TECHNISCHE UNIVERSITÄT
CHEMNITZ

# Effect of Flowcharts on Code Comprehension of Novice Programmers

**Master Thesis**

for obtaining the academic degree
M.Sc. Web Engineering

Fakultät für Informatik
Professur Softwaretechnik

Submission date: 18.11.2022

Supervisor: Prof. Dr. Janet Siegmund
M.Sc. Elisa Hartman

# Abstract

**Background**: There are a variety of articles that have examined problems faced by novice programmers during introductory program courses. Insufficient problem-solving skills has been identified as a significant barrier to programming learning. Furthermore, novices are not able to create appropriate mental models during program comprehension.

**Research goal**: The use of structured flowcharts to support the program comprehension process of novices is investigated. The effect of flowcharts on visual attention, cognitive load, response time, correctness and subjective preference of novices is studied.

**Research Method:** A within-subjects study is conducted among students. Participants performed comprehension tasks with code snippets alone and code snippets in conjunction with flowcharts. An eye tracker and an EEG are used to gain additional insights. A total of 11 students took part in the experiment.

**Results**: Participants used flowcharts significantly as part of their program comprehension process. Additionally, a significant effect on the correction rate and response time was found. Participants using flowcharts answered with a higher correction rate. However, their response time also increased significantly. Finally, subjective preferences are largely in favor of flowcharts as well. No significant effect on cognitive load was found.

**Conclusion**: Flowcharts could be an effective visual assistance during the program comprehension of novices. Although, the use of flowcharts extended the response time, it also increased the correctness ratio of comprehension tasks. This could indicate that novices may develop appropriate mental models that led them ultimately to a correct understanding of the program when flowcharts where available.

**Future Work**: Further studies are necessary to generalize the results. These could consider different algorithm complexities and levels of programming experience. In addition, other methods such as, pretest designs, think-aloud protocols, debug tasks can be used to have a better understanding of the effect of flowcharts in code comprehension.
**Keywords: Flowcharts, program comprehension, novice programmers, visual attention, cognitive load.**

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **AOI** | Area of Interest |
| **BACCII** | Ben A Calloni Coding for Iconic Interface |
| **BPMN** | Business process modeling notation |
| **CSV** | Comma-Separated Values |
| **EEG** | Electroencephalography |
| **EPU** | External Processing Unit |
| **FITS** | Flowchart-based Intelligent Tutoring System |
| **FLINT** | Flowchart Interpreter |
| **ICA** | Independent Component Analysis |
| **IT** | Information Technology |
| **SDK** | Software Development Kit |
| **SFC** | Structured Flow Chart Editor |
| **STEM** | Science, Technology, Engineering, and Mathematics |
| **UML** | Unified Modeling Language |

# 1 Introduction

The demand for graduates in computer science has experienced a significant increase due to technological developments and the diversity of applications [1]. Computer science knowledge plays a key role in the digitalization strategies of companies and institutions. In particular, software systems have become an important competitive advantage in meeting the market requirements [2].

The shortage of computer scientists is slowing down economic growth in Europe. According to the European Union Commission, there was a deficiency of around one million IT specialists in 2020. In Germany alone, 96.000 IT-related jobs remained empty, this is 12% greater than the previous year and it is expected that this number will aggravate in the future [3].

Additionally, the coronavirus pandemic and the climate crisis have worsened the situation. Companies rely extensively on software systems to respond to these new circumstances. More now than ever, there is a necessity for efficient and appropriate training and education in computer science to prepare students for the challenges of actual market labor and socioeconomic crises [4].

## 1.1 Problem Statement

Computer Science degrees have gained popularity in the last decades. More than 2300 computer science-related study programs are offered in Germany [5]. Additionally, other technology-related degrees in the branches of Science, Technology, Engineering, and Mathematics (STEM) include some computer science courses in their program as this type of knowledge is required in many job positions [6].

Computer science degrees are chosen mainly due to their job prospect opportunities. The employability of this career is as high as 90%. Additionally, the tuition fees for computer science degrees have been reduced in the last few years, making it an attractive choice for students [7].

Despite all the advantages, every 3 of 10 students enrolled in a computer science degree abandon their studies in the first year and only 50% manage to complete the course program. The causes of this situation have been studied for more than 40 years and it is still a matter of investigation. Particularly, most of the problems that students face and are the cause of dropout occur in the first year of the degree when introductory programming courses are being taught [8].

The student’s success or failure in introductory programming courses often determines the decision about continuing or dropping out of studies. If a student fails, drops out, or has problems approving the introductory programming courses, it is unlikely that the student continues with the normal program course. Computer science programs are deeply dependent on introductory program courses and the problems will only aggravate if the student is not able to follow the pace of the learning process [9].

Many studies suggest that learning to program is considered a problematic and challenging task. Lahtinen et al. provide a comprehensive review of the difficulties that students face during introductory programming courses. In this study, 559 students and 34 lecturers from different institutions were interviewed. 94% of the students found it complex to learn the core concepts of programming i.e., programming structures, and programming language syntax; whereas 99% percent found it difficult to design a program that solves a certain task. Additionally, lecturers do not always have the correct perception of the problems of each student and are not able to intervene quickly with the personalized help needed [10].

Introductory programming courses are challenging as different skills need to be developed at the same time. Students are required to learn language syntax, control structures, and programming paradigms but also to acquire problem-solving skills. Moreover, a student has to learn how to use different tools (program development environments, software development kits, libraries), which can differ across programming languages. Additionally, further programming courses include more theoretical background e.g., computer architectures, protocols, and networking, to design and implement working programs. This situation could increase significantly stress and fatigue levels, obstructing the learning process [11].

Some methods, tools, and strategies have been developed to alleviate the difficulties that computer science students have during introductory programming courses. Many teaching tools are based on self-assessment, for example, the popular application TouchDevelop allows novices to practice language syntax for different programming languages [12]. This tool aims to reduce syntax errors with different methods, including multiple-choice and fill-in-the-blank tasks. This and other tools have been proven effective to introduce students to the mechanics of the languages. However, they do not train students in problem-solving or algorithm design [13].

