RWTH AACHEN UNIVERSITY

A Live Coding Editor

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

by
Björn Heinen

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Bernhard Rumpe

Registration date: 12.06.2012
Submission date: 12.09.2012

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

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

Aachen, September 2012
Björn Heinen

Contents

List of Figures

Abstract

Live Coding seeks to provide an environment that helps programmers to understand the internal behavior of a program with the help of examples. The goal of this thesis is to create an editor that takes exemplary parameters for a function and continuously executes the current version of the code with those parameters. To help the programmer understand the control flow of the program, not only is the according `return` value of the function continually updated, also the environment shows what happens with the exemplary parameters inside the code between input and output. A qualitative study with 6 users has been conducted to find the best way of displaying the link between the code and the constantly updated evaluations. The result, an environment that keeps a traditional editor on the left-hand side and an evaluated version of the current code (keeping syntax highlighting and indents) on the right-hand side, has been implemented in form of an extension for the open source editor Brackets. Finally, a study with 20 users has been conducted to find out if there is an improvement in the debugging speed of experienced programmers. Although the user study yielded non-significant results, there is evidence that there is an improvement in the debugging speed and that significant results can be obtained.

Acknowledgements

First of all, I would like to thank everybody who participated in my user studies,
Your valuable informations and feedback enlarged my opportunities.

Furthermore, Jan-Peter Krämer deserves my thanks,
Without whom this work would consist of nothing but blanks.

I would like to thank all my good friends, too,
Whose long-standing amity I am looking forward to.

Last but not least, special thanks to my family, particularly my parents,
And, unfortunately, this is where my English ends.

Chapter 1

Introduction

Most of todays's IDEs make a clear separation between editing the code and debugging it. To switch from programming to debugging and vice versa, usually the programmer has to use another mode in the IDE. This way, not only looses the programmer a significant amount of time through the emerging overhead, but the working flow gets interrupted, too [Edwards, 2004].

Overhead results from the separation between editing and debugging

Permanently having to switch between modes has another drawback. The probability of loosing time because the programmer continues coding after a bug has been introduced increases considerably. [Saff and Ernst, 2004b] reported on this phenomenon: They said that larger *Ignorance Times* (the time between introducing a bug and becoming aware of it) yield larger *Fix Times* (the time between becoming aware of a bug and fixing it). Therefor, an IDE should provide tools to keep the Ignorance Time as well as the overhead produced by switching between different modes as low as possible.

Wasted development time through Ignorance Time and Fix Time

When Bret Victor gave his speech "Inventing on Principle", he was talking about something very similar: Direct delay-free feedback. To show what he was talking about, Victor presented an editor that continuously executes the code as soon as it has been entered and not only shows the actual

result but also reveals information about its internal behavior. Figure 1.1 shows how his interface looks like using the example of the binary search algorithm.

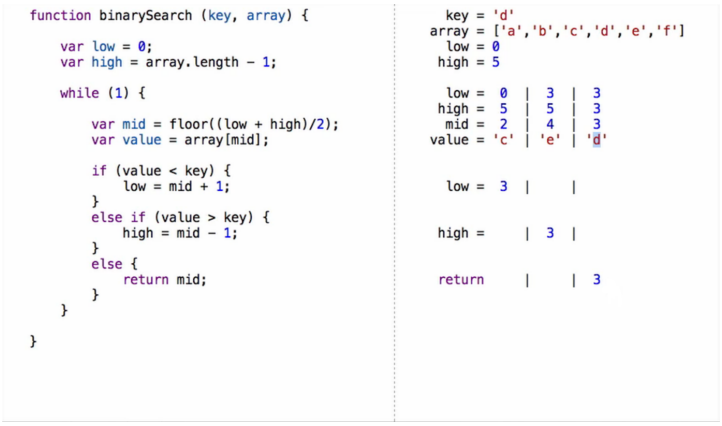


Figure 1.1: A Live Coding Editor presented by Bret Victor [Victor]

Bret Victor presented an editor that gives immediate information about a program's internal behavior

On the left-hand side, the code can be changed as usual. At the top of the right-hand side, exemplary parameters can be entered for which the code then gets executed. Further down, the values for each variable in each iteration and the return value are shown. For example, in the line `var high = array.length - 1;`, the editor says `high = 5` This way, the programmer does not have to imagine what happens between executing the code with exemplary parameters and getting back the according result but can actually see it. As mentioned above, this happens as the code gets edited, so there is no delay between a change and seeing its impact on the code. Figure 1.2 shows an example of how helpful this can be. The key the algorithm searches for has been changed to `'g'`, which is not in the array. Since the invariant of the loop is `1`, the algorithm can only terminate when the key has been found in the array. In this case, however, the loop iterates endlessly. The solution is to replace the invariant by `low` $\leq$ `high`. A solution that is obvious once the programmer sees that, in the fourth iteration, `low` gets greater than `high` and nothing else changes anymore.

The direct, immediate feedback Victor was talking about is the basic idea of my thesis. Since he designed his editor

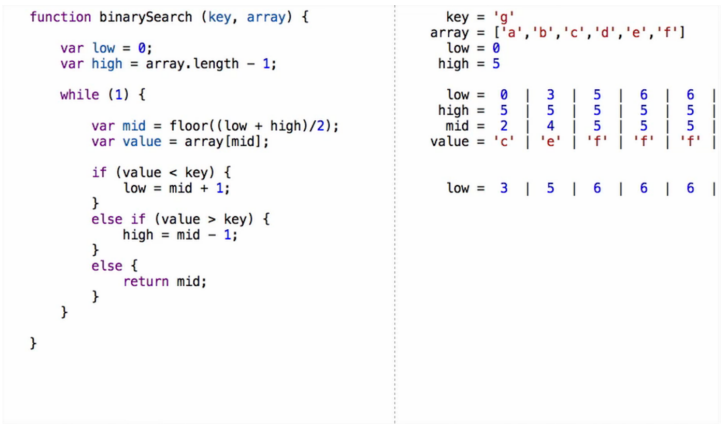


Figure 1.2: The `key` parameter has been changed

only to show what he was talking about in his speech, his prototype is very low fidelity and has some basic usability flaws (most importantly: Only 5 iterations can be displayed and the functionality is very limited). Furthermore, his editor has never been evaluated, neither in a qualitative user study, nor in a quantitative one. My thesis will pick up his idea, develop it and finally evaluate it. To do so, two iterations of the *DIA-cycle* (that is, Design-Implement-Analyze) have been done: We started with a literature research (see Chapter 2) to get an overview what research has already been done with regards to Live Coding. We then implemented a low fidelity prototype and evaluated it in a qualitative user study to find out which basic interface design provides as much analogy between the code and its evaluation (see Chapter 3). Afterwards, we implemented a software-prototype (see Chapter 4) to conduct the second, quantitative user study, which should reveal if the interface enhances the work flow significantly (see Chapter 5).

Overview of the thesis

Chapter 2

Related Work

The idea of maintaining a continuous connection between code and output is not not new. Already back in 1985, [Henderson and Weiser, 1985] stated that the classical Edit-(Compile-)Execute cycle, where program editing and execution are independent activities, is not optimal. They stated that the relation between input and output should continuously be kept consistent.

The idea of continuously executing code has already been presented in 1985

[Pane and Myers, 1996] proposed that a system should support incremental testing and feedback and help detect, diagnose and recover from errors. Although they were recommending heuristics to improve novice programming systems, these advices also help improving the productivity of experts.

Incremental testing and feedback improve productivity

This concurs with [Ko et al., 2004], who reported on learning barriers in programming systems. Among others, they reported on the so called *Understanding Barriers* and *Information Barriers*. Understanding Barriers happen when code behaves differently from what the user expected. If, for example, there is a compile-time error and the user cannot extract which part of the code is deemed wrong by the compiler from the error message, then this is an Understanding Barrier. In the user study Ko et al. conducted, most (34 of 38) of these Understanding Barriers were insurmountable.

2 learning barriers: Understanding Barriers and Information Barriers

Information Barriers are elements in the environment that make the internal behavior of the program hard to understand. They happen when a user builds a hypothesis about the internal behavior of the program but cannot use (or find) a tool the environment provides to verify that hypothesis. In Ko et al.'s user study, most of these understanding barriers (10 of 14) were insurmountable. To help the programmer, an IDE should reveal what a program did or did not do at compile or runtime and, in addition, it should be possible to inspect a program's internal behavior.

