TECHNISCHE UNIVERSITÄT
CHEMNITZ

Chemnitz University of Technology, GERMANY
Faculty of Computer Science
Professorship of Software Engineering

Bachelors Thesis

Eyecatcher - The Effect of three Distinct Question Types on the Solving Time and Correctness of Code Snippet Questionnaires in an online remote Eye-Tracking environment

Advisor:

Univ.-Prof. Dr. Janet Siegmund
Elisa Hartmann, Marc Schwarzkopf

Chemnitz, July 31, 2022

Contents

This study investigates code comprehension studies using neuroimaging and eye-tracking technologies. Due to novelty and different technologies, many studies differ widely. Comparing studies from both fields, many did not provide complete information for reproducing their findings. Missing information often included question types which could result in significant differences in solving behaviour.
An online remote eye-tracking survey with 16 participants was designed and conducted. The analysis shows no details of correlation between question type and solving time or correctness due to missing normal distribution and low sample size. However, a correlation cannot be rejected. Future research may show evidence that the question type affects solving behaviour.

Keywords: code comprehension, remote study, survey, eye-tracking, remote eye-tracking, neuroimaging, REyeker

1 Introduction

1.1 Motivation

Humans have been communicating for over 5000 years in different countries, regions and cultures using different languages [10]. Humans' verbal and written communication has changed and evolved over thousands of years. They contain many different rules and intricacies that are hard to comprehend and grasp [10]. There have also been many different programming languages in over 50 years. These have also changed over the years, but more rapidly and less gradually than natural languages. Programming languages started as machine code - simple instructions for the machines represented as zeroes and ones, which the engineer skillfully chained together to accomplish a larger goal. Additionally, program languages are structurally different to natural languages. The text of natural languages such as English or German is structured in sentences and paragraphs. In contrast, programming languages use statements, blocks, loops and different ways to change the execution order of the program [63]. Later, programming languages started using short character chains as an identifier for those instructions. However, developers still needed a lot of skill and abstraction because they had a minimal set of instructions and specific hardware and memory [79].
Within these years, many programming languages were designed and developed to expand the possibilities of programming, each with different purpose in mind [21, 79]. Some programming languages like 'Brainfuck' were made to confuse, befuddle and interest the code reader [23]. Nowadays, many languages employ feature- and object-oriented languages [60]. Before research in the field of understanding program comprehension started, programmers needed to rely on their intuition and experience, whether it was for designing a new programming language [19], locating syntax and logical errors [14] or more. Researchers could draw many different conclusions using studies with methods like neuroimaging (section 2.3.1), eyetracking (section 2.2.1) or a combination of those two.
These deductions help distinguish novice and advanced programmers by their programming code reading behaviour [5, 45], help understand the importance of syntax structure and highlighting [4, 8, 18, 67]. Research has also shown the differences in activated brain regions between natural text and programming code for comprehension purposes [28, 37], the reading order during software debugging [47, 66] and many more. These research results can help create better programming languages, syntax highlighting, debuggers, and helpful tools. Additionally, those conclusions can help design new teaching methods for people trying to learn to program.

1.2 Problem

The program comprehension research has only started recently, compared to other scientific fields. Nevertheless, the research is done extensively using various techniques and technologies. There may be ways to improve the realization of these researches which have not been considered significant, as will be explained in section 2.4.
Technologies such as fMRI (functional Magnetic Resonance Imaging) impose several constraints and restrictions (section 2.3.2) while producing insightful data to understand program comprehension. The primary motivation behind this study is the low level of attention given to the type of question and their related answer type (section 2.4). The choice of question may have a non-negligible impact on the program code reader's comprehension, reading order, or visual attention. Additionally, the findings of this study may help create new studies and surveys on the topic of program comprehension and their reading order.
Finally, visual attention in code comprehension has been studied for over thirty years [19], and the methods have evolved from natural ocular observations over head-mounted devices to specific to less obtrusive and more accurate eye-tracking tools [52].
This study would normally employ eye-tracking devices to accurately track the gaze, attention and other factors. However, in conditions like the current COVID19 Pandemic (Corona Virus-Infected Disease in the years 2020-2022), surveying test subjects locally is no longer feasible without risking the subjects health or breaking any laws. Instead, an online tool called 'REyeker' will be used to emulate an eye-tracker and track the reader's visual attention. This study considers the movement of visual attention as a factor for head movement to read and understand the code. Therefore a smaller amount of total attention movement may reduce the movement of the head needed. Using the REyeker tool, the test subjects may conduct the study at their favoured location without a need to purchase any expensive physical eye-tracking tools or gadgets. This way, a broader range and amount of test subjects can be part of this survey because the survey is done solely online, and no personnel is required to observe the test subject. The usage of an online tool also introduces a new set of challenges which will also be discussed.

2 Theory

To better understand this paper, several terms, topics and concepts need to be discussed for further comprehension. For this reason, those topics will be introduced step by step before explaining the test setup and procedure, so the reasons for choosing any specific option over another should become transparent.

2.1 Comprehension directions

A piece of information can be understood in different ways. Prior knowledge about the topic and similar things help understanding them. Top-Down and Bottom-Up comprehension are two often used concepts for explaining the order of understanding.

2.1.1 Top-down comprehension

Top-Down comprehension uses the reader's prior knowledge to understand the meaning and context of the written words or syntax. For Top-Down comprehension, the general context and situation are used to identify the purpose and meaning of the information given. This concept can be found in both natural and programming languages and can help identify the objective even if the listener or reader does not know the specific language or environment yet.

Figure 2.1: Latgalian street sign (Latvia)

An example in the non-programming world would be a street sign at an intersection, like figure 2.1. Given the context of two roads crossing and seeing signs with scriptures at every starting road, the viewer can deduce that those signs are likely the name of the streets meeting at the intersection or the directions to settlements following that path. Even if the letters are not known to the reader, they are likely to find the streets intersecting on a map, even if the map is in the native, to the reader unknown, language.

This context-deducing comprehension can also be utilized for reading unknown computer programs. For example, if the name of a method or function is "sort_list", the reader will understand

```python
def quickSort(array=[12,4,5,6,7,3,1,15]):

    less = []
    equal = []
    greater = []

    if len(array) > 1:
        pivot = array[0]
        for x in array:
            if x < pivot:
                less.append(x)
            elif x == pivot:
                equal.append(x)
            elif x > pivot:
                greater.append(x)
        return quickSort(less) + equal + quickSort(greater)
    else:
        return array
```

(a) Quicksort algorithm using descriptive names

```python
def method1(arrayA=[12,4,5,6,7,3,1,15]):

    arrayB = []
    arrayC = []
    arrayD = []

    if len(arrayA) > 1:
        a = arrayA[0]
        for x in arrayA:
            if x < a:
                arrayB.append(x)
            elif x == a:
                arrayC.append(x)
            elif x > a:
                arrayD.append(x)
        return method1(arrayB) + arrayC + method1(arrayD)
    else:
        return arrayA
```

(b) Quicksort algorithm with obfuscated variables

Figure 2.2: The same quicksort algorithm, but forcing a) Top-Down and b) Bottom-Up

that a given list will be sorted. If the reader is also familiar with list sorting algorithms, they may understand the order of sorting the list, even if they do not know the specific programming language. This same approach can also be used to understand named programs and variables. The top-down approach will help understand complex programs as long as the naming of objects like variables and functions is unambiguous. The sorting algorithm of figure 2.2a describes its purpose solely by the naming without the need to delve deeper into the function.
Even though Top-Down comprehension helps understand a wide variety of situations, there may be several chances for misunderstandings. Many words have different meanings in context, so the reader cannot be confident that a method or variable has the initially expected goal or purpose. For example, a method named ”DetectRightPath(...)” can either return the correct turn at an intersection for a pathfinding algorithm or return the path when turning to the right from the perspective of the given path. In these situations, the reader needs to dive deeper into the syntax of the method to understand how it works. This concept of diving deeper to understand the problem is known as Bottom-Up comprehension.

2.1.2 Bottom-up comprehension

Suppose the reader does not know out of context and prior knowledge to understand the observed text, program, picture, or other information. In that case, the reader needs to understand it from the details up to the bigger picture.
Imagine that a human sees a picture of the house design of an alien civilization, like the AI generated figure 2.3. They do not necessarily know that it may be a Building, but they can maybe identify details like windows, doors, floors, a roof, plants or ornaments to deduce its purpose. The viewer use generally known concepts to construct a mental image of the intention. There may be multiple similar buildings next to each other. These groupings of buildings could then be understood as a settlement, village, city or similar.
Likewise, understanding programming syntax builds from comprehending tiny elements like addition, variables and loops and how they interact with each other. Bottom-Up comprehension is the approach to understand new things from small details and recognizing relations between these atomic details to form more significant parts, because the higher contextual cues cannot

be understood or found. Programming requires much abstraction, as computer programs are not structured like natural languages.

Figure 2.3: AI visualisation of ”Alien House Design”, created with Craiyon [20]

Different programmers also write syntax differently, so reading a program written by a different human may feel like reading a different language and understanding it will require understanding small sub-parts. Additionally, Bottom-Up comprehension is beneficial for research purposes, as it reduces both the solving time and quality differences introduced by prior knowledge [58] and introduces a higher activation of brain areas than Top-Down [71]. Research has shown that experienced and novice programmers do not read unknown code in the same order [58]. Furthermore, expert programmers read code less linearly than novice programmers. Experts will jump inside the syntax much more to cross-reference and confirm or deny their ideas of the code. Because Bottom-Up comprehension is beneficial for research purposes, existing programming code snippets can be obfuscated not to show the purpose of the variable or function and renamed to ”variable1”, ”variable2”, ”method1”, and similar. An obfuscated version of the quicksort sorting algorithm can be found at figure 2.2b. These non-descript variable and method names will demand a higher load on the subjects’ working memory to store values throughout the whole program [71].

2.2 Eye-tracking

2.2.1 General

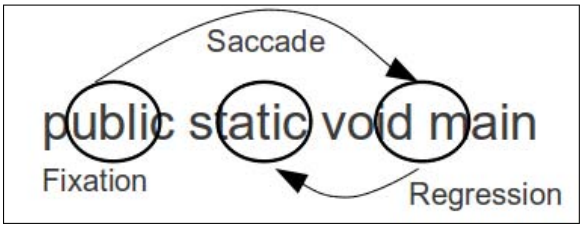
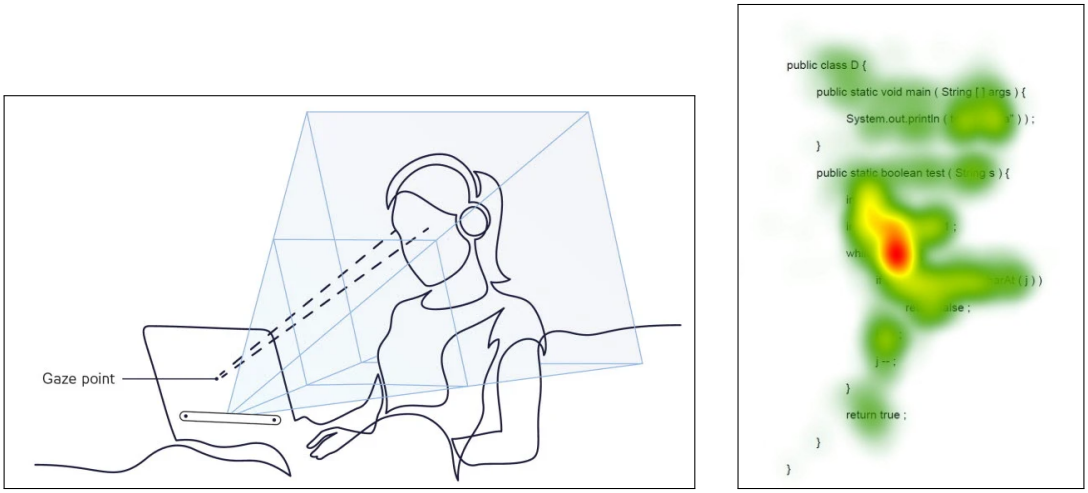


Figure 2.4: The path of visual attention [11]

Eye-tracking is described as the process of tracing the visual attention over a visible stimulus [12]. For example, tracking the viewers’ visual attention to a picture or text reveals essential information about the viewers’ thought processes, which cannot effortlessly be recorded with other methods [12]. When researching reading behaviour for natural text or programming code, various eye-tracking techniques are used to record and understand the readers’ thought processes. A strong indicator of the subjects’ attention is the view area focussed on with their eyes. A model devised by Just and Carpenter [40] links the focus and attention of a text read with their immediate



