unexpected return values from third party APIs. Often times programmers will classify program behaviour as faulty based on the mismatch of observed and expected program behaviour. An example for such a type of faulty behaviour would be a program that draws UI elements in different colours than expected by the programmer. On another occasion the program behaves correct, but the programmer is still interested in what causes a certain program behaviour, for example in order to get a better understanding on how the program works.

Whyline lets the programmer ask questions about program behaviour.

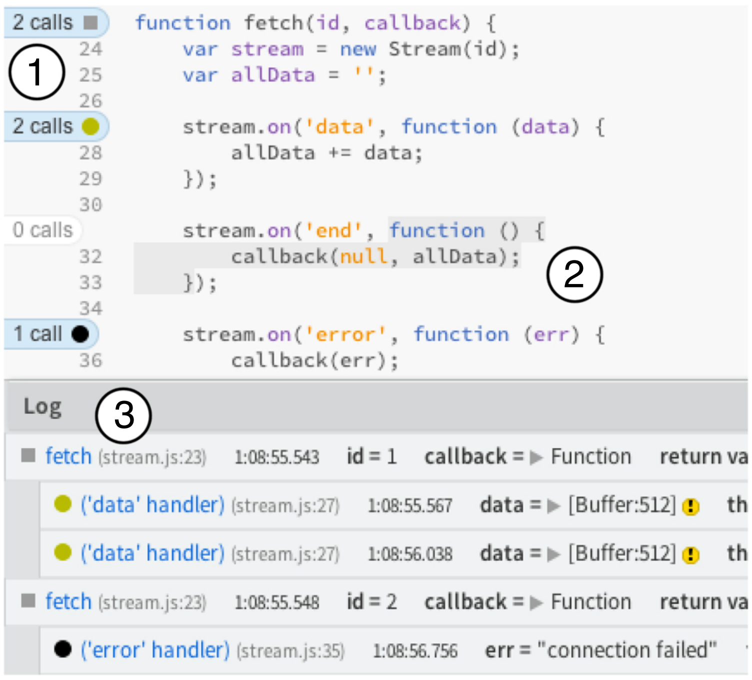
"Whyline" by Ko and Myers [2008] allows the programmer to explore program code and execution by asking *why did* and *why didn't* questions on the program output and behaviour. Whyline lets the programmer demonstrate program behaviour, then set the exact point of time in the behaviour that they want to inquire about. The programmer can then select the part of the program output that they are interested in understanding. Whyline will suggest *why* questions related to the behaviour observed in the selected time frame, by the selected program output (Figure 2.1).



```
        private PaintObjectConstructor objectConstructor

        private ChangeListener colorChangeListener

            public void stateChanged(ChangeEvent e) {

                objectConstructor.setColor(
                    new Color(
                        rSlider.getValue(),
                        gSlider.getValue(),
                        bSlider.getValue()));

                repaint();

            }
```

Figure 2.1: Whyline lets the programmer demonstrate the program behaviour she is interested in (1). She can then select the precise point in time where the behaviour was exercised (2). The programmer can select parts of the output that show the particular behaviour of interest (3). Whyline suggests questions related to the behaviour and shows hints for understanding its causes (4)–(7). [Ko and Myers, 2008]

While they could show that programmers using Whyline could solve bug fixing tasks on a certain type of graphical application, they state that the concept of Whyline requires specific knowledge of the characteristics of the type of output that the program produces in order to suggest helpful *why* questions. Furthermore, questions suggested by Whyline are limited to questions about the implementation of the underlying toolkit that generates the output for the program, e.g. by asking “Why didn’t this JFrame’s repaint() method get called?” instead of asking a more natural and implementation-independent question like “Why didn’t this window change?”[Ko and Myers, 2008].



```
function fetch(id, callback) {
    var stream = new Stream(id);
    var allData = '';

    stream.on('data', function (data) {
        allData += data;
    });

    stream.on('end', function () {
        callback(null, allData);
    });

    stream.on('error', function (err) {
        callback(err);
```

```
Log
■ fetch (stream.js:23)   1:08:55.543   id = 1   callback = ▶ Function   return va
    ● ('data' handler) (stream.js:27)   1:08:55.567   data = ▶ [Buffer:512]   th
    ● ('data' handler) (stream.js:27)   1:08:56.038   data = ▶ [Buffer:512]   th
■ fetch (stream.js:23)   1:08:55.548   id = 2   callback = ▶ Function   return va
    ● ('error' handler) (stream.js:35)   1:08:56.756   err = "connection failed"
```

Figure 2.2: Theseus shows call counts for functions (1). Functions that were not called during execution are marked grey (2). Selecting a call count shows a log of other functions invoked by this function. [Lieber et al., 2014]

Theseus visualises run-time behaviour of programs by displaying function call counts next to functions in the source code.

Theseus (Lieber et al. [2014]) helps the user answering reachability questions by visualising the run-time behaviour of the program inside the editor (Figure 2.2). When a function was called during program execution, Theseus shows the number of calls that the function received next to the function in the editor. Functions that were not called during program execution are marked with a grey background. This visualisation helps the user to understand what parts of the program are involved in certain interactions with the program. By clicking on a function call count next to a function, Theseus provides an overview of all functions involved in an interaction, ordered by the time they were called and nested in a way that relates to who called them.

2.2 Source Code Generation and Code Snippets

When a programmer is unfamiliar with a programming API, her performance in programming against this API will be fairly low compared to her performance when programming against a well known API. The reasons for this are that she needs to look up the documentation of the API very often until she internalises the API classes and functions that relate to the functionality she wants to achieve in her program. Wrong usage of the API as well as wrong assumptions about the internal model of the API will add to this. Finding the proper functions and classes that perform a certain functionality of the API is another important task in using an unfamiliar API that requires the programmer to not only read the documentation carefully, but also to read tutorials and example programs that show how the API is supposed to be utilised. Due to the popularity of community driven knowledge markets like Stack Overflow and open source repository services like GitHub, the internet now offers a huge selection of code examples. It is however still up to the programmer to search through all the examples found on the internet and to adjust the solutions she found to her own program [Brandt et al., 2009]. This section presents two approaches to overcome performance issues caused by unfamiliarity with APIs: *automatic source code generation* and *code snippet libraries*.

Programmers are faster when they internalised an API.

The open source movement provides a huge selection of example code.

2.2.1 Automatic Source Code Generation

CodeHint by Galenson et al. [2014] synthesises source code based on the current execution context of the scope in which the synthesised code is supposed to be executed and a specification that is given by the programmer. An illustrating example on how CodeHint is supposed to help the programmer is the following piece of source code that is called whenever the user of the program presses the mouse button:

CodeHint synthesises program code for the developer.

```
final JComponent tree = makeTree();
tree.addMouseListener(new MouseAdapter(){
```

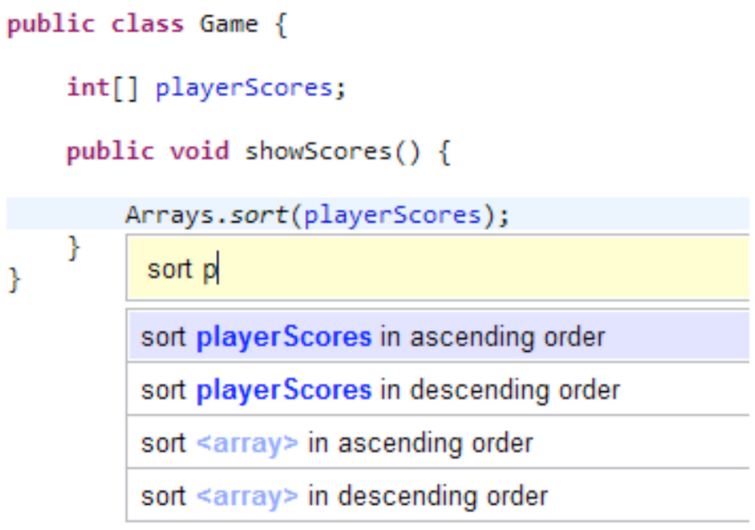
```
3  public void mousePressed(MouseEvent e){
4   int x = e.getX();
5     int y = e.getY();
6     Object o = null;
7     // Get the menu bar
8  }
9 });
```

Listing 2.1: The programmer wants the menu bar of the clicked element to be stored in the variable `o`. [Galenson et al., 2014]

In this example the programmer is interested in getting a reference to the menu bar of the window that contains the clicked element. She know that menu bars are represented as objects of the class `JMenuBar`. She also knows that she wants to store the menu bar reference in the variable `o` (Listing 2.1, line 6). To let CodeHint generate the necessary code, she set a breakpoint after the line 6 and runs the program until the breakpoint is hit. She then provides a specification, for example that as the result of the generated code the variable o should reference an object of class `JMenuBar`. For CodeHint, this specification would look the following: "`o′ instanceof JMenuBar`", where `o′` represents the state of `o` after the execution of the generated code. Using this specification and the run-time information that is provided by the debugger at the breakpoint, CodeHint will search for functions and method calls that are applicable to the objects that are available at this point of execution and executes those in the current context. CodeHint will apply new operations to the objects in context until the specification is fulfilled. When the search is complete, CodeHint will present the user up to five search results that she can use in her code.

2.2.2 Code Snippet Libraries

Code snippets are small programs or pieces of programs that demonstrate the use of programming APIs or demonstrate simple algorithms. Code snippets are often times included in the documentation of an API or can be found on the internet.



```
public class Game {

    int[] playerScores;

    public void showScores() {

        Arrays.sort(playerScores);
    }
}
```

sort p
sort playerScores in ascending order
sort playerScores in descending order
sort <array> in ascending order
sort <array> in descending order

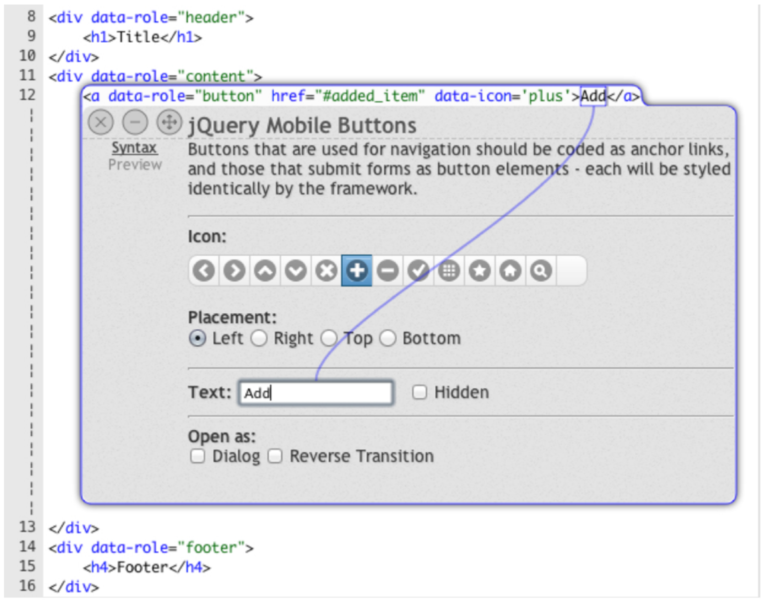
Figure 2.3: SnipMatch uses variables of the context inside the snippet. [Wightman et al., 2012]

Holmes and Murphy use structural context information to recommend code snippets.

In their paper *Using Structural Context to Recommend Source Code Examples* Holmes and Murphy [2005] present a source code recommendation tool that uses the structural context of the code under development to find relevant code snippets from an example repository. The example repository is automatically built from the source code of the framework that is used by the programmer. Since building the example repository for a framework works without requiring previously existing example code or documentation on the framework, this approach aims to help understand programmers in understanding frameworks with little to no documentation or example programs.

SnipMatch offers a markup language to create customisable snippets.

SnipMatch ([Wightman et al., 2012]) introduces a markup language that allows authors of snippet code to build customisable code snippets and an IDE plugin that allows the programmer to search for snippets and tries to automatically integrate them into the code under development. For example, if the programmer wanted to sort an array named `playerScores`, she presses the keyboard shortcut to open the search window and start to type “sort playerScores” (Figure 2.3). SnipMatch will find snippets that match this query and will automatically replace the placeholder that



```
<div data-role="header">
    <h1>Title</h1>
</div>
<div data-role="content">
    <a data-role="button" href="#added_item" data-icon='plus'>Add</a>
```

jQuery Mobile Buttons

Syntax
Preview

Buttons that are used for navigation should be coded as anchor links, and those that submit forms as button elements - each will be styled identically by the framework.

Icon:

Placement:
[✓ Left] Right Top Bottom

Text: Add ☐ Hidden

Open as:
☐ Dialog ☐ Reverse Transition

```
</div>
<div data-role="footer">
    <h4>Footer</h4>
</div>
```

Figure 2.4: Codelets presents code snippets with an interactive helper widget that helps to understand and configure the snippet. Since interactive widgets are associated with the snippet code, programmers can close widgets and bring them up later to configure the snippet. [Oney and Brandt, 2012]

the snippet author placed in the snippet where the name of the array to be sorted would be.

Codelets displays interactive widgets along with code snippets.

Codelets by Oney and Brandt [2012] also allows the programmer to search through a database of pre-defined code snippets using textual queries. Codelets allows snippet providers to create snippets as interactive helper widgets that help the programmer to understand and integrate the snippet into her code (Figure 2.4). The IDE treats Codelet snippets as *first-class objects*, meaning that after closing the interactive editor, it will still be associated with the snippet code. Whenever the programmer wants to change a property of the snippet, she can do so by either changing the snippet code directly or by opening the interactive editor of the snippet. Oney and Brandt could show that programmers using Codelets were able to solve tasks that involved examples faster than programmers who used standard web browsers.

2.3 Live Coding

Live Coding is the concept of development environments where program code is continuously executed and the output of the program execution is permanently presented to the programmer. Live Coding provides a continuous stream of feedback to the programmer, enabling her to constantly monitor the execution state of the program over time. This section presents a short overview of past and current research in the field of Live Coding.

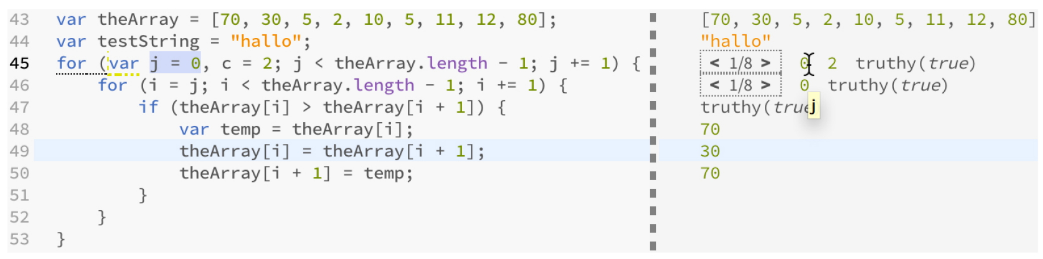
Live Coding continuously executes the program and shows results of execution in the editor.

One of the earliest mentions of Live Coding is the *VisiProg* environment by Henderson and Weiser [1985]. The concept of VisiProg envisioned a programming environment where programs were composed from functional units that can be linked together as a network. Each functional unit would show its internal state as well as the state of its interface (input and output channels that can be connected to other functional units) to the user. When the user changes the state of the input of a functional unit, she will see the changes in the states of other functional units that are connected to this functional unit. Henderson and Weiser compare the conceptual model of VisiProg to the model of analog networks, e.g. electronic circuits. They claim that VisiProg would be "ideally suited for exploratory code development (sometimes called 'prototyping')"[Henderson and Weiser, 1985], but as computers back then did not have the processing power that is needed for an implementation of Live Coding, they could not validate their claims.

VisiProg envisions Live Coding for flow based programming.

As computers became more powerful, implementations of visual programming languages, as envisioned with VisiProg, became practical to be used for everyday programming tasks. In 1997 Wilcox et al. conducted a study on the effects of continuous feedback for visual programming languages. They could not show significant advantages in terms of debugging time for their Live Coding condition overall. Instead, they could show a significant effect on accuracy for low performing participants. This suggests

Wilcox et al. investigated the effect of Live Coding on programmer performance for visual programming.



```
43  var theArray = [70, 30, 5, 2, 10, 5, 11, 12, 80];        [70, 30, 5, 2, 10, 5, 11, 12, 80]
44  var testString = "hallo";                                  "hallo"
45  for (var j = 0, c = 2; j < theArray.length - 1; j += 1) {  < 1/8 >  0  2  truthy(true)
46      for (i = j; i < theArray.length - 1; i += 1) {         < 1/8 >  0  truthy(true)
47          if (theArray[i] > theArray[i + 1]) {               truthy(true
48              var temp = theArray[i];                        70
49              theArray[i] = theArray[i + 1];                 30
50              theArray[i + 1] = temp;                        70
51          }
52      }
53  }
```

Figure 2.5: The Live Coding editor by Kurz [2013] continuously executes the program code and displays the values of variables next to the code. The sliders in line 45 and 46 allow the programmer to browse through iterations of the for-loops. Hovering over a value displays the name of the corresponding variable in a tool-tip.

that although high performing participants could not benefit from the continuous feedback provided by the system, continuous feedback could be a useful tool for the low performing users, e.g. novice programmers.