Current environments offer little help for understanding and adapting examples

Although the idea of helping the programmer understand the internal behavior of a program by example is not new, current environments only offer little help for understanding and adapting examples. Therefore, [Brandt et al.] have implemented *Rehearse*, an editor that provides immediate evaluation and infinite undo of execution. A screenshot of this editor can be seen in Figure 2.1. Although Rehearse embodies the immediate feedback

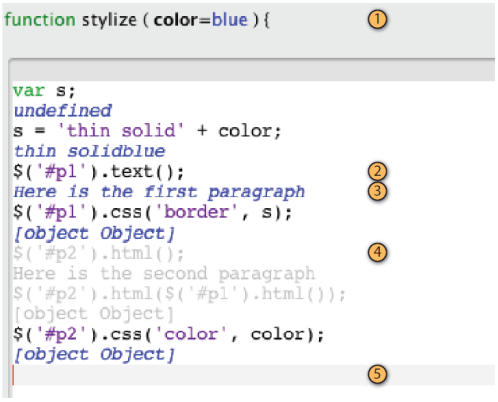


Figure 2.1: "A screenshot of the Rehearse editor. (1) The function declaration, parameter names, and current values; (2) a statement that has been executed and (3) the result of execution; (4) an undone statement; (5) the current line" [Brandt et al.]

Advantages and disadvantages of Rehearse

and the maximal *illumination* we are striving for, there are issues with it. First, Rehearse disturbs the user in his usual programming practices. For example, it is not possible to edit a function in the usual editor while it is open in Rehearse. This means that there are two ways of

editing a function (either the usual way with a standard editor or interactively with Rehearse) and they are mutual exclusive (this issue has also been reported as a result of the user study Brandt et al. conducted). We think it is best to integrate the interactive functionality into the standard one so that the user does not, again, have to switch between different modes. Additionally, Rehearse provides functionality to help the programmer explore alternative paths by undoing arbitrarily many executed lines while the goal of our editor is to show what happens in the code with exemplary parameters to help the programmer understand what internal behavior the code has. This way, the programmer can work uninterrupted and still get information about the behavior of his program.

Similarly, [Edwards, 2004] stated that it improves the work flow if the programmer does not have to leave the environment to see what result the execution of an example yields. He says, the best way to understand code is by thinking of concrete examples. To solve the problem that the programmer has to emulate the behavior of the program in his head, as much code as possible should be *illuminated* with examples. Edwards built a plug-in for Eclipse, for which a screenshot can be seen in Figure 2.2. The results on the

Examples help understanding code

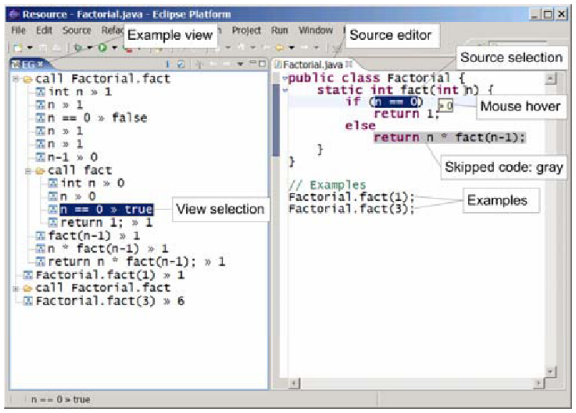


Figure 2.2: A screenshot of the plug-in presented by [Edwards, 2004]

left-hand side of the interface get updated as soon as there is a change in the code. From a theoretical point of view, this is exactly what we intend to do: The code is continu-

Idea and drawbacks of Edwards' plug-in

ously executed for exemplary parameters while the user is working on it. From a practical point of view though, this plug-in has several drawbacks. First, the analogy between code and evaluation is not very strong (e.g. not as strong as the one in Rehearse), so it is not as easy for the user to follow the control flow of the program as it should be. There are further usability issues: Skipped code is greyed out. This is a good idea since the user can see which part of the code has been executed and which part has not. The problem is to see for which parameters the code has been skipped. In Figure 2.2, for example, `fact` has been called with the values `1` and `3` and, in addition, subsequently with the parameters `0` and `2` (resulting from the recursive call of `fact` in the `else` statement). For which of the parameters the `else` statement has been skipped is not as obvious as it could be. Furthermore, it is suboptimal to call the function with exemplary parameters in the code itself. This way, a programmer cannot write code as usual and just extract the additional information but has to insert "foreign" code to be able to see what happens inside the code and then later has to remove the additional code again because it has nothing to do with the functionality the programmer wanted to write in the first place. Finally, Edwards presented the plug-in without evaluation of any kind, so a remaining task is to find out if or how much this kind of feedback improves the work flow (see Chapter 5).

How modifying software works

In [Boehm, 1979], it was reported that modifying software basically consists of three phases:

- Understanding the existing software
- Modifying it
- Revalidating the modified version

We believe that our interface helps improve (thereby speed up) this whole process (our second user study's goal was to find out whether this is true, see Chapter 5). This supposition is supported by [Erdogmus et al., 2005], where it is said that instant feedback whether new functionality

has been implemented properly helps reducing wasted development time.

Finally, our approach concurs with the idea of *Opportunistic Programming* [Brandt et al., 2008], [Brandt et al., 2009a], [Brandt et al., 2009b]. Brandt et al. say that frequent iteration is a necessary part of learning and understanding unfamiliar code. Therefore, an opportunistic programmer will select a tool that speeds up iteration. They also state that the Edit-Debug cycles in opportunistic programming are very short. About 50% of these cycles are shorter than 30 seconds and even 80% are shorter than 5 minutes. Our tool would help reducing the resulting overhead since the outcome is instantly visible without having to change modes or compile and execute.

Opportunistic programmers choose tools that speed up iteration

Chapter 3

First User Study

3.1 The Prototypes

The main goal of our first user study was to find out which basic interface design is the optimal one, especially with regards to the link between the code and the evaluated version of the code. It should work in a way such that the programmer does not get disturbed in his usual work but that the interface still provides as much information as accessible as possible.

Goal of the first user study

To find out how such an interface would look like, we implemented low fidelity prototypes for 5 different interfaces in Ruby. Basically these prototypes consisted of pictures that show how the interface would look like for some exemplary code with exemplary parameters. Each interface has been illustrated using 5 pieces of code as example (Figures 3.1-3.5). Note: The font used in all prototypes was supposed to emphasize that these prototypes are low fidelity so that the users had not inhibitions telling us fundamental flaws in the interface design.

5 low fidelity prototypes for Ruby have been implemented

3.1.1 Interface Version 1

Figure 3.1 shows the first interface with an implementation of the binary search algorithm. In this version of the interface, an evaluated version of the code is inserted in an additional section in the right half of the window. At the top, the exemplary parameters can be manipulated. Then, in the line `high=array.length-1`, the first evaluation takes place. On the right-hand side of the window, the right-hand side of the assignment has been replaced by the according evaluation: `5`. Afterwards, all values are replaced in a similar way. That is, in simple assignments, the right-hand side is substituted completely and in invariants, the condition has been computed up to the penultimate step. This way, e.g. `while(low <= high)` gets evaluated to `while(0 <= 5)` instead of `while(true)`.

Interface variant 1 divides the environment into two. The left-hand side is a usual editor and the right-hand side is the evaluated version of the code

EXCURSUS:
We also thought about showing what would happen inside an `if` statement even if it fails so that the user could see what the code could do alternatively. We discharged this idea however because of two problems: Firstly, we suspected that this might quickly become a confusing experience for the user. Secondly (and most importantly), the Halting Problem would have to be decidable. Suppose the following code had to be evaluated:

```
if(x>0){while(x!=0){x--;}}
else {while(x!=0){x++;}}
```

Then the interface would fall into an infinite loop (for any value of `x` except 0) although the code would not. For `x=1` for example, the actual code would only make one iteration in the first loop. The interface however would try to calculate the values of the second loop, too.

Further down in Figure 3.1(a), there is the following element:

This element will subsequently be called *iterator*. The iterator indicates which iteration of a loop is being shown. Pressing the right arrow increments the shown iteration (Figure 3.1(b)), pressing the left arrow decrements it, accordingly. Furthermore, the code for the `elsif` and the `else` statements is not visible on the right-hand side of the

window. The reason is that the first `if` statement holds and, therefore, the following ones are not executed. In Figure 3.1(b), the first `if` statement failed (greyed out), the second one holds and the third one is hidden again. This way, all code that gets executed is shown (corresponding to Jonathan Edwards who said that as much code as possible should be "illuminated"[Edwards, 2004]).