(a) Common Eye-Tracking setup [74] (b) Attention Heat-map [11]

Figure 2.5: The (a) process and (b) results from recording visual attention via eye-tracking

understanding. The attention stays fixated on the word until the necessary information is processed [19]. Visual attention is the subconscious aspect of processing optically observed information, whether a landscape in nature, a hand-drawn picture, text or a different optical stimulus. From one point to the subsequent, the movement visually focussed on points form a path over time between the points of interest. This path consists of fixations and saccades. Fixations are the points the eyes focus on, usually for a few milliseconds up to several seconds. Areas with many fixations and prolonged fixation time indicate great attention, which implies essential, interesting or complicated information [11, 55, 78]. Saccades are the jumps between fixations. These swift movements are near-instant and usually measured in degrees, as the distance from the eye can vary greatly, so the angle size is a constant measurement. Researchers assume that humans pick up very little information during saccades [11, 55].
A unique kind of a saccade is called a regression. Regressions are backwards jumping saccades to previous fixations. They indicate the re-intake of visual information to control or correct perception that has already taken place. A higher amount of regression indicates a higher complexity of the visual stimulus [10, 11]. Historically, there were many different methods to log the test subjects' attention. For example, the test subject was asked to provide a verbal assessment of the reading sequence, or the researcher watched the eye movements, both of which are prone to error. Direct observations or video protocols and other methods were used to improve the results' validity [73]. Different conventional approaches, such as asking the test subject after completing the task about the reading and understanding order, bring the risk of misreporting an experience. In particular, lengthy tasks are difficult to remember, and each person will assess their results subjectively, which can skew the results. Researchers can minimize the potential for misremembering by asking the test subjects to talk or write about their thought processes. However, this approach is at risk of interrupting the test subject in their thought process during the main task [12]. There is a similar disadvantage when using specific methods such as thinking aloud. The reason is that these techniques always take the programmer away from his or her original task. Even experienced programmers find it challenging to explain how they read the

program with their voice. Subjects must constantly be reminded to communicate their thoughts, as they do not have to express themselves when programming in a natural environment. Furthermore, many unconscious decisions are made but not communicated, such as logical dead ends when reading a program [12]. Due to these complications, it is desirable to record the test subject during their task without interrupting their workflow. Current eye-trackers are head or desk mounted devices that record fixations and saccades more than 50 times each second, like the example depicted in figure 2.5a. This recorded data can then help generate heatmaps and attention information about the visual stimulus, depicted in figure 2.5b.

2.2.2 Eye-tracking on source code

Researchers have used eye-tracking to study the reading behaviour of humans since the 19th century [73], starting with the reading behaviour for natural text. These experiments mainly focused on the reading order and the understanding of unmodified source code. This Programming syntax was most of the time stripped of any comments. Comments are helpful to the reader of the code to understand the methods and ideas presented. Comments will be ignored by the interpreter or compiler and are solely explanations or annotations to document the program. If there are no comments, the reader must understand the program's purpose by understanding the logic written in the program code. As mentioned in section 2.1, understanding directions differ whether the names of variables and functions have descriptive names.
Additionally, different researches explored the impact of various identifier styles [8, 67]. Identifiers are unique names given to each variable, method or function present in a program. They are indispensable in programming as they give a hint for their purpose by their name and type. Identifiers are static during the whole program, meaning they will not change their name and can be an essential tool to understand the code more easily. Modern programming languages allow developers liberties in naming their identifiers [8]. The main restriction of identifier naming is the limitation of some characters. Several special characters are not allowed, such as the empty character "Space", because it would break the program execution.
Some integrated development environments (IDEs) even allow programmers to use special characters such as Emojis as identifiers [32]. Often, the identifiers need to be longer than a single word so the identifier can correctly identify the purpose. The two main formatting styles for identifiers are CamelCase and under_score. The CamelCase style works by stringing together all words in the identifier and capitalizing each first letter per word for faster reading and understanding. The under_score style is an outcome of old programming languages not being case-sensitive. Because of hardware and memory limitations, these languages did not differentiate between lower and uppercase characters. Therefore, for easier and faster reading and program comprehension, the under_score was introduced between each word in the identifier [67].
The compiler is a computer program that translates one program language into another, mainly used to generate machine code from different programming languages so that the computer can execute the program. Each identifier will get renamed to a different string of characters only read by the machine, so the length or structure of the identifier will not have any impact [2]. Similar to compilers are interpreters, which is another way to process programming source

code. It works by directly executing the instructions specified in the program code instead of producing a target program as a translation [2]. Both kinds of language processors have their advantages and disadvantages over each other. Using machine code created by a compiler is usually much faster than interpreted code because the machine code is optimized already for the machine and does not need to be translated anymore. However, the interpreted source code must not be compiled to be executable after each change. This flexibility provides better error diagnostic and changeability [2].

2.2.3 Eye-tracking without an eyetracker

Because of the current worldwide event, also known as the COVID-19 pandemic, the methods used in this study need to be changed to a more home-friendly approach, as test subjects cannot be asked to participate in person without risking their health. This issue is why the study will be conducted online only, bringing its advantages and disadvantages. Usually, this study would be held in the rooms of the Chemnitz University of Technology with equipment provided by the software engineering professorship of computer science faculty.
A new strategy needs to be found, as it is currently impossible to conduct any research using stationary head-mounted or table-mounted eye-tracking devices. There are several requirements for this new procedure, as the study needs to be conducted without supervision at the participant's location. The first requirement is that the survey respondents can participate without any additional hardware. There would be an immense amount of organization to ship and return the devices to and from each participant. Additionally, the participants are not trained in the setup and handling of these devices, likely introducing unknown inaccuracies. The new method would also be required to work on most currently used consumer hardware, Not to exclude participants at an early stage. In addition to not requiring special hardware, the user should not be required to download and install new software. Newman et a. introduced their toolbox called TurkEyes in 2020 with four distinct interfaces for capturing and quantifying attention data [54]. This toolbox works without needing a camera or other external devices. In addition, all TurkEyes interfaces work with current web browsers, so participants can use their current devices without installing additional programs. These interfaces are called ZoomMaps, CodeCharts, ImportAnnots and BubbleView.
ZoomMaps capture the visual attention of multi-scale content like maps with panning and zooming inputs to generate attention heatmaps and quantify the average zoom of each area. The participant must zoom and pan at several different regions of the stimuli to see and understand the essential and relevant details to answer the questions. ZoomMaps is mainly used for capturing attention data for pictures. A higher time spent on a higher zoom level indicates higher attention spent on this area [54].
CodeCharts are a method to record individual attention points (gaze points). This interface shows the stimulus for a set amount of time. Immediately after, a grid of three-character codes will be shown, and the interface requires the participant to self-report the triplet at the position where they looked at the moment. This approach creates one attention point per participant per image. Attention heatmaps are created afterwards by averaging all attention points (one

per participant), so many participants are required. Survey creators must finetune their chart placement because otherwise, gridlike artefacts can diminish the output [54].
ImportAnnots works on the idea of participants marking and annotating essential areas in graphic designs. These annotations are averaged to show the areas the participants found most interesting. This approach collects participants' attention given deliberately. Participants need to mark an area intentionally, so the unwanted viewer's attention will not be recorded. Conventional eye-tracking methods aim to capture the subconscious attention [54].
BubbleView is another tool to capture visual attention by asking the participants to explore one image at a time. The image is blurred at the start to disable legibility and emulate peripheral vision. Regions can be clicked on using the cursor to focus on the area. Clicking on a new point blurs the previous point again [54]. This method records a dataset for each participant consisting of several gaze points and their timing. Adjustable parameters are the blur strength, the size of the viewing bubble and the timing and setup of the experiment [54]. These interfaces distinguish themselves concerning various criteria, such as cost, type of attention recorded, enjoyment of the participants, minimum participants required and resemblance to eye-tracking done by regular eye-trackers.
BubbleView is the interface that solves many problems introduced by the COVID-19 pandemic to conduct eye-tracking experiments. BubbleView is the only tool to capture the gaze path over the whole viewing duration, and it both requires no configuration on the user side and works in a web browser [54]. Participants can even use their mobile devices with touch controls because the interface works click-based with similar success [41].
A recent study has further developed the ideas presented with BubbleView to better study viewing behaviour when reading source code and integrating it into different survey tools. This tool is called REyeker (**R**emote **Eye** Trac**ker**) [72]. It has several adjustable parameters, such as the shape and size of the deblurred area and the transition between the blurred and deblurred area. Additionally, the blur can be smeared in either the x or y dimension. REyeker can generate both attention heatmaps as well as the chronological sequence of click data [72]. Regarding these previously mentioned restrictions and requirements, REyeker will be the primary tool for the survey, as its flexibility and features are fitting well.

2.3 Neuroimaging

2.3.1 General

The study of the human body is a field in science that has been explored for centuries. More discoveries and progress in science led to an increasingly better understanding of the human body and mind. Thanks to the recent surge in technology, previously unthinkable research was no longer a dream. Measuring the structure and activity of the brain of a living organism was possible and improved further over time. Early methods used X-rays, which were low in temporal and spatial resolution and more harmful to the human body than previously assumed. Spatial resolution can be compared to an image's pixel density, where a higher

resolution implies a more detailed image. The temporal resolution expresses how fast brain activity changes can be detected. Temporal resolution can be compared to the shutter time of a camera, where fast changes in the image produce a blur over the area of the change. If the temporal resolution is too low, all different brain activities get added up or averaged out, defeating the purpose of analyzing specific brain activities if they switch activity levels too fast. Scientists have since developed better methods to map the structure of the brain. For example, with functional magnetic resonance imaging (fMRI) researchers could measure the brain's activity via the blood-oxygen level. When the brain activity in an area increases, the need for oxygen increases in that area. This effect is known as the BOLD (Blood-Oxygen-Level-Dependent) effect — the oxygen level increases and peaks after around 5 seconds of brain activity and decreases on average after 12 seconds after finishing the task [59]. Therefore, before, during and after the task, the baseline BOLD level needs to be recorded to detect brain activity accurately. Complicated activities such as solving a puzzle or understanding a sentence activate multiple areas of the brain, as depicted in figure 2.6. These different brain areas are classified as Brodmann areas. There are 52 Brodmann areas associated with different mental processes [29].

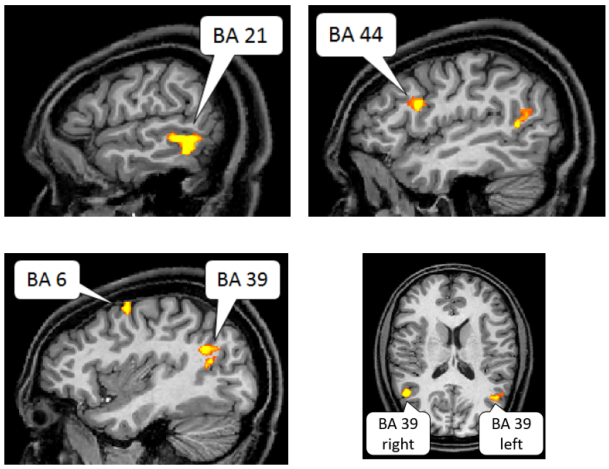


Figure 2.6: Brodman-area activation during comprehension based on semantic cues [11]

Other less restrictive methods, such as functional near-infrared spectroscopy (fNIRS) and Electroencephalography (EEG), can complement each other to measure brain activity. Therefore, researchers use these methods separate from each other and together to study cognitive processes. While research has been done on reading source code since 1990 [19], the first papers to study comprehending programming source code with neuroimaging methods were published in 2014 [35, 53, 69], so this field of study is a recent one.

2.3.2 Constraints

When researching or measuring cerebral activity with neuroimaging, the test implementation and equipment restrict researchers in several ways. The test subject must be conscious and actively comprehend the visual stimulus, such as the source code.