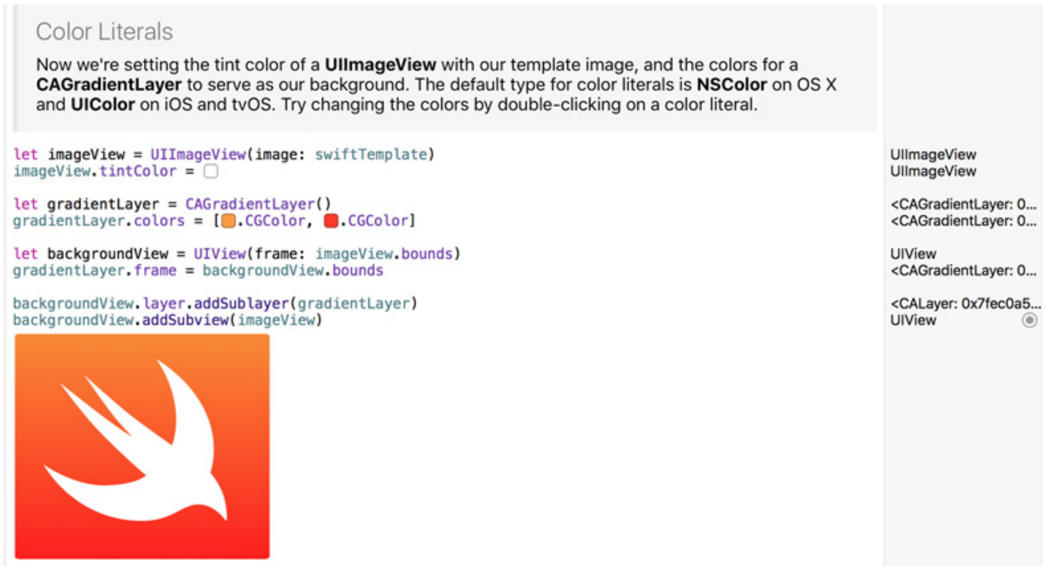
Kurz built a robust Live Coding environment for JavaScript.

Kurz [2013] built a robust Live Coding environment for JavaScript programmers that is based on the open source code editor Brackets by Adobe[1]. The Live Coding environment continuously executes the program that is being written and displays intermediate results next to the code in a separate pane (Figure 2.5). In a user study with 13 participants, all except one having more than 4 years of programming experience, he could not find significant evidence for improved programming speed or correctness when using the Live Coding environment compared to using traditional development tools.

Xcode Playgrounds is a Live Coding environment developed by Apple.

In 2014 Apple introduced *Playgrounds*, a Live Coding environment for the *Xcode*[2] IDE. Similar to the Live Coding editor by Kurz [2013], Playgrounds executes the program continuously and display the results of each line of code in a separate pane next to the code (Figure 2.6). Besides showing variable values, Playgrounds has knowledge about some special classes of Apple's GUI programming framework. This allows the IDE to render the contents of images directly inside the Playground or to display colours using the real colour instead of showing colour codes. It is

[1]http://brackets.io
[2]https://developer.apple.com/xcode/



Color Literals

Now we're setting the tint color of a **UIImageView** with our template image, and the colors for a **CAGradientLayer** to serve as our background. The default type for color literals is **NSColor** on OS X and **UIColor** on iOS and tvOS. Try changing the colors by double-clicking on a color literal.

```
let imageView = UIImageView(image: swiftTemplate)
imageView.tintColor = □

let gradientLayer = CAGradientLayer()
gradientLayer.colors = [🟧.CGColor, 🟥.CGColor]

let backgroundView = UIView(frame: imageView.bounds)
gradientLayer.frame = backgroundView.bounds

backgroundView.layer.addSublayer(gradientLayer)
backgroundView.addSubview(imageView)
```

```
UIImageView
UIImageView

<CAGradientLayer: 0...
<CAGradientLayer: 0...

UIView
<CAGradientLayer: 0...

<CALayer: 0x7fec0a5...
UIView
```

Figure 2.6: Xcode Playgrounds show the result of the evaluation of a line in a pane on the right side of the code. Playgrounds can displays image data by rendering the image into the editor and documentation with rich text formats.

also possible to add documentation for the code in rich text format. Playgrounds are intended to be used as a platform to experiment with code, to learn about programming concepts and languages and to provide interactive documentation.

Live Coding has always been criticised for being resource consuming and leading to bad programming habits ([Henderson and Weiser, 1985, Tanimoto, 2013]). With modern computers, performance problems of Live Coding seem to have vanished for smaller programs. This leaves Live Coding as a valuable tool for educational purposes and for documentation. The problem of bad programming habits is not limited to Live Coding as a practice and has to be addressed outside of the scope of the code editor.

Live Coding can be a valuable tool for learning and documentation.

Although continuous feedback in Live Coding environments could not show significant improvement of programmer performance, this does not mean that continuous feedback has no impact on programmer performance. By utilising excess CPU cycles on a programmers computer to continuously execute the unit tests of the software under

Continuous testing: another source for continuous feedback.

development (Continuous Testing), Saff and Ernst [2003] could show that programmers wasted 92–98% less development time compared to when they had to run the tests manually. However, in order to work properly Continuous Testing requires that a suitable collection of unit tests has already been implemented. Furthermore, the quality of feedback offered by Continuous Testing largely depends on the quality and expressiveness of the unit tests.

2.4 Summary

Debugging and Live Coding build the two ends of a spectrum of programming tools.

Traditional debugging tools and Live Coding build the two sides of a wide spectrum of feedback systems for computer programs. On one side of this spectrum we find debugging as we know it today, which provides a focused feedback about a specific point in time in the execution of a program. The feedback however is not continuous, since changes in the program require the programmer to restart the debugging progress in order to restore the debugging context to further observe the effects of the program change. On the other side of the spectrum we find Live Coding, where program execution is continuous and where feedback about the program state is constantly reported to the programmer. It is the task of the Live Coding environment to keep the stream of information, that is produced by programs under Live Coding condition, comprehensible for the programmer.

Giving too much feedback can reduce programmer performance.

It is noted by Wilcox et al. [1997] and Saff and Ernst [2003] that providing too much information about program execution might as well have a negative influence on programmers performance. We believe that a tool that provides focused runtime information on demand, yet does not depend on complete program runs and the manual setup of a runtime context, will solve many of the problems that Live Coding is suffering from. In the next chapter we present our prototype driven approach to develop such a tool.

Chapter 3

Design

The goal of this thesis is to develop a tool for programmers that could provide fast feedback about program execution, but to limit the scope of this feedback to a comprehensible amount. We want the tool to avoid the information overflow we identified in Live Coding tools as discussed in 2.3 "Live Coding". At the same time we want the tool to provide enough information to the programmer that she understands what is currently happening in the code she is writing.

We want to provide fast feedback, but avoid information overflow.

We went through three design iterations in which we built interactive and non-interactive prototypes of programming tools that would fit our vision. We evaluated each of the prototypes with programmers and incorporated the feedback we received on the prototype into the development of the next prototype. We ended up with a paper prototype for a programming tool that seems to fulfil our requirements.

We used the DIA-Cycle and built prototypes in three iterations.

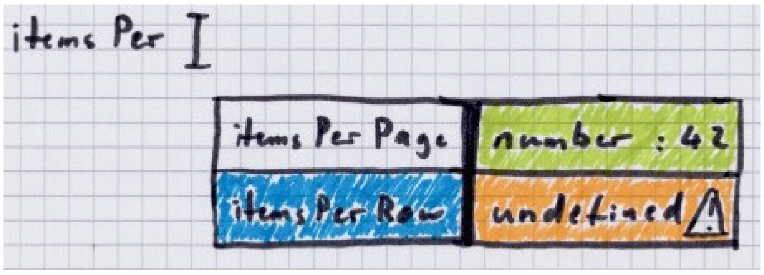


Figure 3.1: Paper prototype of a Life Autocomplete interaction

3.1 First Iteration: Live Autocomplete

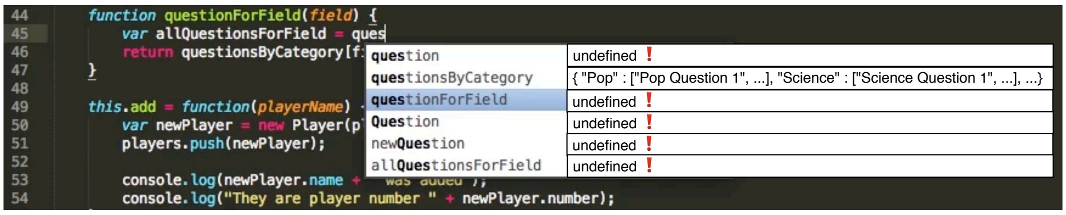
3.1.1 Idea

Live Autocomplete would provide a live preview of the execution of suggested code.

Our first idea was to design a tool that integrates into the autocomplete system of the editor and shows the programmer a live preview of the behaviour that the code suggested by the autocomplete system would achieve. We called this idea *Live Autocomplete*. Every time that the editors autocomplete function is invoked, Life Autocomplete executes the program, but without the line that the programmer is currently working on. Life Autocomplete saves the runtime state for that line and then uses this runtime state to execute the different suggestions made by the autocomplete function against this state. The results of these executions are shown next to the respective autocomplete suggestion that caused them.

Figure 3.1 shows a scenario of how Life Autocomplete is used. The programmer wants to update the `itemsPerRow` variable. She types `itemsPer` and the autocomplete function offers to complete it to `itemsPerPage` and `itemsPerRow`. The Live Autocomplete shows her that the `itemsPerPage` variable currently holds a number object with the value 42. The `itemsPerRow` variable is currently undefined.

A tool like this would allow the programmer to observe the changes in program behaviour that the intended change of



```
function questionForField(field) {
    var allQuestionsForField = ques
    return questionsByCategory[f:
}

this.add = function(playerName) ·
    var newPlayer = new Player(p
    players.push(newPlayer);

    console.log(newPlayer.name +
    console.log("They are player number " + newPlayer.number);
```

question	undefined !
questionsByCategory	{ "Pop" : ["Pop Question 1", ...], "Science" : ["Science Question 1", ...], ...}
questionForField	undefined !
Question	undefined !
newQuestion	undefined !
allQuestionsForField	undefined !

Figure 3.2: In most of the cases executing the autocomplete suggestions did not yield useful results.

the program code would produce. Showing multiple autocomplete suggestions together with their effects on the program behaviour, the programmer could choose the suggestion that fits her intentions the best.

3.1.2 Study: Trivia Implementation

To evaluate the concept of a Live Autocomplete tool, we conducted a study with 3 programmers. We asked them to implement a small JavaScript program according to a given specification (see Appendix A "Live Autocompletion Evaluation: Implementation Requirements". The programmers were asked to capture the progress of implementing the program using screen capture. From the screen capture we selected 9-12 points in their programming progress where we suspected the Live Autocomplete tool to be useful. We then tried to come up with values that the tool could possibly provide. Autocomplete suggestions were taken directly from the editors used by the programmers. Since the programmers used different code editors, as well as different autocomplete settings, not all autocomplete suggestions are considered useful in the selected situations. We still considered those suggestions, since we wanted to see how Live Autocomplete would perform in those cases.

We made an implementation study with three developers...

... and tried to come up with Live Autocomplete previews for various scenarios.

3.1.3 Evaluation

Although we found out that most of the suggestions made by the autocomplete function of the editor did not make sense in the context of the line of code where autocomplete

Most autocomplete suggestions did not make sense.

was invoked (and would even break program execution), this is clearly an issue of the autocomplete systems used by the programmers. Another cause of this issue is the dynamic typing system of JavaScript. We decided that in cases where the suggestions did not make sense in the context of the program, i.e. by leading to runtime exceptions in the program, we decided that the Live Autocomplete should still execute these suggestions and show the error message next to the suggestion. We based this decision on the observation that we often can not programmatically distinguish between errors that are caused by inept autocomplete suggestions and those that are caused by the programmer.

Live Autocomplete would need to provide suggestions for function parameters.

The most apparent issue we found with the Live Autocomplete tool is the need to provide suggestions for parameters if the autocomplete suggestion contains a function. If the autocomplete function suggests a function that has a number of parameters, the Live Autocomplete tool would need to fill in parameters for this function in order to display useful results. We suggest two ways to resolve this issue.

It could substitute parameters with values from the runtime.

In the first solution the Live Autocomplete tool would try to provide different suggestions for the parameters of the function. These parameter suggestions would be drawn from the runtime information that is available for the current line of code. By using type analysis on the runtime information, the Live Autocomplete tool would reduce the suggestions to only contain those parameters that make sense for the function.

The number of parameter suggestions would grow exponentially with the number of parameters.

This solution has several drawbacks. One is that the Live Autocomplete tool will create a list of suggestions for every function which will grow exponentially depending on the number of parameters of the function. The programmer would need to search this list for the most useful suggestion, which will slow her down. Even if the programmer decided to ignore the live preview of the autocomplete suggestions and use the autocomplete function like she would normally do, having more than one entry for each suggestion would render the benefit of the default autocomplete function useless. Another drawback of this approach is that it requires the number of parameters of a function to be known. Programming languages like C or Java allow the programmer to create functions with a variable number of parameters. JavaScript takes this a step further and treats every function as a function with a variable number of parameters[1]. In this case the Live Autocomplete tool would create an endless list of parameter suggestions.

This gets worse with variable numbers of parameters.

The second solution to the parameter suggestion problem is to let programmers themselves enter the parameters for a suggested function and defer the execution of the suggestion until after the parameters have been set. The Live Autocomplete tool would then no longer provide autocomplete functionality, but rather a visualisation and configuration tool for functions.

Leave it to the programmer to provide parameters.

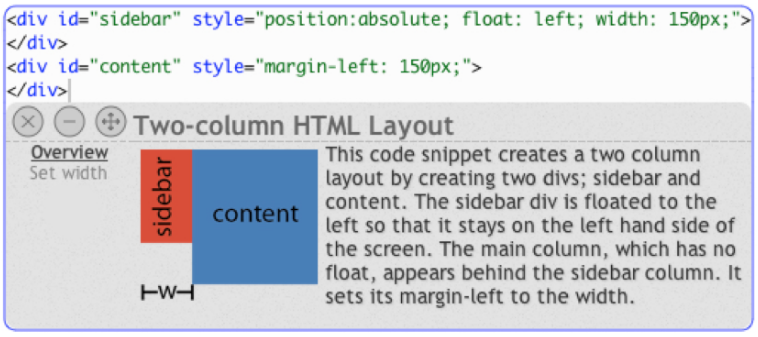
Our considerations on the first solution to the parameter suggestion problem convinced us to drop the idea of a Live Autocomplete tool and instead focus on on-demand visualisation of the runtime state for the current line that the programmer is working on.

3.2 Second Iteration: Runtime Data Widgets

In 2.2.2 "Code Snippet Libraries" we discussed Codelets [Oney and Brandt, 2012], which allows the programmer to search and insert code snippets into her code. Codelets provide interactive widgets to configure the parameters of the code snippet and explain how these parameters affect the properties of the snippet. Codelets focused on code snippets for UI elements, but the widgets for configuring the snippets were static. For example, the widget for a snippet that creates a column layout with HTML and CSS allows the programmer to change properties of the layout (Figure 3.3). But instead of also showing a preview of the resulting layout it just shows an icon of a column layout and a textual documentation. Codelets also offers no snippets for standard programming tasks, like iterating over the elements of an array or sorting an array. Overall, they did not

Codelets widgets are interactive, but static.

[1]http://www.w3schools.com/js/js_function_parameters.asp



```
<div id="sidebar" style="position:absolute; float: left; width: 150px;">
</div>
<div id="content" style="margin-left: 150px;">
</div>
```

Two-column HTML Layout

This code snippet creates a two column layout by creating two divs; sidebar and content. The sidebar div is floated to the left so that it stays on the left hand side of the screen. The main column, which has no float, appears behind the sidebar column. It sets its margin-left to the width.

Figure 3.3: Codelets two-column HTML layout widget lets the programmer configure the width of the sidebar, but does not show a preview of the layout [Oney and Brandt, 2012].

include any snippets for working on data and objects in a program.

3.2.1 Idea

Interactive configuration widgets for functions.

The lack of interactive visualisation for both UI and data related code in Codelets inspired us to focus more on the runtime result visualisation we used in the first prototype. With the second prototype, we aimed to build prototypes of interactive configuration widgets for JavaScript functions. We built three prototypes, using HTML and JavaScript, that demonstrate different use cases for interactive configuration widgets:

- jQuery Selector Widget
- jQuery Insert Widget
- Function Chain Widget

JQUERY:
jQuery (`https://jquery.com/`) is an open source JavaScript library that aims to simplify the interaction with HTML in client-side web applications. jQuery offers functions to select DOM elements, create animations, handle events and many more.

Definition: *jQuery*

We designed all widgets as inline widgets. Inline widgets are activated with a shortcut and are displayed below the line in which the text cursor is placed.

jQuery Selector Widget

jQuery selectors[2] allow the programmer to select various elements of an HTML document by their name or properties or even their relation to other elements. In our *jQuery Selector Widget* (Figure 3.4) we show the programmer a preview of the rendered HTML document (1). When the programmer starts typing a selector, that selector is applied to the HTML document and the matched elements are highlighted in green (2). Additionally, we show a list of the matched elements beneath the preview, which helps when the programmer wants to match elements that are not rendered visible in the document.

jQuery Selector Widget shows the result of a selector by highlighting the matched elements in a preview of the program.

