Detail Visualization for Live Coding

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

by
Hendrik Wolf

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Bernhard Rumpe

Registration date: 08.07.2014
Submission date: 30.09.2014

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 2014
Hendrik Wolf

Contents

List of Figures

Abstract

Programming without receiving a feedback on the internal runtime state of a program demands much from the cognition of the programmer. He has to take over the role of the computer and simulate the program flow continuously in his head. In contrast, live coding uses the capability of the computer for that continuous simulation. The computer executes the program in the background and displays the result after each applied change.

Several existing live coding tools explore detail visualizations with different levels of abstraction and possible user interaction. These tools show the benefit of further extensions. One extension is the capability to regulate the amount of shown information or filter out interesting parts. Furthermore, a majority of prototypes does not yet offer the possibility to choose between different representations. This possibility is essential since there is no visualization which is appropriate for each situation.

This thesis presents a live coding prototype in JavaScript as well as diverse detail visualizations. The prototype enables the programmer to switch between different representations to find the most appropriate one. In case there exists no appropriate visualization, our prototype offers the possibility to enhance it further by adding new visualizations to the existing set.

Überblick

Während des Programmierens erhält der Programmierer keine Informationen über mögliche Programmzustände zur Laufzeit. Er muss die Rolle des Computers übernehmen und die Ausführung des Programms kontinuierlich in seinem Kopf simulieren. Dies erfordert hohe Konzentration. Im Gegensatz dazu wird beim Live Coding die Leistungsfähigkeit und Kapazität des Computers für diese Simulation benutzt. Der Computer führt das Programm im Hintergrund aus und zeigt die Ergebnisse nach jeder Änderung am Quelltext an.

Es existieren bereits diverse Live Coding Werkzeuge, die verschiedene Detail-Visualisierungen mit variablem Abstraktionslevel und möglicher Benutzerinteraktion anbieten. Diese Werkzeuge zeigen das Potential von Live Coding. Eine mögliche Erweiterung ist, dem Benutzer die Fähigkeit zu geben, die angezeigte Datenmenge selbstständig zu regulieren und interessante Teile herauszufiltern. Außerdem fehlt in den meisten Live Coding Werkzeugen und Prototypen bis jetzt die Möglichkeit, zwischen verschiedenen Visualisierungen zu wechseln. Das ist wichtig, da es keine Visualisierung gibt, die in allen Situationen passend und hilfreich ist.

In dieser Arbeit präsentieren wir einen Live Coding Prototypen für JavaScript, sowie verschiedene Detail Visualisierungen. Unser Prototyp ermöglicht dem Benutzer, zwischen verschiedenen angebotenen Visualisierungen zu wechseln. Weiterhin hat der Benutzer die Möglichkeit eigene Visualisierungen zu implementieren und sie den bestehenden hinzuzufügen.

Acknowledgements

I would like to thank Prof. Dr. Jan Borchers for making this thesis possible. I would also like to thank my supervisor Jan-Peter Kraemer, for giving me this amazing topic as well as his valuable and constructive suggestions.

Special thanks go to Moritz Wittenhangen, for taking over as my supervisor and giving me frequently advice. Lastly, I would like to thank all people at i10 for helping me whenever I had a problem or question.

Thanks for all the support!

Conventions

Throughout this thesis we use the following conventions.

Text conventions

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

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

Definition:
Excursus

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

```
myClass
```

The whole thesis is written in American English.

Chapter 1

Introduction

The original programming cycle is divided into four phases: edit, compile, link and run. The programmer received no feedback while editing the source code. As a result, in order to verify that the program works as intended, the programmer has to simulate the changes and their effects in his head. But simulating the flow of a program strains the limits of the human cognition [Snell, 1997]. Additionally, most people do not want to wait frequently for their program to compile, link and finally run so that they can debug it. Thus, they implement as much as possible without debugging the program. This leads to a larger time gap between the editing of source code and the detection of bugs, which increases the required debugging time. More parts of the program are affected by the bugs and as a consequence have to be adapted to fix them.[Saff and Ernst, 2004a].

Programming without feedback is straining

Longer code editing periods increase the required debugging time

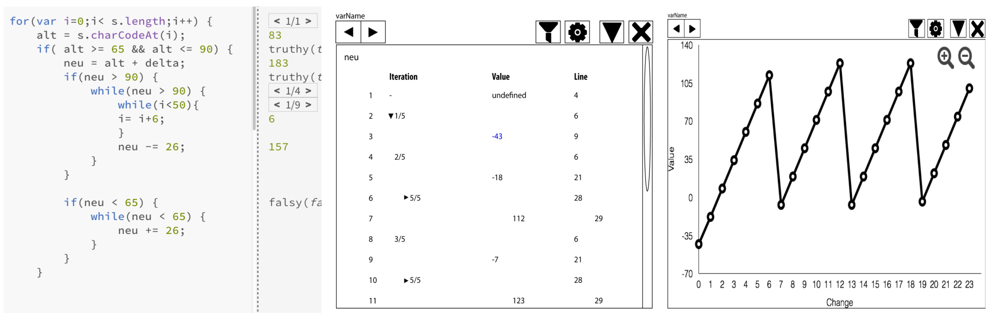
LIVE CODING/PROGRAMMING:
A coding paradigm. The computer continuously executes the program in the background and provides immediate feedback of the program state after each applied change. The feedback can be displayed automatically or on demand.

Definition:
Live Coding/Programming

Recently, increased research on live programming features and tools has been conducted [Choi et al., 2008], Live programming is expected to assist programmers by providing continuous visual feedback

[McDirmid, 2013], [Victor, 2012b], [Guo, 2013]. Live programming intends to close the time gap between programming and debugging by providing immediate feedback while programming. This allows to reduce the four phases of the cycle to just one. Expectations on live programming are, e.g., the minimization of the latency between the visual feedback on a change and it effects on the program, reduction of the strain on the human cognition and the simplification of debugging (faster and more accurate). However, to tap the full potential of live coding, appropriate visualizations are required. Using live programming to provide just arbitrary immediate feedback is not necessarily useful. Victor [2011] states the following in his essay "Ladder of Abstraction":

"The appropriate visualization varies, and often there are multiple good visualizations, each offering a useful perspective. We can look for metrics that summarize some aspect of the behavior. We can also consider transformations that make it easier to visually compare multiple states of the system."



```
for(var i=0;i< s.length;i++) {                  < 1/1 >
    alt = s.charCodeAt(i);                      83
    if( alt >= 65 && alt <= 90) {               truthy(t
        neu = alt + delta;                      183
        if(neu > 90) {                          truthy(t
            while(neu > 90) {                   < 1/4 >
                while(i<50){                    < 1/9 >
                i= i+6;                         6
                }
                neu -= 26;                      157
            }
        }

        if(neu < 65) {                          falsy(fa
            while(neu < 65) {
                neu += 26;
            }
        }
    }
```

neu

	Iteration	Value	Line
1	-	undefined	4
2	▼1/5		6
3		-43	9
4	2/5		6
5		-18	21
6	►5/5		28
7		112	29
8	3/5		6
9		-7	21
10	►5/5		28
11		123	29

Figure 1.1: *Three possible visualizations for a variable of type number. On the left, next to value assignments of type number, the assigned value is shown. The results of assignments in conditions are only visualized if they evaluate to true. This provides a quick overview on the program state. In the middle, a hierarchical list can be seen, which provides a view on all value changes applied to a number variable within its scope. On the right, a visualization in form of a plot is shown, which also illustrates all value changes assigned to the variable.*

For example, a number variable in JavaScript can be visualized in form of a hierarchical list consisting of all value changes over the whole program or as an exact value right to each line consisting of an assignment. In other situations a plot could be the better choice, e.g., when the trend of a variable throughout a loop is of interest (see figure 1.1). Usually, the user is capable of answering the question in which situation, which visualization is the most useful. Therefore, it is important that the user can switch between different visualizations easily.

No appropriate visualization for each situations

Recently, new live coding tools and prototypes, such as the Light Table [1] by Chris Granger or Swift Playgrounds [2] by Apple, have been published, each with a slightly different layout and approach on how to visualize feedback. They enable a programmer to obtain live results in a more detailed way. To give an example, the Light Table enables the user to look at the properties of an object in form of a hierarchical list. The user can click trough the different hierarchy levels to open parts of interest for a more detailed view.
But these new tools and prototypes do not provide functionality for switching between different visualizations. This prevents the programmer to find the appropriate visualization for each situation. The moment their tool chooses not the appropriate one and shows the wrong data, or the right data in a wrong way, the visualized feedback becomes worthless for the programmer.

Almost non tools and prototypes support switching between different visualizations

In this thesis we present a live coding prototype, which enables the programmer to open an interactive visualization for variables and to switch between different visualizations to find the appropriate one for his needs. Furthermore, a programmer can implement his individual visualizations for different data types and add them to the selection. This bachelor thesis is structured as follows:

Present a prototype where different visualizations can be used and added

First, we present in chapter 2 "Related work" inspiring related work and state-of-the-art developments, which we use for comparisons with our own prototype, in the area of live coding. In chapter 3 "Prototype" we present our prototype. We start with a list of reasons why an enhancement of Kurz' prototype is indispensable. Afterwards, we introduce our initial ideas for our layout as well as detail visualizations and illuminate our design process. Then we present our conducted preliminary user study which we

[1] http://www.lighttable.com
[2] https://developer.apple.com/swift/resources.com

used to get some feedback on our concept and design. In the end, we reveal our final design and highlight the applied changes which are based on the user study. Chapter 4 “Implementation” deals with the implementation of our prototype. Since the prototype is an enhancement of Kurz′ prototype, we explain its architecture for a general understanding of its manner of functioning. Then, we describe our newly constructed components, their assigned tasks and the integration into Kurz′ prototype. We precise both in a more detailed way, the management of the frames and the management of the detail visualizations. In chapter 5 “Discussion of our Prototype” we discuss our prototype with regard to its capabilities and limitations. Finally, we describe in chapter 6 “Summary and Future Work” possibilities for further enhancements of our prototype and the need for a future user study after it got refined.

Chapter 2

Related work

In this chapter we present all kinds of previous research and state-of-the-art developments on the topic of live programming. We start with an introduction of the typically used action cycle while programming, its problems and attempts to solve these problems. We then present early research in the direction of live programming, followed by some research on guidelines and taboos regarding the visualization of live feedback. Afterwards, we present various new developments and up-to-date software releases. Finally, we describe some research about the usefulness of live programming.

We present past research as well as up-to-date developments

The assumption that complex cognitive skills are a necessity for programmers exists at least since 1975. Gould [1975] describes in his paper ″Some Psychological Evidence on How People Debug Computer programs″ multiple factors why the error correction task (debugging) is exhausting and cognitive skills are needed. To reduce the strain on the programmer, a high number of debugging tools has been published. They are intended to simplify the debugging process by illuminating the program, its runtime states and the existing data at these states. An example for a debugging tool is the Whyline, a debugging prototype for Alice, a programming environment [Ko and Myers, 2004]. The Whyline allows the programmer to ask predefined questions about the program's runtime failure in form of ″why did″ and ″why didn't″ questions. The set of available ques-

Programming without feedback and debugging takes a great strain on human cognition tions is generated automatically. The tool answers these questions by showing the runtime events which caused the output. In comparison to the usually used approach, stepping through the code after implementing it, this tool simplifies the search for the corresponding parts of code causing the output. To sum up, debugging tools assist the programmer with the error correction task and reduce the strain on the programmer.

However, there is also a strain on the cognition of a programmer while editing source code. Snell [1997] explains, that this strain is caused by the necessity to simulate the whole program flow unassisted while editing. He introduces a tool which already uses the idea of live coding. It merges the edit, evaluate and debug mode (similar to the four phases introduced in the introduction) and provides continuous feedback for the programmer. He calls the method of getting feedback just after typing some code the "ahead of time debugging" feature. But the concept of continuous feedback was introduced even earlier by, e.g., Henderson and Weiser [1985] and their VisiProg environment. They describe in a futuristic way (they built no actual prototype) a system where a programmer receives continuously feedback for his input and illustrates the specifications of their system.

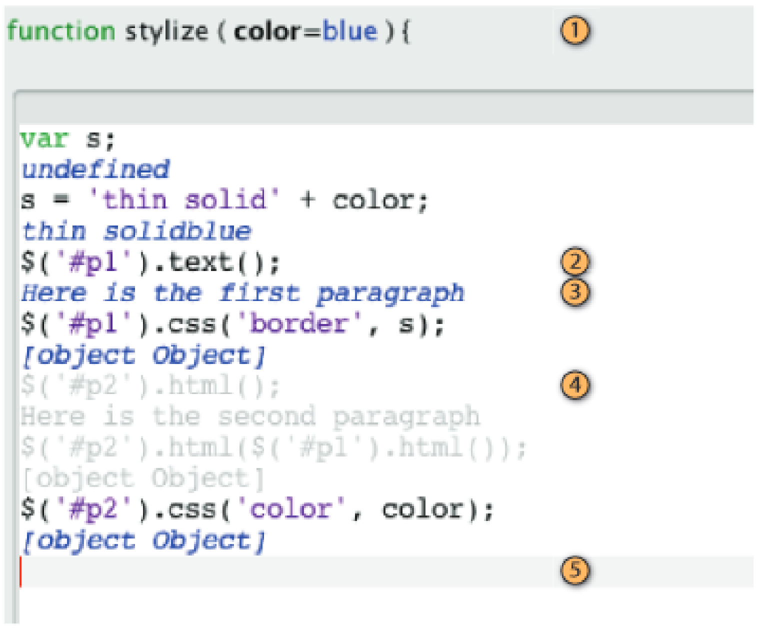
Continuous feedback already introduced 1985 by Henderson and their VisiProg environment

Introduction of VIVA as a language for image processing

The attempt of making programming easier to understand by using not only textual feedback but also more graphical visualizations, such as diagrams or flow charts, was already introduced by visual programming languages. One example is VIVA, a visual language for image processing [Tanimoto, 1990]. In VIVA the programmer can express algorithms by drawing flow diagrams (use of visualizations) instead of writing source code.

Introduction of Rehearse editor and the use of inline visualizations for feedback

Over the last years, plenty of papers, prototypes and tools concerning live coding have been published. One of the earlier tools is the Rehearse editor, introduced by Choi et al. [2008]. Using an inline visualization the programmer receives direct feedback on the results of execution. In addition, undone statements are kept visible to enable backtracking to earlier states of the code (see figure 2.1).



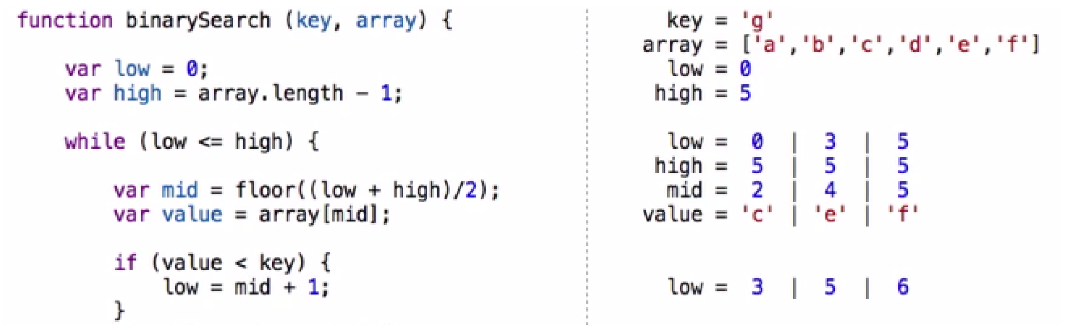
```
function stylize ( color=blue ) {

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

Figure 2.1: *The figure shows a function called "stylize" and its definition in the Rehearse editor. (1) shows the function name, parameters and values. Beneath each line of source code (2), the result of execution is shown (3). Undone statements (4) are still visualized which simplifies backtracking. This screenshot is taken from Choi et al. [2008]*

Another approach of the Rehearse development environment was introduced by Br et al. [2010]. The environment enables a developer to execute and afterwards interact with his application in an extra window. Triggered by each user interaction, recently executed lines become highlighted. The authors claim that using this visualization simplifies connecting results of interactions with the corresponding source code.

A more recent development is the prototype implemented in the programming language YingYang and published by McDirmid [2013]. His prototype uses a combination of probing (an inline visualization) and tracing. Probing enables the user to inspect all kind of variables and expressions by visualizing their current values directly in the next

Probing in form of inline feedback and tracing in form of a second column for marked code



```
function binarySearch (key, array) {

    var low = 0;
    var high = array.length - 1;

    while (low <= high) {

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

        if (value < key) {
            low = mid + 1;
        }
```

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

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


  low =  3  |  5  |  6
```

Figure 2.2: *This figure is a screenshot of Victor's talk "Inventing on Principle" [Victor, 2012b]. On the left side the source code for a binary search method is illustrated. On the right side we can see the used example function parameters for the binary search function, as well as feedback for each time a variable is used. For each iteration of the used while loop a new column for all its variables is created.*

line. Tracing enables the user to place print statements, which then visualize the chosen data in a second column, a so called trace pane. For example, different states of an array can be illustrated. Furthermore, each trace output can be used as a navigation to the source code used to create that line.

