Combining Live Coding and Continuous Testing

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

by
Tanja Ulmen

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof.Dr.Bernhard Rumpe

Registration date: 22.05.2014
Submission date: 19.09.2014

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
Tanja Ulmen

Contents

List of Figures

List of Tables

Abstract

Today software can become complex and due to that it might happen that errors remain undetected during development. To find and to avoid these errors there are several methods. Two of them are live coding and continuous testing. Both help the users to find errors quickly and to reduce the time they would need for debugging. Live coding enables the users to see live what their written code is doing, because the code is continuously executed. This reduces the time the users would need for executing the code manually. Continuous testing is a method where test cases are executed continuously in the background while the programmers are writing their code. If an error occurs the programmers get a small notification and can decide to solve this error quickly or to ignore it.

In this thesis we develop a combination of live coding and continuous testing, we add a testing method to the already existing live coding Brackets extension that was developed by Joachim Kurz. This combination ensures a nearly automatic test case creation. The users have two buttons at their disposal, one to mark a current function call as correct, and one to mark it as false. With a click on this button a test case object is generated and sent to the server side of our program. This server creates and executes the test cases and sends the results back to the client where the results are displayed. Thus the users have an easy overview over their code, which function works as expected and which function still contains an error. With this combination of live coding and continuous testing the time that the users would spend for writing test cases is reduced, because the program writes them on its own.

Überblick

Heutzutage kann Software sehr komplex werden und deswegen kommt es vor, dass Fehler in dieser Software lange unentdeckt bleiben. Um diese Fehler zu finden oder zu vermeiden gibt es mehrere Methoden. Zwei davon sind Live Coding und Continuous Testing, beide helfen den Nutzern Fehler im Code schnell zu finden und sie reduzieren die Zeit, die zum Debuggen benötigt würde. Da beim Live Coding der Code fortlaufend im Hintergrund ausgeführt wird, können die Nutzer live sehen was ihr geschriebener Code bewirkt. Das verringert die Zeit die der Nutzer bräuchte um den Code jedes Mal manuell auszuführen. Continuous Testing ist eine Methode bei der vorher geschriebene Testfälle fortlaufend im Hintergrund ausgeführt werden. So bekommt der Nutzer während des Schreibens Feedback ob die Zeilen die er geschrieben hat Fehler enthalten oder nicht. Wenn ein Fehler auftritt, erscheint ein Hinweis dem der Nutzer nachgehen, oder ignorieren kann.

In dieser Arbeit entwickeln wir eine Kombination aus Live Coding und Continuous Testing. Wir fügen der bereits existierenden Live Coding Erweiterung für Brackets, die Joachim Kurz entwickelt hat, ein Testing Tool hinzu. Diese Kombination ermöglicht es, dass Testfälle automatisch erstellt werden. Die Nutzer haben zwei Buttons zur Verfügung, mit dem ersten können sie den aktuellen Durchlauf einer Funktion als richtig markieren, mit dem anderen als falsch. Mit einem Klick auf einen dieser Buttons wird ein Testfall Objekt generiert, an den Server geschickt, dort wird ein Testfall genereiert, ausgeführt und die Resultate werden wieder zum Client geschickt, der diese dann visuell darstellt. So erhalten die Nutzer einen einfachen Überblick über ihren Code. Sie sehen schnell welche Funktion problemlos funktioniert und wo eventuell noch Fehler auftreten. Da die Testfälle nun automatisch generiert werden reduziert diese Kombination aus Live Coding und Continuous Testing die Zeit, die die Nutzer bräuchten um eigene Testfälle zu schreiben.

Acknowledgements

I want to thank everybody who supported me during my bachelor thesis. Especially Jan-Peter Krämer, my advisor during the first few month of my thesis and Thorsten Karrer my advisor during the last months of my thesis. Then I want to thank my brother Fabian Ulmen for proofreading although he was very busy with his own studies. At least I want to thank Kirsten Kern, who is also writing her bachelor thesis at the moment. With her I always had someone to talk when I was at a loss :).

Conventions

Throughout this thesis we use the following conventions.

Text conventions

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

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

Definition: *Excursus*

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

```
myClass
```

The whole thesis is written in Canadian English.

Chapter 1

Introduction

1.1 Motivation

Software can become complex and that is why it might happen that errors remain undetected during development. Maybe more program code is added before the developers notice that there is an error in their code. Now it is difficult to find the exact code part that causes this error and sometimes it needs a lot of time to find this part. One method to find and avoid these undetected errors is a test suite. A test suite is a collection of different test cases . A test case checks whether a code part, or an entire program works as desired. These test cases are for example helpful if a program has to deal with several edge cases. For example, assume that we want to test a function that is called *isOdd*. This function gets an integer as parameter and it returns a boolean. This boolean is *true* if the given number is odd or *false* if the given number is even. Now it is possible to design a test case that tests exactly this function of correctness. It can check, for example, if the function *isOdd* returns *true* if we use 5 as input parameter. If we define several of this specific test cases, they build a whole test suite for this one function.

Because of the complexity of software, it is helpful to work with test cases.

After - or sometimes before - a code part is written, the developers write different test cases to check the functionality of their code. After finishing the code part, they execute the test cases and receive the results, for example output values or warnings. Now the developers have an overview how their code works and if it works as expected or not. This method of writing a test suite and executing it manually is inefficient in two ways: the developers could use the time they lose while waiting for the test suite to finish its execution and the CPU could also do other tasks while the developers are writing the code [Saff and Ernst, 2003].

Developers lose a lot of time while waiting for test suites to finish.

Saff and Ernst [2003] found out that for a Java dataset with 9114 lines of code, the waiting time for one test suite was 3 seconds and a developer ran this test suite in average every 11 minutes. So with a developing time of 22 hours the developers waited 22 minutes for the test to finish. For a Perl dataset, the developer ran the test suite in average every 5 minutes, with a duration of 16 seconds per run and a total working time of also 22 hours, the developers waited 71 minutes. This results are shown in table 1.1.

Attribute	Perl	Java
lines of code	5714	9114
total time worked (hours)	22	22
total calendar time (weeks)	9	3
total test runs	266	116
total capture points	6101	1634
total number of errors	33	12
average time between tests (minutes)	5	11
average test run time (secs)	16	3
mean ignorance time (secs)	218	1014
min ignorance time (secs)	16	20
median ignorance time (secs)	49	157
max ignorance time (secs)	1941	5922
mean fix time (secs)	549	1552
min fix time (secs)	12	2
median fix time (secs)	198	267
max fix time (secs)	3986	7086

Table 1.1: "Statistics about the Perl and Java datasets" [Saff and Ernst, 2003]

To avoid this waiting time, it is possible to use, for example, live coding or continuous testing. These two methods are explained in the next two sections.