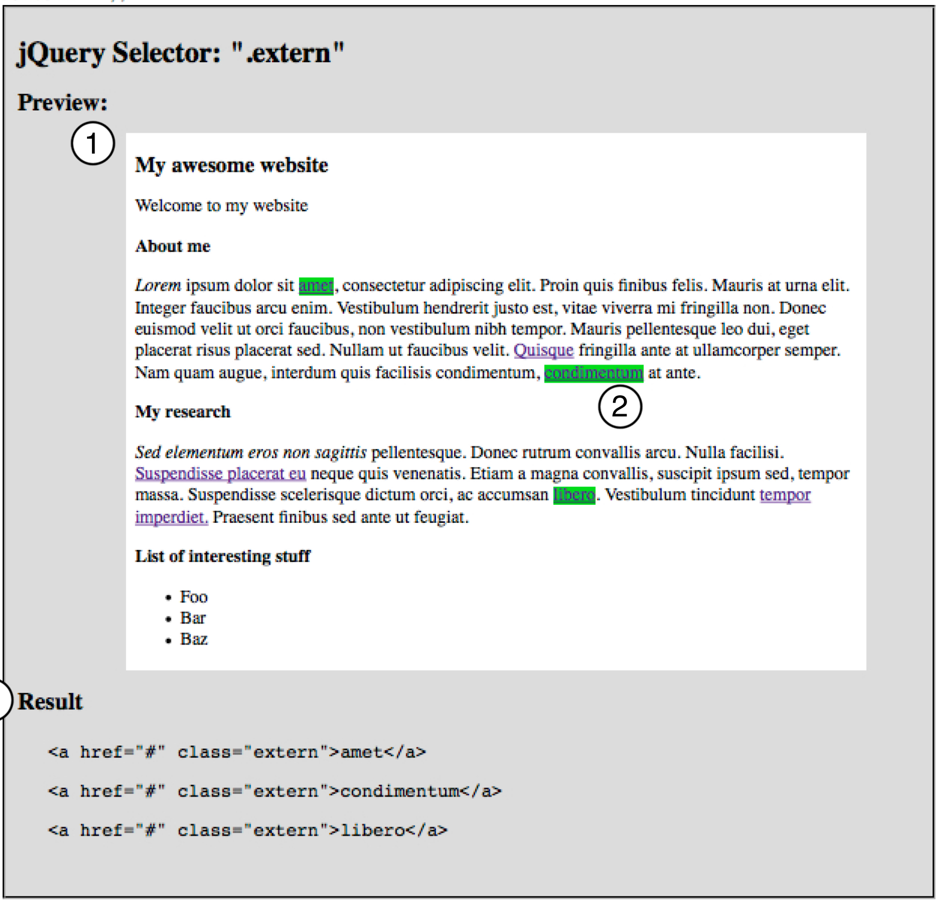
jQuery Insert Widget

jQuery offers an API to manipulate the DOM tree of an HTML document by adding elements[3], removing elements or changing the properties of elements. The *jQuery Insert Widget* shows the results of jQuery's `append` function. The `append` function is called on a jQuery element and appends the jQuery elements that are passed as parameters to the end of the element on which the function was called. `append` can have a variable number of parameters.

jQuery Insert Widget shows the effect of the `append` function by rendering a preview.

[2]https://api.jquery.com/category/selectors/
[3]https://api.jquery.com/category/manipulation/

```
<ul>
        <li>Foo</li>
        <li>Bar</li>
        <li>Baz</li>
</ul>

<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.
<script>
        $(".extern  ");
```



jQuery Selector: ".extern"

Preview:

My awesome website

Welcome to my website

About me

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin quis finibus felis. Mauris at urna elit. Integer faucibus arcu enim. Vestibulum hendrerit justo est, vitae viverra mi fringilla non. Donec euismod velit ut orci faucibus, non vestibulum nibh tempor. Mauris pellentesque leo dui, eget placerat risus placerat sed. Nullam ut faucibus velit. Quisque fringilla ante at ullamcorper semper. Nam quam augue, interdum quis facilisis condimentum, condimentum at ante.

My research

Sed elementum eros non sagittis pellentesque. Donec rutrum convallis arcu. Nulla facilisi. Suspendisse placerat eu neque quis venenatis. Etiam a magna convallis, suscipit ipsum sed, tempor massa. Suspendisse scelerisque dictum orci, ac accumsan libero. Vestibulum tincidunt tempor imperdiet. Praesent finibus sed ante ut feugiat.

List of interesting stuff

- Foo
- Bar
- Baz

Result

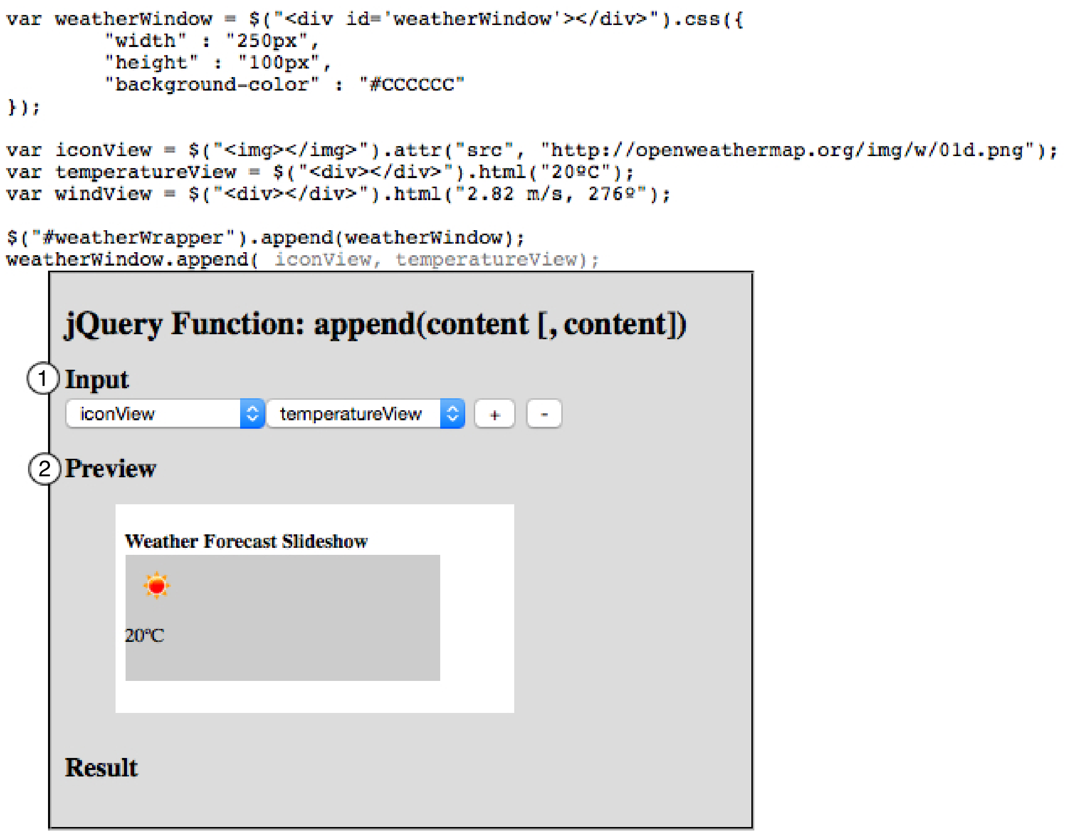
```
<a href="#" class="extern">amet</a>

<a href="#" class="extern">condimentum</a>

<a href="#" class="extern">libero</a>
```

```
</script>
```

Figure 3.4: *jQuery Selector Widget* with live preview

An illustrating example.

Figure 3.5 shows an example on how the widget would be used. The input section (1) allows the programmer to select the parameters of the function call. By pressing the plus button she can add more parameters to the function. Pressing the minus button removes the last parameter. Parameters are shown as dropdown lists. The lists contain names of variables that contain elements suitable to be parameters. In the example, the programmer selected two parameters: the variable `iconView`, which holds a jQuery el-



```
var weatherWindow = $("<div id='weatherWindow'></div>").css({
        "width" : "250px",
        "height" : "100px",
        "background-color" : "#CCCCCC"
});

var iconView = $("<img></img>").attr("src", "http://openweathermap.org/img/w/01d.png");
var temperatureView = $("<div></div>").html("20ºC");
var windView = $("<div></div>").html("2.82 m/s, 276º");

$("#weatherWrapper").append(weatherWindow);
weatherWindow.append( iconView, temperatureView);
```

jQuery Function: append(content [, content])
1 Input
iconView
temperatureView
+
-
2 Preview
Weather Forecast Slideshow
20°C
Result

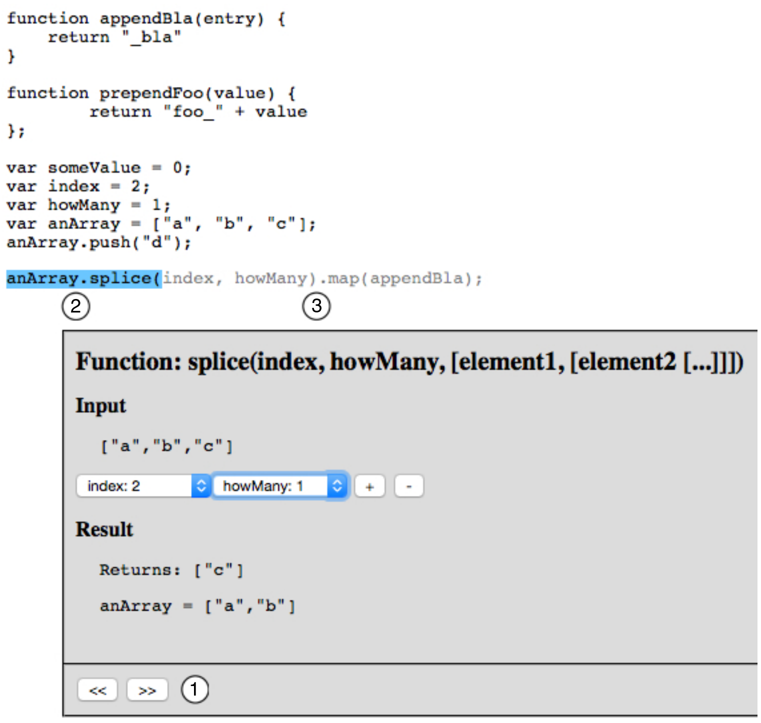
Figure 3.5: Demo of the *jQuery Insert Widget*

ement that references an image element, and the variable `temperatureView`, which holds a jQuery element that references a div element. The preview section (2) shows the state of the HTML document after the function has been executed. In the example we see that the contents of `iconView` and `temperatureView` are appended to the `weatherWindow` container.

Function Chain Widget

Programmers often use the return value of a function to directly call another function on it or use it as a parameter for a different function. jQuery encourages this style of programming by returning the object on which a function has been called as return value of that function. This al-

The chaining of function calls is a common pattern in software.



```
function appendBla(entry) {
    return "_bla"
}

function prependFoo(value) {
        return "foo_" + value
};

var someValue = 0;
var index = 2;
var howMany = 1;
var anArray = ["a", "b", "c"];
anArray.push("d");

anArray.splice(index, howMany).map(appendBla);
```

Function: splice(index, howMany, [element1, [element2 [...]]])

Input

```
["a","b","c"]
```

index: 2 howMany: 1

Result

```
Returns: ["c"]

anArray = ["a","b"]
```

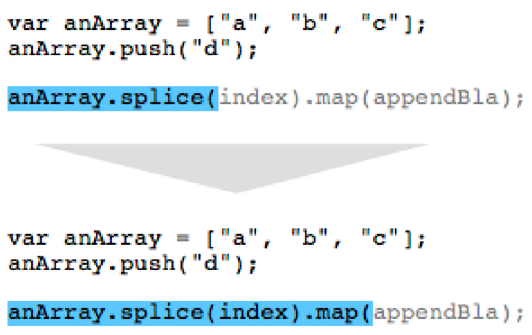
Figure 3.6: The *Function Chain Widget* demonstrates the selection of a specific function within a chain and a visualisation for functions called on `Array` objects.

lows programmers to build arbitrary long function chains. For example, the following function chain finds an element with the id “title”, adds the class “myTitle” to the element and sets the string “Hello world” as the elements text:

```
$('#title').addClass('myTitle')
           .text('Hello world');
```

Since this line covers multiple function calls, it is not obvious which function should be visualised in the widget. It is also not obvious how the programmer decides which function should be shown in the widget. We created an interaction for selecting a function in a function chain and implemented it in the *Function Chain Widget* (Figure 3.6).

The widget demonstrates a situation in which the programmer activated a widget within a chain of function calls.



```
var anArray = ["a", "b", "c"];
anArray.push("d");

anArray.splice(index).map(appendBla);
```

```
var anArray = ["a", "b", "c"];
anArray.push("d");

anArray.splice(index).map(appendBla);
```

Figure 3.7: Moving the focus of the *Function Chain Widget*

When a widget is activated, it will by default display the function on which the text cursor is positioned. The widget displays two buttons that are used to navigate through the function chain (1). The functions that are currently active for the widget are displayed with a blue background (2). The part of the function chain that is not executed is displayed in grey (3).

Function Chain Widget demonstrates the use of a widget visualisation for a function call chain.

In figure 3.7 the programmer uses the navigation buttons to select the next function (`map`) in the function chain. The widget will now display the visualisation of the results of the chained `splice` and `map` function calls. Since the widget now focuses on the configuration of the `map` call, the parameters of the `splice` call are fixed.

3.2.2 Evaluation

We showed the prototypes to programmers and asked them for feedback. Programmers generally liked the idea to see the results of function evaluations inside the editor. They especially liked the jQuery selector widget, since incorrectly defined selectors are a frequent source of errors when programming with jQuery. A selector that does not match any object in the DOM will return an empty jQuery object instead of `null` or `undefined`. Further operations on this object will not fail by throwing an exception, but will instead just do nothing, which regularly poses a hard to find error. Showing the elements matched by a selec-

We evaluated the prototypes by showing them to developers.

tor both in the preview and the result section of the widget helps the programmers understand whether their selector matches the proper objects.

Navigating through the function chain was not intuitive.

Programmers found the navigation and visualisation of the currently active function in the function chain widget confusing. They were not sure whether the inactive parts of the function chain were already present in the source code or whether they were in a suggestive state as they know from autocomplete functions. They further found the navigation buttons inefficient for quick navigation on the functions of the function chain. They would prefer to use keyboard shortcuts to switch between functions or just reposition the text cursor onto the function they are interested in.

Selecting values from a list is not beneficial.

Overall we found out that the list selection, that is used to configure the parameters of the function, does not provide any benefits over writing the parameters directly in the code. The prototype would also still require complete runs of the program in order to generate the data that is shown in the visualisation. It would therefore run into the same problems we already identified for Live Coding and debugging as described in chapter 2 "Related work".

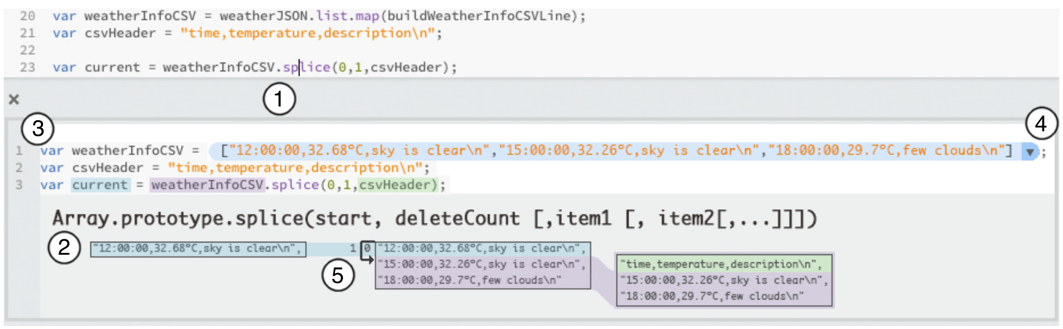
3.3 Third Iteration: Adding Context

Programmers appreciated our efforts.

We found out with the prototype for the second iteration that programmers generally appreciate the inline visualisation widgets we came up with. We kept the idea of these inline visualisations for the third iteration and thought about other functions we could visualise. In order to set our work apart from the work that had already been done in Codelets [Oney and Brandt, 2012] we focused on providing visualisations for data driven functions, i.e. functions that operate on JavaScript arrays.

The second prototype still does not improve Live Coding performance.

With the third iteration we additionally tried to find a solution to the Live Coding problems we introduced in our previous prototype. Our goal was to find ways to perform continuous execution as in Live Coding, but to limit the range



```
var weatherInfoCSV = weatherJSON.list.map(buildWeatherInfoCSVLine);
var csvHeader = "time,temperature,description\n";

var current = weatherInfoCSV.splice(0,1,csvHeader);
```

```
var weatherInfoCSV = ["12:00:00,32.68°C,sky is clear\n","15:00:00,32.26°C,sky is clear\n","18:00:00,29.7°C,few clouds\n"] ▼;
var csvHeader = "time,temperature,description\n";
var current = weatherInfoCSV.splice(0,1,csvHeader);
```

```
Array.prototype.splice(start, deleteCount [,item1 [, item2[,...]]])
"12:00:00,32.68°C,sky is clear\n",        1 0 "12:00:00,32.68°C,sky is clear\n",
                                              "15:00:00,32.26°C,sky is clear\n",
                                              "18:00:00,29.7°C,few clouds\n"
                                                        "time,temperature,description\n",
                                                        "15:00:00,32.26°C,sky is clear\n",
                                                        "18:00:00,29.7°C,few clouds\n"
```

Figure 3.8: The third prototype resembles an actual inline widget in Adobe Brackets

of program code that would be executed to an amount that does not cover the complete program, yet still allows for a realistic feedback on the outcome of the program operations that the programmer is interested in. Solving this trade-off led us to the introduction of a radical new Live Coding concept that manifested itself as a "context editor" within the prototype.