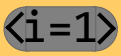
3.1.2 Interface Version 2

Figure 3.2 shows the second version of the interface with an in place implementation of the selection sort algorithm (note that each interface variant has been implemented with each algorithm). The differences to the previous interface variant are:

Interface variant 2 copies a not highlighted, evaluated version of the code to the right part of the window

- The parameter names are red to signal manipulability
- No syntax highlighting
- Evaluated code has no indents and is right aligned
- No dividing line

EXCURSUS:
Note that the iterator in this interface looks like follows:



This design reveals what variable the iterator refers to. This kind of iterator does not depend on the type of interface but on the type of loop. In `for` and `upto` loops, this kind of iterator has been used since it is possible to correlate the values of one variable and the iterations of a loop. In the later prototypes, however, we discharged this idea again because it only was this easy to correlate a variable and the iterations of one loop because the examples were quite simple. In realistic situations, this correlation will oftentimes not be possible.

This version was supposed to take up as little space as possible while still having a complete (evaluated) version of the code in a different place.

3.1.3 Interface Version 3

Interface variant 3 uses small pop-ups that indicate the value of a statement

Figure 3.3 shows version 3 of the interface with an implementation of Bubblesort. Small pop-ups indicate the value of all statements that are evaluable. For example, the (manipulable and therefore red) value of `array` is `[1,42,37,16]` and the value of `array.length-2` is `2`. There are no pop-ups in the code when it is not executed and failed invariants are greyed out. The iterator works like the one in the example for version 2 of the interface but is now inserted into the code. This interface's purpose was to consume as little space as possible while still providing all possible evaluations.

3.1.4 Interface Version 4

Interface variant 4 puts the evaluated version of a line between that line and the next one

Figure 3.4 shows interface version 4 with an algorithm that converts the fractional part of a decimal number into its binary representation and outputs the result to the console. This interface variant slots the evaluated version of a line of code right between that line and the next one. The name of manipulable parameters is highlighted in red and there is no syntax highlighting. Failed invariants are greyed out and not executed code is not displayed, as usual. The goal of this interface version was to find out if the link between code and evaluation becomes clearer when the evaluation is very close and very similar to the code it results from.

3.1.5 Interface Version 5

Interface variant 5 indicates the value of a statement by writing it underneath that statement without keeping keywords

The last variant of the interface can be seen in Figure 3.5 with an algorithm that checks whether a given number `n` is a prime number or not. It is similar to the previous one but only shows "relevant" information. This includes right-hand sides of equations and for conditions the actual invariant but excludes left-hand sides of equations and keywords like `else` or `while`. Apart from this, version 4 and 5 are exactly the same (parameters in red, no syntax highlighting, not executed code is not shown, failed invariants

are greyed out). We developed this variant because, in version 4, the user might confuse actual code with evaluated code. We wanted to check if a variant without replicating control structures is better (less confusing).

3.2 Course of the User Study

The first user study was a qualitative user study. It was supposed to reveal what basic interface design is preferred by users and where there might be flaws in the interfaces. Different techniques for qualitative user studies have been used, in particular the *Model Extraction* method and the *Think Aloud* method. The basic structure was the following:

- First, the experimenter introduced himself and the project. He assured the participant that he would stay anonymous and that he could ask questions or make comments whenever he likes. To make sure the Model Extraction worked fine, the part concerning what the thesis and the user study where about was very firm. The solution we have come up with was to explain the general idea of Live Coding (that we are designing a kind of editor that takes exemplary parameters for a method the programmer works on and then evaluates it continuously while changes are being made) without revealing anything further like "and then there is going to be a button with which you can flick from iteration to iteration"

  Introduction to the user study

- Afterwards, one version of the interface was presented to the user together with the following questions: "What do you think you see here" and "What do you think you can do with it?"

  Model Extraction

- After the user told everything he or she knew or if the user had a gulf of understanding [Norman, 2002], the parts of the interface, which the user did not understand were explained and we went through all iterations once to make clear how the interface works

  Explanation of the interface

- The next goal was to find out if the user had problems in understanding elements of the interface, if he

  Suggestions for improvement?

thinks that there are things that can be done better, et cetera

Going through the remaining interface variants

- Lastly, the other interface variants were shown to the user with the same piece of code and it was explained how they worked, how they differ from the previous ones and we went through all iterations once so that the user could get a feeling for the interface. After each variant, the user was asked what he thought of the interface (and why) and what he thought was better or worse compared to the previous ones

Whenever the user had a question or there was an interesting topic that lead to more interesting information, this point was discussed in more depth.

3.3 Results

The user study was conducted with 6 different users. All of them studied Computer Science, were in the sixth semester and had between 4 and 10 years of programming experience. Only one of them had worked with Ruby before. All users took part voluntarily and were not rewarded in any way. The conditions were randomized and looked like follows:

User ID	Algorithm	Order of versions
1	Bubblesort	3,1,2,4,5
2	Selection sort	4,5,3,2,1
3	Fractional $\text{part}_{10} \rightarrow$ Fractional part_2	2,5,3,4,1
4	Prime number test	1,3,5,4,2
5	Binary search	5,2,4,3,1
6	Binary search	3,4,1,5,2

This means that, for example, user 1 saw interface version 3 first and then versions 1,2,4,5 in that order. In all variants, he worked with the Bubblesort algorithm.

The user study yielded two major results:

- Most users liked version 1 better than the other ones.
- Not a single user understood what the iterator is good for or what it might do.

3 users liked version 1 the most, 1 user liked version 4 the most, 1 user could not decide between version 1 and 2 and 1 user could not decide between version 1 and 3

In more detail: Three users (users 1, 2 and 4) stated clearly that they liked version 1 the most. User 3 was not quite sure whether he liked version 1 or version 2 more. He stated that a mix between the two variants where there is no syntax highlighting would be the best because he got confused the first time he saw version 1 and did not directly see that the code on the right-hand side of the window is not actually code but the evaluation. User 6 was not sure if he preferred version 1 or version 3. He stated that, if there is enough space on the screen, version 1 is better but with all the other elements of a typical IDE, a line of code might oftentimes be too long so that there is no place to show the line itself as well as its evaluated version (a concern that user 3 communicated as well). Without this problem though, user 6 said that version 3 is worse because the interface might get messy and distracting quickly because of all the pop-ups right next to the code. In addition, he said, with more complex code, it might be problematic to see which pop-up refers to which part of the code. Finally, user 5 stated that he likes version 4 the most. The reason was that he did not have to look to the right if he wanted to see the value of a variable and then search it there but that in version 5 the information was exactly at its source. He only said that this version might be problematic to work with since the reading flow is influenced negatively and he feared that he would loose much space on the screen and would therefore have to scroll a lot.

Perceptual model of the code should remain unaffected

Each user stated that the main benefit of version 1 and 2 was that the actual code remained unaffected. This corresponds to what [DeLine et al., 2006] reported on the perceptual model a programmer builds of the code: When navigating through source code, a programmer uses a perceptual model to do so. This way, the programmer does not need to read the code (cognitive) to know which part he is seeing but can do so by looking at its ”visual landmarks” (perceptual). While version 3 affects this perceptual model negatively but does not break it, version 4 and 5 have the

disadvantage to do so.

To most of the users, the basic structure of the interfaces was clear after only a few seconds. Only two of them confused actual code with the evaluation for a short time: User 5 when he saw version 5 the first time and user 4 when he saw version 1 the first time. However, neither of them needed help or much time to figure how the interface worked after they started to read more precisely. Each user's first question or comment concerned the font and how awful they (every one) thought it was. After a brief explanation why we chose this particular font, each user got used to it quickly.

Iterator is not self explaining

No user understood the iterator. None of them got the connection to the `while`, `upto` or `for` next to the iterator. After the explanation what the iterator does, some users said they thought the left and right buttons were "<" and ">" signs.