1.2 Live Coding

Live coding, or Live Programming , is a technique to minimize "the latency between a programming action and seeing its effect on program execution" [Tanimoto, 2013]. To archive this minimization of latency, the written code is executed continuously in the background to receive the results of the program execution, for example output variables or warnings, in real-time. A benefit of this method is, that it also supports the learning process, because the developers immediately see the effects of their written code [Tanimoto, 2013]. Another benefit of live coding is, that the developers are not anymore disturbed in their developing process by debugging code lines manually. Without live coding the developers need to check their code lines frequently by hand. They have to stop their developing process, to build the entire program, to set debugging marks and to look if the previous written code lines are working as desired. This can cost a lot of time. With live coding they are able to develop ideas without breaks to execute the whole program. Finally there is less wasted time that would be needed to execute the program manually.

Live coding reduces the waiting time between writing code and seeing the effect of the code.

A negative point is that live coding needs a lot of "computational resources of a system" [Tanimoto, 2013] and it is not useful for every work. In large programs, for example, it might happen that the developers want to change a part of the code that was already executed, but the program will never return to this point. One idea would be to execute the program in a loop. Another idea is to use breakpoints that can be set to run a special part of the code or a function in a loop [Tanimoto, 2013]. These are two methods to avoid the necessity to re-execute the entire program.

Live coding needs a lot of computational resources.

1.3 Continuous Testing

Continuous testing is a method to combine coding with testing. Test cases are developed before writing the code. While the developers are coding, this tests are running continuously in the background, "using otherwise-idle CPU cycles" [Muşlu et al., 2013]. With this method, the developers receive feedback for their written code in real-time. A disadvantage of running tests continuously in the background is, that the test cases are executed independent of what the developer does. A more efficient variant is to execute the test automatically after finishing a code part [Muşlu et al., 2013].If an error occurs, there will be a notification and the developer can get more information about the error. He also can decide to ignore it, for example if he already knows that there is an error in his code. Continuous testing is good to avoid for example regression errors .

Using the method of continuous testing, the predefined test cases are continuously running in the background, giving hints if errors occur.

Definition: *Regression error*

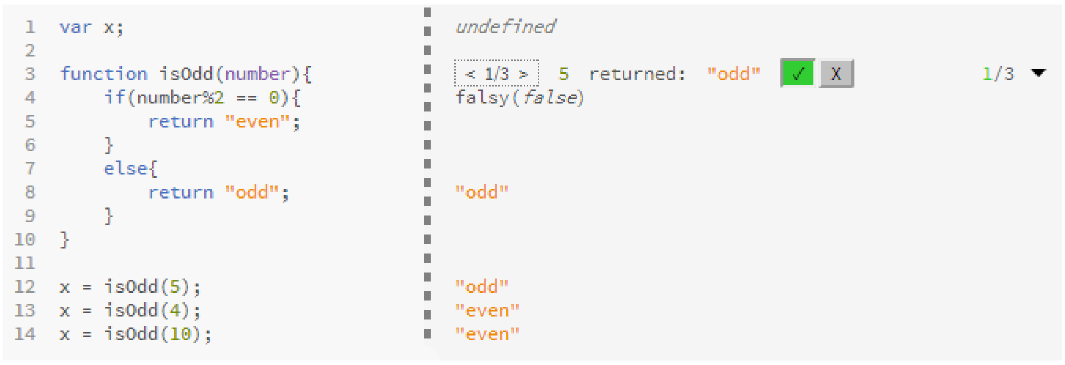
REGRESSION ERROR:
Regression errors are errors that occur if new functionality is added to an old program and due to this new changes an error occurs in the old, former working, functionality.

Another benefit is that with continuous testing the waiting time of the developer to receive test results is less than without this method. This ensures that the developers can concentrate on the implementation of their ideas, otherwise it could be possible that they forget their ideas during debugging the previous code fragments [Saff and Ernst, 2004].

1.4 Combining Live Coding and Continuous Testing

In this thesis live coding and continuous testing are combined to create test cases almost automatically.

The idea of this thesis is to combine live coding and continuous testing. During life coding, the developers already get feedback about what happens with their written code, but it would be more efficient to use also advantages of testing. With a test case it is possible to check various aspects of the program. The combination of the two methods makes it possible to define test cases while writing the code and the results of this tests are received immediately. This method creates test cases on its own, based on the information it already has from live coding and the developers only need



```
var x;

function isOdd(number){
    if(number%2 == 0){
        return "even";
    }
    else{
        return "odd";
    }
}

x = isOdd(5);
x = isOdd(4);
x = isOdd(10);
```

```
undefined

< 1/3 >  5  returned: "odd"  ✓  X          1/3
falsy(false)



"odd"




"odd"
"even"
"even"
```

Figure 1.1: *Screenshot of the button that is needed to define a new test case.*

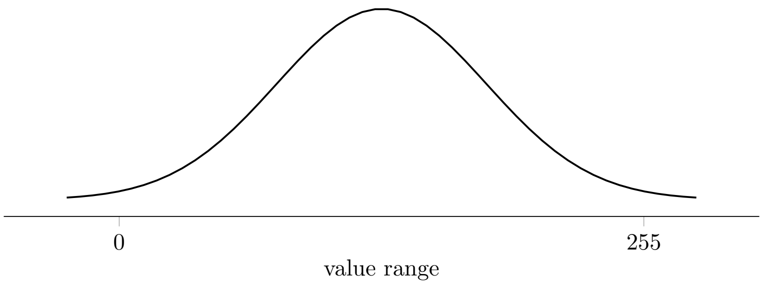


Figure 1.2: *Gained test coverage with the combination of live coding and continuous testing in a Function that works with Integers from 0 to 255.*

to click one of two buttons to add a new test case to the test suite. Figure 1.1 shows these buttons. In this example the function has a numerical parameter of 5 and it returned "odd". If this result is right, it can be marked as correct and a new test case is generated.

With this combinational method it is easy to get a high test coverage of average requests. For example if a function works with Integers from 0 to 255, the average requests are the requests that are in the middle of this range around 127. Usually developers would use this requests in their programs instead of edge cases. Due to that we get a high test coverage for average requests and a low test coverage for edge cases. In Figure 1.2 this distribution is visualized. The expected maximum coverage is around 127 and the minimum coverage around 0 and 255.

This combination of the two methods ensures a high test coverage of average requests.

To enable the idea of combining live coding and continuous testing, we use the live coding extension for Brackets, developed by Joachim Kurz[1] and the Jasmine[2] testing tool for JavaScript. The developers work with the live coding extension that will be expanded by, among other things, the possibility to mark a specific function call as correct. For example, there is a function that decides if a given number is even or odd. With the input 5 it returns the value ”odd”, this is correct thus, this can be marked as correct (see Figure 1.1). At the server side of this extension this new test case will be created and executed with Jasmine. The results of this tests are displayed immediately and the developers get a better overview over the correctness of their code. If there are mistakes in their code they get enough information to find this error quickly.

We work with the live coding extension for Brackets and the Jasmine Testing Framework.

1.5 Chapter Overview

Previous knowledge about live coding and continuous testing.

The next chapter 2 “Related work” contains the previous knowledge concerning live coding and continuous testing.

Details of the user interface and the test case design.

The design is described in chapter 3 “Design”. This chapter is split into two parts: The first part contains the design ideas for the user interface and the differentiation to other design possibilities. The second part is about the design of the test cases.