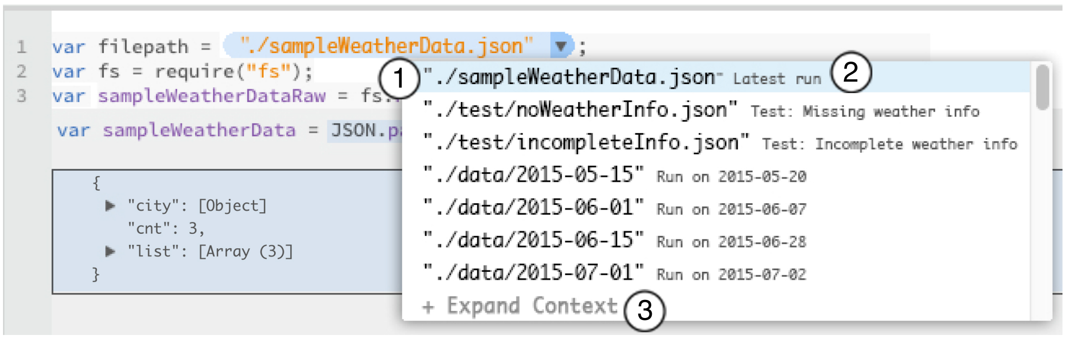
3.3.1 Idea

Figure 3.8 shows the third prototype, which we implemented as a high fidelity paper prototype using Adobe Photoshop. The prototype again shows an inline widget, a style we adopted from the second prototype and refined the visual aspects to match an actual implementation of an inline widget in the Adobe Brackets[4] editor. Instead of showing a function chain as the current line of code, we used a simple function call (1), in order to avoid the confusion that our visualisation of function chains in the second prototype caused. We designed the visualisation of the function more visually appealing and used colours to indicate return values, parameters and the object on which the function is being called (2).

We designed the third prototype in Photoshop.

The most noticeable change compared to the second prototype is the context editor (3), which contains the current line for which the tool was opened along with contextual code that initialises the variables that are being used in the

[4]http://brackets.io



```
var filepath = "./sampleWeatherData.json" ▼;
var fs = require("fs");
var sampleWeatherDataRaw = fs.
var sampleWeatherData = JSON.p

{
  ▶ "city": [Object]
    "cnt": 3,
  ▶ "list": [Array (3)]
}
```

```
"./sampleWeatherData.json" Latest run
"./test/noWeatherInfo.json" Test: Missing weather info
"./test/incompleteInfo.json" Test: Incomplete weather info
"./data/2015-05-15" Run on 2015-05-20
"./data/2015-06-01" Run on 2015-06-07
"./data/2015-06-15" Run on 2015-06-28
"./data/2015-07-01" Run on 2015-07-02
+ Expand Context
```

Figure 3.9: The value selector allows the programmer to initialise variables with values from previous program runs.

The context editor captures contextual code of the current line.

current line. The programmer can edit the variable values in the context editor manually or by selecting values using the "value selector" (4). All the code in the context editor is editable and is executed on every change. The results of the execution are used as input to the visualisation, which contains interactive elements (5) that update the parameters of the function in the context. As an example, the range picker next to the middle table can be extended and dragged, updating the `start` and `deleteCount` parameters of the `splice` function.

Values can be selected from previous program runs.

Figure 3.9 shows an example of the expanded value selector. The values suggested by the value selector represent the values that the respective variable will have in an actual program or unit test run (1). Chapter 4 "Fiddlets" contains a detailed description of how those values are harvested. The value selector further shows a hint where each value originates from (2). As previously mentioned, these values can derive from program executions — denoted here as "Run" — or from running the unit test suite — denoted as "Test", followed by the name of the actual test case that caused the value. Selecting a value in the value selector will change the value that the variable is initialised with in the context editor.

Expanding the context loads more contextual code from the original program.

The last entry of the value selector is the "Expand Context" entry (3), which is only available if either the variable in the context was derived from calling a function on another variable from the actual program that is outside the scope of the line for which the widget was invoked, or the variable was declared outside of the scope of this line.This is often the case when the widget is invoked within a function. When the programmer chooses to expand the context, the context editor will load additional lines that relate to the variable from the original program.

3.3.2 Evaluation

To evaluate the concept of the paper prototype we built paper prototypes for some possible interface states of the editor and created a visualisation for the `map` function. We prepared a paper prototype walkthrough containing 9 different interface states: one state showing the editor without the widget opened, two showing the visualisation of the `map` function and five showing different interaction states of the `splice` function visualisation. The prototypes for the `map` and `splice` visualisations also contain different states for the value selector in the context editor.

We evaluated the third prototype by walking developers through 9 different interface states.

We showed the prototypes to four programmers and asked them to explain the different interface states. The programmers quickly understood the connection between the code in the context editor and the corresponding code in the main editor. They discovered the intended meaning of the colour code we introduced to indicate the various parts of the current line of code.

None of the programmers could explain the functionality of the value selector. They expected it to show an unabbreviated version of the value that was assigned to the variable in that line of code. We attribute this false expectation to the lack of interactivity that the paper prototype provides and expect that in an actual interactive prototype, the programmers will try to click the value selector and draw the correct conclusions from the selectors behaviour. Half of the programmers found it hard to distinguish the context editor from the surrounding program editor, since due to its alignment and colour it did not stand out against the surrounding code.

The value selector was misinterpreted.

One programmer found the use of colours to establish the

We must be aware about visual overload.

visual connection between the parts of the current line and their counterparts in the visualisation problematic, because functions that take many parameters would require many different colours to match them to the visualisation. The use of that many colours would quickly make the UI overflow visually. The programmer suggested to use a single colour for parameters, but to only highlight a parameter whenever the programmer hovers the mouse pointer over that parameters counterpart.

3.4 Summary

We developed and evaluated three prototypes for tools that provide fast feedback about the state of program execution. With the first prototype we tried to reduce the amount of information displayed in a Live Coding environment to the kind of information that is most relevant for the current line and to immediately inform the programmer about the implications introduced through the editing of the code. Our second prototype focused on providing a more detailed visualisation of runtime state for a single line of code. The third prototype added to these visualisations the concept of a “context editor” and drafted the idea of partial code execution for increased Live Coding performance.

Feedback on the third prototype convinced us that we were on the right track to make a valuable tool for programmers in daily use. The context editor and the continuous execution and visualisation of the context code are innovative enough to justify a closer investigation on the performance of such a system. In the next chapter we will further explain how the core elements of the system that we envisioned with our paper prototype are expected to be implemented.

Chapter 4

Fiddlets

In this chapter we will give a detailed explanation of the concepts and rationales for the final implementation of "Fiddlets".

4.1 Concept

Fiddlets is a plugin for the Adobe Brackets IDE that allows the execution of a single line of code of a program without executing the program itself. To achieve this, Fiddlets generates a "context" for the execution of that line. The context consists of code that initialises and manipulates variables used by the line of code, as well as the line itself. The code that makes up the context is taken from the original program. Whenever a line of the context references additional variables, Fiddlets tries to resolve these by either integrating more lines of the original program into the context or by initialising the missing variables with values that are known from previous executions of the program (or its test suite). If Fiddlets fails to find meaningful values, the programmer can provide her own variable values. The context is then executed and the result of the execution is visualised, using a visualisation that matches the operation that is performed by the current line.

Fiddlets generates a context to execute the current line.

Visualisations in Fiddlets are both documentation and playgrounds.

Fiddlets' visualisations are meant to have documentary as well as a playful traits. The visualisation should help the programmer understand what the particular line of code accomplishes and how the parameters of the function relate to the resulting behaviour of this line. The content of the visualisation is updated on every change that is made to the context code. By changing the function parameters of the current line—either in the line itself and observing the changes in the visualisation, or by using the interactive elements of the visualisation and observing the updated parameters in the current line—the programmers can explore the effects of different configurations of function parameters without interfering with the original source code of the program.

Fiddlets allows to execute small parts of a program with data from actual program runs.

Fiddlets lets the programmer investigate the behaviour of smaller parts of the program without having to run the complete program, but with the benefit of using real program data from previous program runs. This provides fast feedback cycles without the need to copy and paste code to and from external tools as described in 1.1 "Motivation". The programmers can further provide their own variable values and observe how different parts of the program react to these changes.

4.2 Implementation

Context, *Value Tracing* and *Visualisation* are the main features in Fiddlets.

In the following section we will explain how the main features of Fiddlets are implemented and how the features interact with each other. The main features are: *Context*, *Value Tracing* and *Visualisation*. Figure 4.1 shows the basic control flow of a Fiddlets instance. Proof-of-concept code for the front-end[1] (visualisation) and the back-end[2] (context collection) are available on GitHub.

On invocation, Fiddlets analyses the line of code in which the cursor is placed, the *Current Line*. Fiddlets identifies all variable and function names used in the current line

[1]https://github.com/laewahn/Fiddlets-Mock
[2]https://github.com/laewahn/Fiddlets-Backend

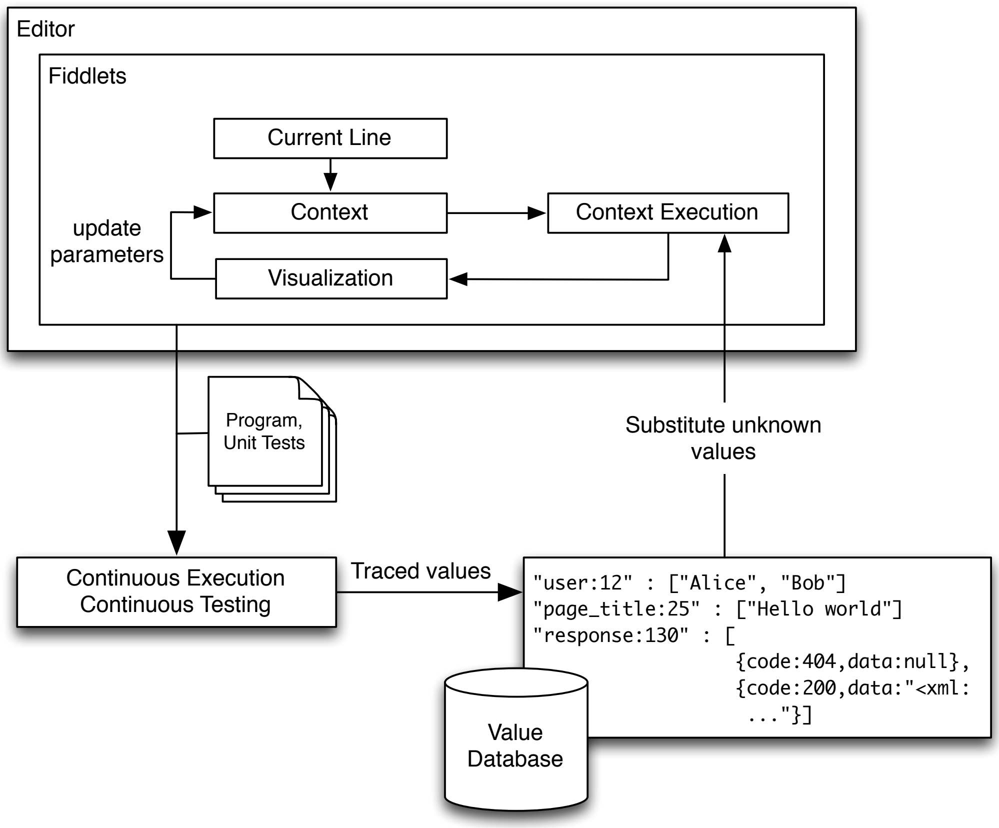


Figure 4.1: An overview over the architecture of Fiddlets.

and uses these to create the context (4.2.1 "Context"). Fiddlets also tries to find out what kind of operation is carried out by this line. The kinds of operation that Fiddlets recognises are: variable declaration, variable assignment and function call, as well as any combination of these (variable initialisation—function calls with the return value being assigned to a variable—, etc).

Fiddlets analyses the current line to build the context.

After context generation has finished, Fiddlets fills in missing variable values (4.2.3 "Value Tracing") with values from previous program runs and executes the context together with the current line. Based on the information about the current line from the previous step, Fiddlets loads and displays a visualisation for the specific operation that is carried out by the current line (4.2.4 "Visualisation"). The visualisation is then updated with values from the execution

Missing variables are initialised with values from previous executions.

of the context and the current line.

As we prototyped it in 3.3 "Third Iteration: Adding Context", Fiddlets will display the context generated for the current line in an editor that is separate from the main editor, the *context editor*. Once the context for the current line from the original program is created and put into the context editor, Fiddlets will use the last line of the context editor as source for the visualisation. When the programmer changes this last line of the context editor, Fiddlets will analyse this line and update the current visualisation with the results of the last execution or, if necessary, replace the whole visualisation with a new one. The programmer can change the current line of code in the context either by directly editing it or interactively in the visualisation.

The context can be edited in the context editor.

The remainder of this section will give a detailed explanation on context, value tracing and visualisation.

4.2.1 Context

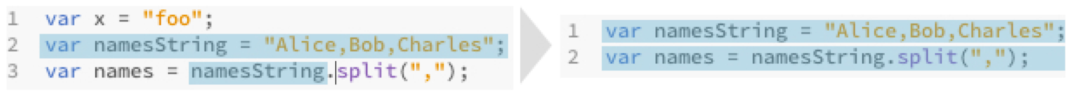
The context builds up a small program that performs the necessary declaration and initialisation of variables and functions that are used by the current line of code. It consists of lines of the original program and aims to build an approximation of the context in which the line would be executed in an actual program run.

The context consists of lines of the original program.

Building a context for a line of code always starts with analysing the line itself and localising all variables and functions used by the variable. In the next step Fiddlets attempts to find other lines that are related to the variables and functions of the current line. For every variable this includes the following lines:

Criteria for relevant lines.

- The line in which the variable was declared.
- Lines where the value behind variable was updated.
- Lines where the variable was given a new value.



```
1  var x = "foo";
2  var namesString = "Alice,Bob,Charles";
3  var names = namesString.split(",");
```

```
1  var namesString = "Alice,Bob,Charles";
2  var names = namesString.split(",");
```

Figure 4.2: A simple program (left) and the context that is generated for line 3 (right)

- If the variable references a function, the complete function is copied to the context.

Fiddlets inserts lines that are matched by these criteria into the context. For every line inserted into the context that references other variables or functions, Fiddlets looks for new lines to insert into the context, following the same rules as before. This continues until no new variables and functions are added to the context.

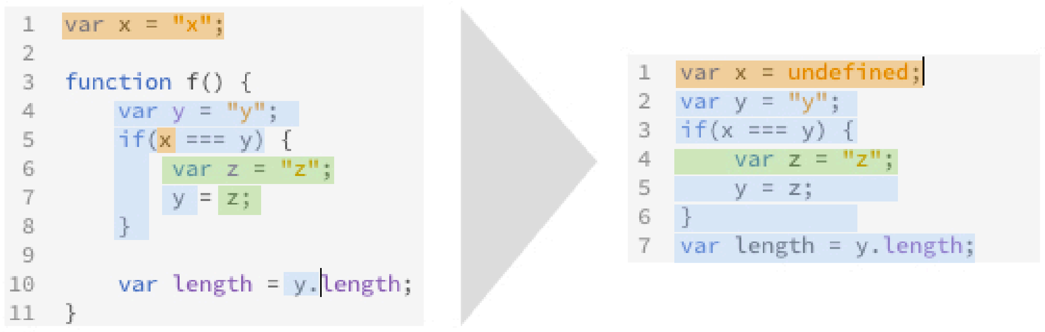
Figure 4.2 shows how the context for a line in a simple program is generated. To generate the context for line 3, Fiddlets identifies the variable `namesString` as relevant. The function `split` is a function that is provided from the built in JavaScript `String` class and is therefore ignored. Fiddlets searches for occurrences of `namesString` and finds its initialisation in line 2, which is copied into the context. Since line 2 initialises `namesString` with a literal and involves no other variables, the context generation terminates.

In order to prevent the context from growing too much, no lines that are outside of the scope of the current line are used in the context. This way, the created context can never grow beyond the size of the function in which the current line is located, thereby keeping the context comprehensible. The restriction also allows a developer to reduce the possible context space by placing the code she is interested in into a new scope.

Fiddlets restricts relevant lines to those within the scope of the current line.

When the context generation has finished, Fiddlets generates declarations for every variable that has been used in the context, but that was not declared in it. This is often times the case when the context should use parameters of a function or variables that are provided by the runtime environment, e.g. command line arguments or values from

Fiddlets add declarations for missing variables.



```
1   var x = "x";
2
3   function f() {
4       var y = "y";
5       if(x === y) {
6           var z = "z";
7           y = z;
8       }
9
10      var length = y.length;
11  }
```

```
1   var x = undefined;
2   var y = "y";
3   if(x === y) {
4       var z = "z";
5       y = z;
6   }
7   var length = y.length;
```

Figure 4.3: Building the context (right) for line 10: Fiddlets looks for lines that reference *y* (blue). Since line 7 references *y*, but also references z, Fiddlets includes lines referencing *z* (green). Since the variable *x* is declared outside of the scope of function *f*, *x* is considered undefined (orange).

external libraries. In order to initialise these variables with values from previous program or test runs, Fiddlets inserts a special tag that contains the name of the variable as well as the number of the line in which the variable was declared initially. An example format for such a tag could be `<#{fallback}:{variable_name}:{line}#>`, where `{fallback}` is a value that Fiddlets should insert if no substitution can be found, `{variable_name}` refers to the name of the variable and `{line}` is the line in which the variable is declared or initialised in the program. Knowing about the line in which the variable was declared is important if a variable name is declared in more than one scope. Variables declared with the same name in different scopes are treated as independent from each other, so manipulating the variable in a certain scope does not affect the value of a variable with the same name in another scope.

An example for a declaration tag for the variable `x` in Figure 4.3 would be `<#undefined:x:1#>`, as the variable `x` was declared in line 1 and `undefined` is the default value for variables that were not initialised in JavaScript. Choosing `undefined` as a default value reminds the programmer that she needs to substitute it with meaningful values.

4.2.2 Expanding the Context

In certain situations, programmers find the scope restriction we imposed on the context collection algorithm too strict. This might happen when some variables that were declared outside of the scope of the current line were derived in a certain way and she wants to see how changing the way that the variable is treated outside of the scope influences the result of the current line. In this case she can use the *Expand Context* option. When expanding the context, Fiddlets tries to resolve any unknown variables in the current context by searching for occurrences of these variables in the parent scope of the current line and copying those lines into the context.