The following tables summarize the pros and contras for each interface:

Version 1	
Pros	Contras
· The control structure (indents) is still present · Because of the indents and the syntax highlighting, there is a strong analogy between the code and its evaluated version · There is a clear separation between code and evaluation · The editor remains unchanged	· Needs much space · User has to look from the left to the right and vice versa to see two pieces that are correlated. In other words, a line of code and its evaluated version are too far away from each other

Version 2	
Pros	Contras
· The editor remains unchanged · Code and evaluation do not get confused because they do not look the same · Needs less space than version 1	· Less analogy than in version 1 · User needs to look from the left to the right and vice versa. This is being complicated by the fact that without indents and syntax highlighting it is even more difficult to find the point of interest · Needs a lot of space

Version 3	
Pros	Contras
· Needs minimalistic amount of space · The code structure itself remains untouched	· Oftentimes not clear which pop-up refers to which part of the code · Can get messy quickly

Version 4	
Pros	Contras
· Strong analogy · Better for lines of code that are too long for version 1 and 2 · Evaluations right next to their source	· Code and evaluation get confused quickly, especially usual reading might be problematic since one has to skip every second line depending on what he wants to read · Perceptual model gets destroyed · Might be demanding to navigate when already long documents get double as much lines

Version 5	
Pros	Contras
· Evaluations right next to their source · Better for lines of code that are too long for version 1 and 2	· Oftentimes it is not clear what a value refers to · Still distracting during code reading · Perceptual model gets destroyed · Might be demanding to navigate when already long documents get double as much lines

3.3.1 Further Comments and Suggestions for Improvement

Interface version 3 can produce syntactically wrong code

One of the users pointed out that in version 3, the iterator has been inserted into the code and therefore it now is syntactically wrong (`for i in 1..4` becomes `for i=2 in 1..4` for the second iteration, for example).

Two users suggested that for version 4 and 5, curly braces could indicate to which part of the code the evaluation refers.

Suggestions for improving the iterator

Some users suggested to make the number on the iterator directly manipulable so that if one would want to go to iteration 42 one would not have to press 41 times a button but could just enter 42. In addition, some users said they wanted a button for jumping to the first and a button for jumping to the last iteration. Two users said it would be good if there is something that indicates directly that the iterator actually shows iterations like the term ”Iteration #” or anything similar.

We noticed that the manipulable parameters and the evaluation results have to be distinguishable more easily .

One user proposed that if the cursor moves over some variable in the evaluation part, a small pop-up could indicate to which variable a value belongs.

One user said that it might be helpful that, for bigger boolean expressions that fail, it would be visible which part of the expression is responsible for the failure.

3.3.2 Conclusions

Version 1 is the favored one

With large screens (24″ and more) becoming more and more common in modern work environments of programmers, we can override the concerns about loosing too much space with version 1, which is the reason why it is the clear favorite. Figure 3.6 shows the basic interface after the first user study.

The iterator has been improved in several ways:

- New buttons for jumping to the first and the last iteration have been added
- There is a text field with which the iteration can be manipulated directly
- A tooltip now indicates the variable to which a value belongs when hovering over the value with the mouse
- The label *Iteration* has been added at the top of the iterator

A new way of manipulating the parameters

The way of manipulating the parameters has been changed. The function header now also is copied to the right side but the names of the parameters are replaced with a text field in which the according value can be entered. This way, both, the analogy to the code and the separation between parameters and evaluation are being enhanced.

```
def function(array, key)

   low = 0
   high = array.length - 1

   while (low <= high)
      mid = low+((high-low)/2).to_i
      value = array[mid]

      if (value < key)
         low = mid+1
      elsif (value > key)
         high = mid-1
      else
         return mid
      end
   end
   return -1
end
```

```
key = 4
array = [0,2,3,4,6,9]
   low = 0
   high = 5

   while (0 <= 5)      1 >
      mid = 2
      value = 3

      if (3 < 4)
         low = 3

      end
   end

end
```

(a) First iteration

```
def function(array, key)

   low = 0
   high = array.length - 1

   while (low <= high)
      mid = low+((high-low)/2).to_i
      value = array[mid]

      if (value < key)
         low = mid+1
      elsif (value > key)
         high = mid-1
      else
         return mid
      end
   end
   return -1
end
```

```
key = 4
array = [0,2,3,4,6,9]
   low = 0
   high = 5

   while (3 <= 5)      < 2 >
      mid = 4
      value = 6

      if (6 < 4)

      elsif (6 > 4)
         high = 3


      end
   end

end
```

(b) Second iteration

Figure 3.1: Version 1: The evaluated version of the code is on the right-hand side and keeps its structure and syntax highlighting

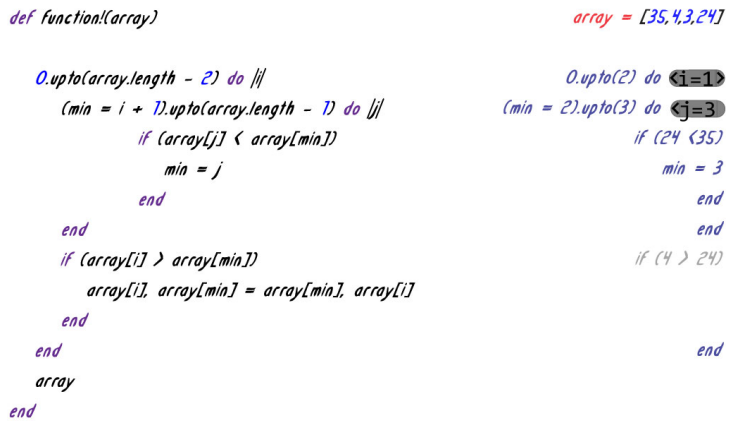


Figure 3.2: Version 2: The evaluated version of the code is on the right-hand side but is ranged right

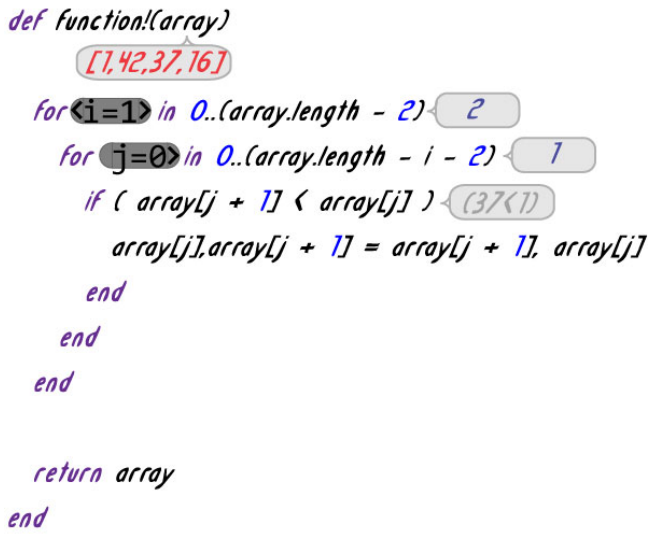


Figure 3.3: Version 3: Small pop-ups indicate the value of a variable

```
def function(value,digits)
value = 5.875
digits = 4

   x=value.to_i
   x=6

   if (x)value)
   if (6)5.875)

      fractional = value - x + 1
      fractional = 0.875

   else

      fractional = value - x


   end
   for i in 1..digits
   for <i=4 in 1..4

      fractional=fractional*2
      fractional=1

      if(fractional >= 1)
      if(0 >= 1)

         puts 1


         fractional=fractional-1


      else
      else

         puts 0
        puts 0

      end

   end
end
```

Figure 3.4: Version 4: The evaluated version of a line is inserted right behind the line itself

```
def function(n)
    n = 25

  p=2
    →2

  while (1) < 3 >
      →1

    if (p)=n)
    →(4 )= 25)

       return true


    elsif(n%p == 0)
        →(1==0)

       return false


    else

       p=p+1
         →5

    end
  end
end
```

Figure 3.5: Version 5: Only the values of the variables are in between the lines

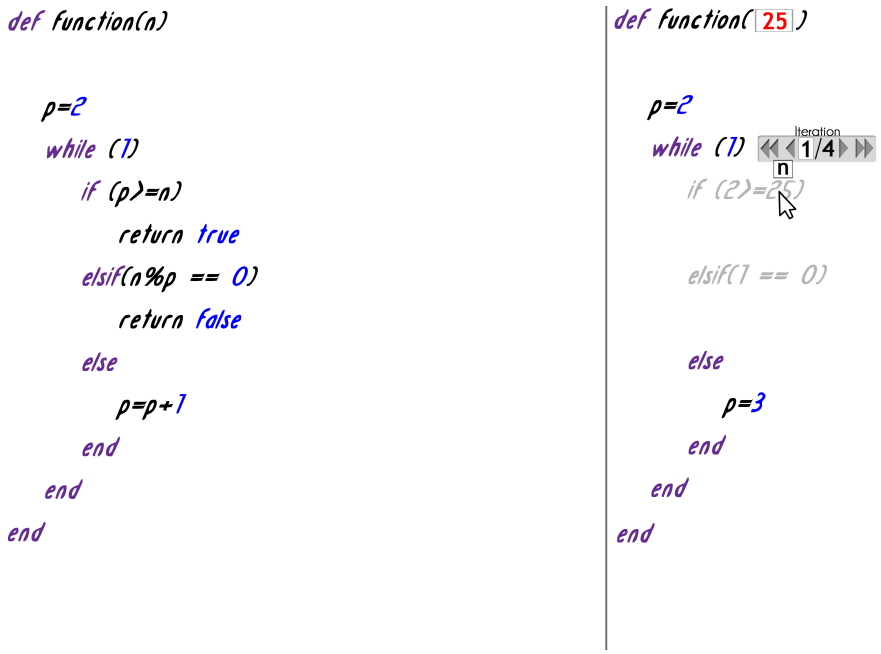


Figure 3.6: Interface after the first user study

Chapter 4

Implementation

After the first user study had been finished and the results had been analyzed, a software prototype with sufficient functionality had to be implemented to conduct the second user study. Hence, we wrote an extension for Brackets[1] , an open source editor that has recently been released officially.