The important parts of the implementation.

After the design chapter follows the chapter 4 “Implementation”. It defines the important parts of the implementation, concerning the server and the client side.

Advantages and disadvantages of the combination of the two methods.

Chapter 5 “Evaluation” discusses the advantages and disadvantages of the combination of live coding and continuous testing. It gives ideas how this method could be evaluated in a study and points out possibilities to extend the program.

A summery and ideas for future work.

The last chapter 6 “Summary and future work” gives a

[1]http://hci.rwth-aachen.de/livecoding
[2]http://jasmine.github.io/2.0/introduction.html

summery of the previous chapters and an idea of future work concerning this topic.

Chapter 2

Related work

How programs present live feedback can be categorized into 6 levels.

In the Literature are a lot of programs that support live coding or continuous testing. One of the first papers that deals with live coding was written by Tanimoto [1990]. It deals with visual programming languages. He was the first one to categorize the degree of "liveness", meaning how programs "present 'live' feedback to the programmer" [Tanimoto, 1990]. Later on Tanimoto [2013] added two more categories to his original hierarchy. All six levels are mapped in Figure 2.1.

Level 1: no semantic feedback.

Level 2: semantic feedback on demand.

Level 3: automatic feedback.

Level 4: automatic feedback and event listening.

Level 5: suggestions about what to implement next.

Level 6: knowledge about the users intentions.

The first and lowest level describes that users do not get any semantic feedback. The second level says that users can get semantic feedback if they want to. All programming enviroments should be on level two of liveness, because they should all enable the users to get semantic feedback of their code. The third level provides automatic semantic feedback on program edits. The fourth step is described as "fully live" by Tanimoto [2013]. This level provides automatic, semantic feedback like in step 3, and a program on this step is able to response to events like mouse clicks for example [Burnett et al., 1998]. The next two steps are ideas which are not yet implemented. A program on level 5 of liveness will be able to make suggestions based of the behaviour and other programs of the developers about what the developers may want to implement next. The last step of liveness, level 6 provides a kind of intelligent program. This program knows what the developers desire or what

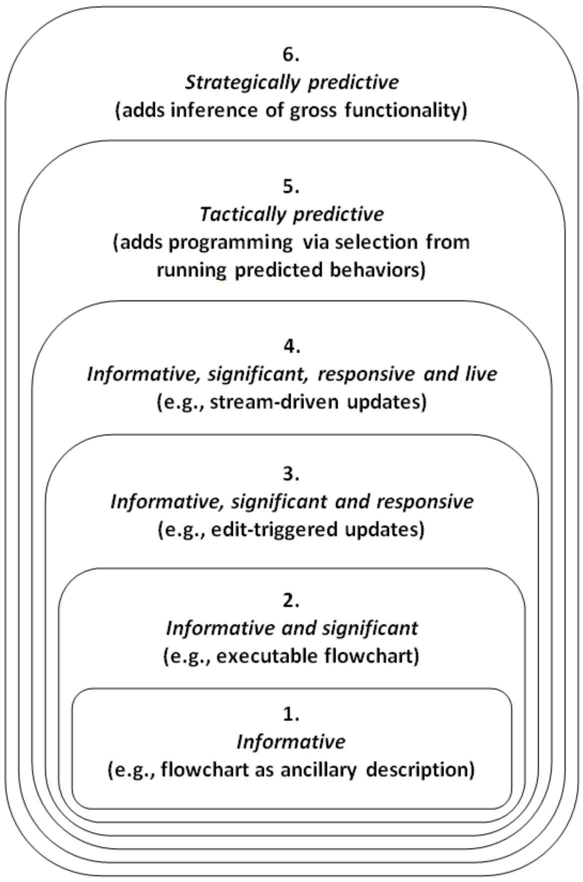

6.
Strategically predictive
(adds inference of gross functionality)
5.
Tactically predictive
(adds programming via selection from running predicted behaviors)
4.
Informative, significant, responsive and live
(e.g., stream-driven updates)
3.
Informative, significant and responsive
(e.g., edit-triggered updates)
2.
Informative and significant
(e.g., executable flowchart)
1.
Informative
(e.g., flowchart as ancillary description)

Figure 2.1: *"Extended version of the liveness hierarchy" [Tanimoto, 2013]*

intentions they have. To ensure this step of liveness, a large knowledge base is needed. The combination of live coding and continuous testing, which we implemented in this thesis, is working on level 4. It provides a "fully live" Tanimoto [2013] output for the user that always shows the current state of the program, and it is possible to interact with this output.

Another early paper about a kind of live coding is the one by Snell [1997]. He developed a combined method of editing and debugging, so that "as the programmer entered the code for a new routine, as soon as each statement was entered, the environment would execute it and display the new program state" [Snell, 1997]. Based on the paper of Tanimoto [1990] it is possible to say that he invented a program with a liveness on level 3. Another point is that his program does not simply display the actual program state, it also supports test cases. The developers need to define several test cases before writing their code, and while entering code the program updates the test cases until they return a result. In this way the developers can easily see which line causes an error because the update of the test cases takes place after every single statement. Snell called this method "Ahead-of-time Debugging" (AOT). With a small study he found out that developers using the AOT environment were faster in writing code and they produced less errors during the coding process than without this environment. Thus, this paper is one of the first ones that work with a kind of combination of coding and testing, but the main objective is the testing tool. The test results are "live" and not the results of the code. In our live coding and continuous testing combination we try to provide both methods in balance.

Ahead-time of Debugging provides live feedback of test results.

Muşlu et al. [2013] wrote a paper about continuous testing with the goal "to shorten the time to detection [of an error] as much as possible". They execute predefined tests in the background during the development of a new code fragment. Therefore, it does not matter if the tests are human written or generated automatically. In the Brackets extension that we developed with this thesis, we use the method of generating test cases nearly automatically and execute them with the method of continuous testing. Our test case development is only "nearly" automatic, because we still need an input from the developers to generate a test case. Muşlu et al. [2013] found out that it is more efficient to execute the tests only if a data update occurs, than to execute the test cases continuously ignoring the actual program state. The continuous execution would lead to an enormous overhead, thus, we decided to execute our tests only after adding a new test to the test suite or after reloading the whole program, like Muşlu et al. [2013] recommended.

Continuous testing is more efficient if test cases are executed after a data update than to execute them continuously ignoring the actual program state.

Another part that Muşlu et al. [2013] discussed in their paper is the question, if it would be efficient to execute all ex-

A test case prioritization would avoid an overhead.

isting test cases. Again it would lead to an overhead, if test cases are executed that are not needed in this state of the program. In our Brackets extension we do not respect this prioritization of test cases, but it would be a useful idea for future work. Our first intention is to ensure the possibility of generating test cases nearly automatically and to ensure a live overview of the test results. Thus, in this implementation of the combination of live coding and continuous testing all generated test cases will be executed without test case prioritization.

A meaningful user interface is important.