Expanding the context considers lines outside of the scope of the current line.

4.2.3 Value Tracing

As mentioned in 4.2.1 "Context", Fiddlets attempts to substitute any unknown variable in the context with a value from previous program or test runs. To achieve this, Fiddlets uses a database of possible values to find an appropriate initial value for the variable. When it finds more than one possible value, it offers the programmer to select between these values. If no matching value could be found, the variable is initialised with the default value that was decided on context creation and the programmer has to enter a value herself.

Unknown values are received from a database of previous values

Fiddlets uses an instrumented version of the program to collect the values for the database. The instrumented program is set up to capture the values of variables along with the variable name and the location in the program where the variable had the respective value. The instrumented version of the program could be run continuously on every change, as is currently done for Live Coding, or on demand. Either of these ways suffers from the same performance problems we discussed in chapter 2.3 "Live Coding". Reachability adds more problems to this: if an execution path of the program flow is not covered by the live execution of the program, the database will not contain values for the variables on this path. However, at the time a set of

Instrumented program runs are used to build the value database.

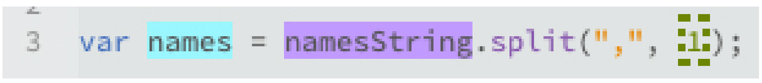
possible values for a variable is present in the database, further executions of the program can be delayed to the point where new variables are introduced in the program or the intended values of a variable change due to modifications in the data processing. For the time that the programmer is experimenting with the available variable values in the context editor, no further execution of the program is necessary.

Unit tests are a great source for variable values.

Another source for variable values is to run the unit test suite of the program against the instrumented program. Unit tests are often designed to perform fast—by mocking external dependencies of the program with defined values—and cover great parts of the program, making them an ideal candidate for variable value collection. They are even able to be run in the background without user interaction. However, the data used in unit tests is often made up by the programmer writing the unit test and therefore it can be claimed that most of the unit test data does not represent the entirety of data that the program would need to operate on. Another problem is that the coverage provided by the test suite depends on which code execution paths were decided to be relevant by the programmers. It is therefore not guaranteed that using a unit test suite alone will cover all relevant execution paths. To expand the coverage provided by the test suite, the programmers will need to write more tests. We therefore advocate a mixed solution to the problem, using live executions as well as unit test suites as sources for the variable value database. Adding a code generator that creates unit tests from live execution sessions (e.g., Kuhn [2013]) would then help in the process of converting slow live execution for uncovered execution paths into repeatable unit tests.

Using an instrumented program version in production is risky, but provides the best values.

A third way to capture variable values is to deploy the instrumented program into production. Collecting runtime information of a deployed application raises serious privacy concerns, but at the same time this information would have the most validity and would therefore be the most valuable data for debugging. Another drawback is that the decreased performance of the instrumented version of the program could possibly lead to user dissatisfaction. A possible solution to this dilemma is to mark values that should



```
var names = namesString.split(",", 1);
```

Figure 4.4: Current line highlights. The parameter highlight can be set by the visualisation.

be traced with special annotations, an idea that was already proposed for Live Coding systems by McDirmid [2013]. This approach would allow for data collection on selected parts of the program, for example parts that have to deal with a variety of different data or parts that often fail due to unknown reasons.

4.2.4 Visualisation

Fiddlets uses several visual clues to help the programmer understand what the code in the context editor does. Additionally, a whole section of the Fiddlets inline widget is dedicated to visualise the execution results of the last line of the context.

As previously mentioned in 3.3 "Third Iteration: Adding Context", the context editor uses colours to highlight certain aspects of the visualised line of code (Figure 4.4). If the line performs an assignment of any kind, then the variable on the left side of the assignment is highlighted blue. If the line calls a function on an object, the object on which the function is called is highlighted purple. The parameters of the function can additionally have a green dashed frame as highlight, but the highlight must explicitly be set in the visualisation.

We use colour clues to establish a connection between code and visualisation.

We expect visualisations to serve both documentary as well as experimental purposes. Visualisations should explain the behaviour of the last line of the context by showing a visual explanation of the operation that is carried out by the line. At the same time should they offer affordances that allow to manipulate and experiment with the parameters and properties of that line.

Visualisations should be used for documentation and experimentation.

Visualisations receive a trace of the context state before and after the last line is executed.

In order to show the difference between the state of an object with and without the execution of the last line, the visualisation is given a trace of the context state before and after execution of the last line. It receives updated traces of the context state whenever the context is changed. Each visualisation is allowed to make changes to the last line of the context. This allows the designer of a visualisation to incorporate interactive elements that allow to change certain properties of the function that is visualised.

Chapter 5

Study Design

The previous chapter introduced a novel interaction with program code, where lines of code are taken out of the context of a complete program, in order to build a new context, independent of said program, that tries to resemble behaviour that is expected in the original program. In order to evaluate the usefulness of this novel concept, we designed a user study with the following research questions in mind:

How does Fiddlets affect programmer performance?

- How will programmers use Fiddlets to solve programming tasks?
- Will programmers using Fiddlets solve certain programming tasks faster than programmers without Fiddlets?

We implemented a proof-of-concept prototype.

In this chapter we present the design of our user study. We implemented a prototype based on the requirements of the previous chapter, but with certain limitations. We describe the functionalities and limitations of our prototype and present the visualisations we implemented according to the requirements imposed by the tasks of our study. We furthermore describe the setup, procedure and tasks used in the user study.

5.1 Prototype

To test the concept of Fiddlets, we implemented a prototype of the system that fulfils most of the requirements we described in chapter 4 "Fiddlets".

5.1.1 Limitations

The prototype contains a proof-of-concept context building algorithm that generates valid contexts for the relevant parts of the source code used in the tasks, but has some limitations. The algorithm is not able to provide the *expand context* feature we discussed in 4.2.1 "Context". In order to work around this limitation, we refactored the code of the example programs in a way that the *expand context* feature would not be needed. For this we moved the relevant functions and variables of the functions that the participants were expected to operate within directly into the scope of said functions. This way, all necessary variables and functions would be available to the context generated by the naive algorithm. Some functions and variables could not be moved into the scope of the functions that the participants will operate on without breaking the example program behaviour. The algorithm generates declarations that initialise said variables and functions with the value `undefined`. We made sure to minimise the number of those undefined initialisations.

The algorithm does not allow to expand the context.

Another limitation of the algorithm is that is does not consider type deduction for the objects in use. For example, if the participant decided to alter the contents of an array by calling the `push` method, the algorithm will expect `push` to be a function that should be defined within the scope of the current function. Since the push function is defined as a method of the `Array` prototype and not defined in the current scope, the algorithm will create the initialisation for a variable named `push` and initialise that variable with the value `undefined`. Since this behaviour does not influence the result of the execution of the context, we decided to explain this case to the participants and ask them to ignore

Our algorithm has no type knowledge.

such cases.

To fill in unknown variables as described in 4.2.3 "Value Tracing" we provided example values and told the participants that these had been already generated by previous runs of the program or the test suite. For tasks that deal with the source code of real world software projects (see 5.3 "Task Design"), we extracted values that were used within the test suite of these projects and offered those as fill-ins for unknown variables.

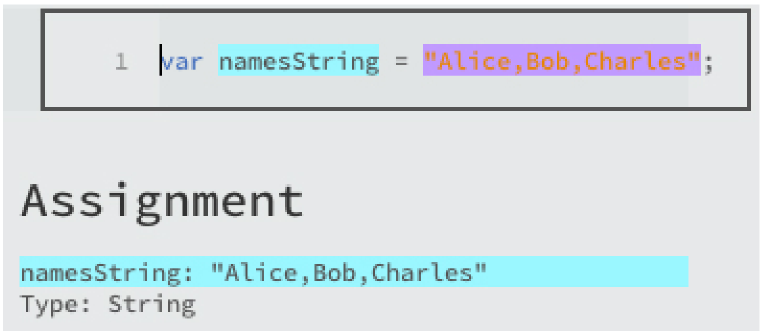
Example values are defined statically.

The context editor of the prototype does not provide an autocomplete function.

5.1.2 Visualisations

We designed and implemented visualisations for the most necessary functions that our participants will have to work with in the study. This includes the following visualisations:

- Assignments
- `String.prototype.replace`
- `String.prototype.split`
- `Array.prototype.map`
- `Array.prototype.splice`



```
var namesString = "Alice,Bob,Charles";
```

Assignment

namesString: "Alice,Bob,Charles"
Type: String

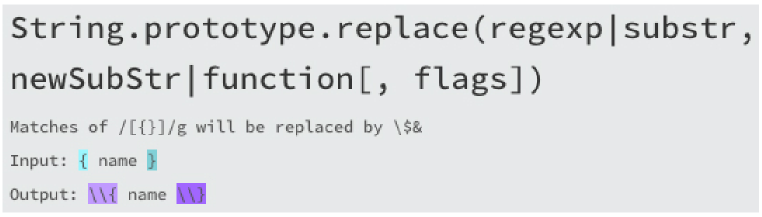
Figure 5.1: Visualisation of a variable assignment in the study prototype.

Definition:
Prototype Methods

PROTOTYPE METHODS:
JavaScript is a prototype based language, meaning it lacks the concept of classes and instead operates on objects with arbitrary key-value properties. To mimic classes, objects in JavaScript have a reference to a prototype object which provides access to the methods that are available for all objects of its type. The prototype of an object is set on creation of the object an can be changed during runtime.
The documentation for a method of a JavaScript type usually references the prototype of that type. In the rest of this section, readers that are used to object-oriented programming may treat the prototype object as the class of the object. For example: `String.prototype.map` in JavaScript would be the equivalent of the method `map` defined by the `String` class in an object-oriented language.

Assignment
visualisation

Figure 5.1 shows the visualisation of an assignment. The visualisation informs the programmer about the new value of the variable on the left side of the assignment and the type of the object that is assigned to this variable. It is also used as a fallback whenever no specific visualisation for a line of code could be found.



```
String.prototype.replace(regexp|substr,
newSubStr|function[, flags])
Matches of /[{}]/g will be replaced by \$&
Input: { name }
Output: \\{ name \\}
```

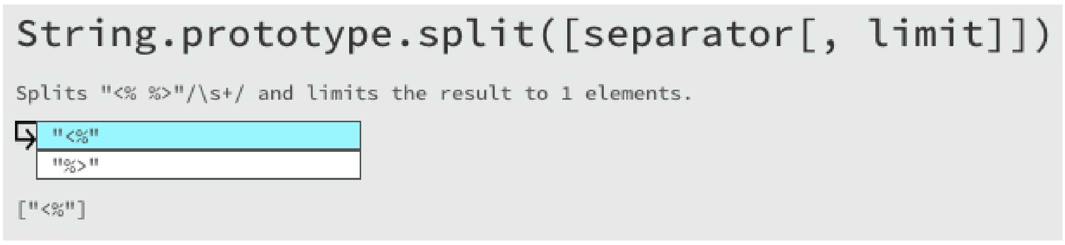
Figure 5.2: Visualisation of the `String.prototype.replace` method in the study prototype

STRING.PROTOTYPE .REPLACE:
JavaScript's `String.prototype.replace` method is used to match and replace parts of a string object. It takes two parameters: the first parameter is a string or a regular expression that should be matched, the second parameter is either a string that replaces every match in the original string or a function that is called for every match. In the second case each matching string is used as an input for the function which should then return the replacement for that match. `String.prototype.replace` returns a new string with the replaced matches and does not alter the original string.

Definition: *`String.prototype.replace`*

`String.prototype.replace`

The visualisation for the `String.prototype.replace` method (Figure 5.2) shows a brief explanation of what the method is supposed to do, as well as the value of string on which the method is being called and the return value. The visualisation highlights the elements matched by the first parameter of the method and their corresponding replacement in the return value. Alternating brightness of the highlights is used to help the programmer mapping matches to their replacements.



```
String.prototype.split([separator[, limit]])

Splits "<% %>"/\s+/ and limits the result to 1 elements.

"<%"
"%>"

["<%"]
```

Figure 5.3: Visualisation of the `String.prototype.split` method in the study prototype

Definition: *`String.prototype.split`*

String.prototype.split:
`String.prototype.split` is a method that separates a string into components. Its first parameter is a string or a regular expression that specifies the characters that are used for separation. It returns a new array with the components of the separated string. An optional second parameter can be used to limit the number of components in the returned array.

`String.prototype.split`

Similar to `String.prototype.replace`, the `String.prototype.split` visualisation (Figure 5.3) shows a short explanation of the methods intended functionality as well as the object it is called on and the return value. An additional visualisation shows an intermediate step of the method call that helps understanding the limit parameter of the method. The additional visualisation shows the array that would be returned by the method if no limit was given in the parameters. All rows of the elements that make up the return value are marked with a blue background. Rows with elements that are not included in the return value have a white background. If the limit is set in the parameters, an arrow on the left side of the table allows the programmer to interactively change it by moving the head of the arrow. The head of the arrow always points to the last element to be included in the return value.

ARRAY.PROTOTYPE.MAP:
Array.prototype.map creates a new array that contains transformed values of the array on which it is called. The method has a single parameter, which is a function that is used to transform the values. This function is called once for every element of the original array, with the corresponding object used as the parameter, and should return the transformed value of that object.

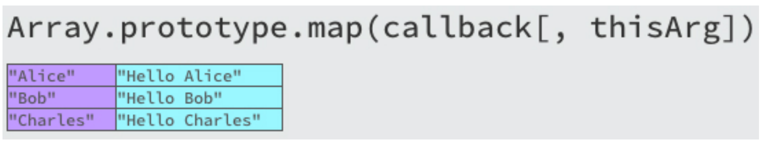
Definition: *Array.prototype.map*

The visualisation for the Array.prototype.map method uses a two column table (Figure 5.4). In the left column it shows the values of the array on which the method is called. The values of the returned array are shown in the right column of the table. It uses the colour scheme we introduced in section 4.2.4 "Visualisation" to make a visual connection to the elements in the last line of the context editor.

Array.prototype.map

ARRAY.PROTOTYPE.SPLICE:
The Array.prototype.splice method changes the content of an array by removing a range of objects from the array and replacing it with a variable number of new elements. All parameters of splice are optional. If splice is called with no parameters, it returns an empty array and leaves the original array unchanged. When parameters are given, the first parameter defines the starting index for the objects that are removed from the array. The second parameter defines the number of elements that will be removed. If this parameter is not set, all elements after the start parameter are removed. After the first two parameters splice takes an arbitrary number of further parameters. Every parameter represents an object that will be inserted into the array at the start index.

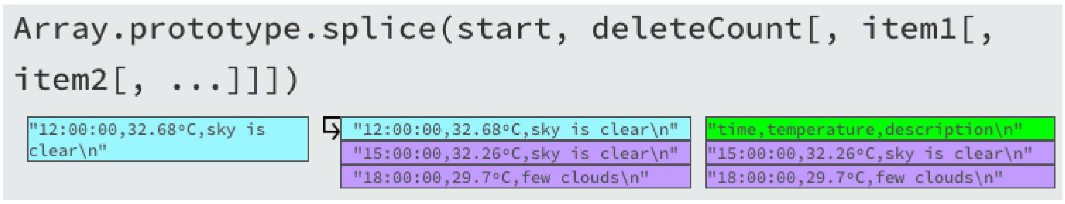
Definition: *Array.prototype.splice*



```
Array.prototype.map(callback[, thisArg])
```

"Alice"	"Hello Alice"
"Bob"	"Hello Bob"
"Charles"	"Hello Charles"

Figure 5.4: Visualisation of the Array.prototype.map method in the study prototype



```
Array.prototype.splice(start, deleteCount[, item1[,
item2[, ...]]])
```

```
"12:00:00,32.68°C,sky is    "12:00:00,32.68°C,sky is clear\n"   "time,temperature,description\n"
clear\n"                    "15:00:00,32.26°C,sky is clear\n"   "15:00:00,32.26°C,sky is clear\n"
                            "18:00:00,29.7°C,few clouds\n"      "18:00:00,29.7°C,few clouds\n"
```

Figure 5.5: Visualisation of the `Array.prototype.splice` method in the study prototype

`Array.prototype .splice`

The visualisation of the Array.prototype.splice method is the most complex visualisation of the prototype. It is composed of three single column tables (Figure 5.5). The table in the middle displays the elements of the array on which the method is called. Elements of that array that are within the range defined by the `start` and `deleteCount` parameters are highlighted with a blue background. Elements that are not affected—and therefore will stay in the original array—are highlighted with a purple background. Moving the head of the arrow on the left side of the table lets the programmer change the `deleteCount` parameter similar to the arrow in the `String.prototype.split` method. The arrow itself can be moved up and down to change the `start` parameter.

The table on the left displays the contents of the return value. Selecting more elements of the original array by either manipulating the arrow UI element or altering the parameters manually will change the number of elements in the output array. The table on the right represents the value of the array on which the `splice` method is called. Elements with a purple background remain in the original array. Elements with a green background are elements that are added in the method call.

Since the parameters of the `splice` method are optional, not all tables are shown all the time: if no parameters are given, only the middle table is shown, otherwise all tables are shown.

Errors

Errors produced by the execution of the context code are shown beneath the context editor. Displaying an error does not remove the current visualisation, but is shown in an additional text output. The error message is displayed with red text colour to stand out against the remainder of the visualisation.

5.2 Study

We designed the study as a between-subjects study with two conditions: the "Fiddlets" condition and the "Control" condition. In the Fiddlets condition participants are allowed to use the prototype to solve the tasks. In the Control condition the prototype is deactivated and participants are allowed to use external tools of their choice to solve the tasks. We neither restrict, nor encourage the usage of specific tools in the Control condition.