The test participant is in the measuring tube when researching brain activity with fMRI, as shown in figure 2.7. This specific test setup was used by N. Peitek et al [60]. Even though different neuroimaging methods like EEG and fMRT do not require lying in a tube, fMRI provides much valuable research data and is used in many research trials on this topic [14, 22, 26, 37, 48]. The subject can only see the distant screen through a mirror, so the source code needs high read-

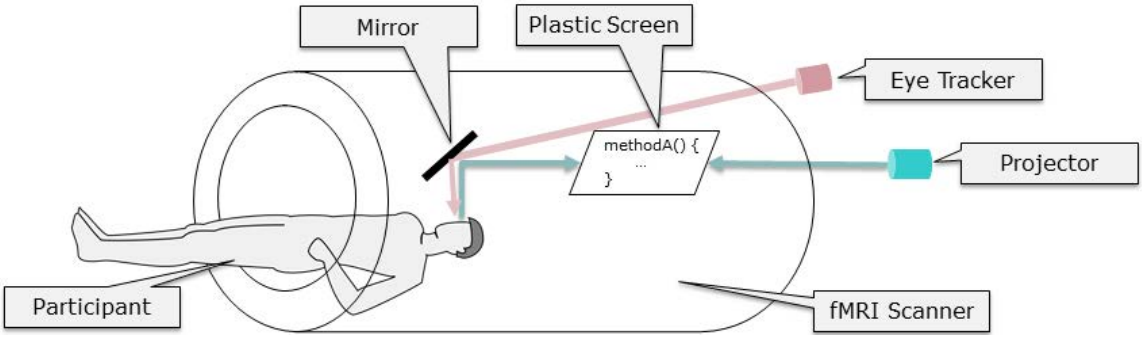


Figure 2.7: Test setup of an fMRI study with eye-tracker [60]

ability and sufficient character size. In addition, all parts used in the fMRI scanner need not be magnetic to affect the measurements, so the eye-tracker, screen, and answering tool must me non-magnetic. Additionally, the scanner has decreased controls to a limited amount of buttons for the test subjects, so scrolling or free inputs are not in the realm of possibility. Widely available fMRI button response systems will only have a small cable connected to a remote with a limited number of buttons [75]. Because of these restrictions, any researcher has to choose their source code carefully to fit these parameters. Furthermore, speaking is not possible during the test, as that would introduce involuntary head movements. Participants often wear earplugs for noise protection [59, 60, 69, 71].
Additionally, the questions given to the test subject in the scanner have to be answerable with the limited amount of buttons present. These questions have to show an understanding of the source code, so they need to be selected thoughtfully, and the answers present should not imply the solution nor be solvable through the exclusion procedure. Because of the need to record the baseline BOLD, the research requires control questions. These control questions should lay in a visually similar context but without any understanding needed. The chosen task was to locate syntax errors several times [24, 59, 60, 69]. The critical difference between answering questions for understanding purposes and pattern matching for syntax error location is that for the second, there is no understanding of the underlying code needed while still looking at similar words and patterns. The brain regions responsible for reacting to the source code will activate for both, while the code understanding areas will only activate when stimulated by comprehension questions [69]. Furthermore, the questionnaire should have a limited time scope, as the test subjects will become tired and restless, which may cause unwanted head movements [26, 43, 60].

2.4 Focus of current source code comprehension studies

While many different studies are researching the understanding of program source code, there are several similarities and diversifying features between them. In this literature analysis, 60 source code comprehension studies were considered. Of these, 28 mainly used neuroimaging and 32 mainly eye-tracking methods. Eye-tracking studies have been conducted far longer, as

the technologies needed were earlier available in the late twentieth century. Due to the many differences like constraints and possible insights, the two methods will be compared independently first and generally afterwards. A direct comparison between both parts would be unwieldy and does not provide valuable insight. Instead, this literature analysis aims to understand how these studies are conducted and where the focus lies. The analysis gives an overview of the research topics, test setup, comprehension direction, question topics, question types, answer methods, and additional information.

2.4.1 Focus of current studies using eye-tracking

The first eye-tracking studies researching code comprehension were conducted around 1990 [19]. These studies mainly focused on the reading direction of source code and general comprehension but also focused on the benefit of syntax highlighting [64], capitalization [7] or indentation [4]. Eyetracking studies are especially interesting for studying computer programming, as the study environment is nearly identical to the work environment for test subjects: The test participant sits at a table in front of a Monitor, similar to figure 2.5a. Close to the monitor is the eye-tracking device placed, and the candidate is instructed to hold their head still not to deteriorate any results. Due to the improving technology, better detail was possible to record and analyze. These studies employed a diverse set of eye-monitoring technology such as eye-movement monitors [19], desk-mounted eye-trackers [6, 9, 12] and head-mounted eye-trackers [39] by a wide variety of manufacturers.
Most eye-tracking studies utilize top-down comprehension (2.1.1). For example, in more than 20 cases [1, 3, 5, 7, 9, 10, 12, 18, 19, 47, 49, 50, 55, 56, 61–63, 65, 66, 78, 80], unmodified source code was used to simulate software developers' typical reading environment. At the same time, only five studies obfuscated their code [4, 6, 11, 27, 38] to force bottom-up comprehension. Only one study utilized both top-down and bottom-up comprehension [58]. These unevenly distributed parts of obfuscated and clean syntax code may be explained by the drive to research the reading behaviour in general and the extent of the impact of visual changes in code, which would be impossible to investigate with non-standard code. For example, these visual changes to the programming code could be the identifier style (CamelCase and under_score) or syntax highlighting [7], which do not impact the code execution.
There are three different main topics asked of the test subject. The most common topics are asking for a summary of the code provided (ten times) [1, 10, 38, 47, 49, 50, 57, 61–63, 78], asking for the output of the function (eight times) [4, 7, 9, 12, 55, 56, 58, 61] and bug finding and correction (seven times) [3, 5, 47, 55, 66, 78, 80]. Several studies [47, 55, 61, 78] employ multiple question topics as main and control questions because of the differences between understanding code and finding syntax bugs. There are also different kinds of question topics asked, such as finding specific variables in code [65] and filling in the blanks [18]. These question topics provide different approaches to help understanding the reading direction, code comprehension, and solution methods of programmers in different qualification categories. Several studies [6, 11, 19, 39] did not describe the specifics of their questions or answer types and at most summarized them as understanding or comprehension questions.

As for question types, there are two distinct main answering methods used. The most common method was to let the test subject answer the question in full. The test subjects of nine studies [3, 5, 47, 49, 50, 63, 66, 78, 80] answered verbally. In addition, five studies provided a text box [1, 4, 55, 58, 62]. Finally, six studies provided multiple-choice answers [8, 9, 27, 57, 61, 65]. These free answer types can be attributed to the minor restrictive nature of eye-tracking devices. The test subject can still move and behave normally in the research environment. These answer types also provide a large amount of insight into the comprehension process of the test subjects. On the other hand, multiple-choice restricts the answering process of the test subject so that they may infer the correct answer from the provided solutions. On the other side, answers can be significantly faster.
Lastly, eye-tracking studies research the error finding process with great interest, as eight studies look into either the finding of syntax errors [19], logical errors [5, 56, 66, 78, 80] or a combination of those [47, 55]. Errors often happen in programming. There are two distinct error types: syntax and logical. A statement is syntactically correct if it follows all rules and regulations of the given programming language. These errors are the easiest to find, as it stops the compiler from processing the code. Fortunately, many IDEs highlight the wrong syntax pieces when writing code so that they can be corrected early. Logical errors happen when the code syntax is correct, but the program's output is wrong or even missing. Unfortunately, these errors are often hard to find because there can be many possible reasons in many different parts of the code. Examples can be the wrong order of code in a function, returning before calculating the result, or having a wrong equation to calculate a distance used in other functions.

2.4.2 Focus of current studies using neuroimaging

The first code comprehension studies utilizing neuroimaging started in 2014 [35, 53, 69]. The main research topics focused on the activation of specific brain areas [14, 17, 22, 34, 37, 45, 69, 71] and brain activation strength [13, 15, 16, 24, 25, 28, 35, 36, 44, 51, 53, 81].
However, many studies also researched both areas [26, 42, 43, 59]. This data helps understand the neurological aspects of source code comprehension compared to natural language. Research is also being done to discern the differences between reading the words of source code and understanding them [14, 24, 42, 59, 60, 69, 71]. Different neuroimaging devices can record different specifics with different accuracies. Concisely, these different devices have different use-cases and constraints. As can be seen in figure 2.8, the different technologies are used with different fre-

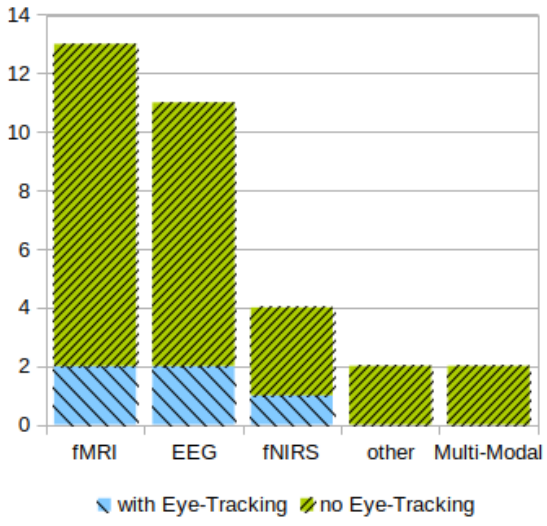
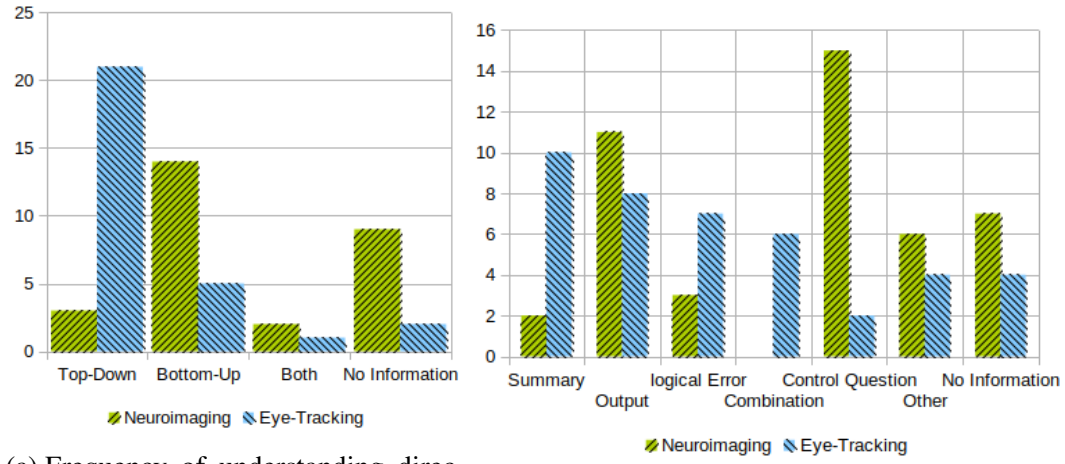


Figure 2.8: Frequency of technologies used in neuroimaging studies

quencies. Of the considered 28 studies using primarily neuroimaging methods, eleven used fMRI [14, 22, 26, 34, 37, 43, 48, 59, 60, 69, 71], nine EEG [17, 28, 30, 36, 42, 44, 45, 51, 81] and four fNIRS [24, 25, 35, 53]. Other methods are heart-rate monitors [16] and the recording of HRV (heart rate variability) and pupil dilation [15]. The two multi-modal studies [13, 33] employed multiple of these devices at once (fMRI, fNIRS, EEG, ECG). Additionally, Eyetracking was employed for studies using fMRI [22, 60], EEG [36, 44] and fNIRS [25]. Figure 2.8 is a visualization for the amount of used technologies.
These neuroimaging studies have primarily used Bottom-up comprehension with obfuscated code. Of the studies examined, 14 used bottom-up comprehension [17, 24–26, 28, 34, 36, 37, 42, 45, 48, 53, 59, 69, 81], and four used top-down comprehension [13, 24, 25, 43]. In addition, two studies [60, 71] used both bottom-up and top-down.
Eleven studies [17, 26, 28, 35, 37, 45, 48, 59, 60, 69, 71] asked for the output of the function to test the comprehension of the employed code snippet. Several studies used error location detection either as primary (semantic errors) [14, 22, 25] or as control questions (syntax errors) [24, 42, 59, 60, 69, 71]. Other studies employed a wide array of different code comprehension methods, such as summarizing the function [42], writing code [13, 43] or finding the value of a specific variable in a specific line. Some studies also did not show standard source code but used either a different question setup like intelligence tests [30] or asked to manipulate computer science data structures mentally [33]. Additionally, several studies [15, 16, 24, 44, 51, 53, 81] did not provide any information about the questions or other specifics like the used code snippets for their studies.
There was also a wide array of different control questions like the previously mentioned syntax error localization, pull request simulation [26], reading natural text [15, 25, 28, 37, 51], mental arithmetics [35], reading clean source code [14] or remembering fake code [48]. Due to the restrictions of the different neuroimaging devices (2.3.2), the answer types were mostly a short open answer [17, 22, 25, 35, 37, 59, 60, 69, 81] or answered via multiple-choice [26, 28, 45, 48, 71]. In addition, these studies employed many different answer methods, for example, writing down the values of variables at multiple checkpoints in the code snippet [53] or requiring the test subject to write code themselves [43]. Neuroimaging studies seem to procure valuable data for understanding source code comprehension. Many different devices, methods and survey techniques are used, and the research will likely continue for several years as new techniques and technologies are uncovered.