The last thing Muşlu et al. [2013] mentioned is the user interface of the test case results. Their "continuous data testing prototype only indicates which test has failed", like in our program. The problem with this kind of user interface is, that the developers only know which function causes an error, but they do not know where the error is caused exactly. We try to help the developers with a detailed error reporting that contains more information than only the message of failure (see 3 "Design") .

Live coding decreases the average fix time of bugs.

A new paper that deals with live coding is by Krämer et al. [2014]. They worked with the Brackets extension where ours is based on, and did a small user study to show that live coding provides a decreased average fix time of bugs. Together with the results of Muşlu et al. [2013] we can expect that a combination of live coding and continuous testing decreases again the average fix time of bugs, in contrast to both techniques on their own.

Rehearse highlights responsible code lines while the program execution.

Brandt et al. [2010] developed the program *Rehearse*. It is a programming environment that highlights "each line of code as it is executed" [Brandt et al., 2010] and if a code line is marked by the users, the program finds other code lines that correspond to the actual marked code line. The first feature, the highlighting of code that is executed, helps users "getting the code right" [Brandt et al., 2008] because this combination of feedback and execution helps the users to quickly identify lines that cause errors. Our live coding and continuous testing combination also shows where errors are located, not in which line but in which function. The continuous testing part furthermore provides a detailed error reporting that also supports the users while "getting the code right" [Brandt et al., 2008].

Too much feedback disturbs a developer.

Saff and Ernst [2003] also focused on continuous testing. They were the first to introduce the technique of continuous testing. They developed this technique to reduce the wasted time that developers need for debugging (see Chapter 1 "Introduction"). Within their user study they came to the point that "more feedback is not always better: an interface that provides too much information [] might interrupt, distract, and overload the developer, perhaps even to the point of retarding productivity" [Saff and Ernst, 2003]. In our program, that works also with continuous testing, we tried to provide a simple feedback. The results of the test cases are shown in a terminal window that can be hidden if the feedback is too much, and our Brackets user interface shows only detailed information of the test results, if the users open a popover.

Chapter 3

Design

"Design is concerned with how things work, how they are controlled, and the nature of the interaction between people and technology. When done well, the results are brilliant, pleasurable products. When done badly, the products are unusable, leaning to great frustration and irritation."

—Norman [2013]

3.1 User Interface

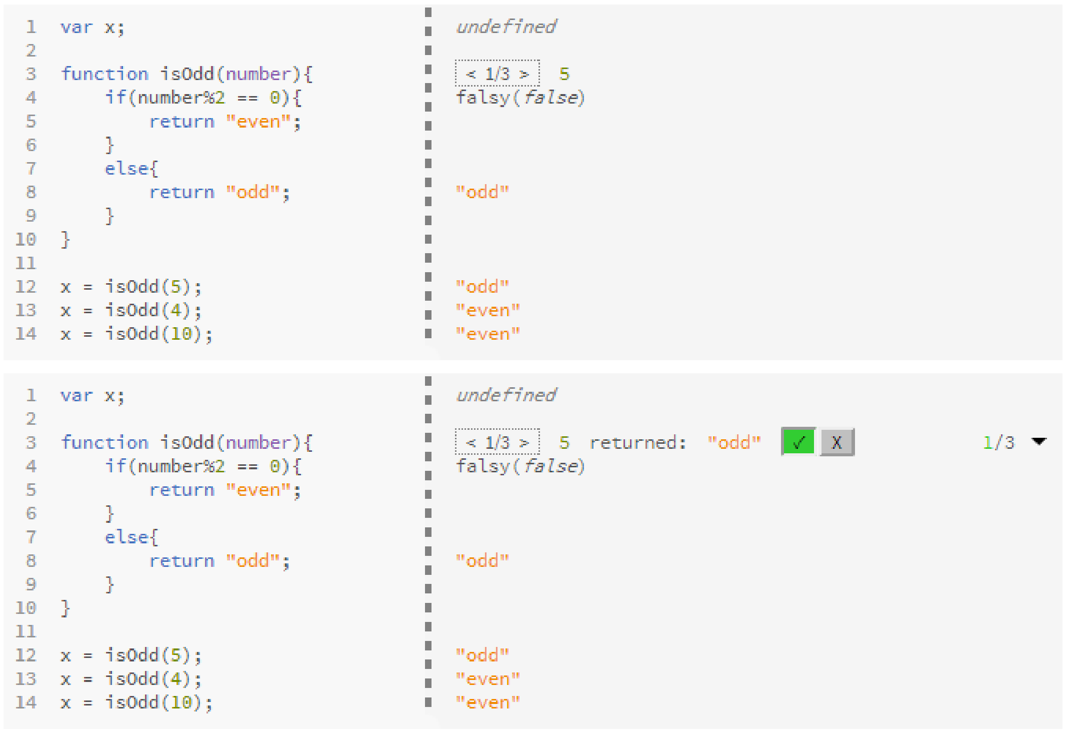
The user interface is based upon the already existing user interface of the Brackets extension that was developed by Joachim Kurz[1]. The actual user interface and the former one are mapped in Figure 3.1.

The user interface is based upon the live coding Brackets extension.

The first modification we made to the user interface, was to change the first line of the function result that is mapped in the second column of the view. The former version contained only one block to skip between the different function calls, and the respective input values at the right side of this block (see Figure 3.1). Now the first line also contains the

The first modification was to add the function result to the live coding column

[1]http://hci.rwth-aachen.de/livecoding



```
var x;                          undefined

function isOdd(number){         < 1/3 >  5
    if(number%2 == 0){          falsy(false)
        return "even";
    }
    else{
        return "odd";           "odd"
    }
}

x = isOdd(5);                   "odd"
x = isOdd(4);                   "even"
x = isOdd(10);                  "even"
```

```
var x;                          undefined

function isOdd(number){         < 1/3 >  5  returned: "odd"  ✓  X            1/3 ▾
    if(number%2 == 0){          falsy(false)
        return "even";
    }
    else{
        return "odd";           "odd"
    }
}

x = isOdd(5);                   "odd"
x = isOdd(4);                   "even"
x = isOdd(10);                  "even"
```

Figure 3.1: *Screenshot of the old user interface above and the new one below.*

return value that corresponds to the actually selected function call. Thus, now the developers have an overview of the main values that correspond to the function, even if the function body is collapsed. This return value comes with a short description in form of "returned:", this helps the users to understand what is displayed.

We added two buttons, to mark specific function calls as correct or false

Another modification in this line are the two buttons to mark the actual function call as correct or false. The first button with the checkmark is to mark the function call as correct, the second button with the "X" to mark the function call as false. This buttons are placed close to each other to make visible that they form a union. They do nearly the same, with the only difference that one marks the function call as correct, the other as false. With the use of the laws of proximity and similarity it is visible for the user that they build a group.