We designed the study as between-subjects with two conditions.

5.2.1 Setup

For the study we use an iMac with a 3.5 GHz Intel Core i7 processor, 24GB Ram and 500GB SSD storage. The iMac has an integrated 27" screen with a resolution of 2560x1440 pixel. We reduce screen resolution to 2048x1152 if participants have trouble with the default resolution. Participants can decide to use either a USB keyboard with US layout or a wireless keyboard with German layout. The iMac is running Mac OS X 10.10 (Yosemite). All tasks are performed with the Adobe Brackets code editor Release 1.2 build 1.2.0-15697.

We provide participants with sufficient documentation to solve each task. This includes the documentation of the `String.prototype.replace` method, the `String.prototype.split` method, the `Array.prototype.map` method, the `Array.prototype.splice` method and the documentation of the JavaScript RegExp type from the official Mozilla JavaScript documentation[1]. We also provide them with a demo of mustache.js. Participants are allowed to

We provide the official documentation for all relevant functions.

[1]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

use any search engine of their choice to browse for further documentation. We advise them not to search for the source code of mustache.js, since that would contain the solution to the tasks.

Task completion time is measures as the time between clicking the "Start task" and "Hand in Task" buttons.

To measure the task completion time we added an extension to the editor showing a task bar with a "Start Task" button above the editor pane. The extension logs the date when the participant presses the button and changes the button title to "Hand in Task". When the participant presses the button again, a modal dialog asks her to confirm to hand in the task. If she confirms the dialog, the extension logs the date that the task was handed in and writes the captured data into a log file. For participants in the Fiddlets condition the log also captures time-stamps for every time that Fiddlets has been invoked and closed. This way we capture how frequent and for how long a participant use Fiddlets.

For further analysis and in case that the logging fails we capture the iMac's screen and microphone input with the screen recording feature of QuickTime. We prepared a pre-questionnaire to be filled out before the study, where we collect age, gender and occupation information, as well as information about programming experience (in years) and JavaScript experience (in years).

We additionally ask users to rank their familiarity with the following technologies and concepts on a scale from 1 (Not at all) to 5 (Very familiar):

Preliminary knowledge.

- Regular Expressions
- Unit Testing
- Live Coding
- Node.js
- mustache.js

We additionally hand out an unchanged version of the SUS scale by Brooke [1996] as a second questionnaire to collect feedback about the usability of Fiddlets. The questionnaires used in the study can be found in Appendix D "Questionaires".

5.2.2 Procedure

We first explain the purpose and background of the study to the participant. We then hand her an informed consent form which we additionally explain to her. We offer her to chose between a keyboard with US or German layout. We also ask about her preferred mouse scrolling direction and hand her the first questionnaire.

Participants can chose between German and US keyboard layout.

After the participant has filled out the questionnaire, we hand her the description for the first task. We ask her to read the description carefully and to ask questions if any aspect of the task remains unclear. The task descriptions contain precise instructions on how to begin and how to finish each task. Participants in the Fiddlets condition are given a brief introduction into the Fiddlets prototype which also contains a brief explanation of the context algorithm. We tell them that the values filled in by the context editor are from actual program runs, although they have been previously defined and are read from a file.

Task descriptions contain precise instructions.

Each task is limited to 20 minutes.

We limit the time for each task to a maximum of 20 minutes. If the participant does not solve the task within this time, we ask her to hand in the task. After each task, we allow the participant to take a rest and offer snacks and drinks, before handing out the next task description. The participant is allowed to ask questions and we answer any question that do not relate to the solution of the task, e.g., where to find certain characters like the degree symbol on the keyboard.

When the participant finishes the last task, we hand her the second questionnaire. We offer the participant to ask further questions regarding the prototype and the tasks. Participants in the Control condition are offered a quick demo of the prototype.

5.3 Task Design

In an empirical analysis of the corpus of widely-used JavaScript programs, Richards et al. [2010] identified `Date`, `RegExp` (Regular Expressions), DOM (Document Object Model, i.e., rendered HTML) `Array` objects as well as runtime errors, as the most frequently used built-in data types of JavaScript. According to this, we designed our tasks around `RegExp` and `Array` objects, since on the one hand, they are among the most frequently used data types in most programming languages, and on the other hand, they provide certain pitfalls to even experienced developers, e.g., off-by-one[2] and out-of-bounds[3] errors for arrays. `RegExp` are especially interesting, as their inherent complexity and versatility makes them both hard to read and hard to compose, justifying the existence of tools[4] that specialise only on the visualisation of Regular Expressions.

Arrays and Regular Expressions are among the most frequently used built-in data types in JavaScript.

Three of five tasks were centred around mustache.js

In order to make the tasks as authentic as possible, we based three out of the five tasks on the source code of mustache.js[5], an open source template system written in JavaScript. We chose mustache.js for the following reasons:

[2]https://cwe.mitre.org/data/definitions/193.html
[3]https://cwe.mitre.org/data/definitions/125.html
[4]http://www.regexpal.com/
[5]https://github.com/janl/mustache.js

- It is widely used and supported by other open source projects.
- With around 600 lines of code it is fairly comprehensible.
- It uses a lot of string parsing and Regular Expressions.
- It provides a comprehensive and fast performing unit test suite.

We built a simple data processing application for tasks four and five.

The two other tasks are designed around a simple data processing application written in Node.js. The application reads Base64 encoded JSON data from a file, parses the JSON data into a collection of separate JavaScript objects and writes the content of these objects into a "Comma Separated Values" (CSV) file. We designed this application to make heavy use of `Array` objects and native JavaScript objects. We chose a Base64 encoded input file to make sure the user cannot imply the content of the processed data by looking into the input file.

The source code for the data processing application can be found in Appendix B "Data Processing Tasks: Source Code". All task descriptions are part of Appendix C "User Study Task Descriptions".

5.3.1 Task 1: Implementing a Regular Expression

Task 1

In the first task we ask participants to provide a regular expression for a call of `String.prototype.replace` to replace special characters with their escaped version. The regular expression should match special characters in a given set of inputs, which is given in a table that contains inputs for the function as well as the desired return values.

Participants are allowed to execute the unit tests of the program, but the tests do not provide feedback about the correctness of the solution. Participants have to find their own way to make sure that their regular expression matches the expected characters.

5.3.2 Task 2: Implementing a Regular Expression 2

Task 2

In the second task the participants have to provide a regular expression for a function that is supposed to escape special characters of HTML (`< > ' " / &`). The program initially has a regular expression matching < and >, but not matching the remaining special characters.

Participants are encouraged to execute the unit test suite of the program to verify their solution. This time the unit tests fail until a correct solution is provided.

5.3.3 Task 3: Bug-fixing a Split Operation

Task 3

For the third task we introduce a bug into the `compileTags` function of mustache.js. `compileTags` receives either an array containing exactly two tags or a string of the two tags separated by whitespace, and builds regular expressions that will later be used to scan a template for the tags to replace. If the `tagsToCompile` parameter of the `compileTags` function is a string, the function calls the `split` method, creating an array of the components of the input string. The `split` method takes two parameters: the first one is a regular expression that matches white space, the second one is a number limiting the size of the array that is returned by the method. In our case the limit is set to two. After the input is split, the `compileTags` function performs a size check on the `tagsToCompile` array to make sure it contains exactly two elements, i.e., the opening and the closing tag. If the `tagsToCompile` is not an array with two elements, an exception is thrown. Otherwise, the function continues to build regular expressions to match the opening and closing tags.

We introduced a simple but easy to overlooked bug by changing the limit parameter of the `split` method call from initially one to two, resulting in the array returned by `split` containing only one element instead of two. The size check on the value of `tagsToCompile` subsequently fails with an exception. Again, we allow participants to use the unit test suite to ensure the correctness of their solution.

5.3.4 Task 4: Transforming an Array

Task 4

The fourth task is to transform JavaScript objects used in the data processing application (Appendix B "Data Processing Tasks: Source Code") into a line for a CSV file using the `map` method. The code calling the `map` method, storing the return value in a variable to be written to file and the function used as parameter for `map` are already implemented in the application. The later function is supposed to return a CSV line from a JavaScript object of the array, but returns the object it was given in the parameter. We ask the participants to implement the code that returns the CSV line string from the function. Participants can run the application and observe the output file in order to verify their solution.

5.3.5 Task 5: Changing Array Contents

Task 5

The fifth task adds an additional requirement to the data processing application: whenever is is run it should replace the first element of the previously built CSV array with a header. The first element of the CSV should then be saved in a separate variable in order to be printed to the standard output.

Participants are asked to implement this functionality in a single call of the `splice` method. JavaScript's `splice` method is a versatile function that can be used to remove, extract and insert objects into an array. We supply participants with an example call of the `splice` method—but without parameters—and ask them to provide the parameters that are necessary to implement the desired functionality. Again, participants can run the program and observe the output to verify their solution.

Chapter 6

Evaluation

In this chapter we discuss the results we obtained from our user study. We start with describing the demographic data of our participants and the data we obtained from the preliminary questionnaire. We then evaluate the collected usage and performance data. Finally we take a look at user reception and observations we made during the study and discuss implications for our interaction design.

6.1 Participants

We recruited 14 participants via a university mailing list, the mailing list of a monthly programmer meet-up and by directly asking computer science students of the RWTH Aachen University. The participants were mostly students (12) except for one PhD student and an IT consultant. The students were enrolled in computer science (10) or computer science related courses (2). The two non-students both have a degree in computer science. 12 participants were male, two were female. Participants age ranged from 19 to 53 years, the average age was 25.79 years (Table 6.1).

Participants were mostly computer science students.

Participants showed a huge diversity in programming experience. The average overall programming experience ("Programming XP") was 8.357 years with a standard devi-

Programming experience was widely distributed.

	Mean	Median	SD	Min.	Max.
Age	25.79	24.50	8.396	19	53
Programming XP (years)	8.357	8	6.902	1	30
JavaScript XP (years)	3.357	1.25	3.505	0.5	10

Table 6.1: Age and programming experience (XP) of the participants

ation of 6.902 years, ranging from one year up to 30 years. The average JavaScript experience ("JavaScript XP") was 3.357 years with a standard deviation of 3.505 years, ranging from half a year up to 10 years (Table 6.1).

Conditions were assigned randomly.

Participants were randomly assigned to either the Fiddlets condition, where they were allowed to use our Fiddlets prototype to solve the task, or the Control condition, where they were allowed to use any external tool that would help them solve the tasks. We ended up having equal numbers of participants in the Fiddlets condition and the Control condition.

	Group	Mean	SD	95% CI
RegExp	c	3.857	0.690	[2.60, 3.40]
	f	3.000	0.577	[3.38, 4.33]
Unit Testing	c	2.714	1.380	[2.12, 3.89]
	f	3.000	1.291	[1.77, 3.66]
Live coding	c	1.429	0.787	[2.63, 3.94]
	f	3.286	0.951	[0.89, 1.97]
Node.js	c	2.000	1.414	[1.65, 3.21]
	f	2.429	1.134	[1.03, 2.97]
mustache.js	c	1.714	1.496	[1.06, 2.37]
	f	1.714	0.951	[0.69, 2.74]

Table 6.2: Reported preliminary knowledge about tools and concepts from the first questionnaire

Fiddlets participants reported higher knowledge in Live Coding and Unit Testing, Control participants in Regular Expressions.

According to the reported preliminary knowledge we surveyed with the first questionnaire, participants in the Fiddlets group had a small bias towards Unit Testing ($M = 3.286$) and Live Coding ($M = 3.000$). They also reported a slightly higher knowledge of Node.js ($M = 2.429$) than participants in the Control group, but since the value is below a rating of "Neutral", we consider this knowledge irrelevant. The same holds true for the reported knowledge of mustache.js, which both groups reported equally low ($M = 1.714$). More detailed information about reported preliminary knowledge is listed in Table 6.2.

6.2 Results

We recall the research questions we stated in chapter 5 "Study Design", which were:

- How will programmers use Fiddlets to solve programming tasks?
- Will programmers using Fiddlets solve certain programming tasks faster than programmers without Fiddlets?

6.2.1 Usage

We looked at average and maximum usage times.

To answer the first question we analysed the usage data, i.e., how often and for how long did the participants use Fiddlets. From the usage data collected in the user study we calculated the following values for each task and each participant:

avg_usage_seconds: Average time in seconds that the participant used Fiddlets.

max_usage_seconds: Longest period in seconds in which the participant used Fiddlets.

avg_usage_rel: Average Fiddlets usage relative to the task completion time.

max_usage_rel: Longest Fiddlets usage relative to the task completion time.

count: Number of times that the participant used Fiddlets in a task.

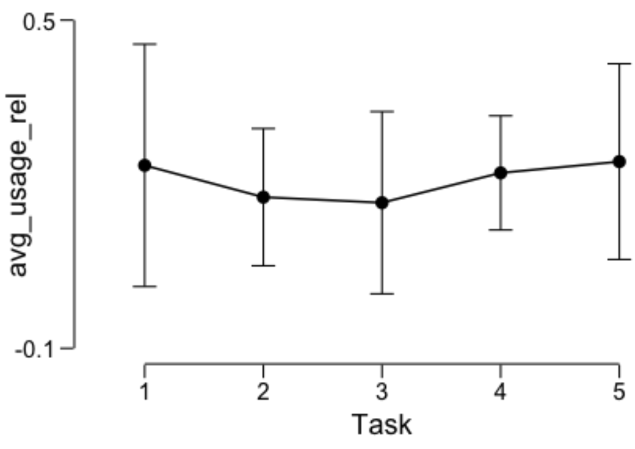


Figure 6.1

Fiddlets was used for around 20% of Task Completion Time.

From the data we see that participants used Fiddlets on average 11.21 times per task. One participant even used Fiddlets 45 times in task 1. Fiddlets usage accounts for an average of 20.68% of participants task completion time. The longest time an instance of Fiddlets was opened accounted for 97.71% of the task completion time, in this case also for task 1. Table 6.3 shows the mean values, standard deviation and maximum values for the collected usage data.

	Mean	SD	Max
avg_usage_seconds	131.4	149.7	799.3
max_usage_seconds	371.8	320.5	1203
avg_usage_rel	0.2068	0.1654	0.6491
max_usage_rel	0.5059	0.2805	0.9771
count	11.21	9.999	45

Table 6.3: Average usage time, maximum usage time and count of usage of all participants

We looked at the usage times for individual tasks.

In order to see how average usage times and maximum relative usage are distributed among the tasks, we sorted the values by task and plotted averages and confidence intervals per task for each measurement. Figures 6.1 and 6.2 show the results. Figure 6.3 shows the distribution of the count of Fiddlets invocations per task.

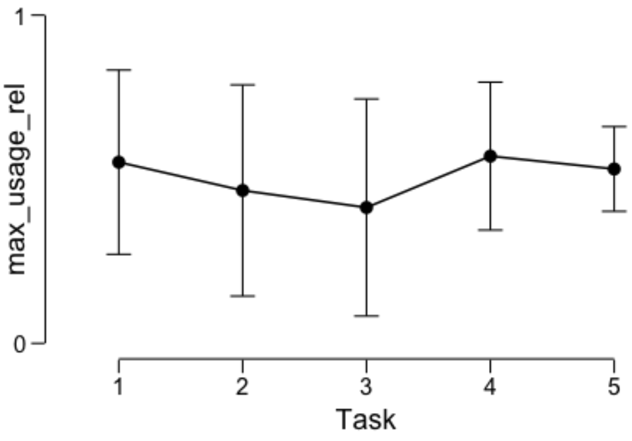


Figure 6.2: Maximum relative usage per task

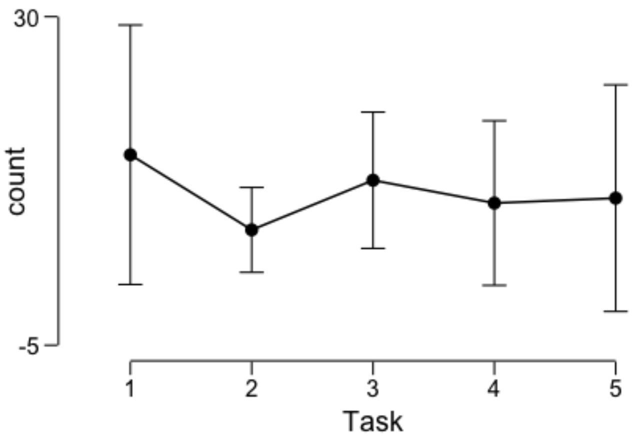


Figure 6.3: Number of Fiddlets invocations per task

We tested the tasks against each other using ANOVA for every measurement, but the analysis reported no significant differences. The values however indicate that participants used Fiddlets in a diverse manner.

Fiddlets usage did not vary significantly among tasks.

6.2.2 Programmer Performance

TCT and TCR were used as measures of programmer performance.

We use the "Task Completion Time" (TCT) as an indicator of programmer performance. The TCT was measured as the time in seconds between the beginning and the handing in of a task. We relate lower TCT to greater programmer performance. We considered the number of successfully completed tasks ("Task Completion Rate", TCR) as another factor for programmer performance: more completed tasks meaning more productivity.

We expected the participants of the Fiddlets condition to perform better in solving the tasks than participants in the Control condition. In order to evaluate our assumptions we stated the following hypotheses:

Hypotheses

1. Participants using Fiddlets will be able to solve more tasks than participants in the Control group (higher TCR in the Fiddlets condition).

2. Participants using Fiddlets will solve tasks faster than participants in the Control group (lower TCT in the Fiddlets condition).

We removed the data points of tasks that were not completed within 20 minutes and calculated averages and confidence intervals for both hypotheses. Table 6.4 shows the results for task completion rate.