Problem-solving is another crucial skill needed to succeed in introductory programming courses. At its core, computer science aims to translate problem specifications into a working computational program. Nevertheless, most computer science students have insufficient problem-solving skills according to many studies [14]. Westphal and Harris found that even when students understand the syntax and elements of a programming language, they cannot put these constructs together in a meaningful and useful way to solve a problem [15]. MacCracken performed a large multi-institutional study with students that completed introductory program courses. It was found that the students had good knowledge of programming elements and programming syntax but they cannot abstract a problem into a solution efficiently [16]. McIver and Conway have stated that this situation occurs when students pay much attention to the syntax and the individual elements of a programming language and exclude the broader picture which is the problem itself [17]. Moreover, Perkins describes how the natural tendency of novices is to go direct to code, in a trial-and-error style, before thinking the problem through and finding a suitable solution [18].

To improve problem-solving skills the focus should be shifted from the syntax to the algorithm design, and how all individual pieces can be combined to solve a problem. This is only possible if the students create the appropriate mental models [19]. Mental models consider the conceptual knowledge and the semantics of programming language features including all the parts, functions, and interactions that are required to execute a program [20].

Shih and Alessi discovered that experienced programmers have appropriate mental models that allow them to precisely describe the flow of a program, its purpose, and its output. In contrast, novices will find difficulties decomposing all essential parts that constitute an algorithm, thus their comprehension skills will also be weak [20]. Ramalingham et al., pointed out that a primary goal in introductory programming courses should be to help students develop appropriate mental models [21]. These studies suggest that students need to pay equal attention to programming knowledge (syntax and semantics) as well as problem-solving strategies [22].

## 1.2 Objective

Many authors support program visualization as a method to develop correct mental models. Tudoreanu et al., stated that program visualization has great potential for conveying information about the behavior of a program [23]. Levy et al., noted that visualization provides a concrete model of execution that all students need in order to understand algorithms and programming [24]. Baldwin et al., also remarked that with the presence of visualization, an accurate mental model can be developed by novices [25]. Westphal et al., suggested that without the use of diagrams or any other form of visualization, it is difficult for novices to communicate the flow of a program [26].

Flowcharts are a type of visualization technique that describes the logic of processes and procedures. They are used to communicate the flow of algorithms, depicting the concepts of iteration, conditions, and sequences they are composed of. Moreover, flowcharts can be easily understood without complicated training; they use a very simple notation that any student can easily master. Additionally, flowcharts are designed to aid program composition and program comprehension, skills that are often weak in novice programmers. Using this resource, a student could focus on the abstract concepts of programming without getting distracted by language syntax [23].

As noted in the previous section, the development of appropriate mental models is a key point that determines the success of students in introductory programming courses. A flowchart provides the novice with a visual representation of the semantics of individual components and how they build up to an algorithm. Therefore, flowcharts could be useful instructional aids for the development of appropriate mental models of programs [24].

Flowcharts could influence the code comprehension of novices, by allowing them to visualize the flow of execution, the semantics of control structures, and how the individual pieces interact to form higher-level concepts [25]. The present master thesis studies the effect of flowcharts on the program comprehension of novices. To this effect, however, this research does not solely rely on performance metrics such as correctness and response time but also examines how novices' visual attention and cognitive load change due to the presence of flowcharts. The inclusion of an eye-tracker and EEG device in this study is an important milestone, toward a clear understanding of the advantages and disadvantages of algorithm visual representations in introductory programming courses. The Eye-tracking device will provide exact information about novices’ gaze behavior over flowcharts, particularly whether flowcharts are actually used. Moreover, an EEG device allows to investigate whether the active usage of flowcharts alleviates the cognitive load in novices. Finally, the correctness ratio and response times are evaluated accordingly. The research question is formulated as:

*RQ: What Is the impact of flowcharts on code comprehension of novice programmers?*

## 1.3 Thesis Structure

The present work is structured as follows:

**Chapter 2 (Literature Review):** As a first step, it is critical to establish the theoretical background used to tackle the problem in the investigation. This forms a frame of reference in which suitable concepts, methodologies, and theories to approach the research problem are defined. In this chapter, the program comprehension theories, methods of measuring program comprehension, and the theory of novice programmers are discussed. Moreover, existing studies related to the problem of the investigation and the status of the knowledge are also presented. Studies concerning the use of algorithm visual representations in program comprehension of novices are considered. Their findings constitute a guide to continue the specific line of research and contribute to the knowledge base.

**Chapter 3 (Methodology):** In this chapter, the suitable methods and strategies to measure program comprehension presented in the previous chapter are selected. This includes variable definition, hypothesis definition, and operationalization. The material and tasks used to measure the effect of flowcharts on program comprehension of novices are also explained. Finally, the design of the study is defined.

**Chapter 4 (Conduct):** The implementation of the design proposed in the previous chapter is presented. This includes the detailed procedure to collect the data and the different data formats.

**Chapter 5 (Data Analysis):** After the data is collected, preprocessing and data cleaning procedures take place. In this chapter, the results are described and analyzed using statistical and visual techniques. Finally, hypothesis testing is conducted and the research question is answered.

**Chapter 6 (Discussion):** Interpretation of the results is given and the findings are related to the research question. The threads to validity are also included.

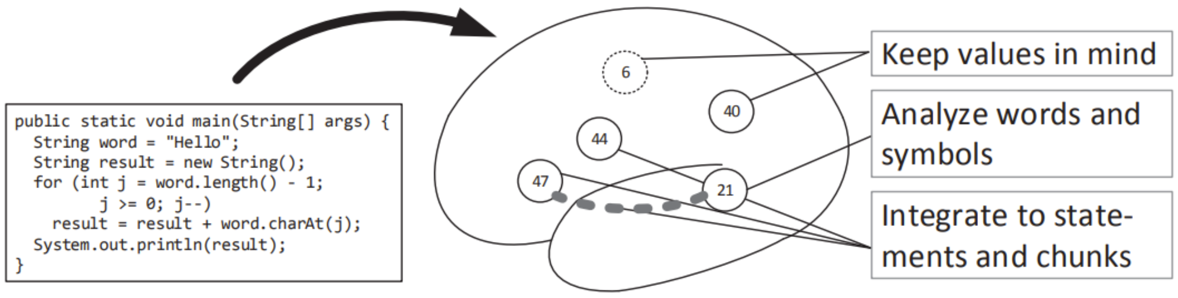
**Chapter 7 (Conclusion and Future Work):** The conclusion and future scope of the research topic are discussed.