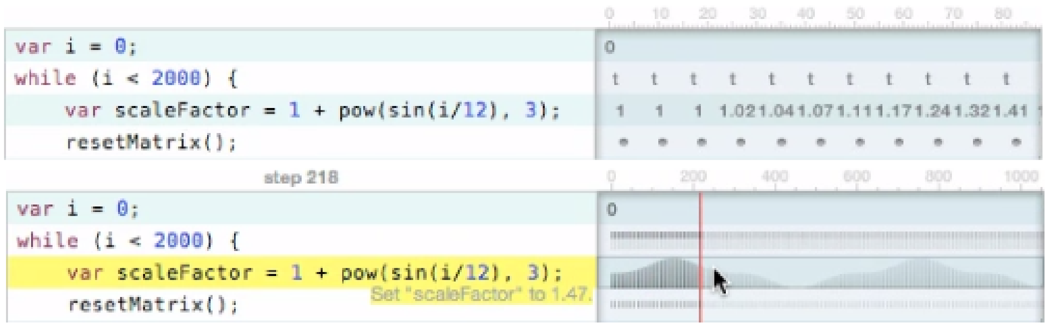
Bret Victor published multiple essays and and held a presentation regarding live coding and visualizations

An influential person in the area of live coding and the design of visualizations for feedback is Bret Victor. He made numerous contributions in the field of live coding and held some talks about it such as "Inventing on Principle" Victor [2012b]. He illustrates plenty of visualization techniques on what and how to visualize feedback. Moreover, he presents in his essay "Ladder of Abstraction" some design guidelines, e.g., that depending on the situation a different visualization with a higher or lower level of abstraction can be appropriate [Victor, 2011]. In his essay "Learnable Programming" Victor [2012a] enumerates multiple criteria a programming environment has to fulfill to make the programming more understandable for the user. Additionally, he names plenty of misconceptions in regard to the content of feedback and its design. Almost all of his examples in his essay "Learnable Programming" refer to the online environment Khan Academy[1] , which was published for beginners to learn programming.

[1]https://www.khanacademy.org/computing/cs

Most of the examples in his essay refer to Khan Academy, since it was a response to this environment. They took his previous talks as an inspiration for it and Victor was not satisfied with the result. Khan Academy's environment offers a live coding feature in form of a two view layout. On the left side the user can write his code and on the right side the result of the program (everything is rendered into a canvas element) is permanently updated, without showing, e.g., intermediate results for variables.

Victor criticizes the live coding environment by Khan Academy and suggest improvements



```
var i = 0;
while (i < 2000) {
    var scaleFactor = 1 + pow(sin(i/12), 3);
    resetMatrix();
```

```
var i = 0;
while (i < 2000) {
    var scaleFactor = 1 + pow(sin(i/12), 3);
    resetMatrix();
```

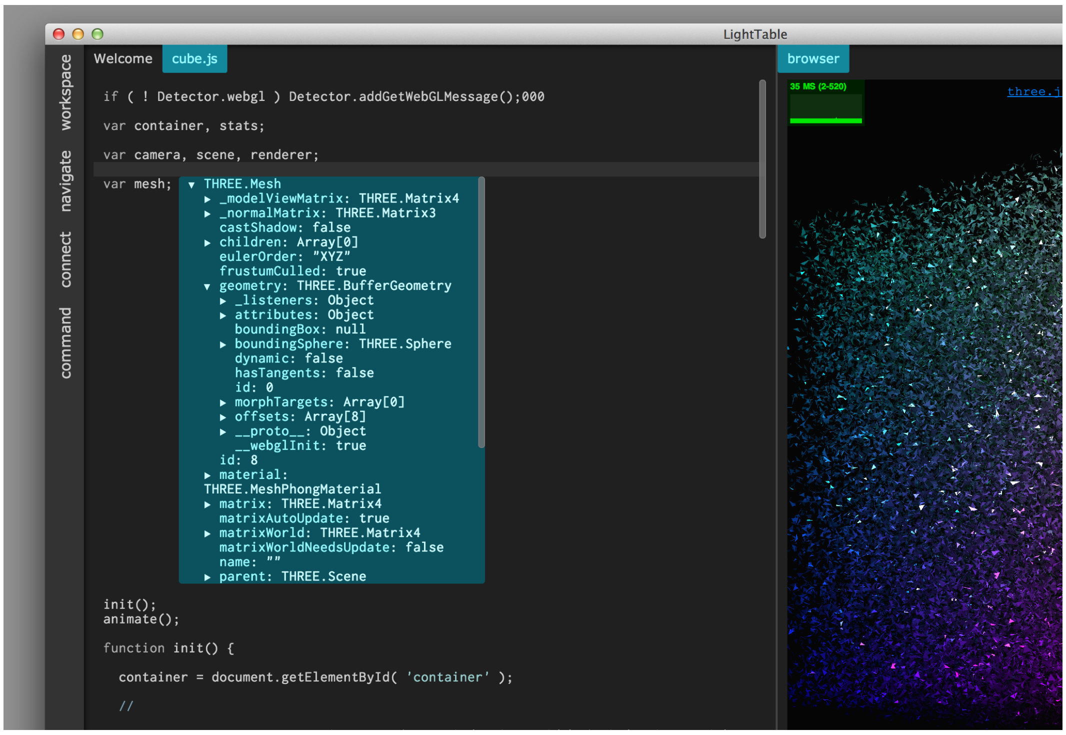
Figure 2.3: *This figure consists of two screenshots from a video used in Victor [2012a]. A source code sample is on the left side of each screenshot. On the right side is the visualized feedback. This visualization is an example for the usage of different abstraction levels. For a small part of the timeline the exact newly assigned value is shown in form of a table. When the programmer zooms out of the timeline the illustration changes from a table to a plot. By moving the mouse over the plot for a variable its concrete values are shown.*

Finally, we present three of the newest published programming environments with live coding features: Light Table [2], Swift Playgrounds[3] and IPython Notebook[4] . All of them offer visualizations of varying abstraction levels, which the other presented prototypes do not. It is interesting to see that all of them actually chose a different layout and placement for their visual feedback.

The Light Table by Chris Granger, for JavaScript, combines an arbitrary number of columns and inline layout for its live feedback (see figure 2.4). Two columns are the standard for the JavaScript version of the Light Table, but the user

Use of a second column and inline visualizations for the live feedback

[2]http://www.lighttable.com
[3]https://developer.apple.com/swift
[4]http://ipython.org/notebook.html



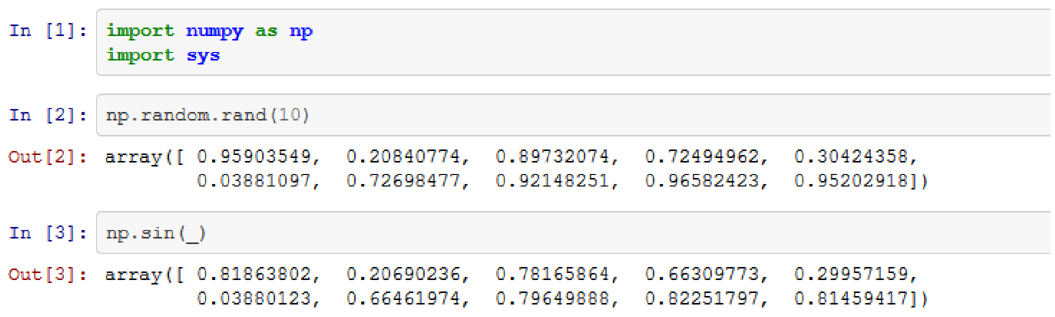
```
if ( ! Detector.webgl ) Detector.addGetWebGLMessage();000

var container, stats;

var camera, scene, renderer;

var mesh; ▼ THREE.Mesh
            ▶ _modelViewMatrix: THREE.Matrix4
            ▶ _normalMatrix: THREE.Matrix3
              castShadow: false
            ▶ children: Array[0]
              eulerOrder: "XYZ"
              frustumCulled: true
            ▼ geometry: THREE.BufferGeometry
              ▶ _listeners: Object
              ▶ attributes: Object
                boundingBox: null
              ▶ boundingSphere: THREE.Sphere
                dynamic: false
                hasTangents: false
                id: 0
              ▶ morphTargets: Array[0]
              ▶ offsets: Array[8]
              ▶ __proto__: Object
                __webglInit: true
              id: 8
            ▶ material:
          THREE.MeshPhongMaterial
            ▶ matrix: THREE.Matrix4
              matrixAutoUpdate: true
            ▶ matrixWorld: THREE.Matrix4
              matrixWorldNeedsUpdate: false
              name: ""
            ▶ parent: THREE.Scene

init();
animate();

function init() {

  container = document.getElementById( 'container' );

  //
```

Figure 2.4: *Shows the Light Table and parts of a JavaScript program. Two columns are currently used and an inline visualization for an object is open.*

can add any desired number of columns. These columns can contain websites as well as other files. Furthermore, a console can be opened at the bottom of the editor. On the left side a programmer can constantly see his source code, on the right side a webpage where the JavaScript code is induced.

Second column shows a website where the JavaScript code is induced

Furthermore, to look at intermediate results, each variable can be inspected in a more detailed view. To give an example, for an object an inline visualization in form of a hierarchical list, containing all its properties, can be opened. The user can open and close properties of interest for further information. Additional information and intermediate results can be also perceived by opening a console at the bottom of the editor. When it is open, it uses the whole width of the editor for feedback visualization. To add results to the console the user has to use the `console.log()` command.



```
In [1]: import numpy as np
        import sys

In [2]: np.random.rand(10)

Out[2]: array([ 0.95903549,  0.20840774,  0.89732074,  0.72494962,  0.30424358,
                0.03881097,  0.72698477,  0.92148251,  0.96582423,  0.95202918])

In [3]: np.sin(_)

Out[3]: array([ 0.81863802,  0.20690236,  0.78165864,  0.66309773,  0.29957159,
                0.03880123,  0.66461974,  0.79649888,  0.82251797,  0.81459417])
```

Figure 2.5: *The figure illustrates an IPython Notebook example. For each line of input, an output line directly beneath that is created whenever an Python object is returned by an expression.*

In IPython a total inline layout is used (see figure 2.5). Every output for an input line, which consists of an expression and returns a Python object, is displayed directly in the next one or multiple lines. Commands, such as `_(last output)`, can be used to access the results later on. One feature, which is similar to our concept, is the offered functionality to choose alternative representations for feedback such as HTML, JSON, PNG or LaTeX.

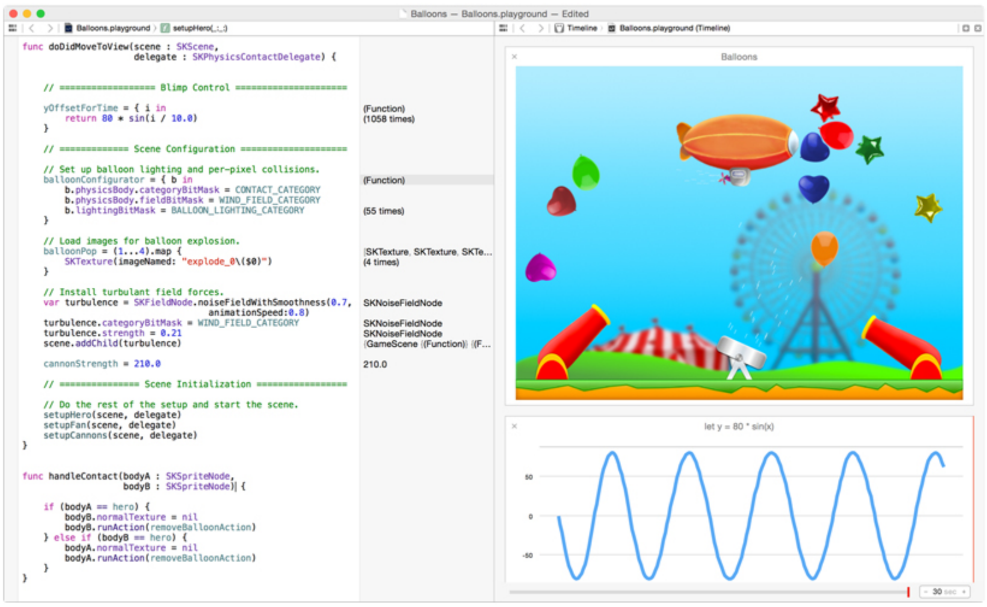
Alternate representations for feedback can be selected

The layout of the recently published Swift Playgrounds (still in the beta phase) builds a strong contrast to that (see figure 2.6). As a standard there are two columns. The left column (called Source Editor Window) contains the source code, while the right one (called Sidebar) contains feedback for every line of source code directly next to it. Whenever a user wants more information, such as more context, a third column named Assistant Editor is used. For each line of feedback in the sidebar, a new visualization in the Assistant Editor can be opened. The Assistant Editor is only visible when there is at least one detail visualization open.

Use of a a sidebar and assistant editor for feedback

For each type of data exists a predefined visualization. The detail visualization, e.g., for a number variable of interest, is a plot which contains all the changes applied to the variable at the chosen line over the runtime. Furthermore, the Swift Playgrounds enable the visualization of a whole application. To sum up, the sidebar is primarily used for textual visualizations with less context around it, whereas the Assistant Editor provides more detailed visualizations, containing more information and possibly with a more graph-

Use of predefined visualizations



```
func doDidMoveToView(scene : SKScene,
                     delegate : SKPhysicsContactDelegate) {

    // ================== Blimp Control =====================

    yOffsetForTime = { i in                                    (Function)
        return 80 * sin(i / 10.0)                              (1058 times)
    }

    // ============= Scene Configuration ====================

    // Set up balloon lighting and per-pixel collisions.
    balloonConfigurator = { b in                               (Function)
        b.physicsBody.categoryBitMask = CONTACT_CATEGORY
        b.physicsBody.fieldBitMask = WIND_FIELD_CATEGORY
        b.lightingBitMask = BALLOON_LIGHTING_CATEGORY          (55 times)
    }

    // Load images for balloon explosion.
    balloonPop = (1...4).map {                                 [SKTexture, SKTexture, SKTe...
        SKTexture(imageNamed: "explode_0\($0)")                (4 times)
    }

    // Install turbulant field forces.
    var turbulence = SKFieldNode.noiseFieldWithSmoothness(0.7,  SKNoiseFieldNode
                                       animationSpeed:0.8)
    turbulence.categoryBitMask = WIND_FIELD_CATEGORY           SKNoiseFieldNode
    turbulence.strength = 0.21                                 SKNoiseFieldNode
    scene.addChild(turbulence)                                 {GameScene {(Function)} {(F...

    cannonStrength = 210.0                                     210.0

    // =============== Scene Initialization =================

    // Do the rest of the setup and start the scene.
    setupHero(scene, delegate)
    setupFan(scene, delegate)
    setupCannons(scene, delegate)
}


func handleContact(bodyA : SKSpriteNode,
                   bodyB : SKSpriteNode) {

    if (bodyA == hero) {
        bodyB.normalTexture = nil
        bodyB.runAction(removeBalloonAction)
    } else if (bodyB == hero) {
        bodyA.normalTexture = nil
        bodyA.runAction(removeBalloonAction)
    }
}
```

Figure 2.6: *The image shows an example of the Swift Playgrounds and their live coding features. It is divided into three columns. The left column contains the source code, the central one short feedback for assignments, functions or loops and the third column shows two detail visualizations. The top one illustrates the result after executing the whole application called "Balloons". Beneath that is a visualization in form of a plot, showing the flight height of a zeppelin over time. It is associated with the variable yOffsetForTime.*

ical representation.

Numerous more prototypes exists, but since we want to limit the number of presented prototypes, tools and concepts to the (in our opinion) most interesting ones we just enumerate a few of them and do not describe them in more detail: Guo [2013] with Python Tutor, Pharo[5], Lively Kernel[6] and Edwards [2004] with the EG prototype.
Finally, after looking at all these live coding tools and prototypes, the question arises how great the benefits of live coding and its continuous visual feedback are.