LAW OF PROXIMITY:
This is the first law of the "Gestalt Principles". Objects that are placed close together seem to be grouped. [Johnson, 2010]

Definition:
Law of Proximity

LAW OF SIMILARITY:
The law of similarity is another law of the "Gestalt Principles". It says that objects that look similar seem to be grouped together. [Johnson, 2010]

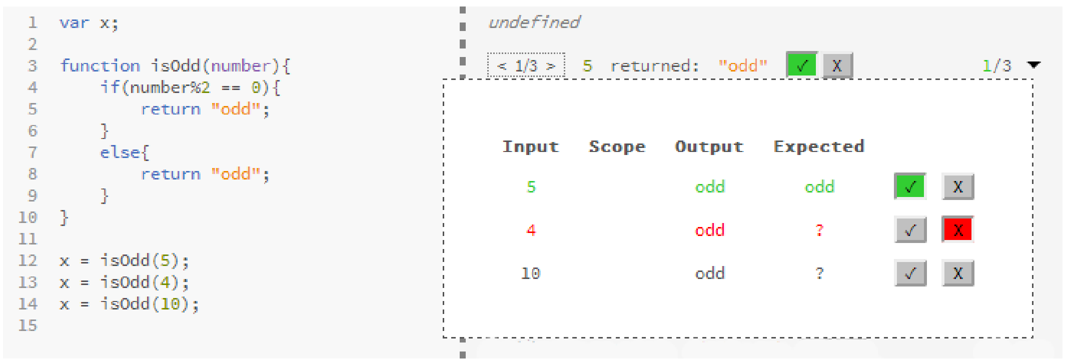
Definition:
Law of Similarity

In Figure 3.1, for example, the developers can mark if it is correct or false, that the function *isOdd(number)* returns *"odd"* with an input of 5. How these buttons are linked with the creation of new test cases, is explained in Chapter 4 "Implementation". If the first button is clicked, the actual function call is marked as correct and the button turns into a green button with a checkmark. Every human will associate the color green and a checkmark with *correct*. If the second button is clicked, the actual function call is marked as false and the button turns into a red button with an "x", because an x and the colour red is associated with *false*. If the respectively other button was marked before, this button turns grey again and the corresponding test case is deleted. A second click on a button turns the button again into the initial state and deletes all test cases that are created for this function call. Thus, it is easy to see if an function call is correct or not. For example, if the actual function call $isOdd\,(5) \rightarrow$ "*odd*" is marked as correct and we change the function code so that the function returns *even* for the input 5. Then the button turns automatically red because it was expected, due to the former marked correct function call, that the function returns "*odd*". If this result is also a correct result, it can be marked as correct and the program accepts both of this possibilities.

The first button marks the function call as correct, the second button marks it as false. A second click deletes the test case.

The next modification is at the right side of the two buttons. It is a simple overview of how many function calls are already marked as correct. The number of correct function calls appears in green (associated with *correct*) and the total number of function calls in grey, the standard text colour in this design. This listing also serves to a better overview of the code results.

Next to the buttons we added an overview of how many function calls are correct.



```
var x;

function isOdd(number){
    if(number%2 == 0){
        return "odd";
    }
    else{
        return "odd";
    }
}

x = isOdd(5);
x = isOdd(4);
x = isOdd(10);
```

```
undefined
< 1/3 >  5  returned: "odd"  ✓ X                1/3 ▾
```

Input	Scope	Output	Expected		
5		odd	odd	✓	X
4		odd	?	✓	X
10		odd	?	✓	X

Figure 3.2: *Screenshot of popover that appears with a click on disclosure triangle on the right side. It gives a detailed overview of all the function calls that belongs to the corresponding function. We use a wrong function that returns odd even if the number is even. Thus, this can be marked as false.*

The disclosure triangle opens a popover with an overview about all function calls and their current states.

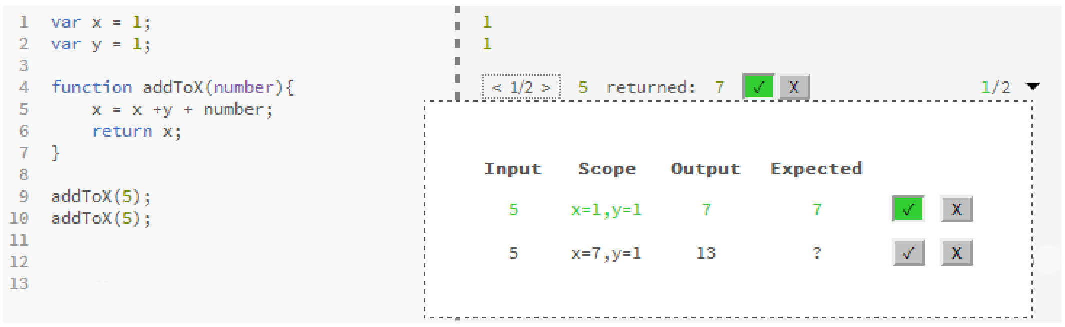
If the developers want to have a more detailed overview, they can use the disclosure triangle at the right side of the window. This disclosure triangle opens the popover shown in Figure 3.2. This popover contains an overview of all function calls that belong to the function in this line. The input and output values of the respective function, the scope, the expected value for the corresponding function call and again the two buttons to generate test cases are listed in this popover. The expected value varies depending on whether there is already a test case for the corresponding input value, that was marked as correct. Like in the example above, if the combination $isOdd\,(5) \rightarrow\ "odd"$ is marked as correct, then the expected value would be *"odd"*.

The column "Scope" shows the needed context for this function call.

Sometimes the return value - if there is one - of a function not only depends on the input value. It may also depend on an additional context, like variables from the global or the closure scope. Therefore, an additional column is added, it contains the context that is needed by the chosen function call (see Figure 3.3). If no context is needed this column is empty (see Figure 3.2). This needed context will also be saved in the test cases.

Definition: *Global Scope*

GLOBAL SCOPE:
The global scope refers to variables that are defined globally. That means they are available everywhere in the code.



```
var x = 1;
var y = 1;

function addToX(number){
    x = x +y + number;
    return x;
}

addToX(5);
addToX(5);
```

```
1
1
< 1/2 >  5  returned:  7  ✓ X                    1/2
```

Input	Scope	Output	Expected		
5	x=1,y=1	7	7	✓	X
5	x=7,y=1	13	?	✓	X

Figure 3.3: *Screenshot of the "Scope" column that shows the additional context that is needed by the function in this function call.*

CLOSURE SCOPE:
A closure is an anonymous inner function that is defined in an outer function. This inner function gets access to the context of the outer function. For example:

```
function greet (first) {
  return function (second){
    return first + second;
  }
}
var helloX = greet('Hello ');
var helloBob = helloX('Bob');

console.log(helloBob);
// returns ''Hello Bob''
```

Definition:
Closure Scope

3.2 Test Cases

The layout of the test case in the "testfile_spec.js" file is based upon the design that is described on the web page of the Jasmine Testing Framework[2]. A test case consists of a *describe* block, an *it* block and a block where expectations are formulated. The *describe* block contains a title for the actual test suite and a function. This function contains the *it* block. This *it* block again contains a title and a function. In this last function the expectations are contained. Thus, a normal Jasmine test suite looks as in Figure 3.4.

Jasmine test cases need a specific kind of design.

[2]http://jasmine.github.io/2.0/introduction.html

```
describe('title one', function(){
  it('title two', function(){
    expect(...);
  });
});
```

Figure 3.4: *Design of a normal Jasmine test suite.*

Our test cases have one *describe* block per function, this *describe* block has several *it* blocks per function and every *it* block has only one expectation.