# 2 Literature Review

This chapter highlights the theory of code comprehension, comprehension strategies, and approaches used to measure code comprehension. The technical features of eye tracking and EEG devices are also discussed. Finally, related work on the effect of visual representations of algorithms over code comprehension is reviewed.

## 2.1 Program Comprehension

Program comprehension is defined as an internal cognitive, problem-solving, and iterative process that aims to discover the purpose and meaning of source code [27]. In other words, program comprehension is the process of understanding what the program is doing and how it is implemented by different methods as illustrated in Figure 2.1.



```
public static void main(String[] args) {
  String word = "Hello";
  String result = new String();
  for (int j = word.length() - 1;
        j >= 0; j--)
    result = result + word.charAt(j);
  System.out.println(result);
}
```

**Figure 2.1 Program comprehension process [28].**

Program comprehension is the main activity performed in software maintenance. In this stage of the software development process, developers need to understand entire codebases in detail making it an extremely demanding, expensive, and difficult task. During the maintenance process, developers must study source code that also changes over time and is continuously growing. Therefore, code comprehension is a key activity during software development, and it is a growing concern for software development companies. It is required developers to have great program comprehension skills as they need to establish a fast and efficient understanding of the source code [29]. Chaparro et al. stated that high code comprehension skills improve significantly the maintenance stage of software [30].

Additionally, software features are changed or added on the go during the software development process. In these cases, developers need to make changes to the source code according to new requirements. The ability to read and comprehend a program that others have made is also a challenging task. It is well known that software engineering is performed by a team rather than by an individual, where the work is divided and assigned across individual members. Consequently, code comprehension is required in multiple software engineering tasks and its role is crucial to successfully assemble and maintain codebases [31].

### *2.1.1 Program Comprehension Strategies*

Different comprehension strategies have been proposed to explain how readers understand code. The strategy used by readers aims to create a mental model i.e., an internal representation of the program being studied. Using these strategies, the reader creates a representation of the concepts, relationships, dependencies, and other dynamic and static aspects of the source program [32].

The program comprehension process depends on how much knowledge the reader can use to understand the source code. Readers who know a program's domain use this knowledge during the program comprehension process. Depending on the amount of domain knowledge, there are three different kinds of program comprehension strategies: top-down models, bottom-up models, and integrated models. These strategies are discussed in the following sections [32].

#### *2.1.1.1 Top-Down Comprehension*

In a top-down comprehension strategy, the reader starts from a higher-level domain and goes to a lower-level domain. It is a fast and efficient strategy where readers use domain knowledge to quickly understand the source code intent. The most influential works on this strategy are the theories developed by Brooks, in 1983, and by Soloway and Ehrlich in 1989 [32].

In his cognitive theory, Brooks defends that program comprehension is hypothesis-driven. That is, the reader creates a primary hypothesis about the source code, which is very general, and goes to a lower-level domain to discover the answer. Meanwhile, that primary hypothesis is subdivided into a hierarchic structure of hypotheses that are tested one after another as illustrated in Figure 2.2 [33].

The hypotheses are validated with the help of beacons in the code. Beacons are a set of features, marks, or structures commonly used to identify concepts in programs.

Readers confirm and refine their hypothesis by quickly jumping between these beacons or other points of interest. If the program confirms their hypotheses, readers have understood its purpose. If any hypothesis cannot be validated, the reader has to use another comprehension strategy [33].