A study about continuous visual feedback while debugging conducted by Wilcox et al. [1997] showed a neutral result and they concluded, that it could be task and programmer dependent whether live feedback actually is of

[5]http://pharo.org/
[6]http://www.lively-kernel.org/

benefit or not. In their user study each participant was confronted with two Forms/3 programs to debug, one with live feedback and one without. Since the participants were divided into two groups, respectively half of the participants worked on each problem with live feedback. For their user study they used three measurements: debugging accuracy, debugging behavior and debugging speed. Their results showed no overall improvement regarding the accuracy and speed. It was quite task dependent whether live feedback improved the results or made them worse.

Usefulness of live feedback probably task and programmer depend

In another study, conducted by Saff and Ernst [2004b] and presented in their paper "An experimental evaluation of continuous testing during development", the results were slightly in favor of live feedback.

To sum up, the positions and findings about live coding vary and it is still necessary to research in which situations it provides a significant advantage. We believe that the results of future users will be in favor for live coding and more significant. Live coding needs to be researched further before we can tap its full potential. To give an example, there are still different approaches (two columns, three columns, two columns and inline,...) on how to optimally integrate visual live feedback into development environments. It is still unclear which approaches are best suited.

Chapter 3

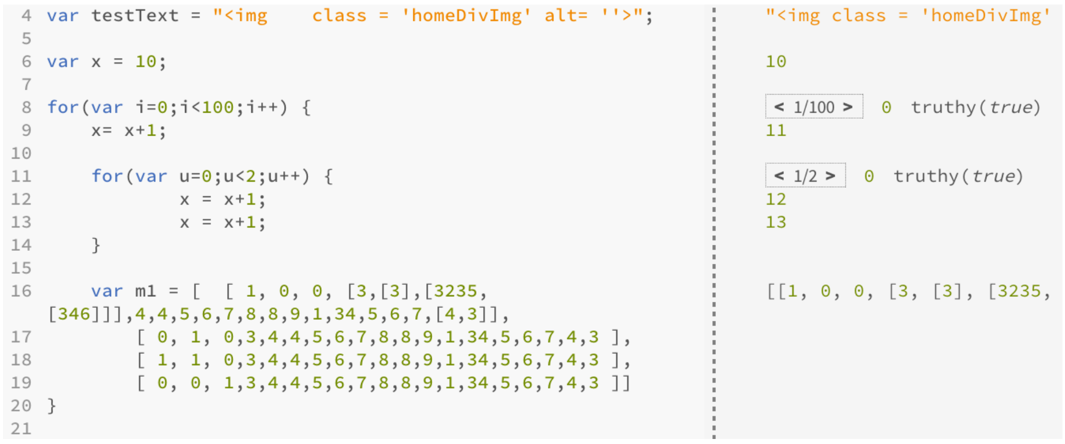
Prototype

This chapter is split into the initial design process, a preliminary user study and our final design

In this chapter we present our prototype, which is an enhanced version of the already existing prototype by Kurz [2013], Heinen [2012] and Belzmann [2013] for JavaScript in Adobe Brackets, illuminate the design process and justify our design decisions. We start with giving reasons for the necessity of an enhancement of the current prototype. Then, we introduce our basic design concept, such as separating the editor in three columns, and detail visualizations. Next, we explain the purpose of our preliminary user study and describe its setup. One reason for this user study was to improve and refine our design. Finally, we present our final design and talk about the applied changes, based on the user study.

3.1 Motivation for our Enhanced Prototype

Our ambition is to develop an interactive and easily expandable software prototype, which enables the user to look at specific variables of interest in detail and perceive their changes while programming. Mainly, as already mentioned in the introduction, it is intended to help the user to keep track of the current state of the program and to fix bugs while programming, without the exhausting task of simulating the program flow and effects of changes to the code in his head. The prototype is going to be used for future user studies after further refinement. One reason is to test the usefulness of our enhanced prototype for programmers. A couple of ideas for possible user tasks will be listed in chapter 6 “Summary and Future Work”. Our prototype is based on the Adobe Brackets plugin by Kurz [2013].



```
 4 var testText = "<img    class = 'homeDivImg' alt= ''>";
 5
 6 var x = 10;
 7
 8 for(var i=0;i<100;i++) {
 9     x= x+1;
10
11     for(var u=0;u<2;u++) {
12             x = x+1;
13             x = x+1;
14     }
15
16     var m1 = [  [ 1, 0, 0, [3,[3],[3235,
   [346]]],4,4,5,6,7,8,8,9,1,34,5,6,7,[4,3]],
17         [ 0, 1, 0,3,4,4,5,6,7,8,8,9,1,34,5,6,7,4,3 ],
18         [ 1, 1, 0,3,4,4,5,6,7,8,8,9,1,34,5,6,7,4,3 ],
19         [ 0, 0, 1,3,4,4,5,6,7,8,8,9,1,34,5,6,7,4,3 ]]
20 }
21
```

```
"<img class = 'homeDivImg'

10

< 1/100 >  0  truthy(true)
11

< 1/2 >  0  truthy(true)
12
13



[[1, 0, 0, [3, [3], [3235,
```

Figure 3.1: *A cutout of the old prototype by Joachim Kurz. The source code is on the left, the live feedback on the right. The user can use the boxes in line 8 and 11 to step through the iterations of the loops.*

3.1.1 Shortcomings of Kurz’ Prototype

Current visualization provides in some cases not enough information

The current live coding editor by Kurz [2013] already provides a rough overview on the state of a program, by showing the immediate results for each assignment and term right next to the corresponding line of source code. Therefore, it is divided into two columns: one for the source code and one for the feedback. We call the first column (source code) from now on **code view** and the second one (visualized feedback) **feedback view**.
In a majority of cases, the current live coding editor with Kurz’ prototype plugin (see the cutout in figure 3.1) is sufficient to spot bugs and responsible parts of code. But occasionally more detailed, clearly laid out visualizations, placed in a larger space than one line, are necessary.

In Kurz’ prototype, we determine the following three major problems we want to solve with our new prototype:

1. **Lack of possible interaction with the visualization, to adapt the representation and amount of information**

   Filtering of information or switching between visualizations is not possible with Kurz' prototype

   There is almost no interaction possible with the current prototype. It is not possible to switch between different visualizations, hide any undesired information or get additional one. For example, when considering a nested object, initially reducing the shown information to the top-level properties could prevent from the confrontation with a large amount of unnecessary information. If required, the user can open properties of interest by himself.

   Using the current live editor, monitoring a couple of selected variables and their changing values can become nerve-racking. The user has to look for all lines where value changes are applied to the monitored variable. That is time consuming and not efficient.

2. **Partly unfitting illustration of information**

   Often not enough space for an appropriate visualization

   In the current prototype each visualization for an applied assignment has the space of exactly one source code line. More precisely, each visualization is about 25 pixel high and infinitely wide. But for visualizations with a width above the width of the feedback view, a scrollbar shows up and only a part of the whole content is visible at the same time. That space is sufficient for the visualization of a single number as a result of a term, but not for most of the other data types, such as an array or an object. In addition, data such as HTML elements, cannot be illustrated properly by visualizations with a textual representation.

   Possibility of a less abstract view is missing

   In these cases a more graphical representation is essential. One possibility is to take the HTML element and render it.
   It becomes notable, that in some cases a visualization needs more space than one line, to allow a clear layout.

3. **Missing context and additional information**

It occurs that context is missing

The current prototype provides for each line in the code view exactly one line in the feedback view. These lines are located right next to their corresponding lines in code view. Thus, a line of code, which changes the value of a variable, and its result are always located next to each other. That can become a problem when, e.g., changes to a variable are applied at different locations in the program. The user has to scroll through the program to compare how its value changes.

Bret Victor states the following in his essay "Learnable Programming":

"Data needs context. It is rarely enough to see a single data point in isolation. We understand data by comparing it to other data." Victor [2012a].

Therefore, the current illustration of loops and value changes during all iterations implies another problem. While clicking through the iterations, the programmer has to remember previous values of an iteration to compare them. Thus, it is difficult to observe value changes. Also, additional information such as the prototype of an object or the length of an array are missing.

3.2 Initial Design Ideas

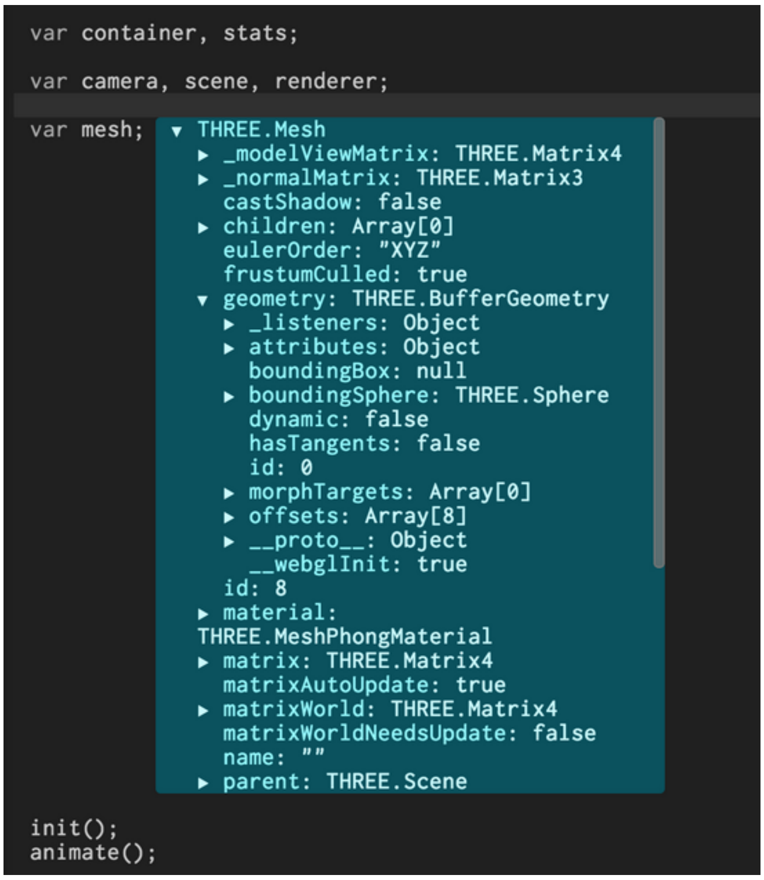
Space for visualizations is limited, therefore thoroughly planing of the layout is essential

During our development process we had to make multiple design decisions concerning how and where to realize our detail visualizations. The limited space for the detail visualizations is a challenge, moreover since the code and feedback view already take a great part of the available space. We looked at various already existing live coding prototypes and tools, collected ideas and finally came up with designs for our separate detail visualizations and our underlying layout.

3.2.1 Three Columns Layout

Since the available space in an integrated development environment (IDE) is quite limited, it is necessary to think about where to place the visualizations for the live feedback and how much space to provide for them. One possibility is to use an inline approach, such as it is done in the Light Table [1], placing the visualizations in the same space as the source code (see figure 3.2).

Inline visualization is one possibility for placing feedback



```
var container, stats;

var camera, scene, renderer;

var mesh;  ▼ THREE.Mesh
             ▶ _modelViewMatrix: THREE.Matrix4
             ▶ _normalMatrix: THREE.Matrix3
               castShadow: false
             ▶ children: Array[0]
               eulerOrder: "XYZ"
               frustumCulled: true
             ▼ geometry: THREE.BufferGeometry
               ▶ _listeners: Object
               ▶ attributes: Object
                 boundingBox: null
               ▶ boundingSphere: THREE.Sphere
                 dynamic: false
                 hasTangents: false
                 id: 0
               ▶ morphTargets: Array[0]
               ▶ offsets: Array[8]
               ▶ __proto__: Object
                 __webglInit: true
               id: 8
             ▶ material:
             THREE.MeshPhongMaterial
             ▶ matrix: THREE.Matrix4
               matrixAutoUpdate: true
             ▶ matrixWorld: THREE.Matrix4
               matrixWorldNeedsUpdate: false
               name: ""
             ▶ parent: THREE.Scene

init();
animate();
```

Figure 3.2: *A screenshot from a video introducing the Light Table. It shows an inline visualization. The source code is pushed away by the detail visualization of a variable named mesh.*

We decided against this approach, because with each line

[1] http://www.lighttable.com/

Inline approaches rip apart the alignment and less source code is visible

consumed by the visualization, the alignment of the source code is altered further and potentially important context disappears from sight. An inline approach could also be applied in the feedback view, but the main idea of that view is that each result is aligned right next to its associated line in the code view. An inline approach would destroy that concept.

Another possibility is to use some vertical space of editor, as it was done by the canvas view in the SuperGlue environment presented by McDirmid [2007], where the visualizations are appended at the bottom in a separate field. However, todays monitors usually have an aspect ratio of 16:9 or 16:10 and offer more horizontal than vertical space. That implies to rather use up the horizontal than the vertical space.

Pharo creates for almost each task a new window, but that provokes disorder and conceals the other windows

In Pharo[2], an IDE wich provides immediate feedback, for a high number of tasks a new indivudal window is created. The windows can be placed wherever the user wants to place them. We believe this solution is open for disorder caused by the user and not properly arranged by its own. Moreover, whenever a new additional window is created, it overlaps existing ones and hides their content.

We use a three column layout

Finally, we decided to use a three column layout, like it is used in the newly published Swift Playgrounds[3]. We took the old prototype and added a third column, which we call **detail view**, for the detail visualizations. Starting from left, the code view is still located in the first column and the feedback view from the old prototype in the second one. Using this layout, we have to think about how to split the available space between these three different views. Since the required space for each view can change depending on the current situation, we made the columns resizable.

[2]http://pharo.org/
[3]https://developer.apple.com/swift

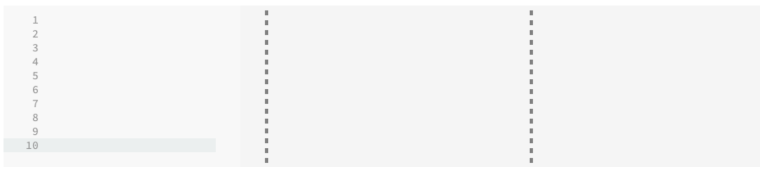


Figure 3.3: *The UI is divided into three columns. From left to right the different views: code, feedback and detail.*

3.2.2 Each Variable in Its Own Frame

Our goal is enabling a detail visualization with possible interaction for each variable. In our prototype each visualization is opened in an own "private" frame by clicking on an interesting value in the feedback view. This frame is, when opened for the first time, positioned at the same height as the associated line, pushing aside overlapping frames. In case the user wants to open an already existing frame, the existing frame becomes repositioned to the position of the associated line. We use this repositioning to point out the link of the frame, which is currently of particular interest.

Each variable is created in its own personal frame with its own settings and options

Each frame consists of a menu bar and an area for the actual detail visualization underneath. The menu bar contains the menus shown in figure 3.4 and provides functionalities such as switching between different visualizations, filtering of specific values or closing the detail visualization.

Menu bar offers diverse functionalities to adapt visualizations and the shown content

There are two main arguments for our "independent frame concept": First, every detail visualization having its own frame results in a clear visual separation of all visualizations. In addition, establishing a link to the code view as well as the feedback view becomes less difficult. Second, every detail visualization can be adapted independently from the rest. The visualization in one frame can be changed without influencing the others.

There is a clear visual separation of all visualizations and individual adaption is possible

Furthermore, it is still possible to add global functionality, affecting all frames, without additional work. The disadvantage of this solution is, that each frame and menu-bar takes up a bit of the limited space. Comparing the advantages and the disadvantages of this approach it seems to us that the idea of separate frames is worth to be explored.