2.4.3 Differences between studies employing neuroimaging and eye-tracking

Even though there are studies for researching source code reading using neuroimaging or eye-tracking means, the approach significantly differs between the two techniques. On one side, eye-tracking devices are great for recording the reading order of source code. On the other side, it does not help much in understanding the neurological differences between reading source code or natural language. Furthermore, eye-tracking devices and methods are very inexpensive compared to neuroimaging devices, which can cost several hundred thousand euros and require



(a) Frequency of understanding directions

(b) Frequency of used tasks

Figure 2.9: Comparison of metrics used by code comprehension studies using Neuroimaging and Eye-Tracking methods

extensively trained personnel for proper use. Both research fields focus on different specifics in order to make better use of their advantages. For example, as seen in figure 2.9a, most neuroimaging studies use Bottom-Up comprehension induced by obfuscated code. Meanwhile, most Eye-tracking studies use unmodified source code not to change too much of the working environment. Eye-tracking studies also induce fewer restrictions on the test subjects. As a result, the researchers can employ different and open response options like asking for a complete summary of the code snippet or filling in a blank, which can be seen in figure 2.9b. On the other side, the answer possibilities of neuroimaging studies are severely reduced due to these restrictions. Therefore, these studies mainly employ short open answers and multiple-choice questions.

Many studies in both fields investigate the solving of software errors. Errors, also known as bugs, are programming mistakes that prevent the expected output. Errors are unavoidable in software development. Finding ways to prevent software problems from occurring would prevent many hours or days of bug-fixing. Time saved could be used to create innovative software [3]. Both fields investigate software errors with a different focus. Neuroimaging studies are primarily interested in finding syntactical errors as a control question. Few neuroimaging studies do focus on the bug-finding process of semantic errors. Several Eye-tracking studies try to understand how programmers find and correct bugs and focus more on logical errors. Neuroimaging studies also mainly focus on a single research point with control questions, presumeably because of the short research done.

Eye-tracking studies generally do not use a different task as control question, but several studies employ a combination of different tasks, all of which are primary research tasks [55, 61, 78]. Control questions used for neuroimaging studies were mainly syntax error location [24, 42, 59, 60, 69, 71], but also non programming related [15, 25, 28, 35, 37, 51] or other. One characteristic that both research fields have in common is the lack of attention given to the questions and answers used in the surveys. For example, many studies did not provide the used

code snippets [14–16, 44, 51, 57], and many studies did not provide specifics about questions asked [6, 7, 11, 15, 16, 19, 24, 39, 44, 51]. Finally, the examined survey papers may only be a sample size, but around one paper out of five did omit data from their survey setup.

3 Study Design

After providing an overview over the topic, the technical details of the study will be presented. This chapter aims to portray every vital facet of the experiment. It contains the study's preparation, participants and execution procedure to provide the best possible data set.

3.1 Objective

This study aims to understand whether there are differences in answer behavior for code comprehension studies when using distinct question types. Specifically, three distinct question types are chosen to be compared regarding answering time and correctness.

3.2 Experimental Material

3.2.1 Question Types

The three chosen question types are:

1. Was ist die Ausgabe dieser Funktion?
   (What is the Output of the function?)
2. In welcher Zeile existiert zuerst die größte ganzzahlige Variable?
   (What is the first line where an integer variable has the highest value?)
3. Welche Beschreibung passt am besten zu dieser Funktion?
   (What summary fits the function best?)

There are several reasons why those three question types were used. Due to the nature of this survey analysing programming code comprehension utilising both eye-tracking (section 2.4.1) as well as neuroimaging (section 2.4.2) means, question types one and three were chosen because they amounted to the majority of in their respective fields. Both question types were used in more than half of the investigated code comprehension studies. Even though bug finding and correction was the third most used question type in previous surveys, it cannot be used in this survey, as the code would need to be adjusted. Any code changes would introduce another study variable, so another question needed to be found.
The second question (What is the first line where an integer variable has the highest value?) has

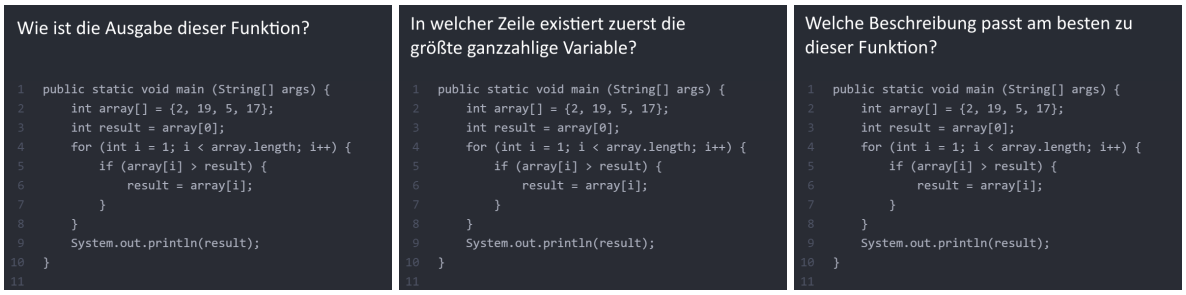
#	Topic	LoC	#	Topic	LoC
1	Factorial calculation	9	6	Exponentiation	9
2	Integer to binary	12	7	Reversing a string	8
3	Finding largest integer in a List	10	8	Median of a List	11
4	Cross sum calculation	9	9	Reversing a list	11
5	Prime number test	11	10	Swapping 2 integers	9

Table 3.1: Code Snippets used in the main experiment in order (#) and length (LoC)

not been used in this way before. A similar question (Example: What is the value of variable x at the sixth line in the second loop?) has been used by Ishida and Uwano [36]. This question needed to be changed to a question that fits to every code snippet without further modification. The variation of the task used here provides two benefits. This version can be used for any code snippet employing integer variables and it would not introduce additional study variables. This question type sparks interest because it does not require solving the function for the output, nor does it prompt any recognition from the reader as this is not a task the reader is likely familiar with. Additionally, the reader needs to keep multiple values in working memory to compare the highest value of each line. The third question type went through multiple iterations because several borderline cases emerged when testing where the correct answer was unclear.
Every code snippet is paired with every question type, so there are 30 different combinations of tasks.

3.2.2 Code Snippets

Ten source code snippets were selected. To reduce the amount of further variables, those used code snippets needed to comply with several restrictions. They needed to be short and of similar length. The complexity and difficulty also needed to be similar to compare the collected data. For this reason, ten different snippets, both linear and non-linear, were used from a collection of code snippets for programming comprehension studies. This list is curated by the computer engineering faculty [68] of the Technical University Chemnitz [76].
The chosen snippets all used a different single function with one concept and had a length of eight to twelve lines of code. These snippets include ”Finding the largest integer in a list”, ”sorting an array in reverse”, ”outputting the median”, or swapping two numbers. The code snippets can be found on the Github repository of this study [46]. A full list of the questions can be seen in table 3.1. For explanatory purposes, three more snippets were created. Those explanation snippets had a very low complexity, so the test subjects can focus on learning to use the tool and explain the sequence of the experiment.
All code snippets used obfuscated code with no syntax highlighting and non-descriptive variable and function names with light grey font on a near black background, also known as dark mods. As mentioned in section 2.1.2, understanding obfuscated code requires a higher mental load from the test subject.



(a) Output (b) Which line (c) Summary

Figure 3.1: Code snippet #3 with every question type as used in the survey

3.3 Experimental Units

Test subjects for this study were 16 people aged 22 to 29, with a median of 24 and an average of 24,375. All people had a background in computer sciences, and most(14) studied at university when the survey was conducted and were programming for 3 to 9 years actively. The target group was late bachelor's students to early master's students. The survey was advertised with multiple methods. Additionally, before conducting the study, several possibly eligible people were sent a link to a short preliminary survey containing only the first part of the main study. This short survey was sent to find several people for a homogenous group of students between 20 and 30 with a computer science background because reading and understanding source code was the topic. Ten test subjects were found by utilising this method.
In addition, on the first day of the experiment time window, an email was sent with the email contribution list of the computer science faculty of the TUC. Further, there were multiple advertisement messages in multiple group chats, which contained a copied message containing the same information and wording as the email. These messages contained the survey topic, the expected duration and the link to the study. It also contained the information that every participant can win a 20€ voucher to a shop of their choosing. All messages were in german because the study was designed in German, so unknown variables are minimised. In total, 35 people clicked the link to the study, while only 16 completed the survey. Several reasons for the low numbers will be discussed in threats to validity in section 4.3.1.
The survey was open for answers for 14 days, from 2021-07-20 to 2021-08-02. The figure 3.3 shows the rate of responses over time. The bump at 2021-07-27 demonstrates the reaction to another wave of advertisements and reminders in group chats. The test subjects were randomly assigned into one of three groups during the main part of the study. These groups were sized 6, 6 and 4 due to the 17th test subject not completing the survey after already being assigned to the third group.

3.4 Tasks

The survey was conducted using SoSci Survey. The tasks provided to the test subjects are split into two main parts. The first part is a test created by Siegmund et al. [70] to self-assess their programming experience and record information about their person, studies and professional programming experience. The personal data collected includes age, gender and colour vision deficiency. Collected information about their studies included their major, year of enrollment, intended degree, and the number of programming courses attended, devised by Siegmund et al [70]. The second group of tasks is the primary purpose of this survey.
Each test subject sees a blurred image and multiple answers in plain text at the bottom. When the user clicks on the blurred image, the user will deblur a rectangle-shaped area around the cursor to read what is underneath. For every question, the test subject is presented with a blurred image at the top and plain text multiple choice answers at the bottom, an example is visible in depiction 3.2. The test subjects need to focus parts of the image to understand the question and the code presented to answer the questions correctly. Finally, they can choose an answer of the four given answer options. Every test subject sees ten code snippets in the same order, but each group has a distinct order of question types. For example, question 3 will show the same code snippet for every test subject, but each group will have a different question, as seen in Figures 3.1a, 3.1b and 3.1c.

Figure 3.2: View of the task for the participant

3.5 Hypotheses, Parameters, and Variables

Due to the aim to explore whether different question types have a different impact on the solving of code comprehension studies, these three hypotheses are derived:

- $H0_1$: The choice of question types will have no significant impact on the solving time
- $H1_1$: The choice of question types will have a significant impact on the solving time
- $H0_2$: The choice of question type will have no significant impact on the correctness
- $H1_2$: The choice of question type will have a significant impact on the correctness

With these hypotheses in place, the variables can be derived. The leading independent variables

are the presented question type and the task choice. Therefore, the dependent variables are the correctness of each question and the solving time. Suppose one of these dependent variables shows a significant difference between the three question types. In that case, the choice of question type may substantially impact the results of surveys conducted for code comprehension studies.

3.6 Experiment Design

This experiment follows a within-subject design where every test participant is subjected to every question type evenly and sees every code snippet. The test subjects were evenly randomized into one of three groups, determining which tasks were shown. Before the study was conducted, it was unclear what exact data were needed for analysis. Therefore, many potential data points were included. These included the personal data of the participants, the answers to the primary survey and the data collected by REyeker [72] such as fixations and solving time.

3.7 Procedure

The survey procedure for the participants from receiving the survey link to successful completion will be presented in this section. Survey participants will open the survey by clicking the link they had obtained via email or chat group for the duration of the study from the 20th of July to the 2nd of August 2021. When the study is opened, they are presented with general information. This information included convincing reasons why it is helpful to participate in such studies and the possibility of winning a gift card. Additionally, it contained the expected solving time, a short explanation about the content of the study, methods to deal with possible software bugs and a reminder that their data will be collected and saved only for the study. This information block was structured to improve readability, and essential words or phrases were also formatted in bold or cursive. If they agree to the data collection mentioned earlier, they will continue and arrive at the first part of the study.
The first part, as mentioned in section 3.4, collects personal information about the test participant, in detail age, gender and colour vision deficiency. Afterwards, the test subjects need to answer questions about their studies and evaluate their programming expertise compared to peers and professionals. Subsequently, they will be presented with another explanatory text about the second part of the study. This information contains a short explanation about REyeker [72], how to use it and a mention that there will follow examples directly after.
These tutorials explain how to solve the three question types and the nature of the expected answer so that every test subject can answer the survey to the best of their knowledge. Every tutorial question is in the same style as the main questions and requires the user to deblur the code snippet to answer the question. After each question type, there is an explanation of why the specific answer was correct, and the code snippet is shown without blur. When the user completes the tutorial, they get an explanatory text informing them about the upcoming central

part of the study. They are also reminded to get hydrated, keep focus, and not get distracted until the end of the central part. In the central part, every survey participant is shown ten questions as detailed in section 3.4. After completing this part, there is a text of thanks and the possibility to enter the email address to enter the gift card giveaway, to receive information about the study's findings or both.