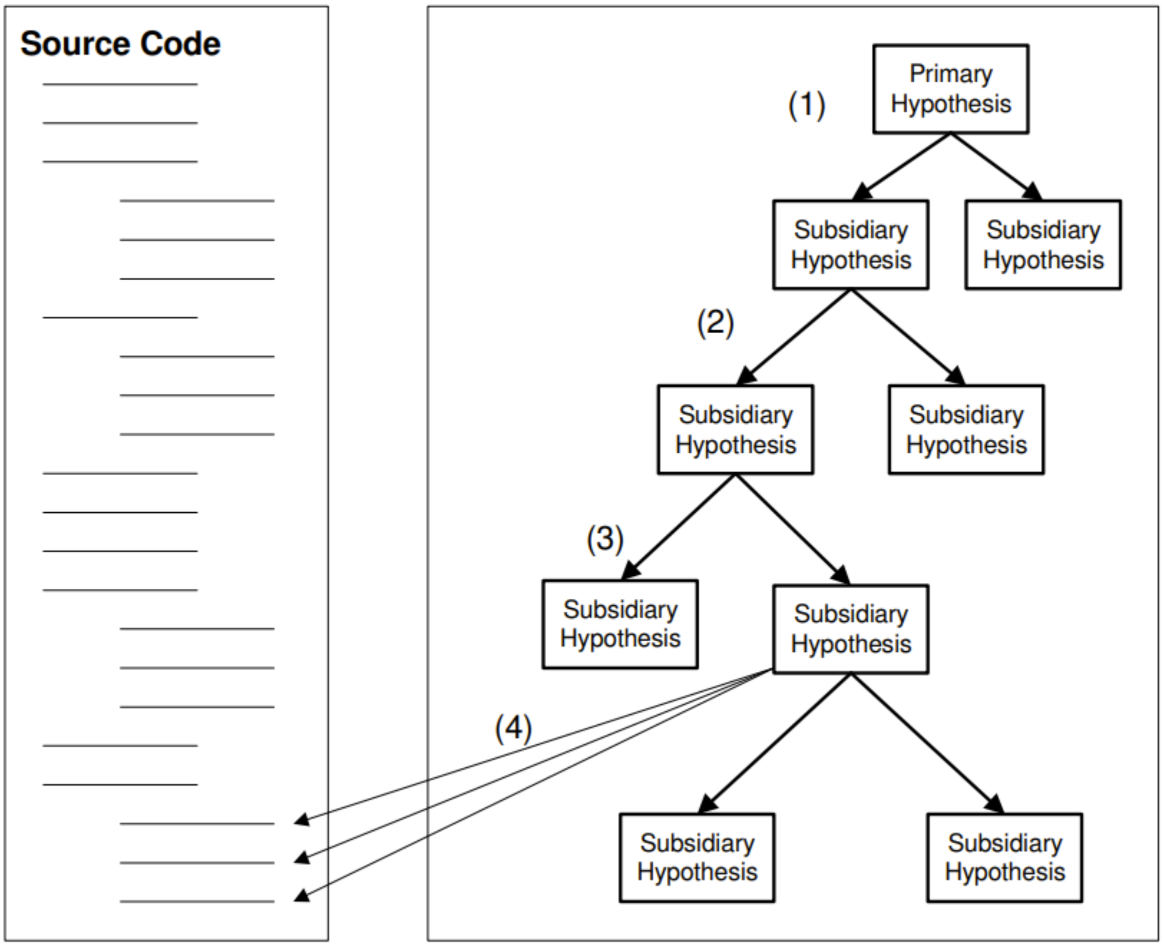


**Figure 2.2 Cognitive theory of program comprehension by Brooks [34].**

Soloway and Ehrlich’s theory also follows a top-down approach, however, it is based on programming plans and the rules of programming discourse. The former are “generic program fragments that represent stereotypic action sequences in programming” [35], and the latter “capture conventions in programming and govern the composition of plans into programs” [35]. The process is complete when the reader has associated the programming plans with the program goals. The authors state that these are features that only expert programmers have. Thus, their theory is only observed on expert readers.

Top-down comprehension requires lower cognitive effort than bottom-up comprehension since the process starts with an initial hypothesis and program statements are understood conjunctively. Thus, readers use top-down comprehension whenever possible [36]. This is of course not viable with novices due to their lack of experience, instead, they mostly use the bottom-up comprehension strategy, which is explained in the following section.

Many studies have identified the importance of beacons during this strategy. Wiedenbeck performed several experiments to evaluate their role in program comprehension. Such studies concluded that beacons are a critical aspect that influences greatly the comprehension process [37]. However, they have also remarked that inappropriate beacons mislead greatly program comprehension [38].

Finally, Shaft and Vessey demonstrated how domain knowledge plays an important role in program comprehension. Experienced programmers may read source code like a beginner if they are not familiar with the domain [39].

#### *2.1.1.2 Bottom-Up Comprehension*

As opposed to the top-down strategy, during the bottom-up strategy, the reader starts from a lower-level domain and goes to a higher-level domain. The most important works on this strategy are the theories developed by Shneiderman and Mayer [40], in 1979, and Pennington [41], in 1987. Another work, although not a theory, is that of Basili and Mills [42], in 1982.

According to Schneiderman and Mayer, the reader uses programming knowledge to abstract source code fragments into chunks which are stored in the short-term memory as illustrated in Figure 2.3. Thus, the source code is decomposed into several code fragments which could be mapped to high or low concepts. These chunks are thoroughly studied, using the different concepts of programming knowledge until they are mapped into higher-level concepts. For this, the reader refers to the long-term memory, where domain knowledge is stored [40].

Pennington states that the reader starts the comprehension process by getting information from the source code to create an abstract control flow. In this first step, the reader discovers the program execution order. Then, in a bottom-up process, the reader creates a more specific model that includes data-flow and functional abstractions of the program [41].

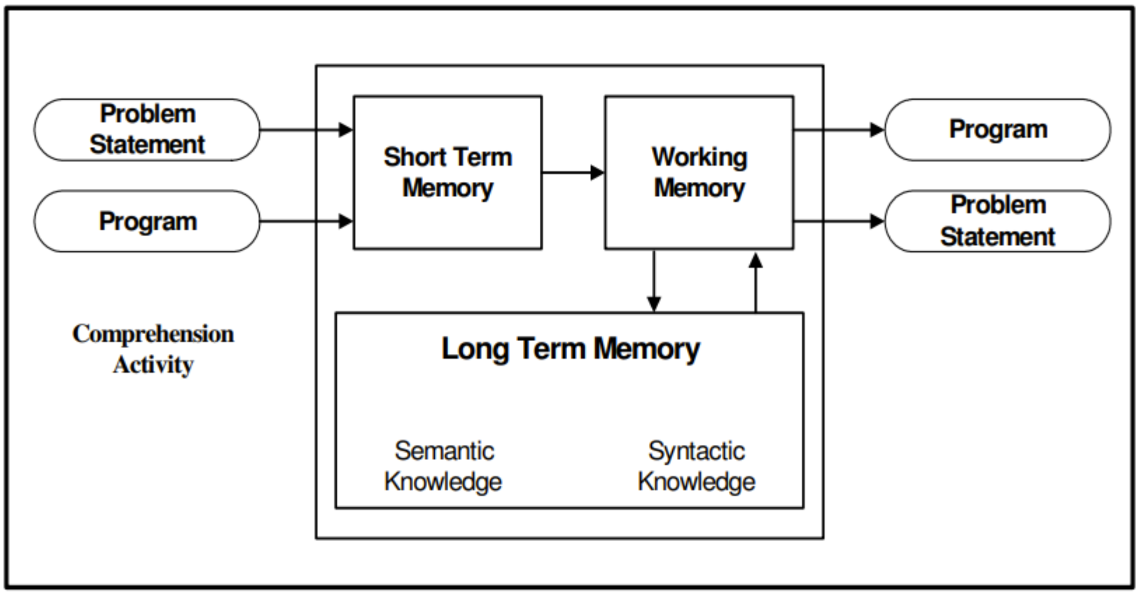


**Figure 2.3 Comprehension strategy by Schneiderman and Mayer [34].**

### *2.1.1.3 Hybrid Program-Comprehension Strategies*

Hybrid strategies are cognitive models that incorporate both top-down and bottom-up strategies of program comprehension. Several authors argue that the program comprehension process is not as rigid as following a top-down or a bottom-up strategy, so alternative strategies have been proposed [43].

Letovsky provides an opportunistic view of the process of program comprehension. This strategy defines three main components: the programming knowledge, the mental model, and the assimilation process. The latter is the process in which the reader uses the programming knowledge base and any other resource e.g., documentation, to generate and improve the mental model as illustrated in Figure 2.4. During the assimilation process, the reader can select the bottom-up or top-down strategy, considering the one that is more valuable to finish the task [44].

Littman et al. suggest that the strategies of program comprehension can be used depending on the needs of the reader. In a given moment, the strategies can be changed from one approach to another, in an opportunistic manner. The authors conclude that when adhering to a single strategy, the time consumption is higher, but the knowledge gathered is also higher. On the other hand, when using a hybrid strategy, the reader takes less time to comprehend the program, and the knowledge acquired is focused on fragments of the source code, and not as a whole [45].