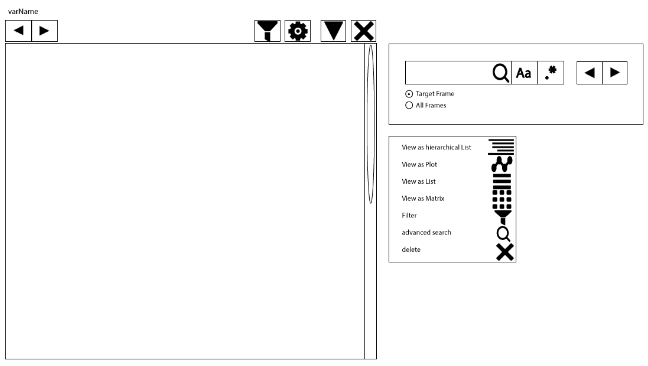


Figure 3.4: *One of the first ideas for the design of the frame. On the left side of the frame are the two buttons for the navigation and above them the associated variable name of the frame. On the right side we put the other buttons for the filter menu, option menu, minimization and closing. Right next to the frame you find the designs for the filter and option menu. At the top is the filter menu, designed like the search menu in the Brackets editor. Beneath that is the option menu, containing possible visualizations and options.*

Each frame offers further functionality such as dragging or resizing

All frames have further functionality. For example, to conform the height and position of each frame to the current situation, they can be resized and dragged. When dragging a frame to the position of an other frame, they are automatically positioned side by side, which enables a better comparison between them.
The menu bar provides one more feature: a navigation function for nested data types, similar to the navigation on websites. When a nested property of an object is opened, the navigation can be used to switch to the visualization of the parent object and backwards. This navigation functionality has to be implemented for each visualization independently and for several ones it does not make any sense to implement such functionality. Due to a lack of time we did not implement this functionality but reserved space for it and kept the idea in our design, in order to add it later on.
The set of possible interactions with an actual detail visual-

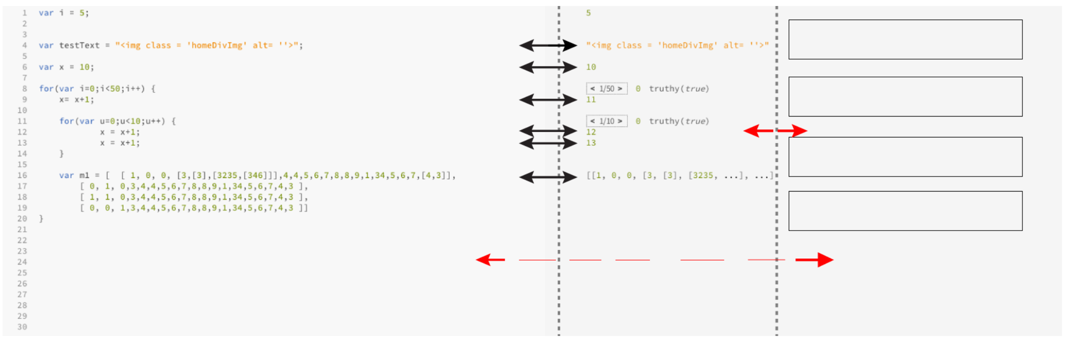
ization depends on its programmer.

Offering numerous features and possible interactions to the programmer represents a trade off. On the one hand, it can increase the value of visualizations and, e.g., help the programmer to find errors and bugs. On the other hand, the user can become too distracted by the visualizations and as a result uses too much time to interact with them. During this time, the programmer does not write any source code. We took that risk into consideration and still believe that offering features to adapt a visualization and the displayed information provides more advantages than disadvantages.

3.2.3 Linking the Three Columns

A link between source code and its illustrated feedback is important

In his talk "Inventing on Principle", Victor [2012b] mentions the necessity of a connection between source code and its output. When using live programming to provide immediate visual feedback, it is also important that the user can see the link between the visualizations and the used data. Without that link, the feedback is of no use to the user. He does not know on which data the visualization is based.



```
var i = 5;


var testText = "<img class = 'homeDivImg' alt= ''>";

var x = 10;

for(var i=0;i<50;i++) {
    x= x+1;

    for(var u=0;u<10;u++) {
            x = x+1;
            x = x+1;
    }

    var m1 = [  [ 1, 0, 0, [3,[3],[3235,[346]]],4,4,5,6,7,8,8,9,1,34,5,6,7,[4,3]],
        [ 0, 1, 0,3,4,4,5,6,7,8,8,9,1,34,5,6,7,4,3 ],
        [ 1, 1, 0,3,4,4,5,6,7,8,8,9,1,34,5,6,7,4,3 ],
        [ 0, 0, 1,3,4,4,5,6,7,8,8,9,1,34,5,6,7,4,3 ]]
}
```

```
5

"<img class = 'homeDivImg' alt= ''>"
10
< 1/50 >  0  truthy(true)
11
< 1/10 >  0  truthy(true)
12
13

[[1, 0, 0, [3, [3], [3235, ...], ...]
```

Figure 3.5: *The screenshot illustrates the missing link from the code and feedback view to the detail view. The black arrows represent an established link between each line in the feedback and code view. The red arrows mark the missing link from the feedback and code view to the detail visualizations in the detail view.*

In Kurz' prototype this link could be established without effort, since the visualizations and their associated lines in the code view are in a *line-to-line relationship*, as you can see in figure 3.5. Additionally, the corresponding line in the source code is highlighted while the mouse is over an visualization.

By introducing the detail view in a third column the establishment of a link between all views becomes more complicated

Unfortunately, it is not that simple when considering our detail view. One problem is the increased amount of space necessary for appropriate detail visualizations, in comparison to the ones in the feedback view. As a result, the line-to-line concept can not be applied for our detail view (see figure 3.5). Even in case that a visualization is created at the same line, the distance to its corresponding line in the code view prevents the user from recognizing the link without difficulties.

To create a link we use highlighting, displaying of the variable name and connecting lines such as used in the version editor in Xcode

We came up with multiple solutions to make the link between the three views more transparent. First, we had the idea to use highlighting like it is done in the prototype by Kurz. While the mouse is tracked over an element in one view, the corresponding parts in the other views are highlighted. Second, each frame contains the associated variable name. Unfortunately, this method connects only the code and detail view and there can be a lot of independent variables with the same name. Therefore, we planned in addition to render lines in order to connect a line in the feedback view with its associated frame in the detail view. We perceived this approach in the version editor of Xcode.

These lines are only visible while the focus of the programmer lies on a concrete frame or line. Through the already existing link between the code view and the feedback view, the link between the code view and the detail view is also improved. By combining these features we wanted to establish a transparent link between all three views.

Synchronized scrolling between the detail view and the other views makes no sense

Additionally, another problem is caused by using an increased amount of space for each detail visualization and enabling the dragging of them. The scrolling between the code and feedback view can be synchronized because of their line-to-line relationship. In contrast, synchronizing the detail view with the others is problematic. After scrolling down to another frame in the detail view, multiple lines of the code and feedback view are no longer visible.

The other way around, when scrolling in the code or feedback view, the same detail visualization stays in the field of vision far too long.
Xcode has a similar problem in its version editor. For example, whenever the newer version of a function is as twice as long as the old one, synchronized scrolling can cause an undesired result. The same function could be still visible on one side, while being already out of the field of vision on the other. This problem is solved by a simple mechanism. When scrolling in the version with the larger part of code (our detail view), scrolling in their version with shorter parts (our code and feedback view) is comparable slower and vise versa. We included this method in our design, but unfortunately had not enough time to implement this feature. Furthermore, whenever all lines of code associated with a detail visualization are no longer visible, the detail visualization should be moved out of the way automatically to open space for other detail visualizations.

A solution is to adapt the speed of scrolling respectively for each view

3.2.4 Grouping of Data Types and Their Detail Visualizations

Higher level program languages always have multiple data types. For some data types using the same detail visualizations is reasonable and they can be grouped together, for others it makes sense to divide them into different subtypes. Before considering to design detail visualizations, splitting and grouping of data types is advantageous. Thereby, an individual set of visualizations can be created for each group. Additionally, one of the visualizations of each group has to be selected as the standard visualization, which is created when a user wants to open a detail visualization for a variable of this group. The visualization, which is most likely useful in common cases, should be selected.

We perform a grouping and splitting of the JavaScript data types and assign each group a standard visualization

JavaScript basically incorporates the following data types: `object`, `array`, `number`, `string`, `boolean`, `undefined` and `null`. We decided to create detail visualizations for these different data types but to do a further distinction between `strings` and `HTML`, as well

Arrays are split into one, two and multidimensional arrays

Undefined and null are grouped together

as between `one dimensional`, `two dimensional` and `multidimensional arrays`. We distinguish between these types of arrays, because of their different grade of nesting. Thus, there are diverse possibilities to visualize them. Furthermore, we group undefined and null together since we do not add any visualizations for both of them.

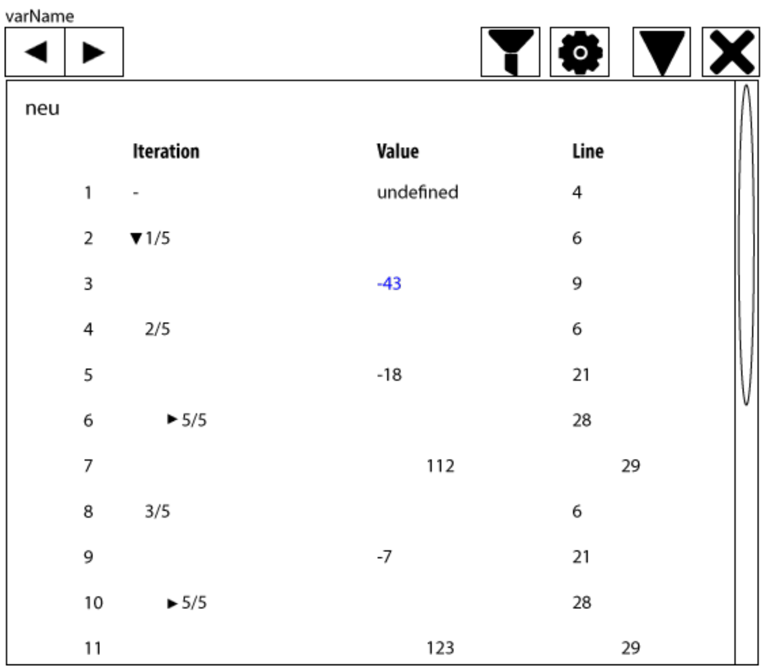
Of course there could be done further distinctions, in case of too many visualizations. For example we could divide our number visualization group into groups such as: plots, textual representations, bar charts and so forth. However, we do not need such a distinction, since our target is not to optimize our prototype for switching between hundreds of visualizations, as it would be necessary for an actual live coding tool. We want to research how much detail visualizations and the ability to switch between them assists programmers with their programming tasks. For that we do not need such a high number of visualizations.
We came up with visualizations for each of our selected groups, except for undefined, null and boolean and explain our design decisions for the visualizations of data types we implemented in detail: `number`, `object` and `HTML`.

A more graphical representation, illustrating more context is necessary

Number Visualization As shown by Victor [2011] multiple visualizations, with diverse levels of abstractions, exists for a variable of type number. One possibility is a textual representation, visualizing an exact value as the result of a term, like it is done by Kurz [2013]. This visualization provides the information whether a term returns the expected result. It hides information such as the general trend of the value, previous values or intermediate results within the term. A different visualization, which includes more information about the context, is necessary.

Unfortunately, including context, such as the value changes within a loop, can induce space problems. We came up with two designs, showing information with different levels of abstraction:

- A *hierarchical list* which contains all value changes applied within the scope of a number variable. This vi-



	Iteration	Value	Line
1	-	undefined	4
2	▼1/5		6
3		-43	9
4	2/5		6
5		-18	21
6	►5/5		28
7		112	29
8	3/5		6
9		-7	21
10	►5/5		28
11		123	29

Figure 3.6: *The figure shows a hierarchical list divided into the three columns: Iteration, Value and Line, containing all changes of a variable over its whole scope.*

sualization provides the user with the information in which line a new value was applied, the exact value and in case it happens within a loop, the number of the corresponding iteration. Therefore, as shown in figure 3.6, the list is divided into three columns: iteration, value and line. In the iteration column the user can hide unessential information or look at (nested) loops in more detail, by closing and opening of iterations. Examples for another possible use are the comparison of the exact results after each iteration of a loop or the identification of an unintentional use of a global variable.

A hierarchical list visualizations divided into the columns: iteration, value and line

So, we identify three important design decisions when using a hierarchical list: the shown content of the initial hierarchical list, the effects of opening and closing loops and the order of the applied changes within the scope of the associated variable.

When a number visualization is opened by clicking on a value in the feedback view, which is assigned within a (nested) loop, all loops and iterations containing the assignment are opened automatically. We assume that the context of this specific change is of interest for the user, since it was used to open the detail visualization. Thus, we anticipate less necessary interactions till the required information are displayed.

Trade-off between necessary number of user interactions and amount of potential unnecessary displayed information

There is a trade-off relationship between the necessary number of interactions, to find the information of interest, and the amount of potentially unnecessary displayed information. For example, when opening a closed loop, it is possible to increase the amount of information in one step more or less.

We decided for higher amount of information

One option is to visualize all applied changes during each iteration. Another option is to visualize all iterations as closed loops by showing just the last assigned value after each incrementation. When using the second method, the user has to open the closed loops of interest on his own, potentially leading to more necessary interactions. To reduce the number of necessary interactions, we decided for the first option and accepted a larger amount of information. A future user study is necessary to show if users prefer rather a large amount of information and less interactions or vice versa.

Order of execution path vs order of appearance

The last design decision is about the order of the applied changes within the scope. There are two possibilities to sort the results: order of execution path and order of appearance. In this visualization the order of appearance is used, which yields the advantage that all calls of a specific function and the resulting changes to a variable are grouped together. Again, it is not clear whether this is the better choice or not and a user study is required to determine the preferable option.

Decided for order of appearance

A graphical illustration in form of a plot

- A *plot* which also displays all value changes applied within the scope of a variable (see figure 3.7). This visualization provides a more graphical view by plotting the development of a variable and its changing values, sorted in the order of appearance. Through this view the detection of outliers is simplified and the programmer can check if the general trend looks right. Such a graphical view could be useful, e.g., to check the implemented flight path of an object (gallons example in swift playgrounds displays a plot to illustrate the flight path of a zeppelin). We did not implement this visualization yet.

One problem of this visualization is that the scale is determined by the highest values on the y-axis and the number of changes (x-axis). That bears the risk to prevent the user from getting any concrete values on specific parts of the program. Therefore, we enable a zoom in both dimensions at the same time at one point of interest. This can be also used to look at a specific intervals, such as a loop. Furthermore, the plot provides more concrete information, such as the line number or the exact value, by moving the mouse over a specific point on the plot. This method limits the amount of unnecessary information displayed at the same time and the visualization is still capable of providing exact values and information.

Zoom control enables user to change the level of abstraction and consider specific parts

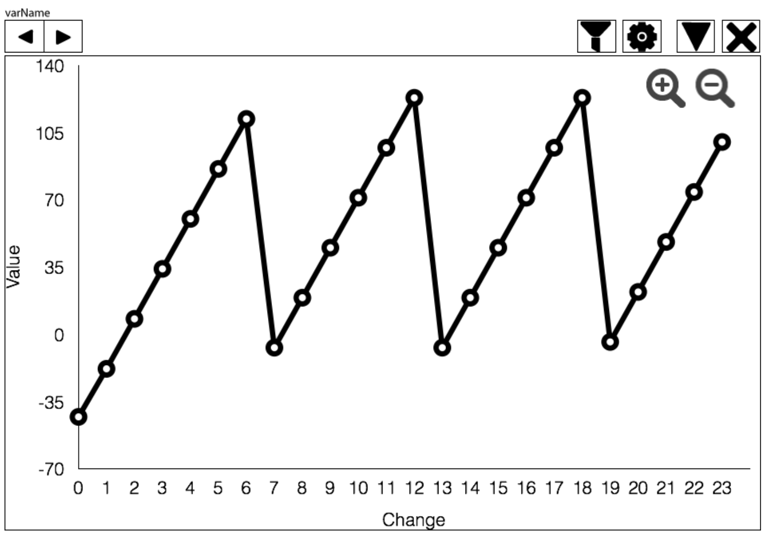


Figure 3.7: *A plot which contains all value changes of a variable. The values are located on the y-axis, the number of the current change is located on the x-axis. Each point in the plot marks a change. The magnifiers can be used to zoom in or out.*

This visualization is just one of many possibilities. Especially for quantitative data (in our case numbers) exist plenty of graphical representations such as quantile plots, histograms or visualizations for multidimensional data spaces. An example of a tool for the visualization of multidimensional data spaces is *Parallel Coordinates* introduced by Inselberg and Dimsdale [1990]. It represents multidimensional data in a two-dimensional visualization. Plenty of other useful visualizations are presented by Tufte [2001] in his book ”The Visual Display of Quantitative Information”.