A software prototype had to be implemented

The language we wrote the extension for was JavaScript. What we needed was an extension that could calculate the values of the variables that had to been substituted.

4.1 First Approach (discharged)

The initial idea was to write an extension that connects via V8-Node[2] (an extension that maintains a connection to a Node.js[3] /V8 debugger) to an external debugger and lets it evaluate the code. Our extension was supposed to supply the actual code to the debugger and set a breakpoint in the first line of actual code. Then it read the *locals* (that is, the local values for all active variables) and stepped over to the

Initial idea: Use V8-Node an Node.js to get the locals

[1] https://github.com/adobe/brackets

[2] https://github.com/DennisKehrig/brackets-v8-node-live/tree/nodeChildProcess

[3] http://nodejs.org/

This extension was not usable for a user study

next line. After all locals had been collected, the extension should substitute all necessary values. There were problems with this approach. Firstly, the extension is not stable enough to be usable in a user study. Secondly, it is too slow. It needs more than half a second for one evaluation. This is way too much since we wanted something that could evaluate the code as the user edits it without any further delay (hence the name "Live Coding"). With this extension it would only have been possible to update the evaluation using a special shortcut or to attach this function to another shortcut like the `Ctrl+S` shortcut for saving. Since this was by far not the optimal way, we decided to start all over again.

4.2 Basic Evaluation Technique

Second approach using `eval` to evaluate dynamically generated code that returns the value of interest

In the second approach, we did not try to externalize the script execution but instead made vast use of JavaScript's built-in `eval` function. The `eval` function takes a String as parameter. If this String contains valid JavaScript code, `eval` will evaluate the code and return the last executed statement. `eval("var i=1;i=i+5")` will return `6`, for example. The extension basically works as follows: A loop that runs over all DOM-elements in the highlighted code searches for substitutable variables. While searching for these elements, the loop captures in which context the actual DOM-element is. The three basic contexts are simple substitutions, substitutions within one loop and substitutions within a nested loop (why this is, will be explained below). But there are also other things the loop checks for. For simple assignments, like `i=x+y;`, `x` and `y` have to be substituted while `i` does not. In lines where a loop begins, the iterator has to be appended, invariants have to be treated differently, et cetera.

Note that right-hand sides of assignments are handled slightly different from what has been presented in the prototypes of the first user study. They do not get evaluated to the last step and invariants do not get evaluated

to the penultimate step anymore. Both of them get only evaluated one step. For `i=1` and `z=2`, the right-hand side of `x=i+z;` does not get evaluated to `3` but to `1+2`. We decided to do it this way because some users in the first user study stated that they did not only want to see what the result of an operation is but also how it got there.

When a substitutable variable has been found, its value will be computed. For that, the `eval` function is used (see section 4.3 for details) and the value will be inserted. This whole process takes place each time there is a change in the code or if one of the parameters gets changed. If another iteration of a loop that is being shown is selected, all values inside that loop will be calculated again.

How the extension reacts to changes

4.3 How Values Are Being Calculated

Depending on the context of the variable, a piece of code will be generated (and then evaluated) that, when evaluated with `eval`, returns the value of the variable of interest. For simple substitutions outside of any loops, this works like follows:

Suppose the following code had to be evaluated for `x=0`:

```
function foo(x){
    var i=0;
    var j=x+1;
}
```

How the values are calculated for simple substitutions

Then, `x` in the third line would have to be evaluated. To do so, the following String would be generated:
`"var x=0;`
`var i=0;`
`x;"`
So, for simple substitutions, the whole code preceding the variable that has to be substituted is copied and each parameter is initialized as `var` with its exemplary value at the

beginning.

Substitutions within one loop

For simple loops, this method has to be extended: First, it has to be found out how often the loop is executed. This serves two purposes: First the iterator now shows the number of iterations of the loop. Second, we have to check if the user wants to see an iteration that does not exist. This is being done by introducing a counter that is incremented in each iteration of the loop and then returned when the loop terminated. The following example shows how the value of a variable within a loop is being calculated:

```
function test(){
    var i=0;
    var j=0;
    while (i<5){
        i++;
        j=j+2;
    }
}
```

Suppose we want to obtain the value for `j` (highlighted in red) in the second iteration of the loop. Then, the evaluation basically works as follows:

```
var myCounter=0;
var myResult;
var i=0;
var j=0;
while (i<5){
    myCounter++;
    i++;
    if(myCounter==2){myResult=j; break;}
    j=j+2;
}
myResult;
```

A counter gets inserted to check which iteration the loop is

in. If it is in the right iteration, the assignment `myResult=j` takes place so that `eval` can return the according value. For a loop within a loop, two counters have to be inserted and both of them have to be checked, accordingly. There are further variants of contexts like the different parts of a `for(firstPart;secondPart;thirdPart)` statement. The first part gets only executed once at the beginning while the second part gets evaluated one time more than the rest of the loop (the third part does not get evaluated). For all cases, the dynamic generation of the code that returns the value of interest works similar to the techniques presented.

Other cases

Obviously, this is a brute-force approach. For each value that has to be calculated, all preceding code has to be evaluated. We used this approach anyway since we spent quite some time developing the first version of the extension (using V8-Node) and were under considerable time pressure. The technique we chose is simple and fast to develop. Furthermore, the prototype only had to work for the second user study, which is the reason why the limited functionality (loop nesting depth of 2 for example) was sufficient. And finally, since the tasks in the second user study were not very expensive with regards to calculation costs, even this very inefficient version of the extension ran live without any delays. Some tests showed that it would even have been quick enough to operate with hundreds of iterations more without any delay and with some few thousand iterations more without dispatching the interface.

Summary: Very inefficient; limited functionality; still completely sufficient for the user study

4.4 Problems

The biggest problem we encountered were infinite loops. Because the interface runs completely live and tries to evaluate the whole code while the user is editing it, infinite loops may occur while the user is typing. Additionally, it might of course be possible that there actually is an infinite loop even though the user thinks the code is correct. Without modification, the interface crashed in

The Halting problem

either of these cases. To prevent this from happening, two minor modifications had to be done. Firstly, if we want to calculate the number of iterations a loop does, we check whether the counter we use for that calculation is still below a certain limit (we used 999 here because it proved to be sufficient and efficient enough to run live). Now, the Halting Problem for this particular loop is approximated. The problem of the infinite loop still exists, though. Suppose, we have the following code:

```
var x=0;
var i=0;
while(x!=10){
    i++;
}
x=i;
```

Substitutions after infinite loops

We now know that the loop does more then 999 iterations, so we do not try to calculate the value for `x` in the invariant or for `i` within the loop for any further iterations. The only problem that remains is that we still try to get the value for `i` in the last line. When we try to calculate this value we do not yet know that there is an infinite loop somewhere above. To find this out, a new global counter gets inserted and limited to 5000 iterations:

```
var globalCounter=0;
var x=0;
var i=0;
while(x!=10){
    if(globalCounter++ >= 5000){break;}
    i++;
}
i;
```

This way, we could keep the incrementation and the check whether there are more then 5000 iterations within one statement (two statements are too many since after a

`while` without curly braces, only the first statement would be executed). The incrementation and test get inserted into every loop ahead of the position we are checking for. It would of course have been possible to set a flag if there is an infinite loop somewhere ahead but this way, we also had the possibility to limit the overall number of iterations and still get a value. The second user study showed that this was the wrong approach: A user introduced an infinite loop in a sorting algorithm. The condition for the loop was never violated although the array was sorted. With the technique of assigning a value to a variable although the limitation of iterations has been hit, the sorted array was shown in the `return` statement. Since the user only paid attention to the output, he did not see that the number of iterations of the loop was `999` and thought that the code is correct.