3.8 Execution

3.8.1 First evaluation

The first test subject wants to remain anonymous, but they have prior programming experience and write software for a living. The participant does not belong to the correct target group as they have graduated already, but they were perfect for a first trial run, as they could clearly articulate their thoughts. As mentioned in the survey overview (section 3.4), the survey was structured into two main parts. The first part is the general questionnaire to gain anonymous data about the person and self-evaluate their programming expertise. The second part is the central selling point of this thesis, the online eye-tracking component. The survey testing was done in the home of the test subject on their private personal computer. This environment was perfect because every respondent will be choosing a setting where they feel comfortable. Additionally, the test subject could be surveyed without additional surveying software or hardware because both test subject and observer were in the same room, so notes could be taken quickly. The test subject was also asked to verbalize anything which seemed interesting, problematic, confusing or questionable, to which they happily obliged.
As mentioned in section 3.3, the survey was created in german because it was conducted on students of the Chemnitz University of Technology [76]. The trial run took about 40 minutes, and ten were spent on the general part and 30 on the eye-tracking part. The trial test subject gave much valid and constructive criticism and found several minor grammatical and consistency errors. For some personal questions in the first part, they did not see the focus of the question, so those questions were changed to highlight the critical information. Additionally, because they were not in the target group, they were missing answers that differed from the target audience. One example of these occurrences was the option "already graduated" in the question for the degree currently sought. The second and central part of the survey starts with a text welcoming the participant and explaining the upcoming methods. One thing that needed to be made more apparent was the eye-tracking method. It was unclear whether an eye-tracking device or a camera was needed to proceed. The central part starts with a tutorial on the three different question types mentioned in section 3.2.1.
All question types had some introduction text explaining the question and usage tips for the eye-tracker. The question asking for the line of code with the highest single value was the most complicated, so an explanation for the question in advance was added. This explanation confused the test subject, as the two other questions did not explain the solution with a deblurred image afterwards. These ambiguities were corrected afterwards. Additionally, the testee was

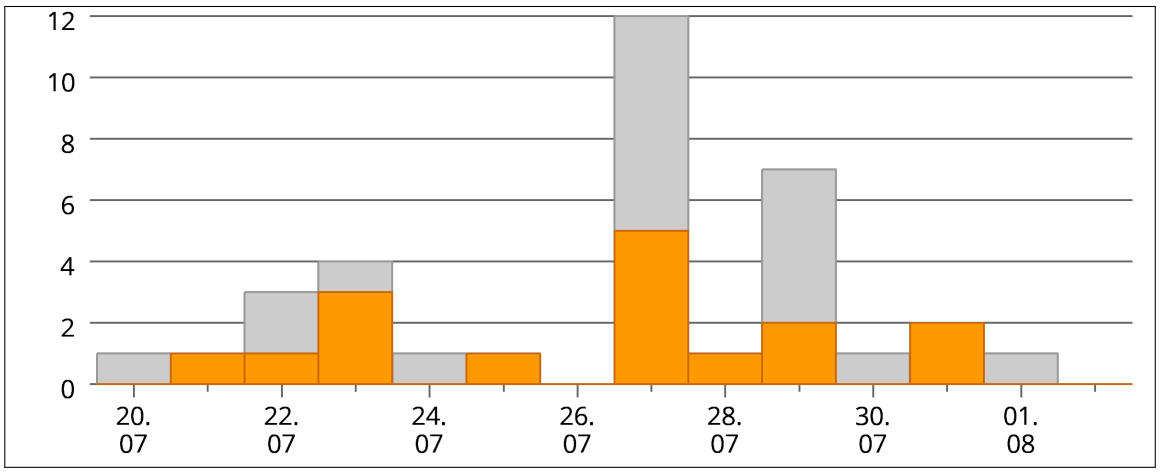


Figure 3.3: Answer statistics of the survey

confused why there was no help text for the non-tutorial part of the survey. Therefore, clarity has been reintroduced by adding an explanatory paragraph in the text just before the central part. Another thing that stood out in the trial run was that the blurred images did not load. This problem did not occur before. Nevertheless, a workaround has been found to this obstacle: to change browser tabs and change back again. Then the images would load. This bug only occurs with the firefox web browser, so a help paragraph was added for the tutorial questions to complete the test. The test subject answered afterwards that the new approach for eye-tracking was a fun time and offered another test run.

3.8.2 Main Experiment run

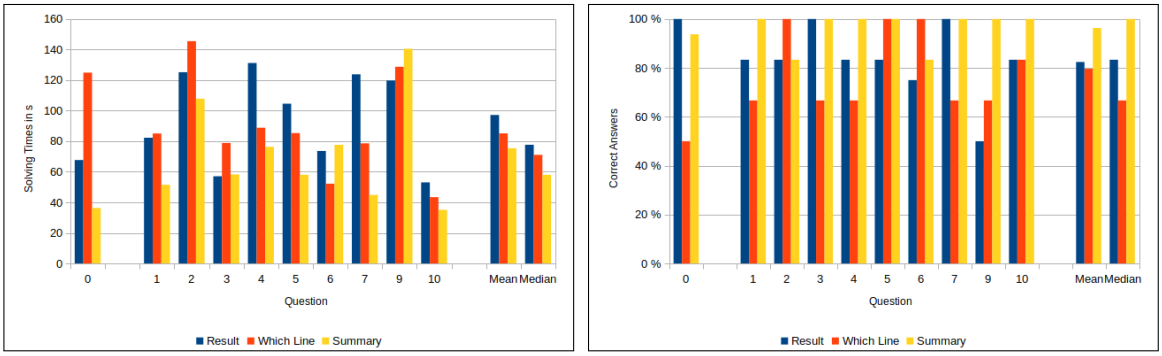
The link to the survey was sent to the mailing list and previously voluntarily registered participants on July 20, 2021. On that day, the link got also sent to several group chats with promising members. The participating students were free to complete the survey at a time of their choosing.
The return statistics can be seen in the figure 3.3. Orange are successful test completions, and grey are aborted tests or page views. The increase in responses on July 27 is due to the additional advertising on that day. However, during the experiment run, one question was found to have a typo, which made the question unanswerable due to missing visuals of the code snippets. This problem was corrected during the survey run to have fewer survey abortions, but data from task 8 will not be considered in the data evaluation.
During the survey, the participants could ask questions if they had any problems. Such responses helped find the error mentioned above in task 8. Generally, the survey was received well and had near to no problems. Additionally, some respondents compared the survey to a game and tried to complete the REyeker park with as few as possible clicks. There were no other inconsistencies or problems founds.

4 Analysis and Results

This chapter aims to present and analyse the data recorded by the conducted survey.

4.1 Overview of collected Data

When conceptualising this research study, many possible ways were considered to achieve the goal of comparing different question types. Due to the many possible hypotheses from the simple statement of ”The choice of question may impact the result”, many different data points were collected to provide a broad set of analysis approaches. This data included personal data from each participant, as described in sections 3.3 and 3.4, as well as answers for the 30 different tasks, general survey statistics and REyeker data points consisting of click-coordinates and click-times in milliseconds. Due to the low amount of participants and the nature of the data, several techniques based on normal distribution cannot be conducted. Outliers can skew the results of statistical analyses, so they are typically removed from the data set. An outlier is a value much higher or lower than most other values in a data set.
First, the data needs to be defined to be analysed further. When trying to define answering times, multiple questions arise.
When does the participant start each task? Does ”being on the task page” already count to solving the task? Does clicking the ”continue” button mean they finished the task? Did the participant finish the task before and go on a walk before clicking continue?
Due to the usage of REyeker and the survey design, there are several possibilities for interacting with those times. In detail, REyeker records the times for every click so that all times can be analysed. For this analysis, the first and last click will be considered for measuring the answering time. SoSci Survey also saves completion times for every task so that these times can function as a completion point for each task. Due to the survey design, the participant will not see the question or the code snippet clearly when starting the task, so it is impossible to start answering the task the second the survey page is shown. For this reason, the first click will count as starting the answering process. This approach altered the solving times drastically. In several cases, the participant waited more than a minute before clicking on the REyeker field. Generally, participants took between one and thirty seconds before clicking, with an average of around 4.5 seconds after discarding significant outliers.
Given the click data recorded by REyeker, one case was found where the participant did not interact with the code snippet. For simplicity, clicking the ”continue” button will be equated to completing the task because the test participant may ponder an indefinite amount of time



(a) Average of solving times

(b) Average of correct answers

Figure 4.1: Comparison of (a) solving times and (b) correctness per task

over the question before deciding on an answer. Additionally, in the information text before the main question set, the participant was urged to complete the central part without any breaks as mentioned in section 3.7.

4.2 Analysis and Results

4.2.1 Impact of question types on solving times

Due to the hypotheses of the experiment, solving times and answer correctness will be compared between the question types. An overview of the average solving times and the average correct answers per task can be seen in figures 4.1. This data was previously cleaned of outliers using the robust test for multiple outliers with a modified Z score $\geq$ 3.5 devised by Iglewicz and Hoaglin [31]. However, due to the uneven group numbers, the total numbers cannot be compared directly but by proportions. Additionally, recorded times often skew heavily towards 0, as shown in figure 4.2. This phenomenon is often seen when recording finishing times in general, as the minimal time to complete a task has a lower bound but no upper limit. More examples of skewed times are competitions where people try to race to complete a goal faster than others, as seen in video game speedrunning (figure 4.3a) and running marathons (figure 4.3b).

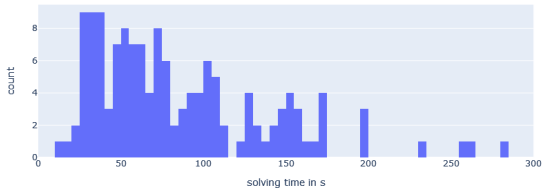
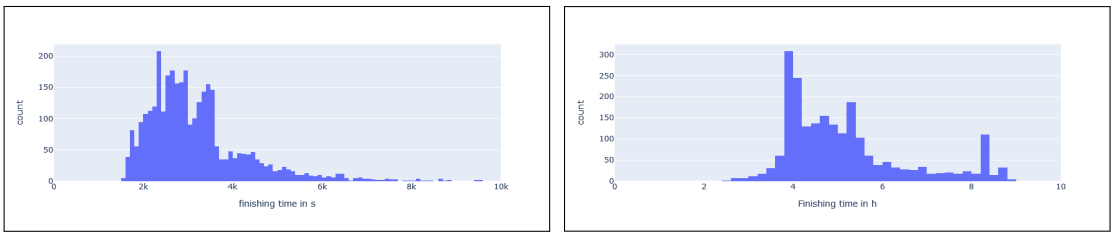


Figure 4.2: Histogram of all solving times without dividing into question types

Due to the upcoming frequent occurrence of the three question types, they will often be shortened to question types 1 (”What is the Output of the function?”), 2 (”What is the first line where an integer variable has the highest value?”) and 3 (”What summary fits the function best?”). When Comparing average and mean times for every question type, the tutorial question times (question 0) have been excluded from calculations. The same exclusion was done for compar-



(a) Celeste Any% speedrunning (b) marathon times of Rennsteiglauf 2018

Figure 4.3: Right skewed histograms of recorded times

Question type	count N	mean $\bar{x}$	median	standard deviation s
1 - Output	46	97.1817 s	77.66 s	55.9735
2 - Which Line	50	85.1316 s	71.05 s	52.4860
3 - Summary	44	75.4186 s	59.42 s	53.2184

Table 4.1: Relevant statistics for solving times

ing correctness. Relevant statistical values can be seen in table 4.1. By comparing mean and median solving times, question type 1 took the longest, 2 was the second longest and number 3 was the shortest. Further, the solving times seem to vary for each task but generally follow the results shown by average and mean calculations. Discrepancies between average and mean can be attributed to the right skew of all answer times, as seen in figure 4.2.
Comparing this raw data for the hypotheses would raise the suspicion that the question types impact the solving time of code comprehension studies. This conclusion is based on the fact that no other variable influences the recorded data, and the hypotheses are independent. When taking a closer look at solving times and correct answers, question type 3 requires extra attention. It shows the highest correctness and the lowest time to complete, suggesting that this question was easier to answer. This statement is not a problem at first, but when comparing the answer possibilities of all three question types, this one is the only one with words as options. When considering the experiment setup as shown in figure 3.2, the answer options are visible before looking at the question and the code. This fact is also correct for question types 1 and 2, but whole words convey meaning far better than alphabetically sorted numbers without a context. The test participants are possibly deducing parts of the solution from reading the answer option, while this approach is not possible for question types 1 and 2.
When approaching the hypotheses statistically, several problems occur. The main problem is that no normal distribution can be detected using the Shapiro-Wilk test for each set of solving times or for all solving times grouped together. Due to many ways to test for significance requiring the data to have a normal distribution, these tests cannot be used. Examples of these tests are the t-test and ANOVA (**an**alysis **o**f **va**riance). Important statistical values are the count N of collected values x_i-x_n, mean $\bar{x}$ of the group and the sample standard deviation s, usually calculated with $s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$. Inserting the solving times for each group in an ANOVA yields no detection of significant difference with a p-value of 0.16187, which is greater than a normally used value of 0.05 to compare against. These results point to $H0_1$ being true and

Question type	count N	mean $\bar{x}$	median	standard deviation s
1 - Output	9	0.8240	0.8333	0.1469
2 - Which Line	9	0.7962	0.6666	0.1620
3 - Summary	9	0.9629	1	0.0735

Table 4.2: Relevant statistics for correctness

$H1_1$ being subsequently wrong. However, this conclusion cannot be drawn fully, as no normal distribution could be found.