Group	Mean	95% CI
c	4.29	[3.73, 4.85]
f	3.71	[2.69, 4.74]

Table 6.4: "Task Completion Rate": number of successfully solved tasks by each participant

The control grouped reached a higher average TCR.

We see that participants in the Fiddlets group were able to solve an average of 3.73 task, whereas participants in the Control group solved an average of 4.29 tasks. Participants in the Control group were therefore able to solve more tasks on average than participants in the Fiddlets group. However, running a t test against these values shows that this difference is not significant ($t(12) = 0.961, p = 0.356$).

Next up, we look at the TCTs. Table 6.5 shows the results for each condition, grouped by task number. Figure 6.4 shows plots of the mean values and confidence intervals for each task.

Task	Group	N	Mean (s)	SD	95% CI (s)
Task 1	c	3	607.7	165.2	[420.76, 794.64]
	f	3	758.7	238.3	[489.04, 1028.36]
Task 2	c	7	346.1	206.3	[193.27, 498.93]
	f	5	499.1	198.1	[325.46, 672.74]
Task 3	c	5	639.8	271.2	[402.09, 877.51]
	f	6	531.9	227.4	[349.95, 713.85]
Task 4	c	7	560.1	278.2	[354.01, 766.19]
	f	6	506.3	127.4	[404.36, 608.24]
Task 5	c	7	254.2	266.7	[56.63, 451.77]
	f	6	381.2	166.1	[248.29, 514.11]

Table 6.5: "Task Completion Time" for conditions and tasks.

Participants in the Fiddlets group solved Task 3 and Task 4 on average faster than participants in the Control group. They were on average 107,9 seconds faster in Task 3 and 53.8 seconds faster in Task 4. Both values are within the standard deviation. Participants of the Fiddlets group solved Task 1, 2 and 5 slower than participants in the Control group. The differences here are 151.0 seconds for Task 1, 153.0 seconds for Task 2 and 127.0 seconds for Task 5. The differences are again within the standard deviation. We again used the t test to check if the differences are significant (Table 6.6). The data shows no significant difference in task completion time for each task.

Fiddlets participant were faster in Tasks 3 and 4, Control participants were faster in Tasks 1,2 and 5, differences are not significant.

When we look at the demographic data of our participants (6.1 "Participants"), we see that they show a huge variation in programming experience (between one and 30 years) as well as JavaScript experience (0.5 to 10 years). High standard deviations in both programming (6.9 years) and JavaScript (3.5 years) experience suggest that our par-

High variation in programmer experience could explain the missing significance.

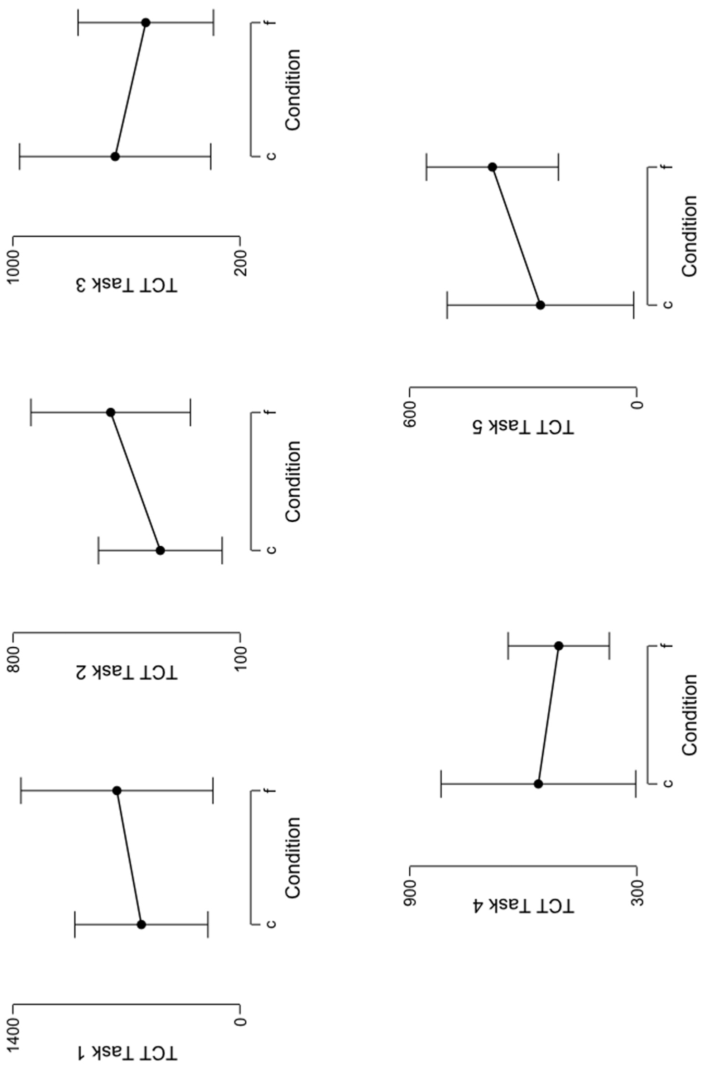


Figure 6.4: "Task Completion Time" in seconds plotted for all tasks. Graphs show mean TCT ± 95% CI.

Task	t	df	p
Task 1	-0.902	4	0.418
Task 2	-1.287	10	0.227
Task 3	0.719	9	0.491
Task 4	0.435	11	0.672
Task 5	-1.007	11	0.335

Table 6.6: *t* test results for TCT, assuming a difference between conditions (Fiddlets vs. Control)

ticipants group represents a vivid mixture of professional as well as novice programmers. With respect to the comparably low overall number of participants in each condition (7 participants per condition) we became curious what insights the data would provide if we compared not only conditions, but additionally include the skill level of participants into our evaluation.

We grouped participants by experience.

We used the programming and JavaScript experience scores and calculated an experience score for each participant. The experience score is calculated as follows:

$$\begin{aligned} xp\_score(participant) = {} & w_{prog} \cdot \frac{prog\_xp_{participant}}{age_{participant}} \\ & + w_{js} \cdot \frac{js\_xp_{participant}}{prog\_xp_{participant}} \end{aligned}$$

with $prog\_xp_{participant}$ being the participant's programming experience in years, $js\_xp_{participant}$ being the participant's JavaScript experience in years and $age_{participant}$ being the participants age, again in years. w_{prog} and w_{js} are used to weight the respective part of the formula. The first part of the formula determines the programming experience of a participant relative to her age. The second part determines the JavaScript experience of a participant relative to programming experience in years. The formula rates participants with only few years of programming experience exclusively in JavaScript high for relative JavaScript experience, but low on overall relative programming experience. A participant with years of programming experience, that had just recently started using JavaScript, would score high for overall relative programming experience, but low for relative JavaScript experience. Using the weights, the formula can be adjusted to either prefer relative programming experience over relative JavaScript experience or vice versa.

JavaScript is easy accessible, but contains concepts unfamiliar to object oriented programming.

The fact that JavaScript combines sophisticated styles of programming—e.g., object oriented programming, structured programming—makes it easily accessible for programmers who are already familiar with those programming styles. The dynamic type system and lack of debugging tools however promote the introduction of errors that are quickly overlooked by inexperienced JavaScript programmers and only show up at program execution. We therefore used 0.4 as value for w_{prog} and 0.6 for w_{js} in order to value JavaScript experience slightly above overall programming experience.

We used the median xp_score to classify participants.

After calculating the xp_score for each participant we used the median xp_score (0.2973) and the condition to classify the participants. Participants with an xp_score equal to or higher than the median were classified as *advanced* participants, whereas those with a score below the median were classified as *beginner* participants. Table 6.7 shows the distribution of participants according to our classification. *f_beginner* and *c_beginner* contain those who were classified as *beginner* in their condition, whereas *f_advanced* and *c_advanced* contain those classified as *advanced* in their condition.

Classification	Frequency	Percent
f_beginner	5	35.7
c_beginner	2	14.3
f_advanced	2	14.3
c_advanced	5	35.7

Table 6.7: Classification of the participants according to their xp_score

Distribution among beginner and advanced participants indicates a bias towards the Control group

The distribution of *beginner* and *advanced* participants between the Fiddlets and Control condition suggests an inequality of skill between those groups, which could explain the low performance of the Fiddlets group in the user study.

We used the classification as factors for an ANOVA analysis against the task completion times to see whether the data would support our suspicion. The results of the analysis are shown in Table 6.8. Again, the data shows no significant differences for task completion time.

Task	df	F	p
Task 1	2	0.309	0.755
Task 2	3	0.829	0.514
Task 3	3	2.398	0.154
Task 4	3	0.198	0.895
Task 5	3	0.594	0.634

Table 6.8: Results of ANOVA against TCT, using the classification based on the median xp_score as factor

In a last attempt we looked at the differences in reported preliminary knowledge across the Fiddlets group and the Control group. Using the data from Table 6.2 in Section 6.1 "Participants", we observed that the participants of the Control group on average reported a higher preliminary knowledge of regular expressions (3.857, mean value) than the participants in the Fiddlets group (3.00, mean value).With $t(12) = 2.52, p < 0.05$ this difference is significant. Participants of the Fiddlets group on the other hand reported on average a higher preliminary knowledge regarding the Unit Testing ($M = 3.0000$), Live Coding ($M = 3.286$) and Node.js ($M = 2.429$), with the difference in Live Coding being significant ($t(12) = 3.98, p < 0.05$). Since we had no Live Coding specific tasks, we don't consider this to be an advantage for solving the tasks. Both groups reported on average the same preliminary knowledge regarding mustache.js ($M = 1.714$), showing that neither of the groups had any advantage in solving the mustache.js tasks, which accounted for more than half of the tasks. Table 6.9 lists the significance values of reported preliminary knowledge.

Control group participants reported a significantly higher knowledge of Regular Expressions.

	t	df	p
RegExp	2.52	12	0.027
Unit Testing	-0.40	12	0.696
Live coding	-3.98	12	0.002
Node.js	-0.63	12	0.543
mustache.js	0.00	12	1.000

Table 6.9: t-values for reported preliminary knowledge, assuming no difference

6.2.3 User Reception and Observations

SUS: 85.71

Fiddlets received an average SUS score of 85.71 ($SD = 8.63$, 95% CI $[79.32, 92.11]$), which according to Bangor et al. [2008] maps to an adjective rating of "Excellent". Although this value has no influence on programmer performance, the SUS shows us that interacting with Fiddlets was not perceived as cumbersome or disruptive and was easy to understand. It also shows us that apparently no major bugs were introduced into the prototype that made using the prototype awkward.

Despite the good SUS score, we identified areas of improvement of the prototype through observation and participant interviews. We will briefly discuss the most striking observations.

Participants largely ignored interactive elements in the visualisations.

None of the participants in the Fiddlets condition used the interactive elements of the `split` and `splice` visualisation. In case of the `split` visualisation, some participants tried to change the number of elements in the returned array by directly clicking the last element they wanted to include in the output array. Participants were so heavily convinced that this was the primary interaction to select the size of the return array, that they tried double clicking the element in question after single clicking it did not show any effect. In case of the `splice` visualisation, the participants preferred to directly interact with the parameters of the method. An explanation for this could be that the effort that is needed to move from the keyboard to the mouse does not outweigh the gains of using our mouse-oriented interaction to manipulate the parameters. Given the size and number of parameters that needed to be entered in the functions, the cost of initially positioning the text cursor in the parameter list of the method and from thereon using the keyboard to change them is much lower than the cost to switch to the mouse and precisely point and drag the interactive elements.

Interactive visualisations required a higher effort than changing parameters manually.

Some of the participants in the Fiddlets group expected that changes made in the context editor would also be applied to the main editor. One of the reasons we did not implement this feature was the assumption that programmers would use the context editor to experiment with the code. We assumed that applying changes in the context editor directly to the main editor would hamper programmers from experimenting freely, as they might fear to break the program. However, contrary to our assumption we observed that participants tended to write experimental code in the main editor and used Fiddlets to quickly execute their experimental code and observe the outcome in the visualisation. One participant scattered log commands throughout the program and tried to use fiddlets to execute them in-place. Since we did not consider such an interaction, the prototype did not offer a visualisation for log commands.

Participants expected changes in the context code to be reflected in the main editor.

Another possible explanation for the low rate of experimentation we observed among participants could be a missing interaction for copying additional lines of code from the main editor into the context code. An illustrating example:

> In Task 5, a participant opened Fiddlets in the line with the `splice` method call. At that point the context editor contained the `weatherInfoCSV` variable initialisation and the current line of code with the `split` method call on the object referenced by said `weatherInfoCSV` variable. The participant entered the first two parameters (`start` and `deleteCount`) for the `splice` method. She then tried to enter the `csvHeader` string as the third parameter, which was defined in the main editor, but not copied into the context editor. It was therefore not available during execution of

Variables initialised in the main program were not automatically available in the context code.

> the context code, which then resulted in a reference error that was being displayed in the visualisation. Confused by this result, the participant started using a literal value for the third parameter in order to observe how the `splice` method behaves, instead of copying the initialisation of the `csvHeader` variable.

The context editor could suggest to copy missing values from the main code.

A solution to this problem would be to search the context code for the usage of variables that have been defined in the main editor, but not in the context editor. Upon finding such a usage, the it could offer the user to copy the initialisation from the main editor. This behaviour could also be integrated into an autocomplete function in the context editor, where when the user types a variable, the autocomplete would offer her to instead copy the initialisation of that variable from the main editor.

6.3 Summary

Participants made great use of Fiddlets, but did not perform significantly different from the Control group.

We evaluated our user study by looking at the usage and performance data of the participants. We saw that throughout the tasks, participants in the Fiddlets condition actively used the tool in diverse ways. From our performance data we could not find any proof for our hypotheses. Neither did the data show a higher task completion rate for participants in the Fiddlets condition, nor did it support our hypothesis that participants using Fiddlets finished the tasks faster than the participants in the control group.

We were however able to identify minor flaws in the design of Fiddlets and received interesting insights into how participants interacted with it. The next chapter will wrap up the ideas and future work that we derive from these insights.

Chapter 7

Summary and future work

This chapter summarises our contributions made to the field of HCI and takes a final look at the results of our user study. We discuss the efforts needed to turn the prototype we developed into a mature program that could be used in everyday work. We conclude with opportunities for future research in the direction of Fiddlets.

7.1 Summary and contributions

We introduced Fiddlets as a novel interaction with source code...

With this theses we introduced Fiddlets, a novel interaction technique that enables programmers to quickly execute small parts of a program without the need to run it as a whole. We introduced the notion of an execution context that captures the minimal amount of related code that is necessary to execute certain program parts. Besides that we described how runtime values can be provided for such contexts by capturing them through the instrumented execution of the complete program. We enhanced the display of the results of context execution with interactive visualisations that are tailored to the line of interest of the context.

... and described the architecture necessary to enable this interaction.

We provided an architectural overview over the components that are necessary to implement Fiddlets and described the parts that are the most critical: generation of the context, capturing runtime values and providing interactive visualisations.

We built a proof-of-concept prototype and evaluated its performance in a user study.

To show the feasibility of our idea, we built a prototypical implementation of Fiddlets. Using the prototype we conducted a between-subjects user study in which we investigated both user acception and the effect on programmer performance. An average SUS score of 85.71 and our evaluation of the usage data showed that Fiddlets was well accepted by the participants. The performance data measured in the study showed no significant differences between those participants who used Fiddlets and those who did not. Some aspects of the data showed indications of an unbalanced distribution of programming experience between the two groups, favouring the Control group.

7.2 Future work

Despite the usage and performance data we identified various areas of improvement to our prototype during the user study. These and other insights of the study offer opportunities for future work in the direction of Fiddlets.

Improved context generation algorithm

In 4.2.1 "Context" we explained our vision of a mature context generation algorithm. The algorithm that was used in the prototype had several limitations which we outlined in 5.1.1 "Limitations". It would be interesting to implement a more sophisticated algorithm that closer matches the features we envisioned and to evaluate its performance in terms of resource consumption and execution time.

More visualisations

For the user study, we restricted the number of visualisations to those that were related to the code in the tasks (5.1.2 "Visualisations". JavaScript provides a plethora of built-in types and function worth a visualisation. Other examples of functions that could benefit from elaborate visualisations can be found in jQuery. We already explored some of the possibilities with our prototypes in 3.2 "Second Iteration:

Runtime Data Widgets". It would be interesting to see how a working implementation of those visualisations in Fiddlets would perform against traditional development environments.

Trace collector implementation

The only part of our architectural sketch (4.2 "Implementation") missing in the prototype for the user study was the collection of runtime traces and variable values. Research on Live Coding (2.3 "Live Coding") already showed that continuous execution and the capturing of program traces is feasible with the capabilities of modern computers. Having a working implementation that collects variable values from both program and unit test execution would allow to design user studies with more open ended implementation tasks.

More user studies

Since the results of our user study showed no significance, we encourage the design of further user studies on the interaction provided with Fiddlets. Besides the aforementioned open ended tasks, we could imagine task designs that are more focused on single functions or types and provide narrower stimuli. A study could for example ask the participants to write regular expressions to separate various sets of strings. Another study could ask the participants to implement functions used along with the `Array.prototype.map` method to perform arbitrary object transformations.

We still believe that the interaction we implemented with Fiddlets can be a valuable tool for programmers in their daily work. With these hopefully inspiring ideas we hand our vision on to future research, looking forward to a time when great tools will allow us to develop even greater software.

Appendix A

Live Autocompletion Evaluation: Implementation Requirements

A.1 Trivia Game Requirements

Implement this game of trivia with any number of players.
No GUI needed, just log what is happening on the console.