Flaw in the implementation

Another minor problem was that, of course, while the user is typing, the extension tries to evaluate code that is syntactically not correct. The solution is to set `window.onerror` to a function that always returns true. This way, errors get just suppressed and the execution stops. If there is a new change in the editor, the execution will start again and work exactly the same way so that working code always is executed and not working code is not.

Suppress errors so that user does not get disturbed

The last problem concerned hiding code that does not get evaluated. For loops, this could easily be done: If the loop is in its very last iteration (the invariant fails), the rest of the loop has to be hidden, otherwise it does not. Finding out if an `if` statement fails (and hence, the according code has to be hidden), however, was more difficult because we had not enough information to be able to do so (remember: we only compute the first step, which includes nothing but substituting all variables). To find out if an `if` statement holds or not, we do the following: all variables in the invariant get substituted and then we let `eval` evaluate just the invariant. For `if(x<y)` with `x=5` and `y=6`, we substitute all values: `if(5<6)` and then call `eval("(5<6)")`.

Hiding code within failed `if` statements

Figure 4.1: Screenshot showing Brackets with our extension and a dummy script

Finally, Figure 4.1 shows how the prototype for our second user study looks like.

Chapter 5

Second User Study

5.1 Course of the User Study

To find out whether our extension brings significant improvement, a second, quantitative user study with 20 participants has been conducted. The hypothesis was that the debugging speed (in this case the time between the user reading the code the first time and fixing it), compared to a usual debugging engine, could significantly be reduced with our extension. If this hypothesis would be supported by the results of the user study, this would indicate that the understanding speed of foreign code would be increased by our method.

Hypothesis: Compared to usual debugging methods, our technique brings significant improvement

No user had worked with Brackets or JavaScript before (notwithstanding that all of them had worked with Java before). Users were Computer Science students with 4-9 (average of 5.8) years of programming experience. Only three of them had professional experience (2-4 years). All users took part voluntarily and were not rewarded in any way.

To test the hypothesis, two searching algorithms (an implementation of Shakersort and an implementation of Bubblesort) have been implemented and a small bug has

2 algorithms; 1 bug each; 2 tasks: debug one time with Brackets' tools and one time with our extension

been introduced in either of them (see Appendix A). The task of the user was to find both bugs - one time with our extension and one time using the traditional way (that is, using the console, breakpoints, variable viewer, web kit inspector and the step over functionality Brackets provides). Each user received an intro to the functionality of both, our extension and the Brackets debugging engine. For both, a small dummy script has been shown with which it has been explained how each part of the functionality works until the user confirmed that he or she understood every part of the functionality. The task for the user was to find the bug in the code (users have been told that there is exactly one bug), fix it and give notice when he or she thought that the code is correct (task to condition assignment and task order were counterbalanced to eliminate side effects).

System Usability Scale to find out whether users liked working with the system

After the two debugging tasks had been accomplished, each user filled out two versions of the *System Usability Scale* [Brooke, 1996] (SUS), one for our extension and one for the standard debugging Brackets provides. The first 10 questions are identical to the initial version that has been presented in 1996 except that the word "cumbersome" in point 8 has been changed to "awkward" as suggested by [Finstad, 2006]. For the extension, the following questions have been added to the questionnaire:

- I found the additional information very helpful for my task
- I was distracted by the additional information
- I found it very easy to follow the control flow of the program
- I found complex control structures hard to understand

For the debugging using Brackets, the following two questions have been added:

- I found it very easy to follow the control flow of the program

- I found complex control structures hard to understand

See Appendix B for a complete version of both questionnaires.

Finally, a short qualitative review has been done with each user. Topics were not set à priori, so the discussion concerned mainly subjects we observed during the debugging tasks (see Section 5.2.3)

5.2 Results

5.2.1 Debugging Times

The box plot in Figure 5.1 represents the time users needed to accomplish the different tasks.

Since it is not possible to assume that debugging the two different code examples is equally difficult without loosing the meaningfulness of the results, they have to be analyzed independently.

On average, users were faster using our extension, but not significantly

On average, participants were faster debugging Shakersort using our extension ($M = 402.00$, $SE = 48.464$) than using Brackets ($M = 495.2$, $SE = 61.896$). This difference was not significant with $t(9) = 1.159$ and $p > 0.05$. However, it did represent a medium-sized effect with $r = 0.36037$. The results for Bubblesort are analogous: Using the extension, users were were faster ($M = 416.33$, $SE = 47.885$) than without it ($M = 477.56$, $SE = 41.597$). The result also is non-significant with $t(8) = 1.046$ and $p > 0.05$. And with $r = 0.34685$, the effect is medium-sized for Bubblesort as well.

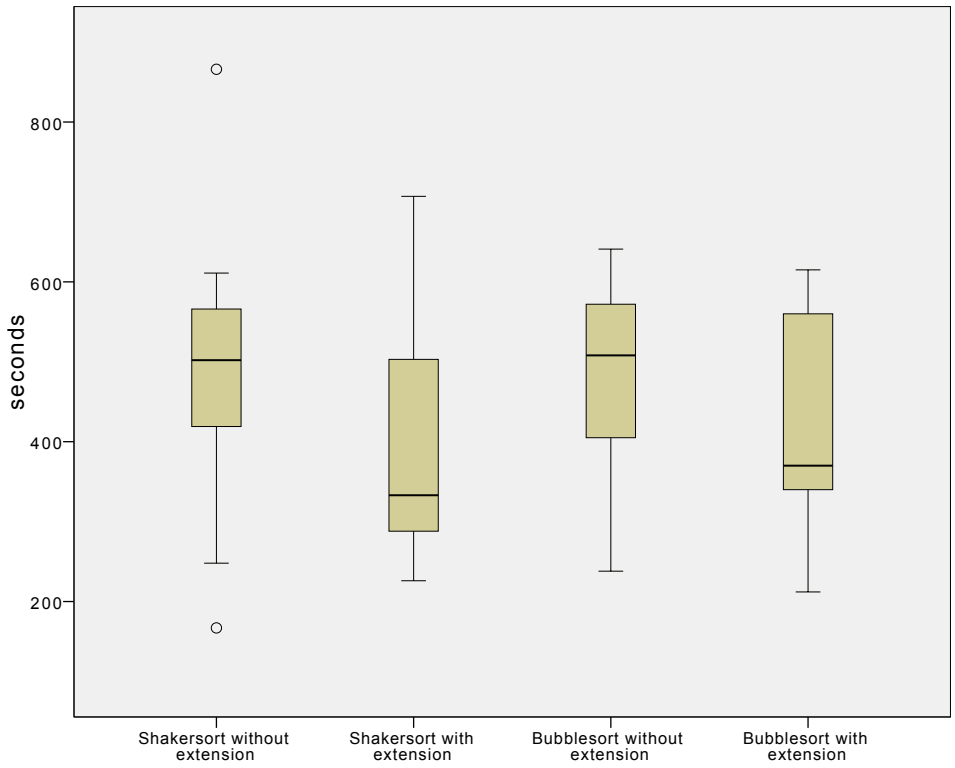


Figure 5.1: Box plot for the different task completion times

EXCURSUS:
Assuming that debugging both algorithms is equally difficult leads to significant results ($t(37) = 1.705$ and $p < 0.05$). As mentioned above, this is not possible without loosing the meaningfulness of the results. However, along with the medium-sized effects of both tasks taken individually, this indicates that just continuing the user study (with 10 or 20 more users) will lead to significant results.