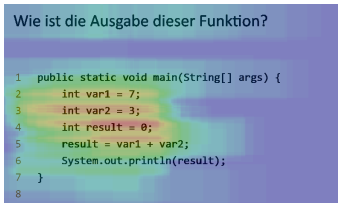
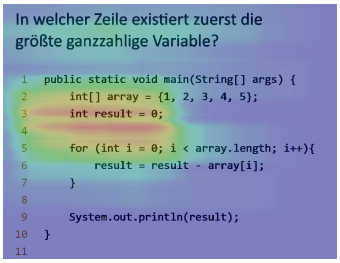
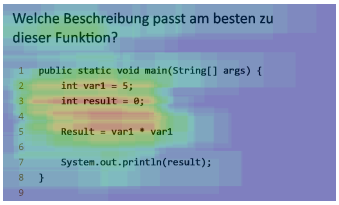
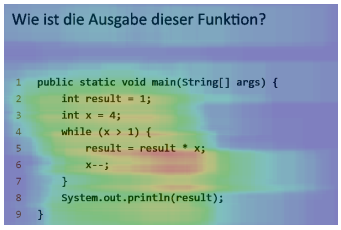
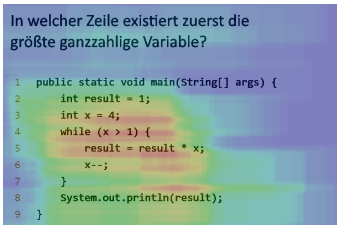
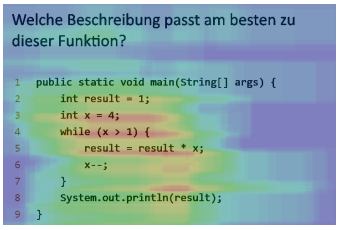
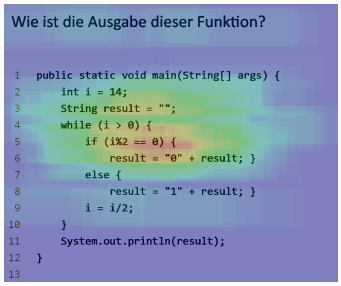
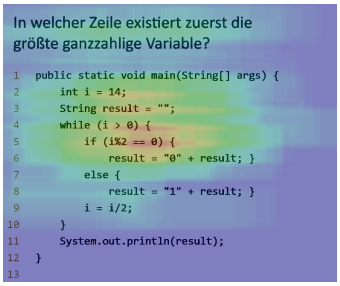
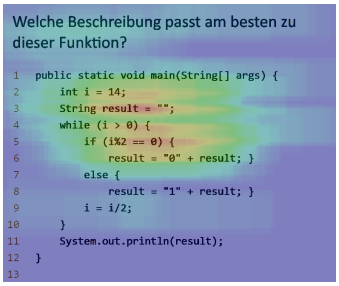
4.2.2 Impact of question types on correctness

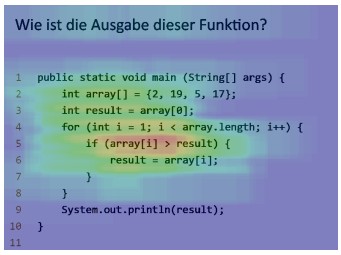
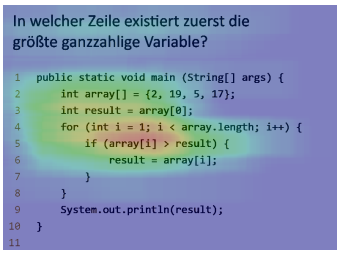
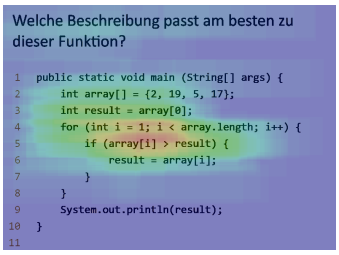
Similar to the comparison of solving times, a definition of correctness is needed before comparing it. Participants provide one answer per task processed. Answers can be either right(1) or wrong(0), depending on the chosen option in the multiple choice array. When grouping answers together, they can be denoted with $A_{3\text{-}2} = 4/6$ when four of six answers in the third question with the second question type ('Which Line') are answered correctly. They can also describe the number of correct answers of a participant, for example, $A_{p7} = 6/9$, which means that participant seven has answered six tasks of a maximum of nine correctly. Figure 4.1b shows the averages of correct answers per task, grouped for the different code snippets and colour coded by question type.
In total, 16 participants provided 144 answers, $A_{all} = 123/144$. Question type 1 provided $A_{\text{all-1}} = 40/48$ correct answers, question type 2 $A_{\text{all-2}} = 39/50$ and question type 3 $A_{\text{all-3}} = 44/46$. When looking deeper into the participant answer statistics, every participant had between $A_{\text{P-lowest}} = 6/9$ and $A_{\text{P-highest}} = 9/9$ correct answers, so no outliers were discarded.
For statistical comparisons, each task's average correctness will provide insights. Nevertheless, due to the small sample size, the Shapiro-Wilk test similarly showed a departure from normality. However, with more data points, a normal distribution could be possible. Therefore, ANOVA will be used on an ill-fitting dataset to show possible significance. For this case, the result is significant at p ¡ 0.05. The f-ratio value is 4.04385. The p-value is 0.030653.
Tukey's HSD (honestly significant difference) [77] will be used to detect the significant difference between which groups. Tukey's HSD procedure facilitates pairwise comparisons within ANOVA data. The F statistic shows whether there is an overall difference between the sample means. Tukey's HSD test allows determining which of the various pairs of means has a significant difference. This procedure shows a significant difference between question types 2 and 3. This result proposes the argument that hypothesis $H1_(2)$ seems correct, but without normal distributed data, this cannot be entirely deduced.

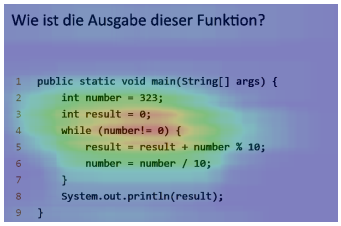
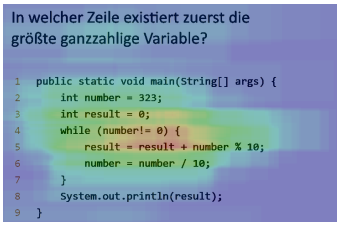
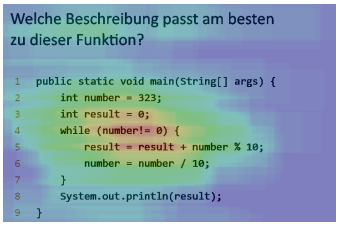
4.2.3 Other analysis possibilities

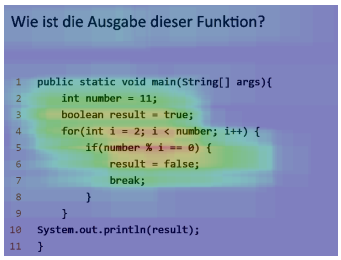
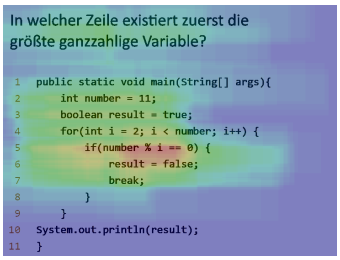
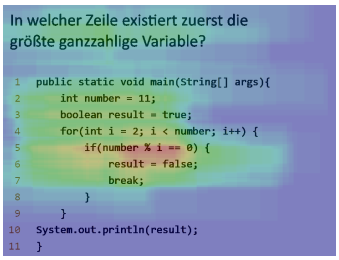
While the hypothesises focus on the differences in solving time and correctness per question type, there are many other methods of comparing question type in a code comprehension environment. For example, data recorded by REyeker can be used for further analysis by creating attention heatmaps or sequence diagrams. An eye-tracking heatmap is a graphical representation of where a participant is looking while viewing a particular code snippet.
The resulting image shows which areas of the task are getting the most attention. High focus areas are red, while low attention areas are blue. This attention data can be added and averaged for each participant of a single task. The resulting heatmaps are visible in table 4.3. Due to the focus area or REyeker being set too large and the low amount of participants, the heatmaps differ only slightly and mainly focus on the parts with the highest difficulty.
Different heatmaps are still visible, for example, Question 9. The underlying task images are the images used in the experiment but inverted and brightened for easier reading. This analysis procedure was also considered when creating the survey, but this analysis would require qualitative and not quantitative comparisons.

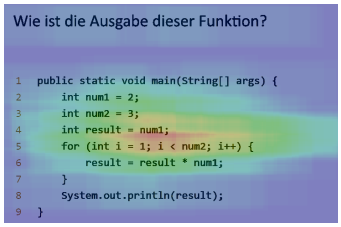
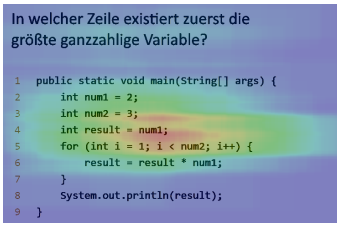
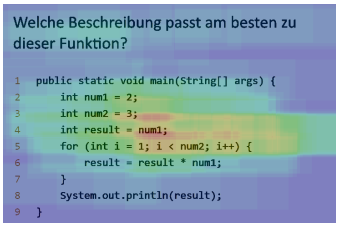
Table 4.3: Average heatmaps for every task

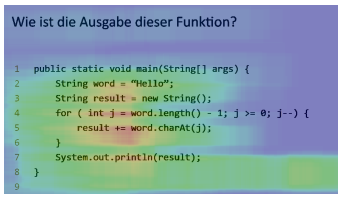
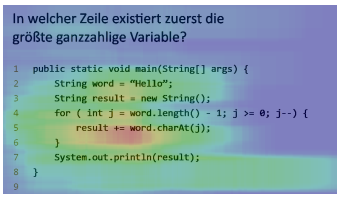
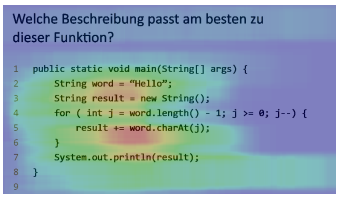
	Output	Which line	Summary
0			
1			
2			

3	 Wie ist die Ausgabe dieser Funktion?
	public static void main (String[] args) {
	int array[] = {2, 19, 5, 17};
	int result = array[0];
	for (int i = 1; i < array.length; i++) {
	if (array[i] > result) {
	result = array[i];
	}
	}
	System.out.println(result);
	}
		 In welcher Zeile existiert zuerst die größte ganzzahlige Variable?
		public static void main (String[] args) {
		int array[] = {2, 19, 5, 17};
		int result = array[0];
		for (int i = 1; i < array.length; i++) {
		if (array[i] > result) {
		result = array[i];
		}
		}
		System.out.println(result);
		}
			 Welche Beschreibung passt am besten zu dieser Funktion?
			public static void main (String[] args) {
			int array[] = {2, 19, 5, 17};
			int result = array[0];
			for (int i = 1; i < array.length; i++) {
			if (array[i] > result) {
			result = array[i];
			}
			}
			System.out.println(result);
			}