Objects can have deeply nested properties

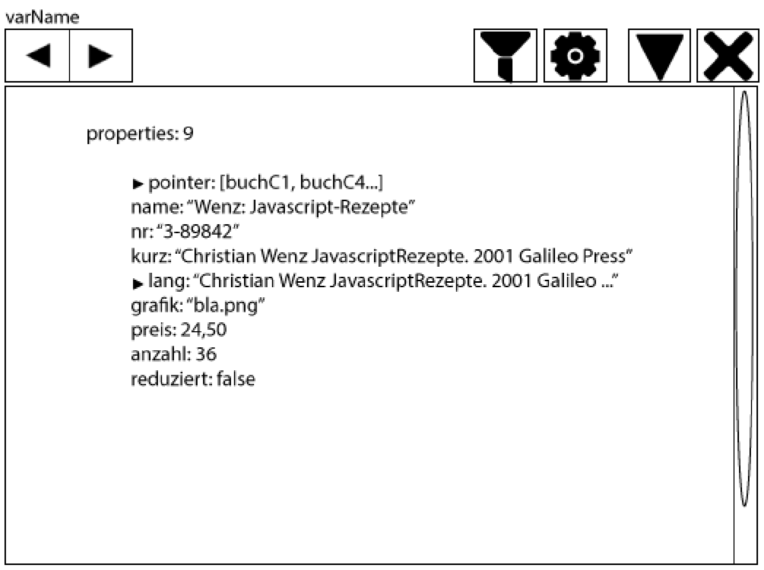
Object Visualization An object in JavaScript can contain an unlimited number of properties of each data type. For instance it can contain, multidimensional arrays or objects, which possibly contain an object element on their own. Since it might be just one or two properties which are of interest for the programmer, it is important to limit the amount of displayed information. In addition, all top-level properties and nested ones are also variables. A programmer can access them, change their value or pass them to a method.

Objects are visualized in form of a hierarchical list

For all properties except the ones of type object an extra detail visualization can be opened

We came up with a hierarchical list visualization where each property of type object can be opened in the same frame. Properties of other variable types are, when opened, created in a complete new frame with the standard detail visualization of their group. Each property is visualized in a private line, containing the name of the property and either an exact value or a preview. A preview is used for properties, such as arrays or HTML elements, which are opened in a new detail frame and can not be compressed to a specific value. It displays information about a property without creating an extra detail view. One benefit of our visualization is the possibility to look up all offered functions and accessible variables of an object, without the necessity to use a documentation. That matches one of the principles used for the Light Table: ”You should never have to look for documentation” [Granger].

Finally, we added one additional detail information to our design. The first property of an object is a list, consisting of all variables which have a pointer to this object. This is



properties: 9

▸ pointer: [buchC1, buchC4...]
name: "Wenz: Javascript-Rezepte"
nr: "3-89842"
kurz: "Christian Wenz JavascriptRezepte. 2001 Galileo Press"
▸ lang: "Christian Wenz JavascriptRezepte. 2001 Galileo ..."
grafik: "bla.png"
preis: 24,50
anzahl: 36
reduziert: false

Figure 3.8: *Visualization of an object in a hierarchical list form. The first property contains all variables which have a pointer at the visualized object.*

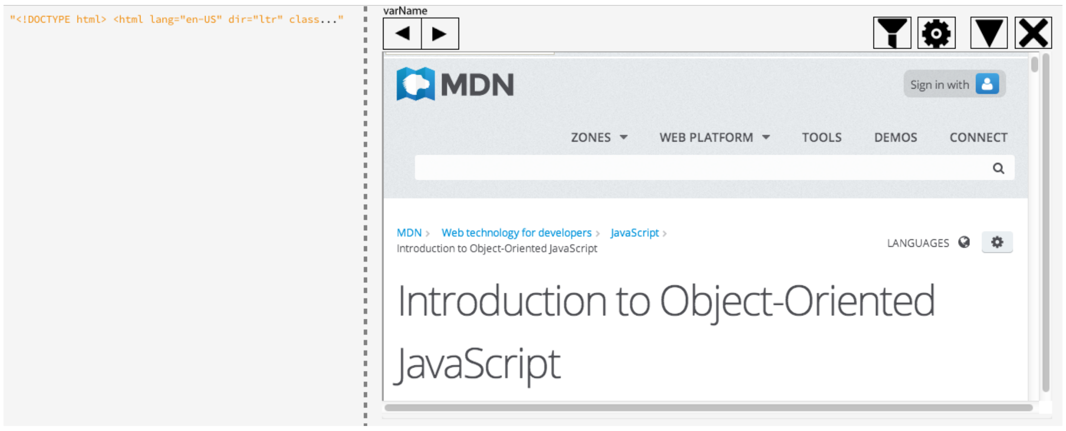
used to show the variables which are directly influenced by changing the selected object (see figure 3.8).

HTML Visualization JavaScript was primarily created to work with HTML elements, to create and alter them. It can access the DOM elements of a website with a list of selectors. To simplify the adaption of HTML elements, jQuery[4] was created. It is a powerful library which assists with selecting as well as changing elements and provides some additional functionality. The result of these changes becomes visible in form of a changed website. It contains new, altered elements or additional functionality.

The process of website creation can be divided into a cycle of three steps: positioning or appending an element or functionality, reloading the page and finally checking whether it is the desired result. The visualization can be used to control the appearance of elements and their position before appending them to a website, which is intended

HTML visualization simplifies the creation of websites and the positioning of elements

[4]http://jquery.com/


"<!DOCTYPE html> <html lang="en-US" dir="ltr" class..."

varName
MDN
Sign in with
ZONES
WEB PLATFORM
TOOLS
DEMOS
CONNECT
MDN > Web technology for developers > JavaScript >
Introduction to Object-Oriented JavaScript
LANGUAGES
Introduction to Object-Oriented JavaScript

Figure 3.9: *The figure shows on the left a XML website request. The resulting HTML string is assigned to a variable. On the right a frame and the detail visualization for the HTML string is illustrated (the webpage).*

to accelerate the creation of websites.

HTML strings are recognized and rendered

Our HTML visualization enables the user to look at one or multiple elements immediately after they were created, by rendering them into their detail visualization frame. Even a whole website can be displayed (see figure 3.9). Also possible is the usage of a CSS document for changing the style of HTML elements.

Summary of Other Visualizations First, our string visualization is identical to our number visualization, except that it shows string values instead of number values. Next, our array visualizations. One dimensional arrays are illustrated as a list, where each array element is initially closed (not whole content is visible) and can be opened by the user. We display a preview for each array element. For two dimensional arrays we have two possible visualizations: a hierarchical list and a matrix. For the hierarchical list we also display previews of the content of the arrays. Furthermore, to illustrate the number of dimensions and to simplify accessing the shown values, we display colored brackets and color their corresponding indices the same. The multi dimensional array is visualized as the same as a two dimensional array. It is the same except for a deeper level of nesting. Images of the other visualizations can be found in appendix A.

3.3 Preliminary User Study

Munzner presents a model of visualization creation

In her paper "A Nested Model for Visualization Design and Validation" Munzner [2009] presents a model of visualization creation. It contains the following four nested layers in a descending order: domain problem characterization, data/operation abstraction design, encoding/interaction technique design and algorithm design. Each of these layers represents one category which threads the validity of a design. In our case, the domain problem characterization is pretty clear and refers to the task of simultaneously programming and debugging. The next two layers are of interest for the user study. In our case, they address the challenge of using the correct data type for the right data and for these data, appropriate visualizations.

Conduction of a user study to get some feedback and assure the potential of our visualizations

Before we start with the implementation we wanted to assure that our detail visualizations are meeting the requirements of Munzner [2009]. Furthermore, we want to get ideas for further improvements, expose confusing parts and show that there are actual situations where our visualizations help to understand source code or solve a problem.

3.3.1 Setup and Tasks

To acquire brief feedback and some opinions on our visualizations we conducted a qualitative user study. Therefore, we presented five people our concept and visualizations in form of low fidelity paper prototypes. The persons were on average between the age of 22 and 27 years old. All of them used JavaScript once before but were not experienced JavaScript developers.

The process was roughly the same for each participant. We presented them an incorrect source code sample, explained what the sample originally was supposed to do and tested if they could solve the problem, or at least notice which results were buggy through our visualization. We asked

Process consists of: presentation of code scenario as well as our visualizations and interviewing them

them if there are not self-explanatory parts of the visualizations or frame. Furthermore we asked for suggestions for improvement and let each participant explain our own frame and visualizations to us. We used these explanations to control, if the participants really perceived the visualizations as we supposed them to. In other words, if our mental model, the system image and the users mental model match [Norman, 2002]. Additionally, a couple of design problems were exposed.

For each type of visualization we created an own scenario

Due to the fact that all visualizations utilize their potential in different situations we came up with a different setup for each group of visualizations. An object visualization for example is not useful when there is an error within a loop consisting of numbers. However, we describe only the scenarios of visualizations we also implemented.
When creating scenarios for the different visualizations, it is a challenge to keep them simple enough to be understood easily and in an adequate amount of time, but to provide at the same time situations where more detailed visualizations than in Kurz' prototype become necessary. Hence, the setups have a variable level of difficulty and a variable needed amount of time to explain them. These differences pose no problems for the user study, since we do not want to compare the visualizations and its assessment. Rather we want to get a short review for each one individually.

Difficult to find meaningful scenarios of a similar length as well as complexity

HTML Visualization Scenario We chose a simple setup for the HTML visualization. A HTML image element which uses the wrong CSS class is assigned to a variable. Thus, a placeholder image was rendered into a detail frame instead of the intended image in form of the Google logo.

Object/1-dim Array Visualization Scenario For the object visualization we implemented a small online shop-alike structure where one could add products to a sales list. The error was, that the same product could be added to the list of sales (in form of an array) multiple times.

Number Visualization Scenario We used a small base64 encoding algorithm for the number visualizations. The algorithm of a base64 encoding is quite short and in our mind easy to understand. More important, no specific knowledge of JavaScript is necessary to understand it. We implemented an incorrect exit condition for one used `while loop`, which led to a wrong result of the encoding.

3.3.2 Results

Participants believe the visualizations look promising

All participants stated that the visualizations have potential and could prove as useful in programming tasks. However, at the same time they pointed out some shortcomings of our visualizations and partly suggested possibilities to improve them. The number of shortcomings were quite different for each visualization. Whereas the HTML visualization was not criticized at all, four of the participants found multiple flaws in the number visualizations.

Illustration of the line of the initial loop declaration is irritating

First, the illustration of the location (line) for each initial loop declaration in the hierarchical list visualization is confusing. Second, all of the participants criticized the design of an opened loop. The privation of a button to close the loop again at only the line of the first iteration and not at all iterations irritated them. Two of these participants also remarked that they expect the visualization to automatically open all loops which contain the line used to open the detail visualization. This was fortunately exactly as we designed it but since we presented them only a paper prototype that behavior was not observable. When we presented our plot visualization, most of them remarked the absence of exact values. They expected the visualization to provide in addition to the value and the line, also the number of iteration (when the mouse is moved over a specific point in the plot). All of the participants preferred the list visualization as the standard visualization. They mentioned that in their opinion the plot is less suitable for a majority of situations.

Hierarchical list is the preferred standard visualization

The navigation icons are not self-explanatory

For our menu bar we also got some suggestions for improvement. It was not obvious for the participants that the two arrows on the leftmost side of the menu-bar could be used as a navigation. None of them understood the use of these arrows at first sight.

Participants want to open a new frame through interaction with the code view

The last point of criticism was about our design decision to initialize the creation of a visualization by clicking on a line of interest in the feedback view. Four of five participants wanted to be able to open it somehow by interacting with the code view. They did not find it obvious to use the feedback view for that.

3.4 Final Design and Applied Changes

By using the results of our preliminary user study (see subsection 3.3.2) we changed and improved our design of the frame, the object and both number visualizations.

Appending the opening of a new frame to the code view is possible by using the right click menu of the code view or offering of an extra method

Most of the participants wanted to be able to use a line in the code view to open a detailed visualization. Moreover, they thought that there is no visual implication that opening a detailed visualization is possible by clicking on a line in the feedback view. To solve the problem of no visual implication, the image of the mouse cursor is changed, from the standard HTML cursor to a pointer cursor, whenever the mouse is moved over a line in the feedback view. There are multiple options where to add the functionality to open frames in the detail view. Opening a new frame whenever the user clicks on a line in the code view is not an option. Programmers click around source code frequently, to correct or add something. This method would lead to numerous of not required open frames. We came up with two solutions: append it as an additional option to the right click menu of the code view or implement an extra method such as openDetailView(varName, line). For now, we added the functionality just at the feedback view and eventually will add it to the code view later on.

3.4.1 Frame Design

We did not change anything at the frame concept except for the menu-bar. Additionally, we refined the style of the

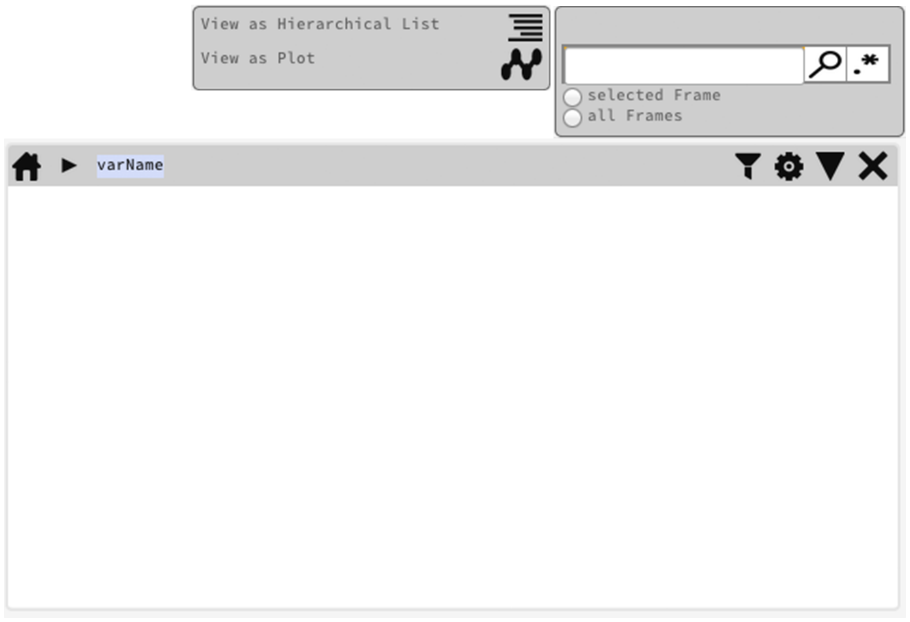


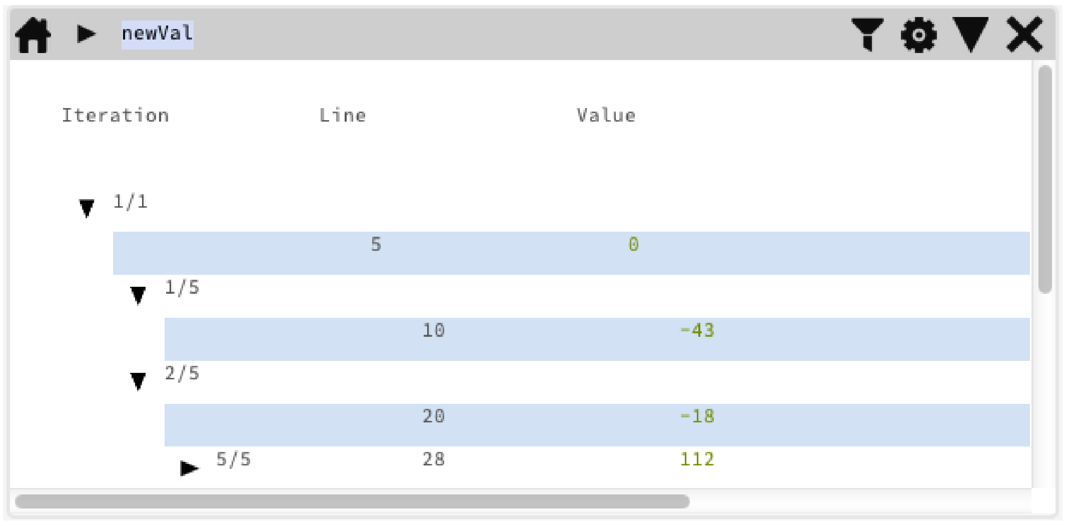
Figure 3.10: *The new frame and menu design. The filter menu on the right, the option menu on the left. The new option menu contains only the visualizations which are available for a specific data type.*

frame. First, we replaced the icons for navigation through a history site navigation box as visible in figure 3.10. This provides more information on the currently displayed content as well as past one. This design should be more self-explanatory than the two arrow icons. Additionally, we chose to provide an individual option menu for each data type group.