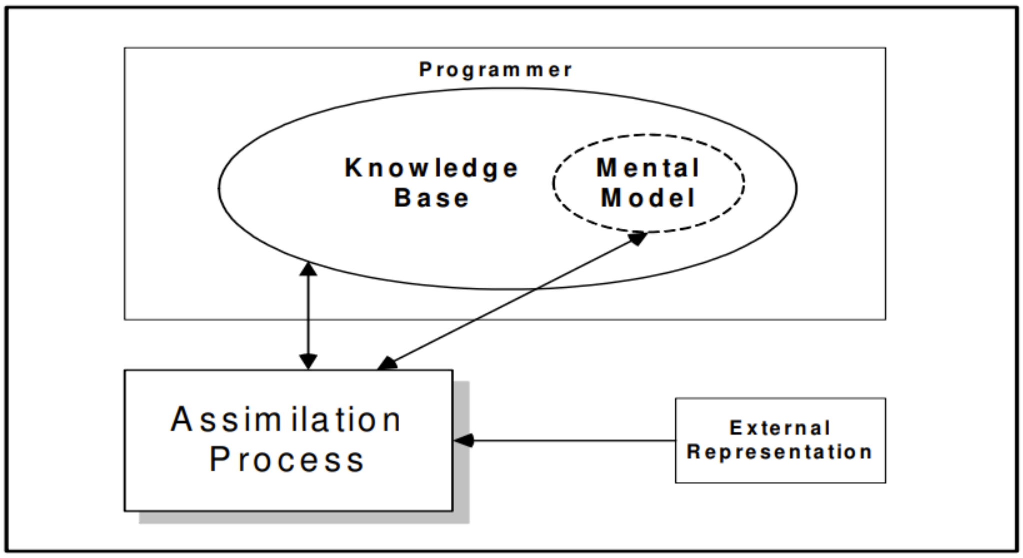


**Figure 2.4 Comprehension strategy by Letovsky [34].**

Koenemann and Robertson claim that “program comprehension is best understood as a goal-oriented, hypotheses-driven problem-solving process” [46]. The authors also state that the reader focus on the parts of the source code that are relevant for the program comprehension. Readers will employ first the top-down approach to establish hypotheses. The bottom-up approach is then used when hypotheses verification fails.

Von Mayrhaser et al. analyzed the strategies developed by Soloway and Ehrlich and Pennington, and created the Integrated Model. This model embodies the program model, situation model, the top-down model, and programming knowledge. Depending on the experience of the reader, the integrated model can be started with a top-down approach, if the reader has experience in the domain; or by the construction of the program model, otherwise [29].

Other works include the one of Xu [47], who presented a new approach based on the constructivist learning theories and on Bloom and Krathwohl’s taxonomy of the cognitive domain [48]. Furthermore, Guéhéneuc [49] proposed a theory of program comprehension that includes vision science aspects, in order to extend Brooks’ and von Mayrhauser’s models.

### *2.1.2 Approaches to measure Program Comprehension*

Several approaches have been used to measure program comprehension. In the following sections, the relevant approaches to this study are discussed.

#### *2.1.2.1 Task Performance*

Task performance is a method for measuring participant behavior from an objective and quantitative perspective. In program comprehension research, response time and correctness are often measured. Task performance is an indirect way to measure participants' understanding of a certain domain. Participants who have deep knowledge domain will probably solve the tasks effectively. Though, participants who do not have the necessary knowledge or skills will need more time to complete the task (response time) or will solve tasks incorrectly (correctness) [50].

One of the main advantages of task performance is that can it be used to study the behavior of large groups. The measurements can be collected individually or in groups, in presence or in an online setting. Thus, it is less demanding to implement a study with a large number of participants as with other approaches. This is especially advantageous in reaching the minimum significance levels [50].

Response time is the time a participant takes to complete a comprehension task. This measurement can be used both for studying the differences between participants and also for comparing the performance of the same participant with different tasks. For example, a participant can study a task with different levels of complexity which will potentially change the comprehension process. The underlying assumption is that the response time is also an indicator of perceived difficulty [50].

Correctness measures the ratio of correct responses. This metric is useful to measure the limits of participants' understanding of a domain. If a participant cannot solve a task correctly, their thought process and subsequent mental model are not appropriate. Plenty of program-comprehension studies have used task performance as the main method to measure program comprehension. For example, Soloway and Ehrlich designed a fill-in-the-blank code task for novice and expert programmers. Using task performance, they found significant differences between novices and expert programmers [51].

#### *2.1.2.2 Interviews*

Interviews aim to understand the internal thought processes of participants [52]. In most cases, the verbalization process happens after a given task or after completing a study. Interviews allow to have a better understanding of participants' experiences.

Interviews are generally used in combination with quantitative methods in program comprehension studies. For example, Xia et al., observed 78 programmers during their workday to study the importance of program comprehension in everyday work. They found that program comprehension accounts for an important part of the tasks, especially for less experienced programmers. The authors also conducted interviews to confirm their quantitative data [53]. Similarly, Roehm performed a study for 45 minutes and a subsequent interview, where participants could give more details about the experience. The study concluded that programmers focus on doing a given task without having a deep understanding of the underlying concepts required [54].

Interviews are a simple method to collect more information about a participant's behavior and thoughts, however, its correct implementation could be challenging [54]. The researcher may introduce bias during the interview or the analysis of qualitative data [55]

#### *2.1.2.3 Subjective Ranking*

Subjective rating is a method where participants state their perceived comprehension. For this, the Likert scale is often used (e.g., “How well did you understand the presented code snippet?” with five answering options: “not at all”, “a little”, “about half”, “mostly understood”, “fully understood”, [56]) or a semantic differential scale (e.g., “How difficult is the presented code snippet?” with five answering options: “simple”, “somewhat simple”, “medium”, “somewhat complex”, “complex”, [57]). Subjective rating maps participant’s perception to quantitative data, but is limited to the question.

Subjective Ranking is also frequently used in program-comprehension studies. For example, Miara et al. used subjective ranking to find the best indentation levels for programming. The study found that an indentation level of 2 of 4 spaces is ideal according to participants' perceptions [58].