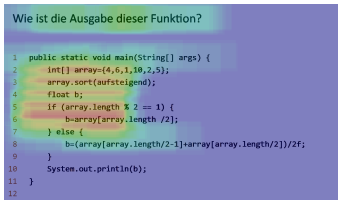
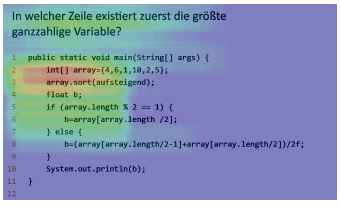
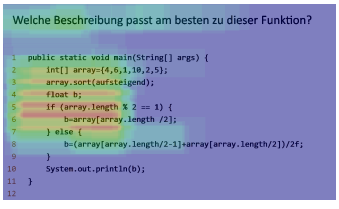
4	 Wie ist die Ausgabe dieser Funktion?
	public static void main(String[] args) {
	int number = 323;
	int result = 0;
	while (number!= 0) {
	result = result + number % 10;
	number = number / 10;
	}
	System.out.println(result);
	}
		 In welcher Zeile existiert zuerst die größte ganzzahlige Variable?
		public static void main(String[] args) {
		int number = 323;
		int result = 0;
		while (number!= 0) {
		result = result + number % 10;
		number = number / 10;
		}
		System.out.println(result);
		}
			 Welche Beschreibung passt am besten zu dieser Funktion?
			public static void main(String[] args) {
			int number = 323;
			int result = 0;
			while (number!= 0) {
			result = result + number % 10;
			number = number / 10;
			}
			System.out.println(result);
			}

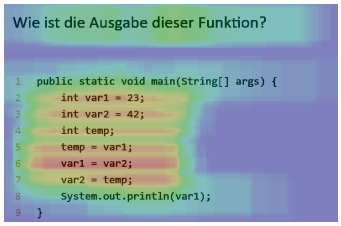
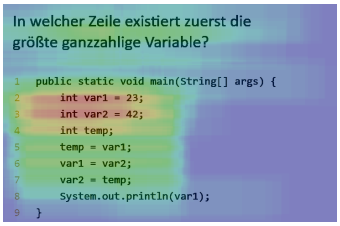
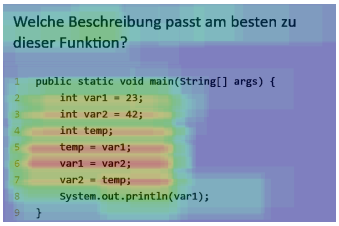
5	 Wie ist die Ausgabe dieser Funktion?
	public static void main(String[] args){
	int number = 11;
	boolean result = true;
	for(int i = 2; i < number; i++) {
	if(number % i == 0) {
	result = false;
	break;
	}
	}
	System.out.println(result);
	}
		 In welcher Zeile existiert zuerst die größte ganzzahlige Variable?
		public static void main(String[] args){
		int number = 11;
		boolean result = true;
		for(int i = 2; i < number; i++) {
		if(number % i == 0) {
		result = false;
		break;
		}
		}
		System.out.println(result);
		}
			 In welcher Zeile existiert zuerst die größte ganzzahlige Variable?
			public static void main(String[] args){
			int number = 11;
			boolean result = true;
			for(int i = 2; i < number; i++) {
			if(number % i == 0) {
			result = false;
			break;
			}
			}
			System.out.println(result);
			}

6	 Wie ist die Ausgabe dieser Funktion?
	public static void main(String[] args) {
	int num1 = 2;
	int num2 = 3;
	int result = num1;
	for (int i = 1; i < num2; i++) {
	result = result * num1;
	}
	System.out.println(result);
	}
		 In welcher Zeile existiert zuerst die größte ganzzahlige Variable?
		public static void main(String[] args) {
		int num1 = 2;
		int num2 = 3;
		int result = num1;
		for (int i = 1; i < num2; i++) {
		result = result * num1;
		}
		System.out.println(result);
		}
			 Welche Beschreibung passt am besten zu dieser Funktion?
			public static void main(String[] args) {
			int num1 = 2;
			int num2 = 3;
			int result = num1;
			for (int i = 1; i < num2; i++) {
			result = result * num1;
			}
			System.out.println(result);
			}

7	 Wie ist die Ausgabe dieser Funktion?
	public static void main(String[] args) {
	String word = "Hello";
	String result = new String();
	for (int j = word.length() - 1; j >= 0; j--) {
	result += word.charAt(j);
	}
	System.out.println(result);
	}
		 In welcher Zeile existiert zuerst die größte ganzzahlige Variable?
		public static void main(String[] args) {
		String word = "Hello";
		String result = new String();
		for (int j = word.length() - 1; j >= 0; j--) {
		result += word.charAt(j);
		}
		System.out.println(result);
		}
			 Welche Beschreibung passt am besten zu dieser Funktion?
			public static void main(String[] args) {
			String word = "Hello";
			String result = new String();
			for (int j = word.length() - 1; j >= 0; j--) {
			result += word.charAt(j);
			}
			System.out.println(result);
			}

8	—-	—-	—-

9	 Wie ist die Ausgabe dieser Funktion?
	public static void main(String[] args) {
	int[] array={4,6,1,10,2,5};
	array.sort(aufsteigend);
	float b;
	if (array.length % 2 == 1) {
	b=array[array.length /2];
	} else {
	b=(array[array.length/2-1]+array[array.length/2])/2f;
	}
	System.out.println(b);
	}
		 In welcher Zeile existiert zuerst die größte ganzzahlige Variable?
		public static void main(String[] args) {
		int[] array={4,6,1,10,2,5};
		array.sort(aufsteigend);
		float b;
		if (array.length % 2 == 1) {
		b=array[array.length /2];
		} else {
		b=(array[array.length/2-1]+array[array.length/2])/2f;
		}
		System.out.println(b);
		}
			 Welche Beschreibung passt am besten zu dieser Funktion?
			public static void main(String[] args) {
			int[] array={4,6,1,10,2,5};
			array.sort(aufsteigend);
			float b;
			if (array.length % 2 == 1) {
			b=array[array.length /2];
			} else {
			b=(array[array.length/2-1]+array[array.length/2])/2f;
			}
			System.out.println(b);
			}

10	 Wie ist die Ausgabe dieser Funktion?
	public static void main(String[] args) {
	int var1 = 23;
	int var2 = 42;
	int temp;
	temp = var1;
	var1 = var2;
	var2 = temp;
	System.out.println(var1);
	}
		 In welcher Zeile existiert zuerst die größte ganzzahlige Variable?
		public static void main(String[] args) {
		int var1 = 23;
		int var2 = 42;
		int temp;
		temp = var1;
		var1 = var2;
		var2 = temp;
		System.out.println(var1);
		}
			 Welche Beschreibung passt am besten zu dieser Funktion?
			public static void main(String[] args) {
			int var1 = 23;
			int var2 = 42;
			int temp;
			temp = var1;
			var1 = var2;
			var2 = temp;
			System.out.println(var1);
			}

4.3 Evaluation

As written in sections 4.2.1 and 4.2.2, the hypotheses can currently neither be confirmed nor denied. The main question was whether the choice of question type impacts the execution of code comprehension studies. For this reason, solving time and correctness were chosen as metrics for determining significant differences. Additionally, many more metrics were recorded without meaning to use them if a chance presented itself. Additionally, question type and answer possibility choices were potentially not the most informative.
An open answer strategy may have reduced the effects of deducing the solution by seeing the answer options and levelling the field for all question types. On the other side, open answer procedures may require much work to deduce correctness afterwards if the number of experiment participants is high. Therefore, the multiple-choice approach was chosen for straightforward quantitative analysis. In hindsight, many points of this study should have been changed.

4.3.1 threats to validity

Even though the hypotheses cannot be supported or denied, threats to validity will be discussed.

Internal Validity

The internal validity of an experiment is essential to demonstrate a causal link between two variables. Without strong internal validity, the conclusions of a causal relationship may be called into question.
There are several types of internal threats to validity, including Confounding variables. These variables are not controlled for in the study and can potentially skew the results. For example, in this experiment, all test participants had similar computer science backgrounds but different Java or programming experiences which could heavily impact the program comprehension part of the study. Additionally, due to the online nature of the study, the test environment could not be controlled. Therefore, daytime, hydration, exhaustion levels, monitor size, chair comfort, and many more variables can impact the results. The code snippets were chosen from previously performed code comprehension studies to choose a set of similarly difficult samples. Additionally, the questions were not randomized in order, which may produce a learning effect. This learning effect can be seen when comparing solving times for question types 1 and 2. Question type 2 took, on average, longer than question type 1, but this pattern changed during the experiment. The selection of test subjects was also heavily biased. For example, only german speaking students from the Chemnitz University of Technology took the test. Additionally, most test subjects were personally asked to participate by the survey creator. The study employed a dark background with light colored code which many programmers are used to, and no test participant commented on the choice of dark mode, but the choice could nontheless impact the result.
However, choosing REyeker as an approximation of visual attention helps to bring reliable data

if the size of the focus area is chosen correctly.

External Validity

The results of this study used only a limited set of simple iterative programs. Since experienced programmers with advanced skills are recruited, it is difficult to generalize these results to other situations, such as B. complex types of programs, novice programmers or expert programmers, because the results are only applicable to similar situations. However, the setting is a valuable representation because the choice of question type is not challenged beforehand.

4.3.2 lessons learned

There are many lessons learned during the creation of this bachelor's thesis. One of the main lessons learned is the need for a shorter survey, as many possible test subjects disregarded the test early. The survey also presents too much text and a whole sub-survey that is not part of the central survey. Reducing text and slimming the survey would have been severely needed. Additionally, more intuitive test questions and survey procedures should be desired to avoid several paragraphs explaining the question. As the return statistics have shown, advertising the study brings in more participants. There should have been more and better advertising at the Chemnitz University of Technology and perhaps at different universities. It is also not a good idea to conduct the survey during the exam period when many students have no time. Additionally, the hypotheses should be defined earlier and more clearly to help create the survey with the hypotheses in mind. The hypotheses should also be chosen with analysis procedures in mind to prevent situations where the data cannot be normally distributed. The type of answer possibilities should also be reconsidered, as it should be possible to create regex terms to determine correct answers automatically. A manual pass of all answers should also help weed out false assessments.

5 Summary and further research

This thesis outlines code comprehension studies done with neuroimaging and eye-tracking technologies. It shows requirements, constraints and research areas, and an overview of used question and answer types used in those studies. The result of the focus analysis showed that the question types used in these kinds of studies were not important for the survey creators. Afterwards, a survey was developed using SoSci-survey and a remote eye-tracking technology called REyeker to analyse the impact of the choice of question on the solving time and correctness. The survey was conducted with 16 participants. Unfortunately, analysing the data showed that the samples did not have a normal distribution and therefore could not be analysed statistically. Even so, it cannot be ruled out that the question type may impact the solving of code comprehension studies, which could be analysed further with a different study and a better choice of hypotheses.
In future studies, survey designers should pay more attention to the choice of questions, as their choice of question type may have a non-negligible impact on solving these.

Bibliography

[1] N. J. Abid, B. Sharif, N. Dragan, H. Alrasheed, and J. I. Maletic. Developer reading behavior while summarizing java methods: Size and context matters. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 384–395. IEEE, 2019.

[2] A. V. Aho. *Compilers: Principles, Techniques and Tools (for VTU)*. Pearson Education India, 2007.

[3] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin. Do developers read compiler error messages? In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 575–585. IEEE, 2017.

[4] J. Bauer, J. Siegmund, N. Peitek, J. C. Hofmeister, and S. Apel. Indentation: simply a matter of style or support for program comprehension? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 154–164. IEEE, 2019.

[5] R. Bednarik. Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *International Journal of Human-Computer Studies*, 70(2):143–155, 2012.

[6] R. Bednarik and M. Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 125–132, 2006.

[7] T. Beelders and J.-P. du Plessis. The influence of syntax highlighting on scanning and reading behaviour for source code. In *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 1–10, 2016.

[8] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.

[9] T. Blascheck and B. Sharif. Visually analyzing eye movements on natural language texts and source code snippets. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications*, pages 1–9, 2019.