Frame concept stayed the same but we changed the style of a frame and eliminated the vagueness

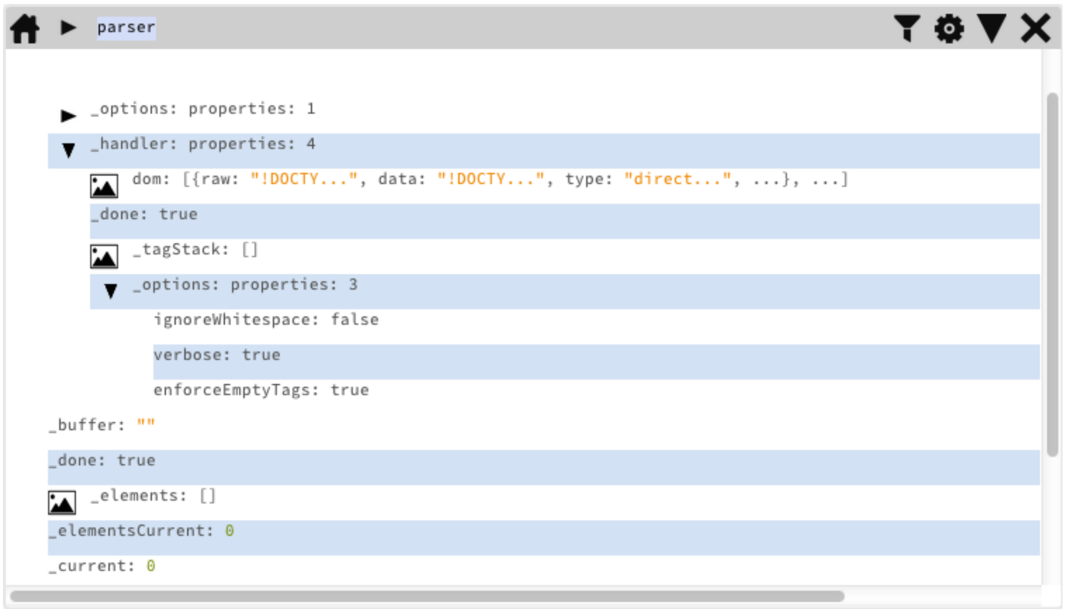
3.4.2 Detail Visualizations

As you can see in figure 3.11, we adapted the following parts of our hierarchical list visualization for numbers: we added possible interaction to all iterations of an opened loop, removed the number of the line which illustrates the begin of loops and added a line coloring to simplify the differentiation of the different lines in the list as well as their contained values.



```
newVal

Iteration          Line          Value

▼ 1/1
                      5              0
   ▼ 1/5
                      10             -43
   ▼ 2/5
                      20             -18
      ▶ 5/5           28             112
```

Figure 3.11: *Hierarchical list after opening it by clicking on a number value of interest within a loop. All loops containing the line of value change are opened automatically.*



```
parser

▶ _options: properties: 1
▼ _handler: properties: 4
    dom: [{raw: "!DOCTY...", data: "!DOCTY...", type: "direct...", ...}, ...]
    _done: true
    _tagStack: []
  ▼ _options: properties: 3
      ignoreWhitespace: false
      verbose: true
      enforceEmptyTags: true
_buffer: ""
_done: true
  _elements: []
_elementsCurrent: 0
_current: 0
```

Figure 3.12: *Hierarchical list of an object. Each property has a preview of its value or the whole value. Object properties are opened within this frame, changing the visualization. The other properties have a frame icon and can be clicked to open a detail view for them in a new frame*

In the object visualization (see figure 3.12) we removed the list of pointers till we find a better way to visualize that without using to much additional space. Finally, we added the same line coloring to this visualization as applied for the hierarchical list visualization for numbers.

Chapter 4

Implementation

In this chapter we present the architecture of Kurz' application, the architecture of our enhancement as well as its components and the integration into the old prototype. Since we enhance the prototype developed over multiple iterations by Heinen [2012], Belzmann [2013] and Kurz [2013], there are components we have to access in order to add our detail view and its visualizations. For example, we use the available structure which contains the feedback data from the server or the update function for the views. By explaining the architecture our prototype bases on, we get a general understanding of how the program works.
After giving an overview about our general concept, its components and their relation to each other, we give more details about the management of frames and our detail visualizations.

We present our newly added components and the existing foundation of the prototype

4.1 Existing Foundation for Our Prototype

The first version of our live coding plugin for Brackets was developed by Heinen [2012]. His prototype uses a loop in combination with the intern `eval function` of JavaScript, which evaluates and executes a given argument, to create live feedback. Next, Belzmann [2013] improves the capability and performance of the system by discarding the idea of using the intern eval function of JavaScript. Instead, his plugin uses a client-server architecture. Finally, Kurz [2013] modified the client by adding components, such as the LiveDataView, LiveDataController and LiveDataPane, implementing the feedback view and adapting the backend to remove some limitations.

All these modifications led to the current architecture:

A client-server architecture is used

The basic concept is the division of the program into a client and a server, called backend. The client continually sends a whole JavaScript file to the server and takes care of the visualization process.
Therefore it consists of the major components: Main, LiveDataView and LiveDataController, as well as multiple sub components, such as LiveDataPane and LiveCodingEvaluator. Each component is exactly one JavaScript file. Most of the components are plugged together by Main. It has access to the major components, creates the LiveDataController and tells it, e.g., to use the LiveDataView as the display.

Kurz implemented LiveDataView and LiveDataController as a Model-View-Controller pattern

The LiveDataView and LiveDataController are following the concept of the Model-View-Controller pattern. The LiveDataController decides what content to display, when to update the content and talks to the LiveCodingEvaluator, which receives the messages from the server, in order to obtain all necessary data. Afterwards, the LiveDataController forwards the received data to the LiveDataView.

The LiveDataController manages the content to display and when to update the content while LiveDataView manages how and where to display the content

The LiveDataView defines for each piece of content its position and representation. It displays error messages, creates the line-to-line layout and decides for each line the displayed content as well as its appearance. Furthermore, it attaches event handler for possible interactions and updates the view whenever the controller tells it about an occurred change. To sum up, every feedback and additional information is visualized by the LiveDataView. The component LiveDataPane supports LiveDataView, by synchronizing the scrolling between the two views as well as adding new views (feedback view).

The server instruments and executes the code and finally sends the results (runtime state of the program) back to the client. Whenever new code is received by the controller component, it communicates with the client, it spawns a new child process in form of an ExecutionManager. An ExecutionManager instruments the code, creates a sandbox and spawns two new child processes on his own. One child process generates a source map which maps from the instrumented code to the original position in the source code and vice versa. The source map is necessary because uncaught exceptions can falsify registered source code locations. The other process executes the instrumented code.

The server uses child processes to execute newly received code

By creating a new child process, each time new code is received, the crash of the whole program can be prevented. Usually, in case the executed code has an infinite loop or does not stop for other reasons it would be necessary to shut down the whole program to fix it. However, since each child process can be shut down independently, without influencing the client, the actual program remains responsive.

One child process generates a source map and one executes the code

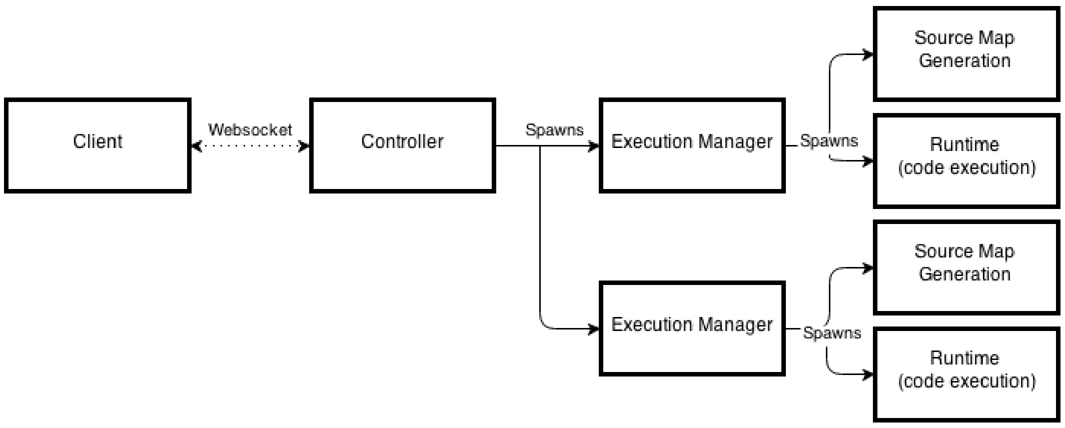


Figure 4.1: *The picture presents the server architecture of Kurz' prototype. The controller communicates with the client through a WebSocket protocol and receives its messages. In case of newly received source code it spawns a new process named ExecutionManager. The ExecutionManager also spawns new child processes, one for the generation of a source map and one for the code execution.*

The whole client-server communication, its message structure and setup, is an important part of the architecture. Communication between the client and the server is established by using a WebSocket protocol. Most messages are sent in form of JSON strings and contain code, but there are also control type messages to control the server.

Communication, in form of JSON strings, is established via a WebSocket protocol

```
{
    type: "code",
    code: code,
    [options: options object],
    id: id
}
```

The messages send by the server are mostly messages about source code changes. The messages contain information such as the number of iterations of a loop, a result of a term or the associated line and column in the source code. However, it can also send control messages such as error messages. A more detailed description of the client-server communication, the content and structure of the messages and diverse server tasks can be found in Belzmann [2013]. Plenty of functionalities, such as the instrumentation of source code or usage of multiple processes, are realized by third party modules. The most important ones are the following: *ws*, *escodegen*, *esprima*, *contextify* and *child_process*.

Use of Node.js and its modules e.g. for the client-server architecture and child processes

All of them are modules of the platform Node.js, which can be used for server-side as well as network applications and is built on Chrome's JavaScript runtime. *Ws* (WebSocket) enables the communication to the client. *Esprima* is used to parse source code into an AST (abstract syntax tree). *Escodegen* can generate code out of the instrumented AST as well as a source map. *Contextify* enables the execution of source code in a sandbox and *child_process* to spawn new child processes like it is done by the controller component of the server.

Most needed information for our detail view are already provided by the foundation

These components build the foundation for our prototype. They provide almost all important data we need for our detail view and its visualizations. The messages contain, e.g., the results of terms, the corresponding lines in the code view or the number of iterations we need for our visualizations. LiveDataView and LiveDataPane provide us with a foundation where we can attach our detail view and implement functionality such as the scrolling or updating of our visualizations.

4.2 Newly Added Components

Use of further third party modules such as escope.js for getting all scopes of an AST

To realize our detail view and visualizations we added further third party modules: *escope*, *estraverse* and *htmlTokenizer*.
Estraverse is used in connection with *esprima* and *escope*. Through *estraverse* it is possible to move and search trough an AST. The AST is created by the *esprima* parser. By using *escope* we acquire all scopes and can filter the scope of specific variables, which is necessary for our number visualization. In addition, we believe that it could be pretty useful for future visualizations and changes. Lastly, *htmlTokenizer* helps to detect errors in HTML elements. Since we add a third view to the surface and are interested in visualizing existing data, we mainly adapt, extend and make use of the components LiveDataView, LiveDataPane and LiveDataController.
Furthermore, we implement our own components to create an architecture where it is possible to easily add an own visualization and adapt the functionality of frames as well as visualizations. We add the following components to the existing foundation: FrameManager, FrameFunctionality, DetailVisManager as well as the various visualizations: NumberVisualization, ObjectVisualization and HTMLVisualization.

Adaption of the update and other functions in LiveDataView for our third view

We create our third view and implement the scroll functions and resize functions for the widths of the views within the LiveDataPane.
In the LiveDataView we implement all functions concerned with the creation of a detail frame: mostly creating HTML elements, adding classes to them and setting a CSS class). In short, almost everything that changes the appearance of the interface. In this component we had to adapt some methods, such as the `update function`, to enable the immediate update of our detail visualizations after each applied change.

LiveDataView requires functionality from FrameManager and DetailVisManager

First, the position of the associated line in the code view and feedback view can possibly have been changed or deleted with each update. Henceforth, we check if the associated line to each detail frame is still at the same position. In case the position changed or the line was deleted, we

search for our associated line in a certain radius. Second, the data of our line can possibly have changed. Thus, each open detail frame compares their current data with the new ones, whenever the function is called. In case of a difference, the visualization is adapted automatically.
Additionally, LiveDataView is the link to the newly added components: FrameManager and DetailVisManager. By using these components we separate large sections of the detail view functionality from the architecture of Kurz' prototype. Thus, new functionality can be added more easily and changing the behavior of the detail view and its elements becomes easier.

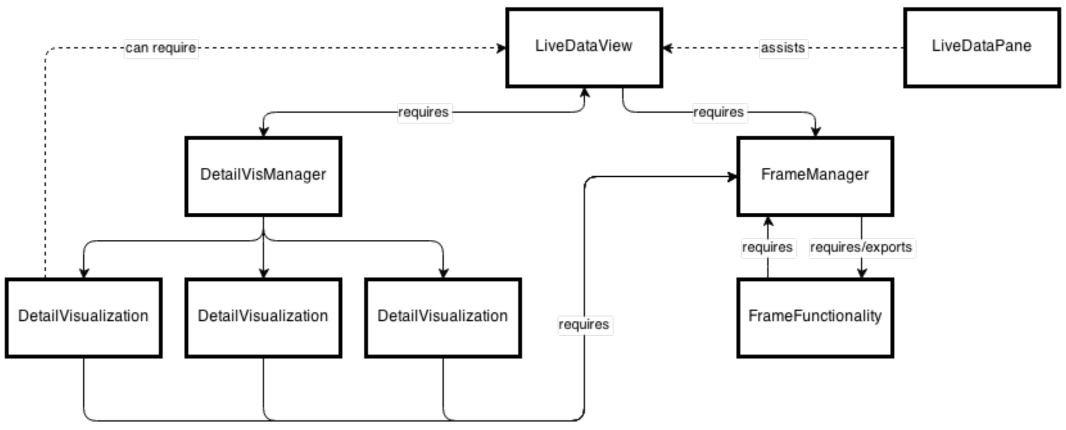


Figure 4.2: *The figure shows the LiveDataView as well as our newly added components and their relationship to each other. LiveDataView is our major interface to the already existing architecture. It requires DetailVisManager as well as the FrameManager and therefore has access to all their exported methods. On the right sight the FrameManager is shown. It requires and exports all functionality provided by FrameFunctionality. The FrameFunctionality requires the FrameManager since it needs the global frame attributes and functionality. On the left side is the DetailVisManger, which requires all DetailVisualizations and exports a method called* `createDetailVisualization`*. Each detail visualization has to require the FrameManager to keep it informed and to provide information such as the used data for the visualization. Each DetailVisualization has the possibility to require the LiveDataView to use its offered functionalities which can assist with the creation of a new visualization.*

We offer further functionality the programmer can use to create his own visualizations. First, as already mentioned we use *escope* to get all existing scopes of the currently opened JavaScript file. We then implemented a function called `getScopeOfVariable`. This method is called with a line number and a variable name as parameters. It determines the whole scope of the chosen variable. Second, we implemented a method called `getVarValuesLinesDepth`. This method provides information on the changes of a variable within its whole scope. For example it contains all applied value changes and their corresponding lines in the code view. However, this method does not work for variables which are created and incremented in the header of a loop declaration. In addition, there is other functionality such as the method `getLineDataForAllIterations`, which provides for all iterations the results at a certain line in the feedback view, respectively code view.

4.3 Frame Management

FrameManager contains global frame attributes and accesses and forwards all functionality from FrameFunctionality

The FrameManager contains all global attributes which are used to manage the frames, such as an unique identifier for each frame. It has an up-to-date list of existing frames, checks for overlaps and has access to all functionality provided by FrameFunctionality. Most of these functions are just forwarded to the LiveDataView. Furthermore, he has information about each frame such as the currently used data and whether the visualizations refers to multiple lines of code or just one line.
The component FrameFunctionality exists as a place for all additional frame functionality a programmer wants to add. We did this separation of the FrameManager and the FrameFunctionality to clearly separate the management and global attributes from the set of additional functionality. Additionally, it limits the amount of source code to check in case of an undesired behavior.