5.2.2 System Usability Scale

According to the SUS-values, users liked our system better than Brackets

According to the SUS-values, users liked the Live Coding extension considerably better ($M = 81.95$, $SE = 1.804$) than the developer tools Brackets provides ($M = 51.20$, $SE = 2.117$). This result is significant with $t(19) = 15.831$

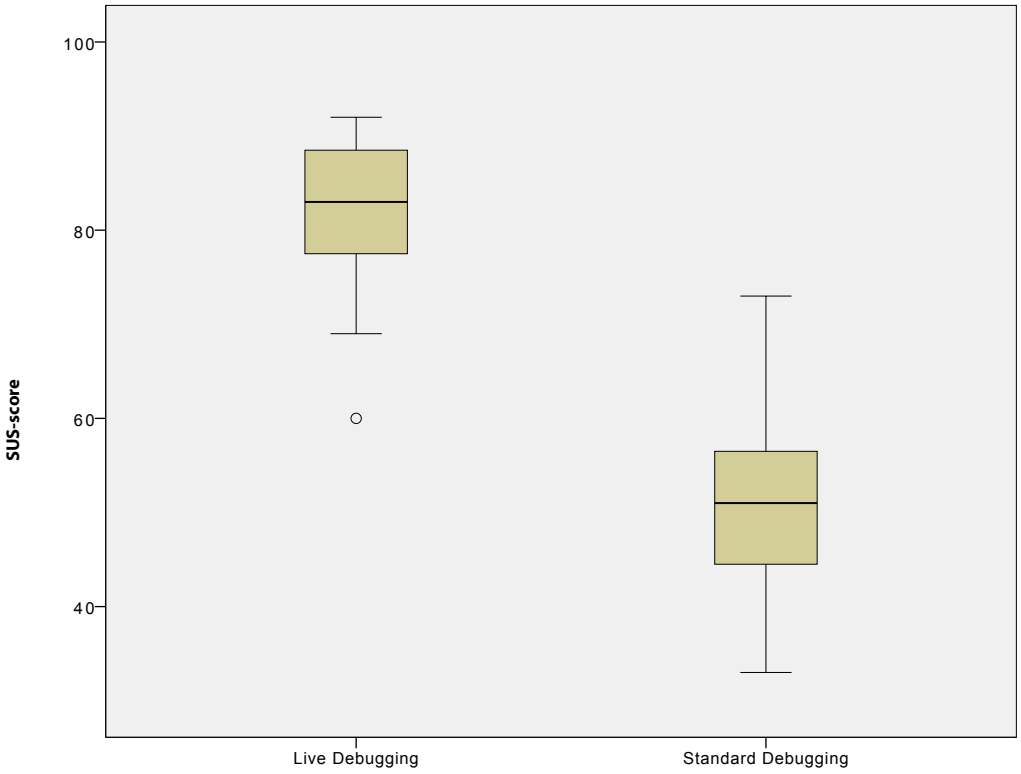


Figure 5.2: Box plot for the SUS-values with and without extension

and $p < 0.01$. This can also be seen in the box plot of the SUS-scores in Figure 5.2. Finally, Figure 5.3 shows how the users responded to the additional questions on the questionnaire

5.2.3 Qualitative Results

We found two major issues in the qualitative follow-up survey. Firstly, the iterator and parameter fields are too small. Users had difficulties hitting the intended buttons on the iterator and reading the content of the parameter fields. Secondly, many users expected the interface to show the first iteration of a loop first and not the last one. This was particularly observable with nested loops. When the outer iteration had been changed, most users expected the interface to first show the first iteration of the inner loop because that reflects their way of comprehending the code.

Iterator and parameter fields are too small

Users expected interface to show first iteration first

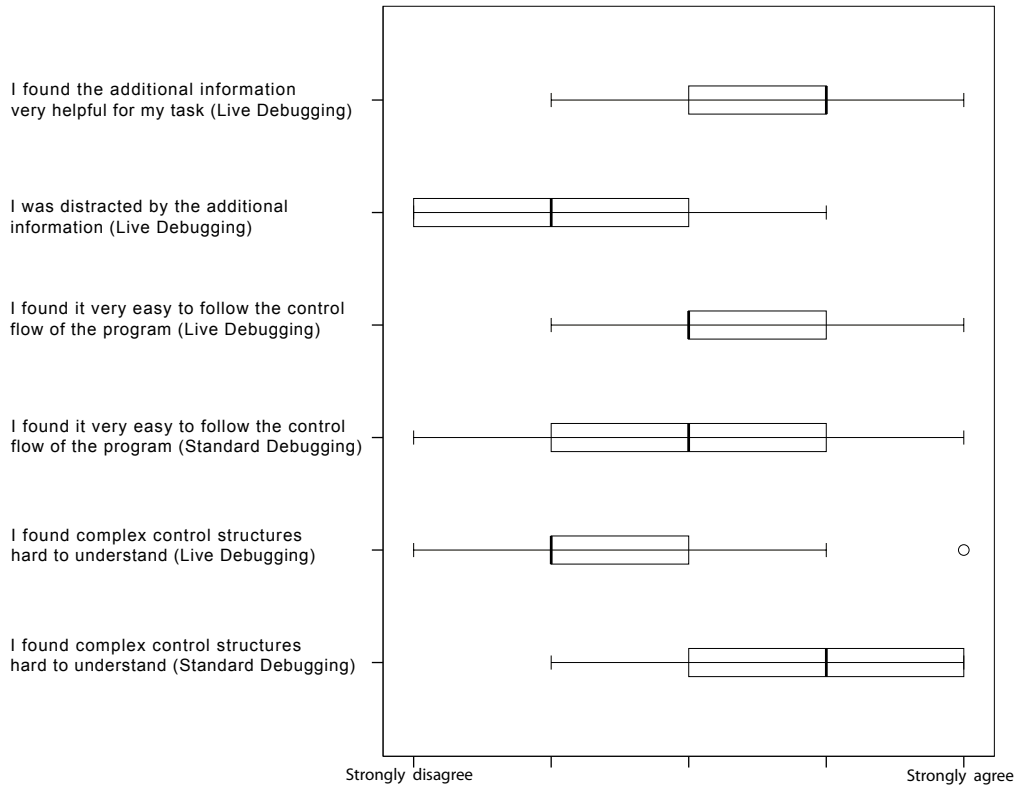


Figure 5.3: Box plot for the additional points on the questionnaire

Flaw in the implementation: Despite endless loop, values are assigned

One user uncovered a flaw in the implementation. As already explained in section 4.4, even if an infinite loop occurs somewhere in the code, values are still assigned to the variables (either after the 999th iteration of one loop or after there have globally been more than 5000 iterations). In this particular case, the algorithm was finished with sorting the array but the invariant was wrong in such a way that it did not get violated anyway. Nevertheless, the extension assigned a value to the `return` statement. The user could have seen that there is an infinite loop by looking at the iterator which showed "`999/999`" after each edit. However, since he changed the exemplary input array and looked only at the result (which was a correctly sorted array), he thought that the code was correct although it was not. This means that, in a future implementation, no substitution should take place in unreachable assignments.

Finally, many users said it would be good to highlight, which element of an array is addressed by an index. For [45,3,2,4,5,42,7,5,3][5], for example, users should not have to count which element is the fifth one but that element should modestly be highlighted.

5.2.4 Discussion

Most users stated that the extension would help improve the understanding (debugging) process. Considering the System Usability Scales, the additional questions on the questionnaires and the qualitative statements of the users, users liked our extension notably better than the developer tools Brackets provides. However, no significant result could be obtained in this quantitative user study that confirms the initial hypothesis. Some users reported that, despite the introduction to the extension, it was not as easy to involve the extension in the understanding process as it was for the Brackets tools because they were used to the way the traditional tools work and not to the way the extension works. It is quite possible that just continuing the user study or refining the prototype and the user study will both lead to significant results. The user study presented in this chapter, however, does not provide any conclusive manifestations concerning improvements in debugging or understanding speeds.

Users liked extension and said that it would help. Nevertheless, no significant improvement could be shown in the user study

Chapter 6

Summary and future work

6.1 Summary

Live Coding enables the programmer to illuminate code during editing it in order to understand its internal behavior. A user study with low fidelity prototypes has been conducted to find out how the basic interconnection between code and a continuously updated, evaluated version of the code can best be displayed. A software prototype of the result, an evaluated copy of the code with syntax highlighting and indents in a separate section of the IDE, has been implemented in form of an extension for Brackets. It has been explained how this prototype essentially works, what drawbacks it has and which problems occurred during the implementation. With this prototype, a second user study has been conducted, which should reveal if there is a significant enhancement in the debugging speed of experienced programmers. Although this user study only yielded non-significant results, there is evidence that significant results can be obtained by continuing the user study.

6.2 Future Work

2 possible approaches to significant results

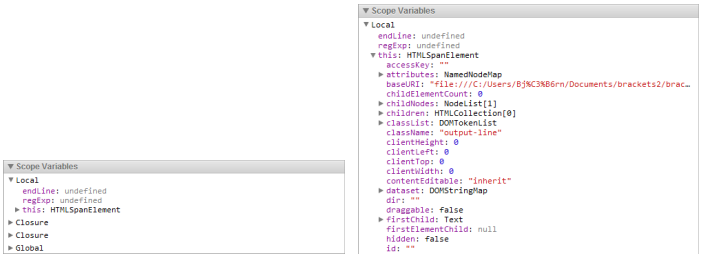
The quantitative user study only yielded non-significant results but there is evidence that a significant result can be obtained (see Chapter 5). Therefore, two possibilities of continuation exist. Firstly, it is possible to refine the software-prototype, then to design and conduct a further user study. Secondly, it is possible that simply continuing the user study presented in Chapter 5 with another 10 or 20 users will already yield a significant result.