#### *2.1.2.4 New approaches to measure Program Comprehension*

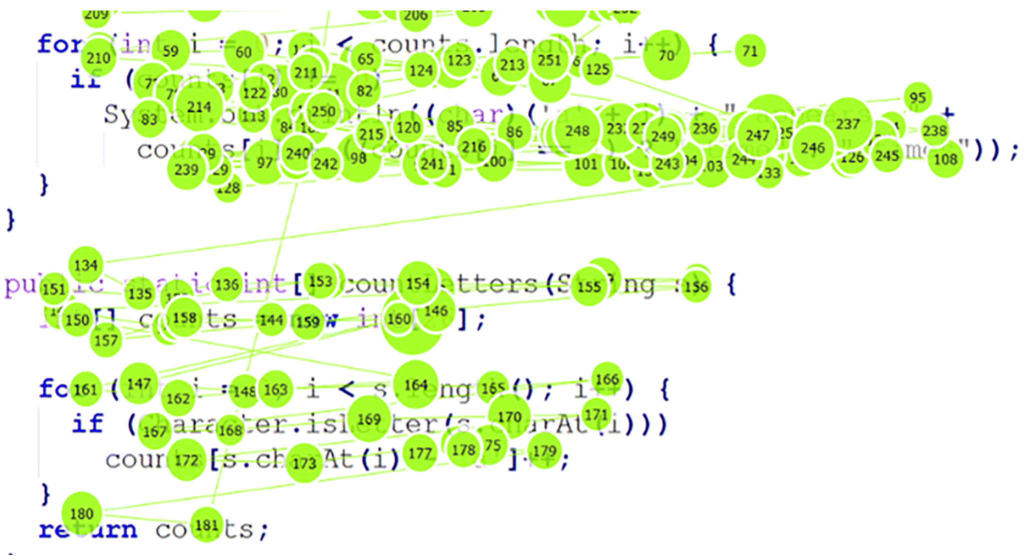
Different tools and technologies have been used to get additional insights into the program comprehension process. These techniques are discussed in the following sections.

### 2.1.2.4.1 Eye Tracking

Eye tracking is one of the most popular tools used during program comprehension research. Eye tracking tools capture participants' visual attention by following their eye movements over a screen [59]. Modern Eye tracking technology can provide eye-gaze data with high temporal and accurate spatial resolution. This technology has been used widely in many domains where human visual attention is critical. For example, in the usability domain, this technology is used to test the design of web pages [60].

Eye tracking data consists of a sequence of horizontal and vertical coordinates of the screen. As screen sizes may differ, it is necessary to perform a calibration and validation process to ensure precision and accuracy. After the data is collected, an event detection algorithm is used to calculate fixations and saccades over the raw gaze data. A fixation is a stable spatial eye gaze, which typically lasts for 100-300 ms. When fixation occurs, the participant cognitively processes the fixated visual point. A saccade is a transition between fixations. During these transitions, the participant is unable to process the visual stimulus [59].

A scan path is the sequence of saccades and fixations that the participant followed to visualize the stimulus. An example of a scan path is illustrated in Figure 2.5. A heat map portrays the distribution of fixations, where the intensity and density of fixations are coded into colors as illustrated in Figure 2.6. Many metrics have been proposed to analyze eye gaze data quantitatively: the number of fixations, average saccade length, and the number of visits to a specific area of interest (AOI) [61].

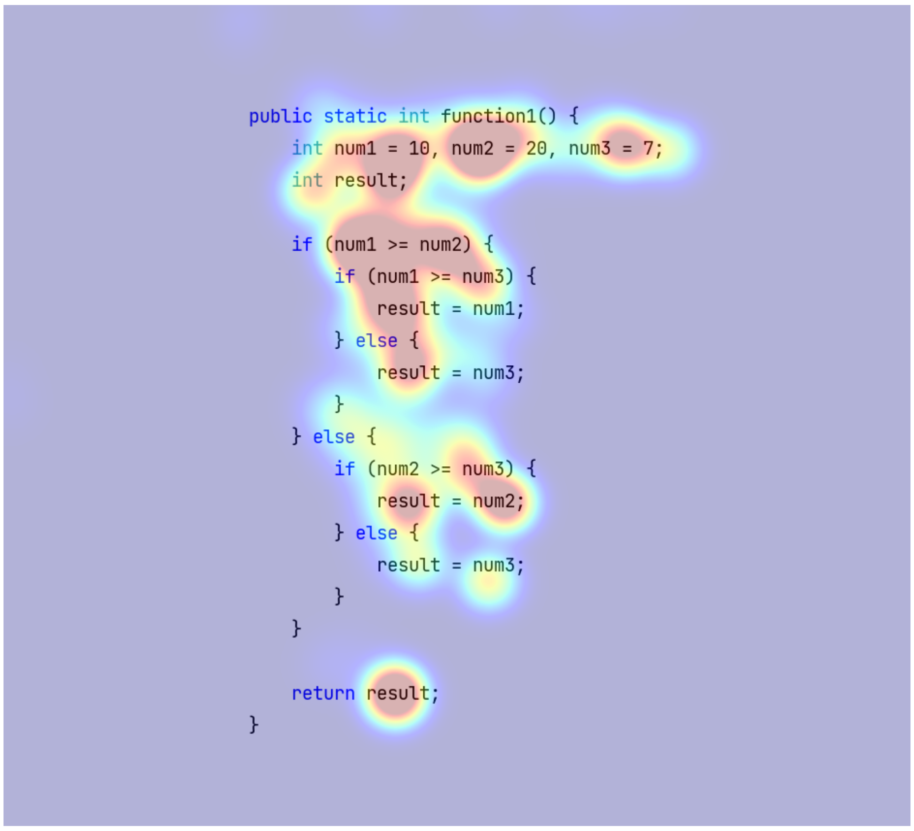


```
    for (int i = 0; i < counts.length; i++) {
      if (counts[i] != 0)
        System.out.println((char)('a' + i) + " " +
          counts[i] + ((counts[i] == 1) ? " time" : " times"));
    }
  }

  public static int[] countLetters(String s) {
    int[] counts = new int[26];

    for (int i = 0; i < s.length(); i++) {
      if (Character.isLetter(s.charAt(i)))
        counts[s.charAt(i) - 'a']++;
    }
    return counts;
```

**Figure 2.5 Visualization of a sequence of fixation and saccades during program comprehension [62].**



```
public static int function1() {
    int num1 = 10, num2 = 20, num3 = 7;
    int result;

    if (num1 >= num2) {
        if (num1 >= num3) {
            result = num1;
        } else {
            result = num3;
        }
    } else {
        if (num2 >= num3) {
            result = num2;
        } else {
            result = num3;
        }
    }

    return result;
}
```

Figure 2.6 Heatmap visualization over a code snippet.