4.3.1 Frame Creation

Creation of frames is triggered in LiveDataView

The creation of a frame is initiated by clicking on a line in the feedback view. In LiveDataView all necessary HTML elements for a new frame are created: the menu bar and its menus, as well as the extra space for the detail visualization. All methods and global attributes necessary for the frame creation, to guarantee uniqueness or to register to an event handler, are provided by the FrameManager and indirectly by the FrameFunctionality. During the creation of a frame, the LiveDataView checks the data type of the concerned variable. With that knowledge it orders the DetailVisManager to create the detail visualization for the frame. Here we also distinguish between the data type groups we defined in section 3.2.4.

4.4 Visualization Management

All visualizations are registered at the DetailVisManager

The DetailVisManager manages all defined data type groups and their visualizations. It knows of all detail visualization files and can use them to create a new detail visualization. The DetailVisManager also offers this functionality to the LiveDataView. We created the DetailVisManger with the target to simplify the adding, deletion and change of visualizations, without the need to apply considerably changes in other components. It also offers the functionality to change the currently used standard visualization of each group.

4.4.1 Data Type Dependent Visualization

Each visualization is represented by an own component which can consists of multiple JavaScript files. Yet, one file, which can be required by the DetailVisManager and is able to control the whole visualization process, is necessary. In case a programmer wants to add his own detail visualization, he can implement it and add it to its belonging group. The same visualization can be possibly added to multiple groups. The DetailVisManager has an array variable for each visualization group which contains all the visualizations belonging to this group. If a detail visualization is requested by the LiveDataView, it searches for the concerned group, get its current standard visualization and starts the creation of that visualization.

Chapter 5

Discussion of our Prototype

5.1 Capabilities and Advantages

Our enhancements of the prototype by Joachim Kurz introduces new capabilities and advantages. First, in case the provided feedback of the second column is not sufficient, the programmer has now the possibility to open a detail visualization which is located in the third column. He can perceive feedback in other forms than textual representations, which can illustrate additional and contextual information.
By displaying each visualization in its individual frame, we enable the programmer to adapt each visualization without influencing the others. At the same time it ensures a clear separation between the open visualizations. In addition, these frames can be dragged to other positions to enable a better comparison between visualizations and the programmer is able to adapt the size of each frame to his requirements.
Another advantage of our prototype is that the user can choose in each situation the visualization which fits the best. For the case that no representation fulfills the requirements, the foundation of our prototype offers a programmer to implement his own visualizations and add them to

the existing sets.
We provide functionality to simplify the implementation of new visualizations, such as a method called `getScopeOfVariable` which returns the whole scope of a variable.
Finally, we added the live programming feature to our detail visualizations. Whenever parts of source code are changed, our detail visualizations check for differences and adapt accordingly. In addition, whenever the programmer clicks through the iterations of a loop, the affected visualizations update their content. If the programmer is not satisfied with the standard update behavior after code changes or the iteration of a loop was changed, he can use his own implemented update functions for his visualizations.

5.2 Limitations and Shortcomings

Our enhanced prototype has a couple of limitations. Some of these limitations we took over from Kurz' [2013] prototype. First, the application can only capture console output but no input. Second, feedback is only provided for lines consisting of an assignment. Furthermore, the plugin does only work with Node.js compatible programs. For more details about these see Kurz' [2013] and Belzmann's [2013] theses.

Changing the type grouping is cumbersome

Our enhancements introduce some new limitations. First, the programmer has to use our grouping of data types. The grouping can not be enhanced by just adding a new group. Changes to parts of the DetailVisManager, LiveDataView as well as all detail visualizations, which use our current grouping, are inevitable.

Detail visualizations for multiple assignments in one line are not possible

Second, the prototype has problems with multiple assignments in one line. Due to the concept that each line has exactly one attached frame, detail visualizations are only possible for one variable of each line. Currently, when the user initiates the creation of a frame, we check if a frame already exists for that line.
To open multiple detail visualizations for one line in the feedback view, we have to save the position of each value

in that line and adapt the identifiers accordingly.

One shortcoming, already mentioned by the participants of our preliminary user study, is the absence of possibilities to open visualizations by interacting with lines in the code view. This has nothing to do with our architecture and can be added later on without necessary changes to it. We already proposed possible solutions for that in section 3.4.

User interaction to open a detail visualization is only possible in the feedback view

Another limitation is the necessity to reload the extension, every time an user changes the currently shown JavaScript file. In the course of events our attached resize and drag handlers become partly inoperative.

Further, our number visualization has some problems and in some cases it does not work appropriate.

Finally, visualizing a representation which contains a huge amount of data leads to performance issues. Visualizations such as our object and HTML ones typically pose no problems, cause they refer to the data of one assignment and not to the data of the whole scope. In contrast, our number visualization consists of all value changes within the whole scope of a variable. There is a performance degradation due to the necessity to collect all data as well as by the creation of the visualization itself. Moreover, each event handler attached to the visualizations decreases the performance. For example, the creation of a visualization for a variable within a nested loop of depth 2, where the outer loop has 100 and the inner one 10 iterations, already takes a few seconds and there is a small delay when interacting with it. However, the performance of our prototype should be sufficient for a user study. We describe a potential future user study in 6.2.

Illustration of a large amount of data and usage of many event handler leads to performance issues

Chapter 6

Summary and Future Work

In this chapter we shortly summarize our results and contributions. Afterwards, we present possible future work, possible improvements for the live coding editor and a potential future user study.

6.1 Summary and Contributions

Gathered ideas, designed our layout as well as detail visualizations and finally implemented the detail view and three visualizations

First, by considering multiple live coding prototypes and tools, we gathered design ideas and guidelines for our visualizations. Moreover, we figured out some requirements for our detail view by looking at Kurz' [2013] prototype. Afterwards, we grouped and split the data types of JavaScript. We then came up with designs of detail visualizations for all these groups and a layout for our extension. We decided to use a three column layout, where each visualization belongs to its own unique frame.
Furthermore, we came up with an use case for each group and conducted a small qualitative user study by using these scenarios. After we got feedback and improved our design, we enhanced the prototype by Kurz, implemented the detail view as well as three detail visualizations and extended the live coding functionality to our frames and their detail

visualizations. We implemented our extensions in a structure that enables a programmer to enhance the prototype with his own visualizations and provide functionalities to assist him. In addition, the user can switch between visualizations of the same group to find a suitable one for each situation. Also interactions, such as filtering of information, are possible. Finally, we described limitations we inherited from the previous prototype as well as limitations of our extensions.

Our main contributions are:

- Determination of the requirements for our prototype by looking at the old one, as well as collection of design ideas by considering other live coding tools and prototypes.
- Creation of visualizations for all data type groups and conduction of a preliminary qualitative user study for these designs.
- Implementation of the detail view, the live coding capability for detail visualizations, our final detail visualizations and the foundation for further visualizations.

6.2 Future Work

Diverse possibilities for enhancement

There are still possible improvements and tasks for our prototype which have not been realized yet. The capabilities and performance of the prototype can be improved further, issues need to be fixed and user studies are yet to conduct. First, numerous designs for visualizations are yet to implement. Second, there are still parts of the application which can be enhanced.

Rest of designed visualizations and functionalities need to be implemented

The participants of the user study wished to be able to initialize the creation of a detail visualization by interaction within the code view. Therefore, we already presented some design ideas in section 3.4. These ideas have not been implemented yet. There are other functionalities we in fact designed but still not integrated in the new version of the prototype: the navigation, dimension reduction of arrays, recognition of changes in the data type of a variable within its scope, lines from the feedback view to a frame to emphasize the link (see section 3.2.3) as well as the planned scrolling behavior between the detail view and the other views.

Adding of the real documentation to the object visualization

Furthermore, while observing the Light Table [1] we came up with another advancement of the object visualization. We want to add the available online documentation, containing function descriptions and variables, to the object visualization.

Outsourcing of resource-demanding tasks to the server

Another issue is the performance of the program. There are diverse ways to improve the performance of our added components. First, functionality, such as the determination of the scope, could be handled by the server. Thus, the application would stay responsive while child processes on the server are occupied with these tasks. Another task, which is resource-demanding and could be done by the server, is the filtering of one specific scope out of the whole set of scopes.

Reduction of the number of jQuery operations

A further possibility to increase the performance is to refactor the whole source code by minimizing the number of necessary traverses through the Document Object Model (DOM) as well as jQuery operations in general. These are resource-demanding.

Finally, it is essential to test our enhanced prototype after more refinements and implementing further visualizations in a more defined user study than our preliminary one (see 3.3 "Preliminary User Study"). For example, it is not verified yet if there are situations where our visualizations improve the debugging time and accuracy. We already came up with an idea for a potential user study. A combination of a screen and a video recoding can be used to collect as much data as possible.

Moreover, more data can be achieved by letting each participant fill out a questionnaire for each visualization. Questions of interest are for example, how useful the participants found the visualizations, the amount of illustrated information or what kind of functionality is still missing

[1] http://www.lighttable.com/

or not necessary. Parallel to the screen recording, we could measure the task completion time and how frequently the different UI elements are used. UI elements of interest could be the offered functionalities of our frame and our detail visualizations. An additional idea is to measure the number of interactions with each detail visualization. That could show whether the visualizations need too many user interactions and whether we have to reduce or increase the amount of displayed information. As in the preliminary qualitative user study with paper prototypes, a different setup for each group of visualizations is advisable. For our implemented visualizations we already have some ideas.

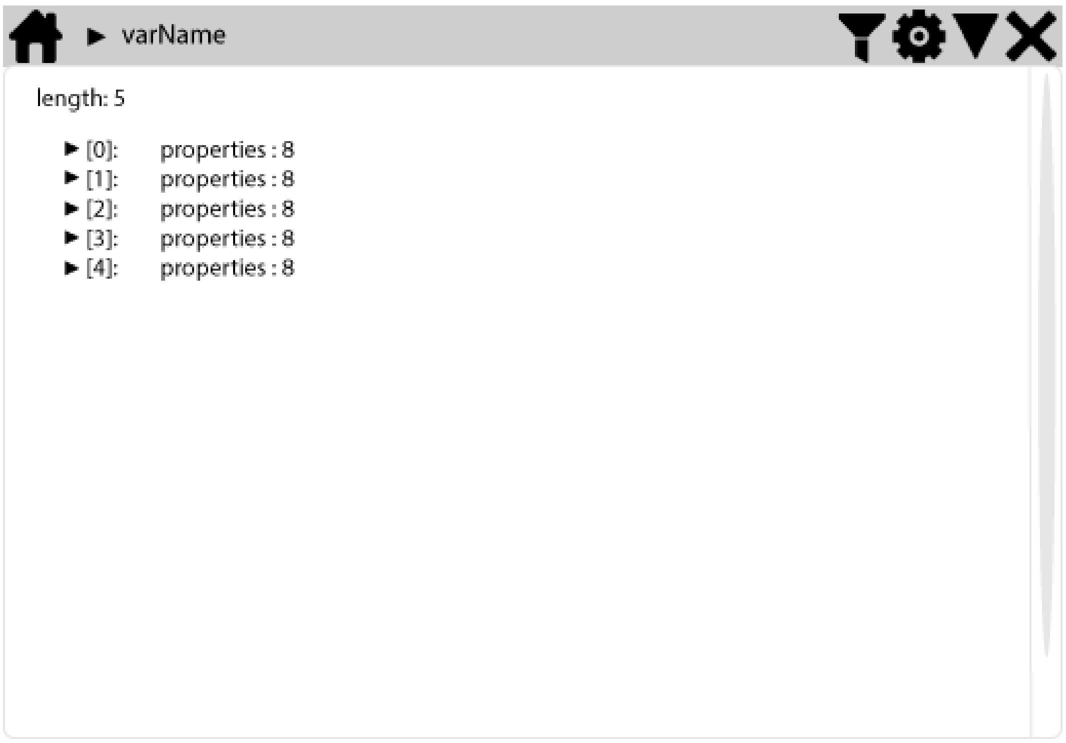
Use for each visualization an individual scenario

In the scenario for the HTML visualization the participants could create a prescribed website or adapt an existing one. Here we can test how much faster participants can place new elements and change their representation with the help of our live coding tool in comparison to those without. For testing our object visualizations, the participants could develop a small program by using an unfamiliar API. In this way we can test whether our object visualization assists with the task of understanding a new API and with the usage of all its offered functionalities. In addition, it is a common task to use new unfamiliar libraries and it enables us to check whether the participants need to pay less attention to the actual documentation.

Finally, an idea for an implementation task for number visualizations could be, e.g., a Base64 encoding. When implementing this, number variables needs to be changed at diverse locations of the program and within a loop over many iterations. Another suggestion is to use a task where the user has to simulate more graphical events such as the movement and curves of a roller coaster.

Appendix A

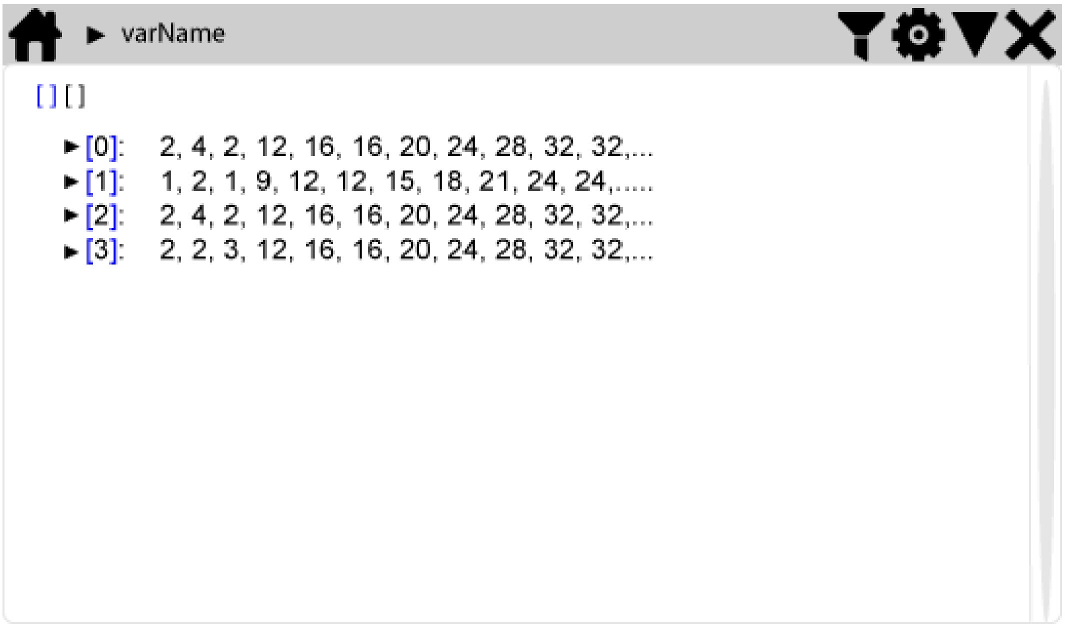
Not Implemented Detail Visualizations before User Study With New Frame



length: 5

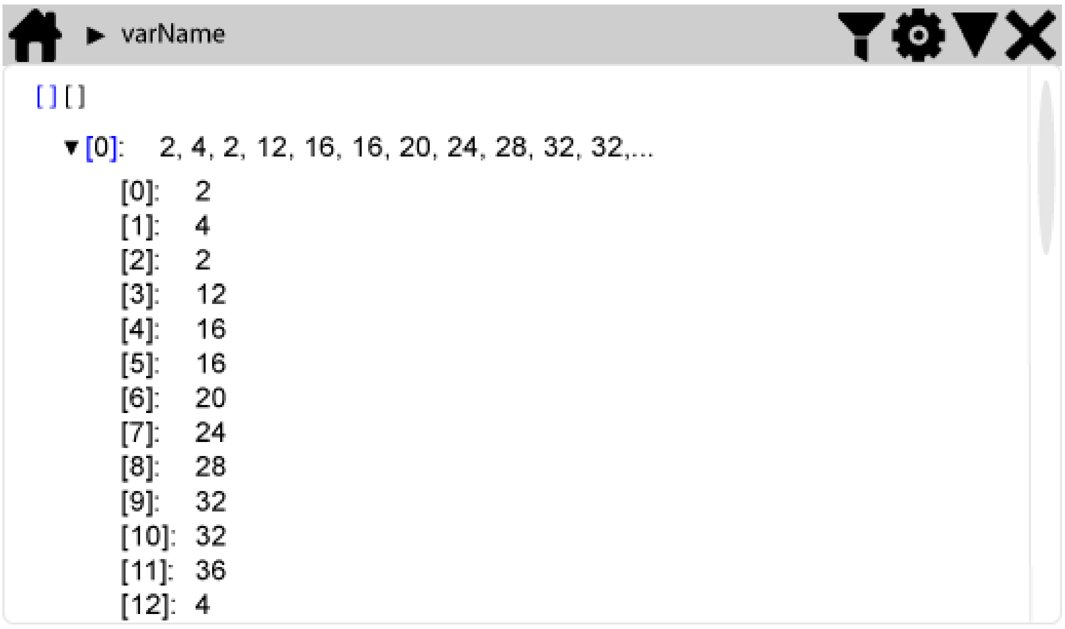
```
► [0]:    properties : 8
► [1]:    properties : 8
► [2]:    properties : 8
► [3]:    properties : 8
► [4]:    properties : 8
```

Figure A.1: A list containing all elements of the array. We illustrate the number of the elements at the top. Each element can be opened on its own and a preview is displayed.