Complex data structures have to be displayed...

The biggest issues for a continuation of the design of the interface are caused by complex data structures. Firstly, complex data structures have to be displayed. In the prototypes presented in this paper, only simple data types (Strings, Arrays, Integers,...) have been displayed. If, however, the interface is supposed to be used on a day-to-day basis, complex data structures (e.g. DOM-elements or self-defined structures) have to be representable. It would be conceivable to use a representation similar to the one Bracket's web kit inspector uses. Figure 6.1(a) shows how this looks likes for the DOM-element `this`. If the user presses the small arrow, the element "unfolds" and its attributes can be inspected (Figure 6.1(b)). This approach, however, has the drawback that information is not quickly available and therefore another approach that concentrates on instantly showing relevant information might be interesting.

...and to be entered

Furthermore, complex data structures do not only have to be displayed but also to be entered. An approach that might be a solution for this problem has been presented by [Edwards, 2004]. When a program calls an external function in the IDE presented by Edwards, only the fact of their execution and their return value are traced. Similarly, it might be possible for our environment to remember with what parameters the function that has to be evaluated has been called for an exemplary execution of the context the function is used in. This way, not only does the user save time by not having to enter the example but also an example is available that indeed occurs.



(a) DOM-element folded (b) DOM-element unfolded

Figure 6.1: Version 1: The evaluated version of the code is on the right-hand side and keeps its structure and syntax highlighting

Another aspect that might be a sensible enhancement is the incorporation of the *Test-Driven Development* [Williams et al., 2003],[Beck, 2002], [Saff and Ernst, 2004a] and the *Continuous Testing* approach [Saff and Ernst, 2004b]. The idea of the Test-Driven Development is to first write test cases and then implement the according production code. Although Continuous Testing is predominantly used for large scale systems, there is common ground between Continuous Testing and the Test-Driven Development. Therefore, checking how our approach can be integrated into and combined with these techniques is a promising field of research.

Potential field of research: congruence with TDD and Continuous Testing

Appendix A

Code for the Second User Study

```
function sort(inputArray){
    var array=inputArray;
    var n=array.length;
    var newn;
    var swap;
    while(n>1)
    {
         newn=1;
         for(var i=0; i<n-1;i++)
         {
            if(array[i]>array[i+1])
            {
                swap=array[i];
                array[i]=array[i+1];
                array[i+1]=swap;
                newn=i+1;
            }
         }
         n=newn;
    }
    return array;
}
```

The Bubblesort implementation (red highlighted `+1` has been left out)

```
function sort(inputArray)
{
    var array=inputArray;
    var begin=-1;
    var end=array.length-2;
    var swapped=true;
    var swap;

    while(swapped){
         swapped=false;
         begin++;
         for(var i=begin; i<=end;i++)
         {
             if(array[i]>array[i+1])
             {
                 swap=array[i];
                 array[i]=array[i+1];
                 array[i+1]=swap;
                 swapped=true;
             }
         }
         end--;
         for(var i = end; i>=begin;i--)
         {
             if(array[i]>array[i+1])
             {
                 swap=array[i];
                 array[i]=array[i+1];
                 array[i+1]=swap;
                 swapped=true;
             }
         }
    }
    return array;
}
```

The Shakersort implementation (red highlighted `-1` has been changed to `0`)

Appendix B

System Usability Scales

On the following two pages, the System Usability Scale questionnaires used in our second user study can be seen.

System Usability Scale - Standard Debugging

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

Figure B.1: System Usability Scale questionnaire for debugging using Brackets

System Usability Scale - Live Coding Extension

	Strongly Disagree				Strongly Agree
1. I think that I would like to use this system frequently					
2. I found the system unnecessarily complex					
3. I thought the system was easy to use					
4. I think that I would need the support of a technical person to be able to use this system					
5. I found the various functions in this system were well integrated					
6. I thought there was too much inconsistency in this system					
7. I would imagine that most people would learn to use this system very quickly					
8. I found the system very awkward to use					
9. I felt very confident using the system					
10. I needed to learn a lot of things before I could get going with this system					
11. I found the additional information very helpful for my task					
12. I was distracted by the additional information					
13. I found it very easy to follow the control flow of the program					
14. I found complex control structures hard to understand					

Figure B.2: System Usability Scale questionnaire for debugging using the Live Coding Extension

Bibliography

Kent Beck. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002. ISBN 0321146530.

B. W. Boehm. Classics in software engineering. chapter Software engineering, pages 323–361. Yourdon Press, Upper Saddle River, NJ, USA, 1979. ISBN 0-917072-14-6. URL http://dl.acm.org/citation.cfm?id=1241515.1241536.

Joel Brandt, Vignan Pattamatta, William Choi, Ben Hsieh, and Scott R. Klemmer. Rehearse: Helping programmers adapt examples by visualizing execution and highlighting related code.

Joel Brandt, Philip J. Guo, Joel Lewenstein, and Scott R. Klemmer. Opportunistic programming: how rapid ideation and prototyping occur in practice. In *Proceedings of the 4th international workshop on End-user software engineering*, WEUSE '08, pages 1–5, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-034-0. doi: 10.1145/1370847.1370848. URL http://doi.acm.org/10.1145/1370847.1370848.

Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Opportunistic programming: Writing code to prototype, ideate, and discover. *IEEE Softw.*, 26(5):18–24, September 2009a. ISSN 0740-7459. doi: 10.1109/MS.2009.147. URL http://dx.doi.org/10.1109/MS.2009.147.

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*

27th international conference on Human factors in computing systems, CHI '09, pages 1589–1598, New York, NY, USA, 2009b. ACM. ISBN 978-1-60558-246-7. doi: 10.1145/1518701.1518944. URL http://doi.acm.org/10.1145/1518701.1518944.

J. Brooke. SUS: A quick and dirty usability scale. In P. W. Jordan, B. Weerdmeester, A. Thomas, and I. L. Mclelland, editors, *Usability evaluation in industry*. Taylor and Francis, London, 1996.

Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven Drucker, and George Robertson. Code thumbnails: Using spatial memory to navigate source code. In *Proceedings of the Visual Languages and Human-Centric Computing*, VLHCC '06, pages 11–18, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2586-5. doi: 10.1109/VLHCC.2006.14. URL http://dx.doi.org/10.1109/VLHCC.2006.14.

Jonathan Edwards. Example centric programming. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 124–124, New York, NY, USA, 2004. ACM. ISBN 1-58113-833-4. doi: 10.1145/1028664.1028713. URL http://doi.acm.org/10.1145/1028664.1028713.

Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31:226–237, 2005. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2005.37.

Kraig Finstad. The System Usability Scale and Non-Native English Speakers. *Journal of Usability studies*, 1(4):185–188, 2006. URL http://www.upassoc.org/upa_publications/jus/2006_august/finstad_sus%_non_native_speakers.pdf.

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.

Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pages 199–206, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8696-5. doi: 10.1109/VLHCC.2004.47. URL http://dx.doi.org/10.1109/VLHCC.2004.47.

Donald A. Norman. *The design of everyday things*. Basic Books, [New York], 1. basic paperback ed., [nachdr.] edition, 2002. ISBN 0-465-06710-7.

J. F. Pane and B. A. Myers. Usability issues in the design of novice programming systems. Technical report, Carnegie Mellon University, August 1996.

David Saff and Michael D. Ernst. Continuous testing in eclipse. *Electron. Notes Theor. Comput. Sci.*, 107:103–117, December 2004a. ISSN 1571-0661. doi: 10.1016/j.entcs.2004.02.051. URL http://dx.doi.org/10.1016/j.entcs.2004.02.051.

David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. *SIGSOFT Softw. Eng. Notes*, 29(4):76–85, July 2004b. ISSN 0163-5948. doi: 10.1145/1013886.1007523. URL http://doi.acm.org/10.1145/1013886.1007523.

Bret Victor. Inventing on principle, cusec 2012.

Laurie Williams, E. Michael Maximilien, and Mladen Vouk. Test-driven development as a defect-reduction practice. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 34–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2007-3. URL http://dl.acm.org/citation.cfm?id=951952.952364.

Typeset September 12, 2012