A *describe* block can contain more than one *it* block, and an *it* block can contain more than one expectation. In this thesis we work with one test suite per function, that means one *describe* block per function. This *describe* block will contain one *it* block per function call and this *it* block always contains just one expectation. In Figure 3.5 such a test suite is shown. The first title describes which function is tested with this test suite, the title has the form `'function: FUNCTION_NAME'`. The second title describes which input is tested and what result we expect. It is of the form `'checks input INPUT_NAME (INPUT CONTEXT) toBe EXPECTED_RETURN_VALUE'` or if a return value is not expected `'checks input INPUT_NAME (INPUT CONTEXT) not.toBe EXPECTED_RETURN_VALUE'`. Thus, with these two titles it should be clear what a test does. The *expect* part is the part that finally executes a test. It calls the respective function with the given input value and the maybe needed context. With the matchers *toBe* or *not.toBe* it is going to test whether the output is as expected.

As parameters the function gets the needed context values, this-variables and the function parameters.

The *expect* block has also always the same form in this thesis. It is build as follows:

```
expect(
  FUNCTION_NAME.call(
    {THIS_VALUES},
    INPUT_VALUES,
    [CONTEXT_VALUES])
  ).toBe(EXPECTED_RETURN_VALUE);
```

The `THIS_VALUES` are the values that appear in the origi-

```
 eval(require('fs').readFileSync('currentCode.js', 'utf8'));

 describe('function: isOdd', function(){
   it('checks input 10 () not.toBe odd', function(){
     expect(isOdd.call({} ,10,[])).not.toBe('odd');
   });
   it('checks input 5 () toBe odd', function(){
     expect(isOdd.call({} ,5,[])).toBe('odd');
   });
});
```

Figure 3.5: *Design of the test cases in the "testfile_spec".*

nal function with a *this.* keyword. The `INPUT_VALUES` are the parameters that are needed to call the function and the `CONTEXT_VALUES` are the values out of the global or the closure scope that are needed by the function.

These mentioned parts together form a valid test case that can be tested with Jasmine.

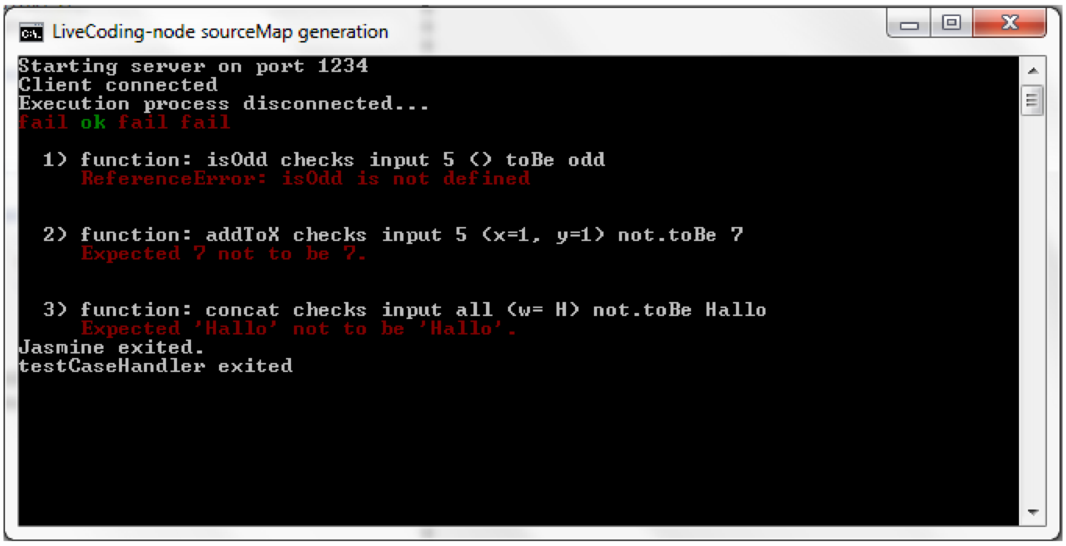
Results of the TestCases

The results of the different test cases are mapped in the terminal that is needed to execute the server behind the Brackets extension. Figure 3.6 shows how the results are reported. The first line of the test results contains a small overview of the result. If a test case is true, means the real output of a function matches the expected output, a green "ok" appears. If a test case is false a red "fail" appears and below all the failures are described. The description of the failed test cases are based on the titles that are given in the *describe* and the *it* block. These titles should help to understand which error occurred in detail. In Figure 3.6 the first failure description is:

The results of the test cases are shown in the terminal that is needed to run the server.

```
function:  addToX checks input 5 (x=1, y=1)
not.toBe 7
Expected 7 not to be 7.
```

Thus, we can see that the function *addToX* is checked. The



```
Starting server on port 1234
Client connected
Execution process disconnected...
fail ok fail fail

  1) function: isOdd checks input 5 () toBe odd
     ReferenceError: isOdd is not defined


  2) function: addToX checks input 5 (x=1, y=1) not.toBe 7
     Expected 7 not to be 7.


  3) function: concat checks input all (w= H) not.toBe Hallo
     Expected 'Hallo' not to be 'Hallo'.
Jasmine exited.
testCaseHandler exited
```

Figure 3.6: *Terminal screenshot of the Jasmine test results*

The first line of the results are a short summery, below are detailed information of the failures.

input is the value *5*, the scope is *x=1, y=1* and the result is expected not to be *7*. The description in the second line shows that the result of this function is exactly *7* and the test case resulted in false. Like this it is easy to identify the wrong function.

Chapter 4

Implementation

The whole implementation of the live coding and continuous testing combination is split into two parts: the client and the server. The client is responsible for the design of the user interface, that was described in Chapter 3 "Design", and to bundle all the information that are needed by the server to create a new test case. The server is responsible to create a test case, to execute it and to send the received results back to the client.

The implementation is split into a client and a server part.

4.1 Client

The main tasks of the client are building the user interface and to bundle all the information that are needed to create test cases that can be executed with Jasmine. To create a new test case the developers need the different buttons that are described in Chapter 3 "Design". These buttons are created as objects of the following form:

The client has to ensure the user interface, saves information for test cases if a button click is noticed and sends them to the server.

```
var newButton = {};
newButton.type = "button";
newButton.identifier = FUNCTION_NAME;
newButton.name = TEXT_TO_SHOW_ON_HOVER;
newButton.neededContext = NEEDED_CONTEXT;
newButton.initValue = FUNCTION_PARAMETER;
newButton.returnValue = RETURN_VALUE;
```

The function name is calculated from the function start location and the original source code.

The function name is calculated by a separate function that gets the entire code and the start location of the function body (that comes from the Esprima parser[1]). A special case are anonymous functions. They exist in the following forms:

```
var doSomething = function () { ...
//the identifier is "doSomething"

... return function(parameter){ ...
// there will be no testing option
```

Closures can not be tested.