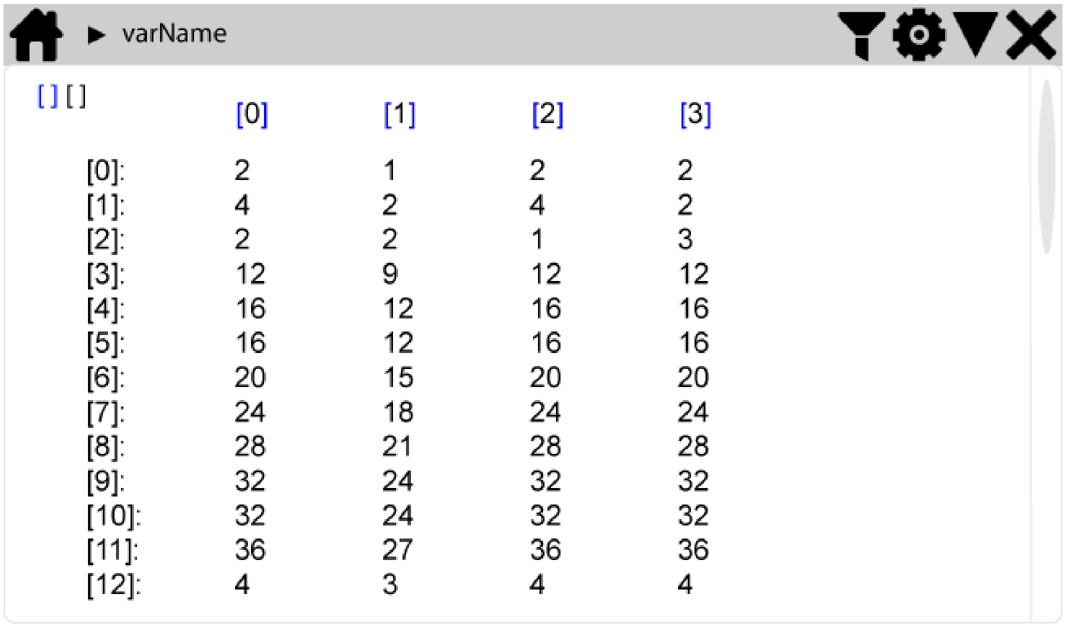
```
[][]

▸[0]:   2, 4, 2, 12, 16, 16, 20, 24, 28, 32, 32,...
▸[1]:   1, 2, 1, 9, 12, 12, 15, 18, 21, 24, 24,.....
▸[2]:   2, 4, 2, 12, 16, 16, 20, 24, 28, 32, 32,...
▸[3]:   2, 2, 3, 12, 16, 16, 20, 24, 28, 32, 32,...
```

Figure A.2: A two dimensional array in form of a hierarchical list. The colored brackets show how to access the visible elements and the dimension of the array. Again, for each element a preview is displayed.



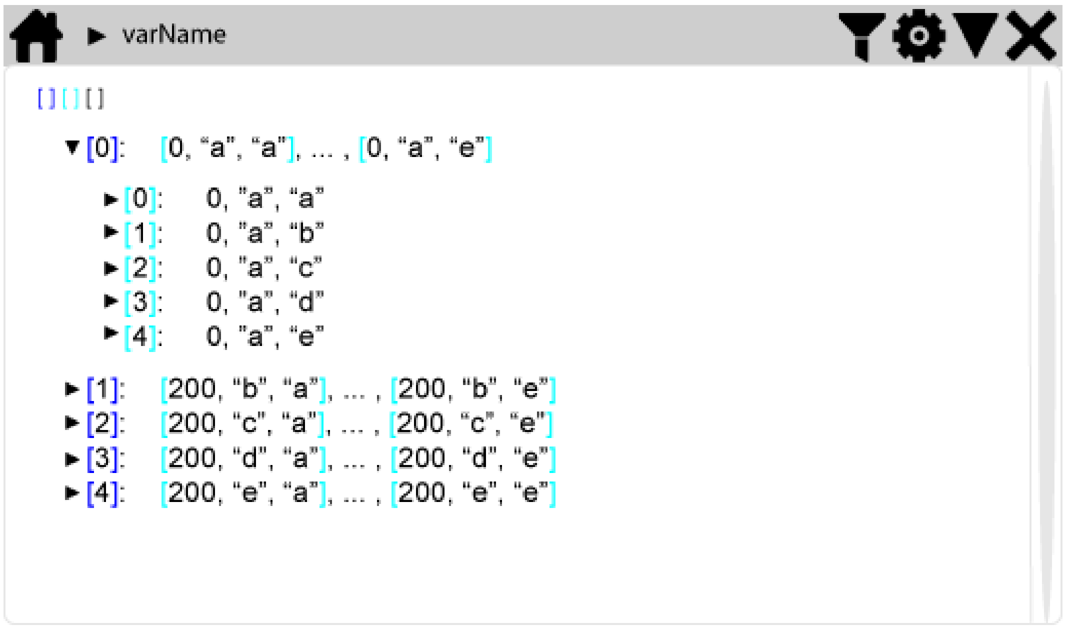
```
[][]

▾[0]:   2, 4, 2, 12, 16, 16, 20, 24, 28, 32, 32,...
    [0]:   2
    [1]:   4
    [2]:   2
    [3]:   12
    [4]:   16
    [5]:   16
    [6]:   20
    [7]:   24
    [8]:   28
    [9]:   32
    [10]:  32
    [11]:  36
    [12]:  4
```

Figure A.3: An open two dimensional array in form of a hierarchical list.



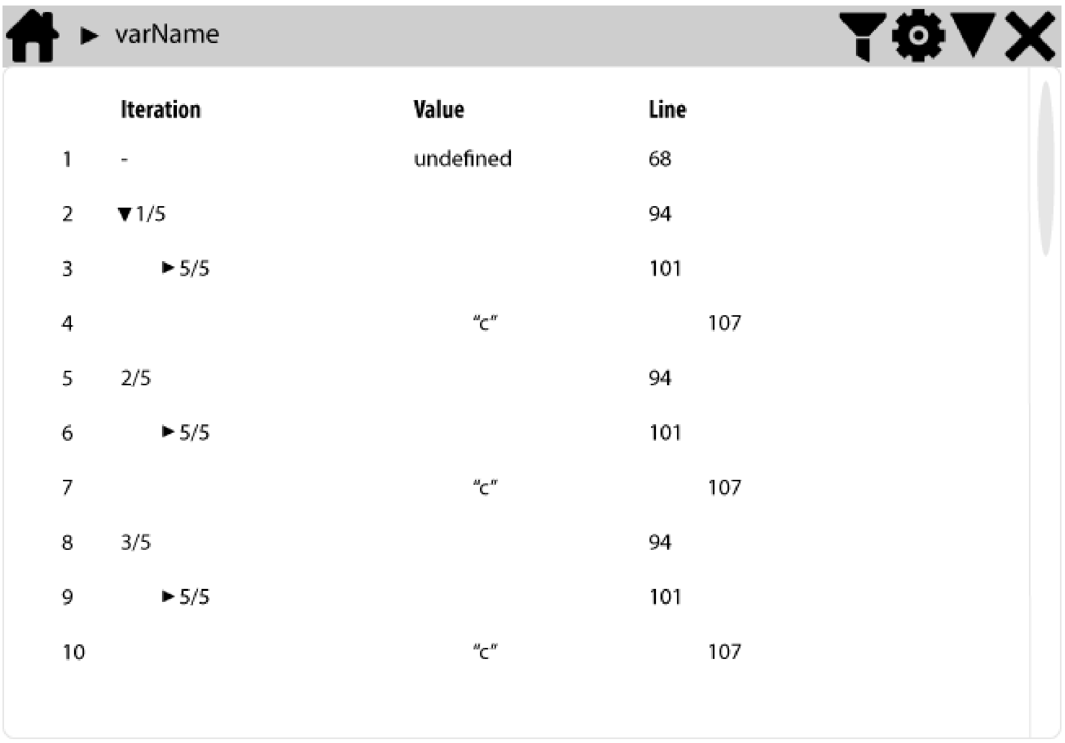
[] []	[0]	[1]	[2]	[3]
[0]:	2	1	2	2
[1]:	4	2	4	2
[2]:	2	2	1	3
[3]:	12	9	12	12
[4]:	16	12	16	16
[5]:	16	12	16	16
[6]:	20	15	20	20
[7]:	24	18	24	24
[8]:	28	21	28	28
[9]:	32	24	32	32
[10]:	32	24	32	32
[11]:	36	27	36	36
[12]:	4	3	4	4

Figure A.4: A two dimensional array in form of a matrix. For the use of the brackets look at figure A.2



```
[] [] []
▼[0]:   [0, “a”, “a”], ... , [0, “a”, “e”]
    ▶[0]:   0, “a”, “a”
    ▶[1]:   0, “a”, “b”
    ▶[2]:   0, “a”, “c”
    ▶[3]:   0, “a”, “d”
    ▶[4]:   0, “a”, “e”
▶[1]:   [200, “b”, “a”], ... , [200, “b”, “e”]
▶[2]:   [200, “c”, “a”], ... , [200, “c”, “e”]
▶[3]:   [200, “d”, “a”], ... , [200, “d”, “e”]
▶[4]:   [200, “e”, “a”], ... , [200, “e”, “e”]
```

Figure A.5: A multidimensional array. The form of representation is the same as in figure A.2



	Iteration	Value	Line
1	-	undefined	68
2	▼1/5		94
3	►5/5		101
4		"c"	107
5	2/5		94
6	►5/5		101
7		"c"	107
8	3/5		94
9	►5/5		101
10		"c"	107

Figure A.6: A table which consists of the three columns: Iteration, Value and Line. Has the exact layout as the number visualization before we improved and refined it.

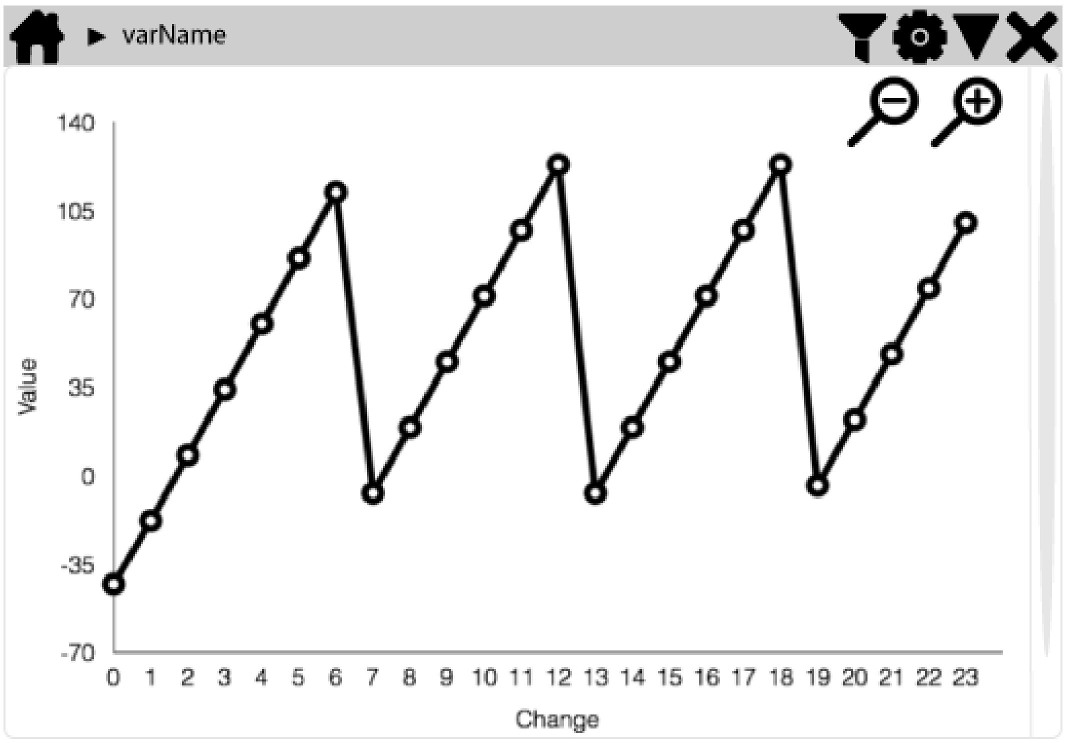


Figure A.7: Our already presented plot visualization with the new Frame.

Bibliography

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 Br, Vignan Pattamatta, William Choi, Ben Hsieh, and Scott R. Klemmer. Rehearse: Helping programmers adapt examples by visualizing execution and highlighting related code, 2010.

William Choi, Joel Brandt, and Scott R. Klemmer. Rehearse: Coding interactively while prototyping, 2008.

Jonathan Edwards. Example centric programming. *SIGPLAN Not.*, 39(12):84–91, December 2004. ISSN 0362-1340. doi: 10.1145/1052883.1052894. URL `http://doi.acm.org/10.1145/1052883.1052894`.

John D. Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7(2):151 – 182, 1975. ISSN 0020-7373. doi: http://dx.doi.org/10.1016/S0020-7373(75)80005-8. URL `http://www.sciencedirect.com/science/article/pii/S0020737375800058`.

Chris Granger. Light table. URL `https://www.kickstarter.com/projects/ibdknox/light-table`.

Philip J. Guo. Online Python Tutor: Embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1868-6.

doi: 10.1145/2445196.2445368. URL `http://doi.acm.org/10.1145/2445196.2445368`.

Björn 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`.

Alfred Inselberg and Bernard Dimsdale. Parallel coordinates: A tool for visualizing multi-dimensional geometry. In *Proceedings of the 1st Conference on Visualization '90*, VIS '90, pages 361–378, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. ISBN 0-8186-2083-8. URL `http://dl.acm.org/citation.cfm?id=949531.949588`.

Andrew J. Ko and Brad A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM. ISBN 1-58113-702-8. doi: 10.1145/985692.985712. URL `http://doi.acm.org/10.1145/985692.985712`.

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

Sean McDirmid. Living it up with a live programming language. *SIGPLAN Not.*, 42(10):623–638, October 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297073. URL `http://doi.acm.org/10.1145/1297105.1297073`.

Sean McDirmid. Usable live programming. In *SPLASH Onward!* ACM SIGPLAN, October 2013. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=189802`. To appear.

Tamara Munzner. A nested model for visualization design and validation. *IEEE Transactions on Visualization* *and Computer Graphics*, 15(6):921–928, November 2009. ISSN 1077-2626. doi: 10.1109/TVCG.2009.111. URL `http://dx.doi.org/10.1109/TVCG.2009.111`.

Donald A. Norman. *The Design of Everyday Things*. Basic Books, New York, reprint paperback edition, 2002. ISBN 0-465-06710-7.

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

David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 76–85, Boston, MA, USA, July 12–14, 2004b.

James L. Snell. Ahead-of-time debugging, or programming not in the dark. In *Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97) (Including CASE '97)*, STEP '97, pages 288–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7840-2. URL `http://dl.acm.org/citation.cfm?id=829539.831969`.

Steven L. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages and Computing*, 1(2):127 – 139, 1990. ISSN 1045-926X. doi: http://dx.doi.org/10.1016/S1045-926X(05)80012-6. URL `http://www.sciencedirect.com/science/article/pii/S1045926X05800126`.

Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, second edition, 2001. ISBN 0961392142.

Bret Victor. Ladder of abstraction, 2011. URL `http://worrydream.com/#!2/LadderOfAbstraction/`.

Bret Victor. Learnable programming, 2012a. URL `http://worrydream.com/#!/LearnableProgramming/`.

Bret Victor. Inventing on principle, 2012b. URL `http://vimeo.com/36579366/`.

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`.

Index

Typeset September 30, 2014