The quality of the data depends on the accuracy and precision of the eye tracker tool. Accuracy describes the difference between the actual and measured eye gaze and precision refers to the consistency of the difference between the actual and measured eye gaze. Accuracy and precision are illustrated in Figure 2.7 [62].

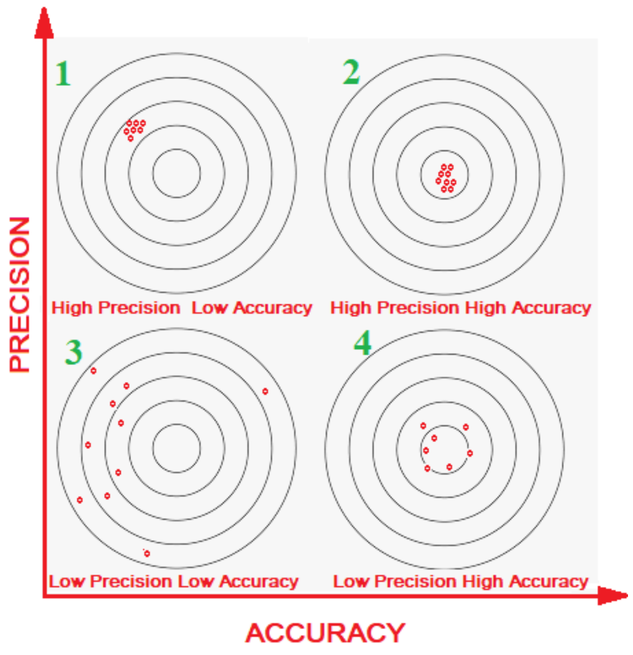

PRECISION
1
2
High Precision Low Accuracy
High Precision High Accuracy
3
4
Low Precision Low Accuracy
Low Precision High Accuracy
ACCURACY

Figure 2.7 Accuracy and precision in eye tracking [63].

Eye tracking has been widely used in program comprehension studies. Crosby and Stelovsky et al., found differences between the eye-gaze patterns of novices and expert programmers [64]. Sharif and Maletic conducted a study with eye tracking to analyze the effect of variable naming on program comprehension. They found that programmers can read underscore-style variable names faster than camel case style [65].

Busjahn et al. studied the linearity of the reading order in programmers. The authors concluded that programmers do not follow a strict linear order as in natural text. Additionally, experts read source code less linearly than novice programmers [66]. Additionally, Ikehara and Crosby showed that eye-tracking data could be used to predict task difficulty in program comprehension studies [67].

Eye-tracking tools provide also other measurements that could be used to analyze participants' performance. One of these measurements is pupil dilation. This metric is often associated with the cognitive load when subjects perform cognitive processes [68]. Beatty and Kahneman’s studies demonstrated that the number of digits to be remembered correlates with pupil dilation [69]. Similarly, Hess and Polt concluded that pupil dilation correlates with the difficulty of mathematical calculations [70]. Overall, several studies have established that pupil dilation can measure accurately mental states [71]. However, the use of pupil dilatation has many disadvantages as it is affected by many external factors especially light. Thus, preprocessing is often needed in addition to stable environmental conditions during the experiment [72].

Pupil dilation has also been used as part of program comprehension research. Ford et al. used eye-tracking metrics including pupil dilation, saccades, and blink rates to identify the mental states of programmers [73]. Behroozi et al., used also pupil dilatation to measure cognitive load while programmers write code [74].

Blink rate and duration are other eye-tracking metrics that can reveal processes of goal-directed behavior [75]. Using blink rates as a research tool is also challenging as many factors influence it. Fatigue, air humidity, and room temperature can affect the blink rate [76]. Ford et al. proposed blink rates as a possible measure for the analysis of remote interviews [77]. However, Behroozi et al., stated that blink rates are not a significant metric to distinguish mental states. Nevertheless, blink duration has exhibited important differences, when participants’ cognitive workload increases [78].

*2.1.2.4.2 Electroencephalography (EEG)*

Electroencephalography (EEG) is a technique that detects brain electrical activity with sensors placed on the participant's scalp. These electrodes receive weak electrical potential (5-–100μV) of the activity of large groups of neurons in the brain [79]. An EEG device is illustrated in Figure 2.8.

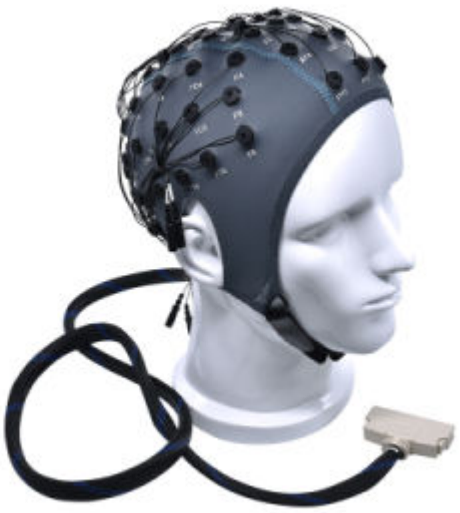


**Figure 2.8 EEG Cap [80].**

The real-time recording of brain electrical activity lacks a reproducible pattern, as can be seen in Figure 2.9. It resembles a very irregular signal with a small amplitude of around 10-100μV. The signal captured by the electrodes is weak since it is attenuated by the scalp and the outer layers of the brain, therefore it is easy to confuse it with random noise [79].