A.2 Board Setup

- There are 12 fields numbered from 0-11
- The players traverse the fields in sequential order
- After field 11 comes field 0

A.3 Questions

Every field links to one out of four question categories

- 0,4,8 are “Pop”
- 1,5,9 are “Science”
- 2,6,10 are “Sports”
- 3,7,11 are “Rock”

There are 50 questions for each category

- The questions consist only of their category plus their index (e.g. “Pop 1”) instead of being actual questions
- Questions are in sequential order

A.4 Rules

- Players take turn in the order they were added to the game
- On each turn, a player throws a dice (1-6), then moves the according number of fields and answers the question corresponding to the field she lands on
- If the question is answered correctly, the player wins a gold coin
- If the question is answered incorrectly, the player is put into penalty
  - The player can get out of penalty by throwing an odd number when it is his turn. If she does so, she moves to the field according to the number she threw
  - If the player throws an even number, she does not move, is not asked a question and has to wait for the next turn
- The first player to have 6 golden coins, wins the game

A.5 Things to log

- Player creation
- Dice rolls
- Moves
- Questions asked
- Correct or incorrect answers
- Entering and leaving penalty
- Received gold coins

Appendix B

Data Processing Tasks: Source Code

```javascript
var fs = require("fs");

function readWeatherInfoFromFile(filepath) {
    var sampleWeatherDataRaw = fs.readFileSync(
        filepath, "utf8");
    var buffer = new Buffer(sampleWeatherDataRaw, "
        base64");
    var sampleWeatherData = buffer.toString();

    return sampleWeatherData;
}

var weatherJSON = JSON.parse(readWeatherInfoFromFile
    ("./info.dat"));

var i;
var myWeatherInfo = []
for(i = 0; i < weatherJSON.list.length; i++) {
  var info = {};
  info.mtemp = weatherJSON.list[i].main.temp;
  info.wdesc = weatherJSON.list[i].weather[0].
      description;
  info.atmpress = weatherJSON.list[i].main.pressure;
  info.dt_txt = weatherJSON.list[i].dt_txt;
  info.wnd = {
    v: weatherJSON.list[i].wind.speed,
    dir: weatherJSON.list[i].wind.deg
  };
  myWeatherInfo.push(info);
}

```

```
var weatherInfoCSV = myWeatherInfo.map(
    buildWeatherInfoCSVLine);
exportWeatherAndPrintCurrent();

function buildWeatherInfoCSVLine(weather) {
  var date = new Date(weather.dt_txt);
  var localeTimeString = date.toLocaleTimeString();
    return localeTimeString + "," + weather.mtemp +
        "C," + weather.wdesc + "\n";
}

function exportWeatherAndPrintCurrent() {
    var csvHeader = "time,temperature,description\n"
        ;
    var weatherNow = weatherInfoCSV.splice(0,1,
        csvHeader);

    fs.writeFile("forecast.csv", weatherInfoCSV.join
        (""), function(err) {
        if(err) {
            console.error(err);
        }
    });

    console.log("Weather now: " + weatherNow);
}
```

Appendix C

User Study Task Descriptions

Task 1

In the following tasks, you will be working on mustache.js, a JavaScript templating system. A demo on how mustache.js works can be found at https://mustache.github.io/#demo.

For this task you will complete the function `escapeRegExp`, which escapes characters that are special characters in regular expressions. For a given string, the function should return a string with the regular expression symbols escaped. The following table provides example input and outputs for the function:

Input	Output
`{`	`\\{`
`{{`	`\\{\\{`
`|{`	`\\\|\\{`
`{ name }`	`\\{ name \\}`

Input	Output
`?`	`\\?`
`[%`	`\\[%`
`<[{`	`<\\[\\{`
`**`	`\\*\\*`

Your task is to write a regular expression that matches the reserved special characters as shown in the table.

1. Read this instruction carefully. If you have any questions left, ask the study supervisor.
2. When you feel prepared for the task, click the „Start Task" button.
3. Go to line 37. This is where the `escapeRegExp` function is located.
4. Fill in the missing regular expression
5. If you feel confident about your solution, click the „Hand in task" button

You are allowed to use any online resources you might need to fulfill the task, except the original source code of mustache.js.

Hint: When used in the second argument of the string replace method, "$&" will be replaced by the substring matched by the first argument of the replace method.

Task 2

In the next task you will complete the `escapeHTML` function, which escapes string used in HTML documents. The replacer function is already implemented. However, the regular expression that should match HTML special characters (<, >, ‘, “, &, /) is not correct.

Your task is to fix regular expression in the escapeHTML function.

1. Read this instruction carefully. If you have any questions left, ask the study supervisor.
2. When you feel prepared for the task, click the „Start Task“ button.
3. Go to line 60. This is where the `escapeHTML` function is located.
4. Fix the regular expression in line 61.
5. If you feel confident about your solution, click the „Hand in task“ button

You are allowed to use any online resources you might need to fulfill the task, except the original source code of mustache.js.

Hint: you can test your implementation by running `npm test` on the command line inside the tasks folder.

Task 3

For the next task, a bug was introduced into the compileTags function, which takes an array with the opening and closing tag or a string consisting of the opening and closing tag separated by whitespace, and creates regular expressions to search for these exactly these tags in a template.

You can observe the bug by running the test suite (using `npm test`): the test suites `render-tests.js` and `parse-tests.js` are failing.

Your task is to fix the error in the `compileTags` function.

1. Read this instruction carefully. If you have any questions left, ask the study supervisor.
2. When you feel prepared for the task, click the „Start Task“ button.
3. Go to line 128. This is where the `compileTags` function is located.
4. Find the bug inside the function and fix it.
5. If you feel confident about your solution, click the „Hand in task“ button

You are allowed to use any online resources you might need to fulfill the task, except the original source code of mustache.js.

Task 4

For the next tasks you will work on a small node.js application that reads weather forecast information in a proprietary format from a file and transforms parts of it into csv format.

After the weather forecast information is read and decoded, it is transformed into an intermediate array of JavaScript objects, from which you will extract the specified fields, which will make up your csv data, by using the array `map` function.

For every weather forecast object in the array, the csv should contain the following information:

1. The time of the forecast
2. The temperature with the unit symbol (°C)
3. The description of the forecast

An example cvs line could look like the following:

“2015-11-01 12:00:00,13.36 °C,light rain”

Your task is to implement the function that is used by map to create csv lines.

1. Read this instruction carefully. If you have any questions left, ask the study supervisor.
2. When you feel prepared for the task, click the „Start Task“ button.
3. Run the application by calling `node weather.js`. It will create a new file `forecast.csv`.
4. Go to line 40. This is where `map` is called on the forecast array.
5. Implement the `buildWeatherInfoCSVLine` function to return a csv string as specified before
6. If you feel confident about your solution, click the „Hand in task“ button

You are allowed to use any online resources you might need to fulfill the task.

Task 5

The csv you generated in the task before needs a header that describes its fields. The program should also exclude the first forecast from the csv and instead write it to the terminal.

Removing and replacing elements of an array in JavaScript is done with the `splice` function. Use `splice` to accomplish this task.

Your task is to implement the function that is used by map to create csv lines.

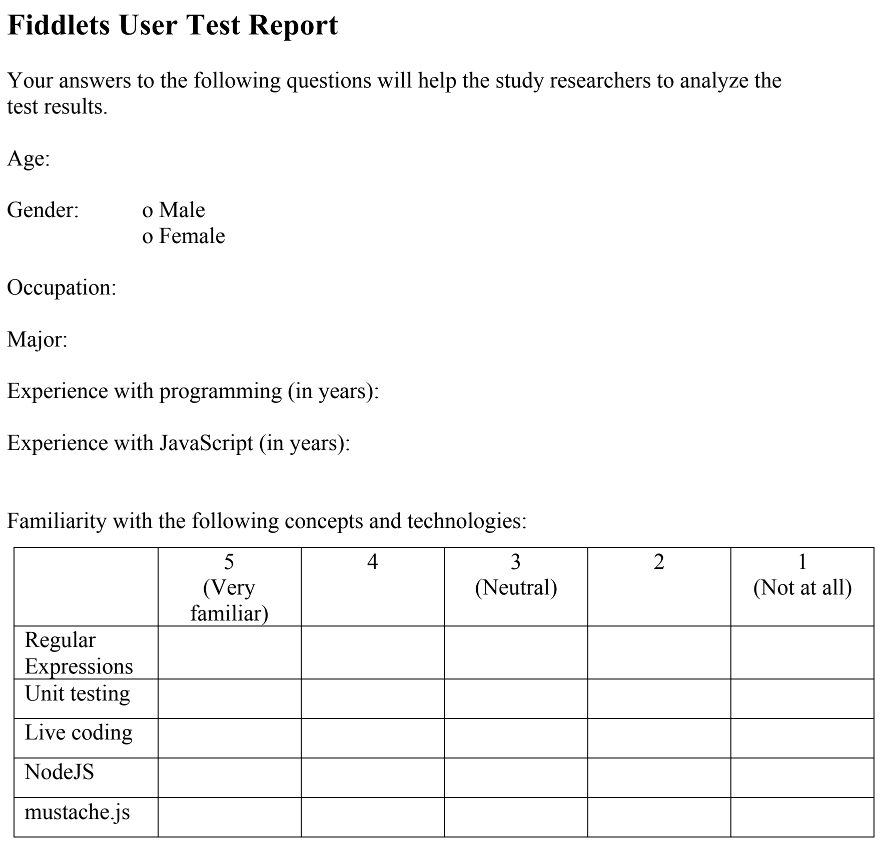
1. Read this instruction carefully. If you have any questions left, ask the study supervisor.
2. When you feel prepared for the task, click the „Start Task“ button.
3. Run the application by calling `node weather.js`. It should create a file called forecast.csv.
4. Line 38 defines the header line you should include in your csv.
5. In line 39 you will find the splice call that should replace the first element with the header line. Fix this line to behave as explained above.
6. If you feel confident about your solution, click the „Hand in task“ button

You are allowed to use any online resources you might need to fulfill the task.

Appendix D

Questionaires

D.1 Pre-Questionaire



Fiddlets User Test Report

Your answers to the following questions will help the study researchers to analyze the test results.

Age:

Gender: o Male
o Female

Occupation:

Major:

Experience with programming (in years):

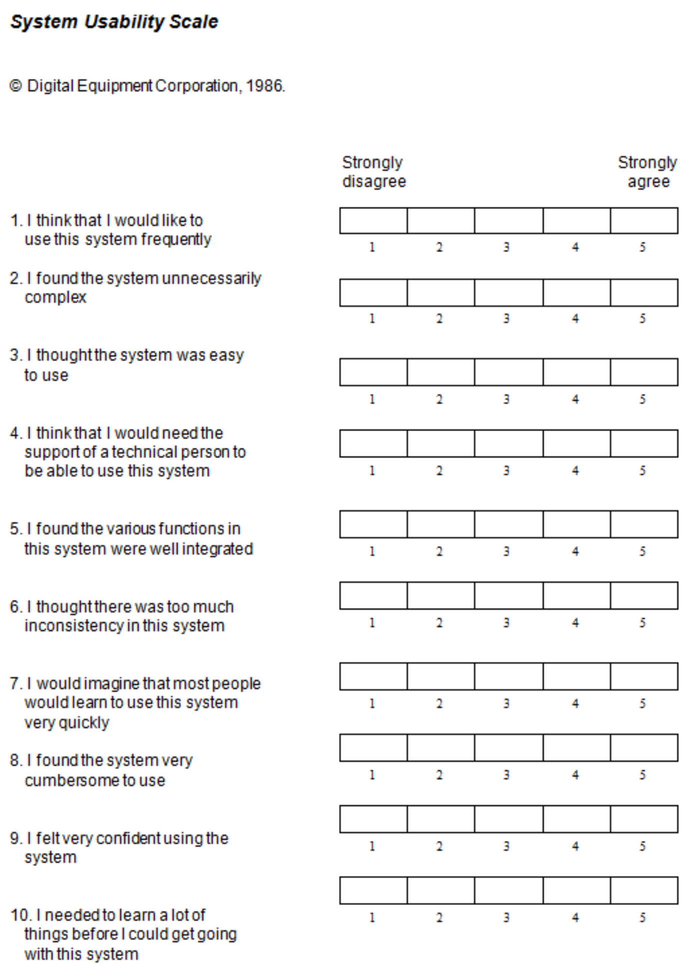
Experience with JavaScript (in years):

Familiarity with the following concepts and technologies:

	5 (Very familiar)	4	3 (Neutral)	2	1 (Not at all)
Regular Expressions					
Unit testing					
Live coding					
NodeJS					
mustache.js					

Figure D.1: *Questionaire handed out before the tasks*

D.2 SUS Scale



System Usability Scale

© Digital Equipment Corporation, 1986.

	Strongly disagree				Strongly agree
1. I think that I would like to use this system frequently	1	2	3	4	5
2. I found the system unnecessarily complex	1	2	3	4	5
3. I thought the system was easy to use	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this system	1	2	3	4	5
5. I found the various functions in this system were well integrated	1	2	3	4	5
6. I thought there was too much inconsistency in this system	1	2	3	4	5
7. I would imagine that most people would learn to use this system very quickly	1	2	3	4	5
8. I found the system very cumbersome to use	1	2	3	4	5
9. I felt very confident using the system	1	2	3	4	5
10. I needed to learn a lot of things before I could get going with this system	1	2	3	4	5

Figure D.2: *Questionaire handed out after all tasks were completed: SUS scale by Brooke [1996]*

Bibliography

Aaron Bangor, Philip T Kortum, and James T Miller. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, 24(6):574–594, 2008.

Earl T. Barr and Mark Marron. Tardis: Affordable time-travel debugging in managed runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '14, pages 67–82, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660209. URL `http://doi.acm.org/10.1145/2660193.2660209`.

Ewgenij Belzmann. Utilization and visualization of program state as input data in a live coding environment. Diploma thesis, RWTH Aachen University, Aachen, April 2013.

Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1589–1598, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-246-7. doi: 10.1145/1518701.1518944. URL `http://doi.acm.org/10.1145/1518701.1518944`.

John Brooke. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

Thomas G. Evans and D. Lucille Darley. Debug&mdash;an extension to current online debugging techniques. *Commun. ACM*, 8(5):321–326, May 1965. ISSN 0001-0782. doi:

10.1145/364914.364952. URL `http://doi.acm.org/10.1145/364914.364952`.

Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 653–663, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568250. URL `http://doi.acm.org/10.1145/2568225.2568250`.

Zhongxian Gu, Earl T. Barr, Drew Schleck, and Zhendong Su. Reusing debugging knowledge via trace-based bug search. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 927–942, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384684. URL `http://doi.acm.org/10.1145/2384616.2384684`.

Stephen José Hanson and Richard R. Rosinski. Programmer perceptions of productivity and programming tools. *Commun. ACM*, 28(2):180–189, February 1985. ISSN 0001-0782. doi: 10.1145/2786.2791. URL `http://doi.acm.org/10.1145/2786.2791`.

BjÃ¶rn Heinen. A live coding editor. Bachelor's thesis, RWTH Aachen University, Aachen, December 2012.

Peter Henderson and Mark Weiser. Continuous execution: The visiprog environment. In *Proceedings of the 8th International Conference on Software Engineering*, ICSE '85, pages 68–74, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press. ISBN 0-8186-0620-7. URL `http://dl.acm.org/citation.cfm?id=319568.319582`.

Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2. doi: 10.1145/1062455.1062491. URL `http://doi.acm.org/10.1145/1062455.1062491`.

Lingxiao Jiang and Zhendong Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 184–193, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321660. URL `http://doi.acm.org/10.1145/1321631.1321660`.

Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368130. URL `http://doi.acm.org/10.1145/1368088.1368130`.

Adrian Kuhn. On extracting unit tests from interactive live programming sessions. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1241–1244, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL `http://dl.acm.org/citation.cfm?id=2486788.2486974`.

Joachim Kurz. Evaluating developer strategies in a live coding environment. Master's thesis, RWTH Aachen University, Aachen, August 2013.

Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 185–194, New York, NY, USA, 2010a. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806829. URL `http://doi.acm.org/10.1145/1806799.1806829`.

Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 8:1–8:6, New York, NY, USA, 2010b. ACM. ISBN 978-1-4503-0547-1. doi: 10.1145/1937117.1937125. URL `http://doi.acm.org/10.1145/1937117.1937125`.

Bil Lewis. Debugging backwards in time. *arXiv preprint cs/0310016*, 2003.

Tom Lieber, Joel R. Brandt, and Rob C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2481–2490, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2557409. URL `http://doi.acm.org/10.1145/2556288.2557409`.

Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 480–486, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co. ISBN 0-201-84705-1. doi: 10.1145/223904.223969. URL `http://dx.doi.org/10.1145/223904.223969`.

Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 592–615, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8. doi: 10.1007/978-3-540-70592-5_25. URL `http://dx.doi.org/10.1007/978-3-540-70592-5_25`.

Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 53–62, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509585. URL `http://doi.acm.org/10.1145/2509578.2509585`.

Stephen Oney and Joel Brandt. Codelets: Linking interactive documentation and example code in the editor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2697–2706, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1015-4. doi: 10.1145/2207676.2208664. URL `http://doi.acm.org/10.1145/2207676.2208664`.

Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806598. URL `http://doi.acm.org/10.1145/1806596.1806598`.

David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 281–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2007-3. URL `http://dl.acm.org/citation.cfm?id=951952.952340`.

Steven L. Tanimoto. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, pages 31–34, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-6265-8. URL `http://dl.acm.org/citation.cfm?id=2662726.2662735`.

Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. Snipmatch: Using source code context to enhance snippet retrieval and parameterization. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 219–228, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1580-7. doi: 10.1145/2380116.2380145. URL `http://doi.acm.org/10.1145/2380116.2380145`.

E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, CHI '97, pages 258–265, New York, NY, USA, 1997. ACM. ISBN 0-89791-802-9. doi: 10.1145/258549.258721. URL `http://doi.acm.org/10.1145/258549.258721`.

Typeset December 4, 2015