The first kind of anonymous function gets the function name that is given to the function with the variable initialization. The second kind of anonymous function is a function that only exist locally in another function. Thus, it is impossible to call this second kind of anonymous function, regardless whether it has a function name or not. Hence it follows that closures can not be tested, because on the one hand we can not call the return-function from outside and on the other hand it is impossible to test if a function as return value is exactly the function that we expected. Therefore an algorithm would be needed that checks if two functions do exactly the same, although they are build in different ways.

The return value is calculated from the Esprima syntax tree.

Return values and the function parameters are calculated in the same way, by external functions. These functions get the code that is parsed by Esprima and return the wanted values.

The context capturing is also based on the Esprima syntax tree. To find the context the tree is searched for identifiers.

The context that is shown in the overview popover (see Figure 3.3), is calculated during the first interpretation of the source code after reloading the actual window. The syntax tree, that is calculated with the Esprima parser is searched for identifiers. The identifiers that are created within the respective function, just as the function parameters are subtracted from the former found identifiers. Thus, the remaining identifiers are the ones that are needed from the closure or the global scope. To get the respective values of these identifiers, every function gets the following hidden line:

[1]http://esprima.org/

```
var thisScopeVariables = 1;
try{var scopePARAMETER = PARAMETER;}catch(e){}
try{ ....
```

Hidden in this case means, this line is added right after the function head, but this line is not visible for users of this program. The line is only needed to get the values of the current scope. The first variable initialization (`var thisScopeVariables = 1`) is only an identifier to show where scope variables are saved. The try-catch block is needed, because it might be that variables are defined as scope variables, that are not defined. The try-catch block ensures that the other variables are saved and no error occurs if something is undefined. Therefore, it is possible to filter the respective scope so that only the needed variables are left over. This filtering avoids an overhead of information that would occur if we save all identifiers that occur.

The values of the variables that are needed from the scope are saved with a hidden line after the function head.

Every function call has an own button with the respective information. With a click on this button, a function sends all the data that is contained in this button object to the server with the note to create a new test case based on this information. At the same time the information is saved locally in the client to update the user interface. Like this the buttons are working without delays. A delay would occur, if the client has to wait for the test case results from the server to update the user interface. The local test cases are saved as multidimensional objects of the following form:

A click on the button creates and saves a local test case object based on the information that are contained in the button-object.

```
FUNCTION_NAME:
  PARAMETER_VALUES:
    SCOPE_VALUES:
      OUTPUT_VALUES:
        MARKED_AS: TEST_RESULT
```

This object is needed to create all the buttons in the correct colour and the popover with the detailed information about the function calls. The colours are set related to the accord of the test results with the expected value. If these values agree, the line is coloured in green, if they do not agree, the line is coloured in red and if there is no expected value and the function call is not marked as false the line stays grey. When the server finishes the test case execution this local test case object is overwritten with the actual test

The specific form of the local test case object in the client ensures the colouring of the result (green if the test result is as expected, red if not).

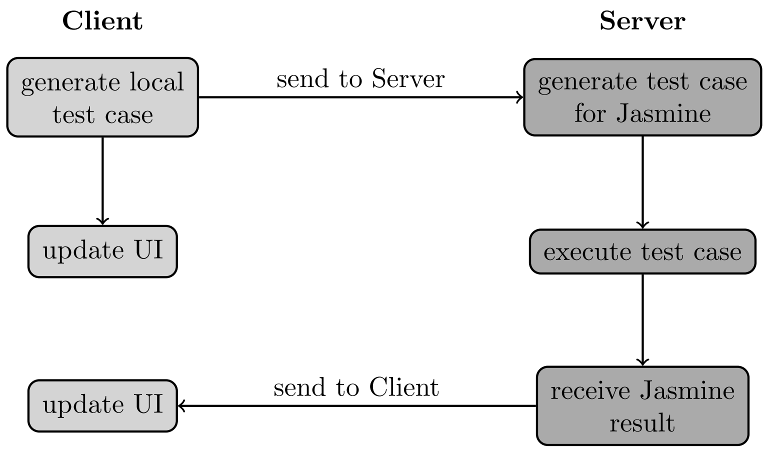


Figure 4.1: *Graph of the interaction between Client and Server.*

results from the server and the user interface is updated again. The last update of the user interface ensures the correct colouring of the lines in the popover, because it might be that there is a new expected value. The client server interaction is mapped in Figure 4.1.

4.2 Server

The server is responsible for executing the test case with Jasmine and sending the results back to the client.

The server is responsible for creating the test cases, for executing them and for sending the results of the test cases back to the client. To create a new test case, the developers click on a specific button in the user interface, the client collects the needed data and sends them to the server (see Figure 4.1). The server needs to change the form of this data so that it can be executed with Jasmine. The specific form, that is needed for Jasmine, is already described in Chapter 3 “Design”. To handle these test cases we created a new file that is called ”testCaseHandler.js”. In this file the data is changed from object into a Jasmine compatible string, Jasmine is started and the results from the jasmine execution are changed from a jasmine object into an object that has the same form as the test case object in the client.

To execute the test cases in Jasmine, they are written into a file named "testfile_spec.js". This local test case file also ensures that the test cases are still available after restarting the program. After the test cases are written into the file, Jasmine is started by the "testCaseHandler.js" and it executes the test cases on the original program code. Before Jasmine writes the test results into the terminal, like described in Chapter 3 "Design", it sends the test results to the "testCaseHandler.js" where the results are written into an object with the specific form that is needed in the client (described in the section above). The object is send via a web socket to the client and there the local test case object is overwritten with this new object from the server.

Jasmine executes the test cases, sends test results to the server and displays the test result to the users.

To ensure a fast execution of the server, the "testCaseHandler.js" and the Jasmine framework are running in child processes of the server. Jasmine is started by "forking" the "cli.js" inside the jasmine-node package. This package has to be installed with `npm install jasmine-node` inside the server folder. The contend of this package is only changed to send the test results to the "testCaseHandler.js" before writing them into the terminal and to change the design of the test results in the terminal. The changes are located in the "reporter.js" and the "cli.js" file.

Jasmine is running in a child process of the server.

Jasmine is always executed if a new test case is created or if the Brackets extension is reloaded. This ensures that the developers always have the current state of their program, also if they restart the program. A test case is only deleted if the function call is marked again as *undefined* or if the test case is already marked as *true*, and it is changed to *false*, then the *true* test cases is deleted and vice versa. Thus, with the time the developers get a large test suite for the respective functions, without writing a test case on their own.

The test cases are executed if the users create new test cases or if the program is restarted.

Chapter 5

Evaluation

With this combination of live coding and continuous testing we wanted to develop a testing tool for the former implemented live coding Brackets extension. This testing tool should create test cases automatically, it should work with the method of continuous testing and it should provide an user interface that is easy to understand.

With this program we wanted to develop a tool that creates test cases automatically.