[10] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265. IEEE, 2015.

[11] T. Busjahn, C. Schulte, and A. Busjahn. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, pages 1–9, 2011.

[12] T. Busjahn, C. Schulte, B. Sharif, A. Begel, M. Hansen, R. Bednarik, P. Orlov, P. Ihantola, G. Shchekotova, and M. Antropova. Eye tracking in computing education. In *Proceedings of the tenth annual conference on International computing education research*, pages 3–10, 2014.

[13] A. Calcagno, S. Coelli, R. Couceiro, J. Durães, C. Amendola, I. Pirovano, R. Re, and A. M. Bianchi. Eeg monitoring during software development. In *2020 IEEE 20th Mediterranean Electrotechnical Conference (MELECON)*, pages 325–329. IEEE, 2020.

[14] J. Castelhano, I. C. Duarte, C. Ferreira, J. Duraes, H. Madeira, and M. Castelo-Branco. The role of the insula in intuitive expert bug detection in computer code: an fmri study. *Brain imaging and behavior*, 13(3):623–637, 2019.

[15] R. Couceiro, R. Barbosa, J. Durães, G. Duarte, J. Castelhano, C. Duarte, C. Teixeira, N. Laranjeiro, J. Medeiros, P. Carvalho, et al. Spotting problematic code lines using nonintrusive programmers' biofeedback. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 93–103. IEEE, 2019.

[16] R. Couceiro, G. Duarte, J. Durães, J. Castelhano, C. Duarte, C. Teixeira, M. C. Branco, P. Carvalho, and H. Madeira. Biofeedback augmented software engineering: monitoring of programmers' mental effort. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 37–40. IEEE, 2019.

[17] I. Crk, T. Kluthe, and A. Stefik. Understanding programming expertise: an empirical study of phasic brain wave changes. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 23(1):1–29, 2015.

[18] M. E. Crosby, J. Scholtz, and S. Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *PPIG*, page 5. Citeseer, 2002.

[19] M. E. Crosby and J. Stelovsky. How do we read algorithms?Acase study. *Computer*, 23(1):25–35, 1990.

[20] B. Dayma and P. Cuenca. Craiyon: Ai model drawing images from any prompt! https://www.craiyon.com, 2022.

[21] R. Dewar. Setl and the evolution of programming. In *From Linear Operators to Computational Biology*, pages 39–46. Springer, 2013.

[22] J. Duraes, H. Madeira, J. Castelhano, C. Duarte, and M. C. Branco. Wap: understanding the brain at software debugging. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 87–92. IEEE, 2016.

[23] B. Easter. Fully human, fully machine: Rhetorics of digital disembodiment in program-

ming. *Rhetoric Review*, 39(2):202–215, 2020.

[24] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 286–28610. IEEE, 2018.

[25] S. Fakhoury, D. Roy, Y. Ma, V. Arnaoudova, and O. Adesope. Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Software Engineering*, pages 1–39, 2019.

[26] B. Floyd, T. Santander, and W. Weimer. Decoding the representation of code in the brain: An fmri study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 175–186. IEEE, 2017.

[27] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th international conference on software engineering*, pages 402–413, 2014.

[28] D. Fucci, D. Girardi, N. Novielli, L. Quaranta, and F. Lanubile. A replication study on code comprehension and expertise using lightweight biometric sensors. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 311–322. IEEE, 2019.

[29] L. J. Garey. *Brodmann's' localisation in the cerebral cortex'*. World Scientific, 1999.

[30] B. Helmlinger, M. Sommer, M. Feldhammer-Kahr, G. Wood, M. E. Arendasy, and S. E. Kober. Programming experience associated with neural efficiency during figural reasoning. *Scientific reports*, 10(1):1–14, 2020.

[31] D. C. Hoaglin, B. Iglewicz, and J. W. Tukey. Performance of some resistant rules for outlier labeling. *Journal of the American Statistical Association*, 81(396):991–999, 1986.

[32] B. (https://stackoverflow.com/users/575085/butterdog). Using emoji as identifier names in c++ in visual studio or gcc, 2015.

[33] Y. Huang, X. Liu, R. Krueger, T. Santander, X. Hu, K. Leach, and W. Weimer. Distilling neural representations of data structure manipulation using fmri and fnirs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 396–407. IEEE, 2019.

[34] Y. Ikutani, T. Kubo, S. Nishida, H. Hata, K. Matsumoto, K. Ikeda, and S. Nishimoto. Expert programmers have fine-tuned cortical representations of source code. *Eneuro*, 8(1), 2021.

[35] Y. Ikutani and H. Uwano. Brain activity measurement during program comprehension with nirs. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6. IEEE, 2014.

[36] T. Ishida and H. Uwano. Synchronized analysis of eye movement and eeg during program

comprehension. In *2019 IEEE/ACM 6th International Workshop on Eye Movements in Programming (EMIP)*, pages 26–32. IEEE, 2019.

[37] S. Ivanova, AnnaAand Srikant, Y. Sueoka, H. H. Kean, R. Dhamala, U.-M. O'reilly, M. U. Bers, and E. Fedorenko. Comprehension of computer code relies primarily on domain-general executive brain regions. *Elife*, 9:e58906, 2020.

[38] A. Jbara and D. G. Feitelson. How programmers read regular code: A controlled experiment using eye tracking. *Empirical software engineering*, 22(3):1440–1477, 2017.

[39] S. Jeanmart, Y.-G. Gueheneuc, H. Sahraoui, and N. Habra. Impact of the visitor pattern on program comprehension and maintenance. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 69–78. IEEE, 2009.

[40] M. A. Just and P. A. Carpenter. A theory of reading: From eye fixations to comprehension. *Psychological review*, 87(4):329, 1980.

[41] N. W. Kim, Z. Bylinskii, M. A. Borkin, A. Oliva, K. Z. Gajos, and H. Pfister. A crowdsourced alternative to eye-tracking for visualization understanding. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, pages 1349–1354, 2015.

[42] M. V. Kosti, K. Georgiadis, D. A. Adamos, N. Laskaris, D. Spinellis, and L. Angelis. Towards an affordable brain computer interface for the assessment of programmers' mental workload. *International Journal of Human-Computer Studies*, 115:52–66, 2018.

[43] R. Krueger, Y. Huang, X. Liu, T. Santander, W. Weimer, and K. Leach. Neurological divide: an fmri study of prose and code writing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 678–690. IEEE, 2020.

[44] S. Lee, D. Hooshyar, H. Ji, K. Nam, and H. Lim. Mining biometric data to predict programmer expertise and task difficulty. *Cluster Computing*, 21(1):1097–1107, 2018.

[45] S. Lee, A. Matteson, D. Hooshyar, S. Kim, J. Jung, G. Nam, and H. Lim. Comparing programming language comprehension between novice and expert programmers using eeg analysis. In *2016 IEEE 16th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 350–355. IEEE, 2016.

[46] B. Leister. Data from this study. https://github.com/Surlix/Eyecatcher_QuestionTypeImpact, 2022.

[47] Y.-T. Lin, C.-C. Wu, T.-Y. Hou, Y.-C. Lin, F.-Y. Yang, and C.-H. Chang. Tracking students' cognitive processes during program debugging—an eye-movement approach. *IEEE transactions on education*, 59(3):175–186, 2015.

[48] Y.-F. Liu, J. Kim, C. Wilson, and M. Bedny. Computer code comprehension shares neural resources with formal logical inference in the fronto-parietal network. *Elife*, 9:e59340, 2020.

[49] I. McChesney and R. Bond. Eye tracking analysis of computer program comprehension

in programmers with dyslexia. *Empirical Software Engineering*, 24(3):1109–1154, 2019.

[50] I. McChesney and R. Bond. Observations on the linear order of program code reading patterns in programmers with dyslexia. In *Proceedings of the Evaluation and Assessment in Software Engineering*, pages 81–89. 2020.

[51] J. Medeiros, R. Couceiro, J. Castelhano, M. C. Branco, G. Duarte, C. Duarte, J. Durães, H. Madeira, P. Carvalho, and C. Teixeira. Software code complexity assessment using eeg features. In *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 1413–1416. IEEE, 2019.

[52] A. O. Mohamed, M. P. Da Silva, and V. Courboulay. A history of eye gaze tracking. 2007.

[53] T. Nakagawa, Y. Kamei, H. Uwano, A. Monden, K. Matsumoto, and D. M. German. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement:A controlled experiment. In *Companion proceedings of the 36th international conference on software engineering*, pages 448–451, 2014.

[54] A. Newman, B. McNamara, C. Fosco, Y. B. Zhang, P. Sukhum, M. Tancik, N. W. Kim, and Z. Bylinskii. Turkeyes:A web-based toolbox for crowdsourcing attention data. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[55] M. Nivala, F. Hauser, J. Mottok, and H. Gruber. Developing visual expertise in software engineering: An eye tracking study. In *2016 IEEE Global Engineering Education Conference (EDUCON)*, pages 613–620. IEEE, 2016.

[56] R. Orlov, PavelAand Bednarik. The role of extrafoveal vision in source code comprehension. *Perception*, 46(5):541–565, 2017.

[57] P. Peachock, N. Iovino, and B. Sharif. Investigating eye movements in natural language and c++ source code- A replication experiment. In *International Conference on Augmented Cognition*, pages 206–218. Springer, 2017.

[58] N. Peitek, J. Siegmund, and S. Apel. What drives the reading order of programmers? an eye tracking study. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 342–353, 2020.

[59] N. Peitek, J. Siegmund, S. Apel, C. Kästner, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. A look into programmers' heads. *IEEE Transactions on Software Engineering*, 46(4):442–462, 2018.

[60] N. Peitek, J. Siegmund, C. Parnin, S. Apel, J. C. Hofmeister, and A. Brechmann. Simultaneous measurement of program comprehension with fmri and eye tracking: A case study. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2018.

[61] C. S. Peterson, T. Saddler, JonathanAand Blascheck, and B. Sharif. Visually analyzing students' gaze on c++ code snippets. In *2019 IEEE/ACM 6th International Workshop on*

Eye Movements in Programming (EMIP), pages 18–25. IEEE, 2019.

[62] P. Rodeghero, C. Liu, P. W. McBurney, and C. McMillan. An eye-tracking study of java programmers and application to source code summarization. *IEEE Transactions on Software Engineering*, 41(11):1038–1054, 2015.

[63] P. Rodeghero and C. McMillan. An empirical study on the patterns of eye movement during summarization tasks. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE, 2015.

[64] A. Sarkar. The impact of syntax colouring on program comprehension. In *PPIG*, page 8, 2015.

[65] Z. Sharafi, Z. Soh, Y.-G. Guéhéneuc, and G. Antoniol. Women and men—different but equal: On the impact of identifier style on source code reading. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 27–36. IEEE, 2012.

[66] B. Sharif, M. Falcone, and J. I. Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 381–384, 2012.

[67] B. Sharif and J. I. Maletic. An eye tracking study on camelcase and under_score identifier styles. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 196–205. IEEE, 2010.

[68] J. Siegmund. Toward measuring program comprehension with functional magnetic resonance imaging. https://www.tu-chemnitz.de/informatik/ST/research/material/fMRI/index.php, 2022.

[69] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th international conference on software engineering*, pages 378–389, 2014.

[70] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[71] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 140–150, 2017.

[72] J. Siegmund, M. Schwarzkopf, and J. Mucke. Reyeker: Remote eye tracker, 2021.

[73] L. Thite and R. Brown. The history of eye tracking. 2015.

[74] TobiiAB. This is eye tracking, https://www.tobii.com/group/about/this-is-eye-tracking/, 2021.

[75] TroykaMED. fmri response button system, https://troykamed.com/en/urunler/fmri-

system/fmri-tepki-buton-seti/fmri-tepki-buton-seti/, 2021.

[76] TUC. Technical university chemnitz, 2021.

[77] J. Tukey. Multiple comparisons. *Journal of the American Statistical Association*, 48(263):624–625, 1953.

[78] R. Turner, M. Falcone, B. Sharif, and A. Lazar. An eye-tracking study assessing the comprehension of c++ and python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 231–234, 2014.

[79] R.-G. Urma. Programming language evolution. Technical report, University of Cambridge, Computer Laboratory, 2017.

[80] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 133–140, 2006.

[81] M. K.-C. Yeh, D. Gopstein, Y. Yan, and Y. Zhuang. Detecting and comparing brain activity in short program comprehension using eeg. In *2017 IEEE Frontiers in Education Conference (FIE)*, pages 1–5. IEEE, 2017.