The automatic test case creation works without problems. The users only have to click one button and a test case is generated and executed automatically. We also added the possibility to capture the input context, not the output context. Without the input context a test case can be wrong, for example assume that we have the following part of code:

This automatic test case creation works with just one click.

```
var x = 1;

function addValueToX( number){
  x = x +  number;
  return x;
}

addValueToX(5); //returns 6
addValueToX(6); //returns 12
```

The fist call of the function *addValueToX* would return 6 and the second call would return 12. If we want to test the second call, and we do not respect the global scope (means that the global variable x is equal to 6 before this call) we would

The output context is not captured like the input context, because it does not affect the main functionality of our program.

receive that this call returns 7 and this is not the expected value 12. Therefore, this actual version of the Brackets extension respects the input scope but not the output scope. An output context would be needed if, for example, a function changes a global variable without returning it as value. This change is not noticed in the actual version, but the test cases are still correct if this context is missing. Thus, it is not essential for the execution of our program.

A problem occurs if functions are renamed, because the test cases are not changed.

An essential problem occurs if users change a function name. The change does not change the function name in the test cases. This means if a function gets a new name the test cases will not work anymore because they need the function with the former name. It is possible to change the names manually in the test file but it would be better to ensure that the program changes the test cases automatically.

Test cases are only deleted if a button is clicked twice or if a test case is overwritten by another one, test cases should also be deleted if users delete a respective function.

Along with the problem of the renaming of test cases goes the problem of the deletion of test cases. In this version of the Brackets extension test cases are deleted if the developers mark them as *undefined* or if they are overwritten by another test case (see 4 "Implementation"). A functionality that should be added to the actual version is, to delete test cases if the function where the test cases belong to, is deleted. This should be added to avoid a massive overhead by executing test cases that are not necessary anymore. This functionality is also not implemented, because it does not affect the main functionality of this Brackets extension.

Beside the mentioned problems the program works as desired.

Like mentioned, only the problem of renaming a test case is a problem that affects the main functionality of the program. Beside this, the program works as desired. It creates test cases automatically, executes them automatically and displays the results in a readable way to the users.

A user study would give another overview of the functionality of the program.

To get a better overview of how users will work with this program it would be useful to do a small user study. During this thesis we focused on the functionality of the program without evaluating it in a user study. In the following are ideas how this program could be evaluated in such a user study.

For receiving a significant result during a study, it will be necessary to work with the former brackets extension without the testing methods and with the extension that was build during this thesis. The volunteers that work with the former Brackets extension may use an external testing framework to ensure the same conditions for both parts. Both groups may solve programming tasks, maybe to find errors in a predefined code or to write new code with the given frameworks. Then the time can be stopped and hopefully the group with this new live coding and continuous testing framework will solve the exercises quicker than the participants that work with the former version. To get more information it might be useful to do audio or video recording of the participants during the test. Like this it will be easier to see where are still difficulties and what problems occur.

In a user study it is important to give the same conditions. One group of participants can work with the new program, another group works only with the live coding extension and an additional testing tool.

Chapter 6

Summary and future work

6.1 Future work

The program still has limitations.

As mentioned in Chapter 5 "Evaluation" our program still has limitations. The deletion of test cases, the renaming of functions and test cases and the capturing of the output context are features that still remain.

The output context capturing leads to a problem with the test cases, because the output context is not returned.

To fix the output context capturing it would be possible to add the following line at the end of each function:

```
var thisScopeVariables = 1;
try{var scopePARAMETER = PARAMETER;}catch(e){}
try{ ....
```

Thus, the output context capturing would work similar to the input capturing. The problem with this solution will be the question how to test this output scope. The input scope can be tested by adding the wanted scope with a "call" to the executing function, but in this way the output scope can not be tested. To enable the output context capturing a new method is required.

Another task for the future is to solve the problem that if function names are changed all defined test cases for this

Another task for the future is to solve the renaming problem (test cases are not renamed if functions are.

functions still exist with the old function name. Therefore, we need an algorithm that compares the old code and the changed code and finds out if a function name is changed. With this algorithm it will be possible to change also the test cases if a function is changed. In our Brackets extension only the new code is analyzed without relation to the former code version.

The problem of the test case deletion should also be fixed in the future.

The last problem that was mentioned, is the deletion of the test cases. This can be solved with a function that gets all the function names of the existing test cases (that can be found in the description of the *describe* block in the test case file') and the function names of the new source code (that can be found with help of the Esprima tree). With this information the deleted function can be found and the test cases can be deleted.

An additional idea is to provide the manual addition of test cases in a guided form.

With this improvements the Brackets extension should work without any problems. Anyway there will be always functionalities that would improve this Brackets extension. One idea of improvement is, to enable a possibility to add test cases manually. This might be a button with "add test case" where a window appears that ensures the developers to add test cases that do not occur in their code. Thus, it would be possible to cover the edge cases, that are not highly covered in this version, like mentioned in Chapter 1 "Introduction" (see Figure 1.2). It will be necessary to guide the developers while entering a new test case, because the specific form of a jasmine test case has to be ensured. Therefore, it is possible to give input boxes for "function name", "function parameter" and "expected return value", this would ensure a correct test case design.

Another idea is to provide a test case prioritization.

The last idea is a test case prioritization. In our Brackets extension all existing test cases are executed, although some test cases may belong to other JavaScript files. With a lot of test cases this would lead to an enormous overhead and it would be better to filter the test cases. Thus, only the needed test cases for the actual file are executed.

6.2 Summary

During this thesis we implemented a combination of live coding and continuous testing for Brackets. This extension is based on the already existing live coding Brackets extension developed by Joachim Kurz. The design of our program is leaned on the design of the previous version of this Brackets extension. A few objects are added to ensure a better overview of the code results, like the output variables of a function. The most important thing we added are the buttons to create test cases, because this combination of live coding and continuous testing creates test cases automatically after a click to this buttons. The information for these test cases are collected by the client side of our program and created and executed by the server side using the JavaScript testing framework Jasmine. The server sends the test results back to the client, where they are displayed to the users. Thus, the users get a detailed overview how their code work, if the functions return the expected values or if they contain errors. This method ensures a high test coverage of the average requests. The edge cases are less covered. In the future work we want to enable a possibility to add test cases manually in an easy way. This should provide a better edge case test coverage.

Bibliography

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

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

M. M. Burnett, J. W. Atwood Jr, and Z. T. Welch. Implementing level 4 liveness in declarative visual programming languages. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 126–, 1998.

Jeff Johnson. *Designing with the Mind in Mind*. 2010.

Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. How live coding affects developers' coding behavior. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, 2014.

Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Data debugging with continuous testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 631–634. ACM, 2013.

Don Norman. *Design of Everyday Things*. 2013.

David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 281–192. IEEE Computer Society, 2003.

David Saff and Michael D. Ernst. Continuous testing in eclipse. *Electron. Notes Theor. Comput. Sci.*, 107:103–117, 2004.

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)*, pages 288–. IEEE Computer Society, 1997.

Steven L. Tanimoto. Viva: A visual language for image processing. *J. Vis. Lang. Comput.*, 1:127–139, 1990.

Steven L. Tanimoto. A perspective on the evolution of live programming. In *Live Programming (LIVE), 2013 1st International Workshop on*, pages 31–34, 2013.

Index

Typeset September 19, 2014