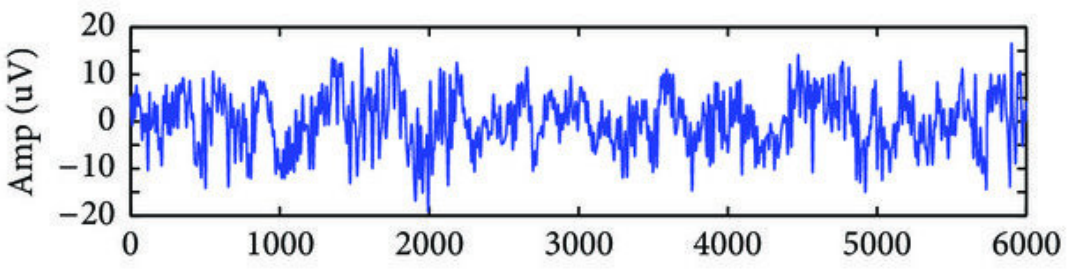


**Figure 2.9 EEG Signal [81].**

The brain waves recorded by the electroencephalograph can be analyzed in the frequency and time domains. Due to the complexity of the shape of the signal in time, analysis of the EEG signal often focuses on the power spectrum. The spectrum of an

EEG signal has historically been divided according to some frequency bands denoted with the Greek letters α, β, δ, θ, and γ; However, over time, new bands have been discovered and, in some cases, overlap with one another, and differ in characteristics such as location or brain functions. The most important bands are described next and illustrated in Figure 2.10 [82]:

- **Delta Band:** Delta waves have great amplitude and low frequency. They are typically between 1 and 4 Hz and have amplitudes of 20 to 200 μV. It is found in healthy adult individuals, exclusively during deep sleep. If detected in an awake person, it may indicate that there is some kind of abnormality in the brain.

- **Theta Band**: Theta waves have frequencies between 4 and 8 Hz and appear in childhood, although adults can also present them in periods of emotional stress and frustration. In addition, these waves are present when fatigue is in its earliest phase. It is located in the parietal and temporal zones.

- **Alpha Band:** The alpha rhythm is manifested mainly in the frequency band from 8 to 13 Hz, with amplitudes ranging between 20 and 60 μV. They are found on the electroencephalogram of most healthy adults, with eyes closed or with visual rest, awake in a calm and resting mental state. The alpha rhythm is blocked or attenuated by attention, especially visual attention, and mental or physical exertion. During deep sleep, alpha waves also disappear. It is observed mainly in the posterior area of the head, in the occipital, parietal, and posterior temporal regions.

- **Beta Band:** These small amplitude signals, below 20μV, have a frequency between 13-30 Hz. They are common and predominate during adulthood. It is usually divided into low beta, medium beta, and high beta. The low beta rhythm is usually localized to the frontal and occipital lobes, and the other two are less localized. More irregular than the alpha rhythm, it is associated with psychophysical activity, states of agitation, and alertness.

- **Gamma Band:** This rhythm is manifested at frequencies greater than 30 Hz and amplitudes between 5 and 10 μV. It is a harmonic activity that occurs in response to sensory stimuli, such as aggressive sounds or flashing lights. This signals can be observed in a large area of the cerebral cortex, manifesting mainly in the frontal and central areas.

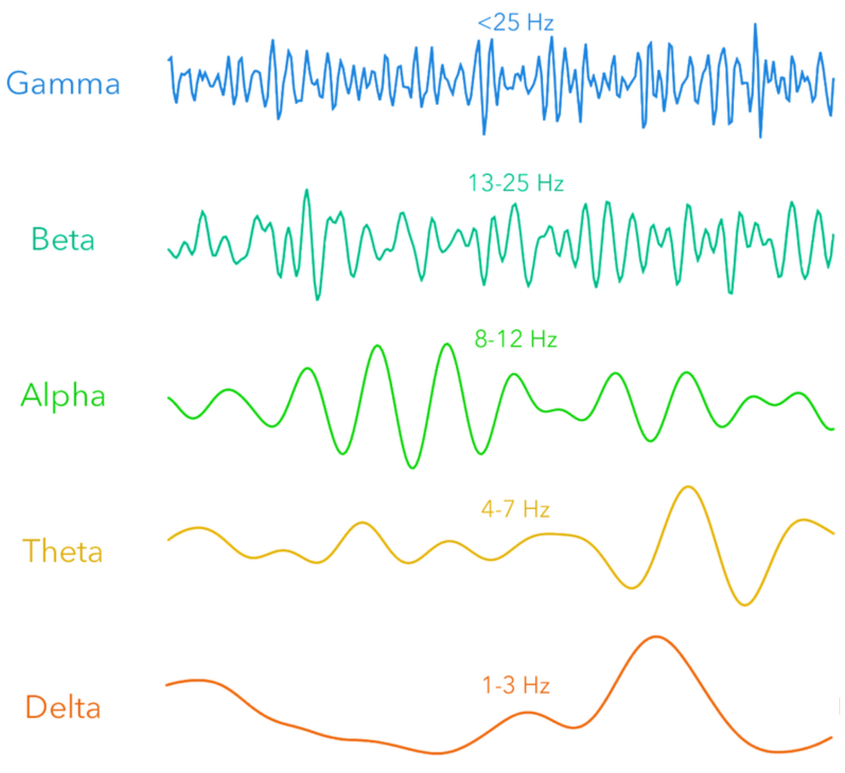


**Figure 2.10 EEG Bands [83].**

Technical and biological artifacts may affect the analysis of EEG signals as illustrated in Figure 2.11. Technical artifacts are all those that are related to the electronic part of the signal acquisition process and external agents i.e., magnetic fields, noise from the input preamplifier, aliasing, or A/D converters. Biological artifacts are those inherent to the organism. Some of the artifacts found in the EEG signals are presented below [84]:

- **Ocular Artifacts:** Ocular components are generated by the movement of the eye. Each retina creates an electrical field that can be observed with the EEG. This type of signal is divided into 2 components: horizontal and vertical. In general, any ocular component can be observed in the range of frequencies below 5 Hz.

- **Muscle Artifacts:** The origin of muscle artifacts is located usually in the frontal and temporal regions. The duration of the potential is twenty milliseconds, and can have amplitudes between 0-10 mV and frequencies from 50Hz to 150Hz; it depends on the muscle involved and the subject.

- **Respiration Artifact:** This component lasts for several seconds and is very slow. In this artifact, the cut of the current in the electrodes due to the reduction of the impedance can be seen.

- **Cardiac Artifacts:** These components are generated by cardiac activity, which is why they are related to electrocardiograms. They have a very typical pattern known as a QRS complex that becomes easily recognizable in the signal. These artifacts occur at frequencies of about 1 Hz.

- **Line Artifacts:** Generated by the proximity of the EEG electrodes to electricity-generating sources, this artifact is characterized by having frequencies between 50 and 60 Hz, depending on the region. This type of artifact is relatively easy to remove by applying filters with a lower frequency range.

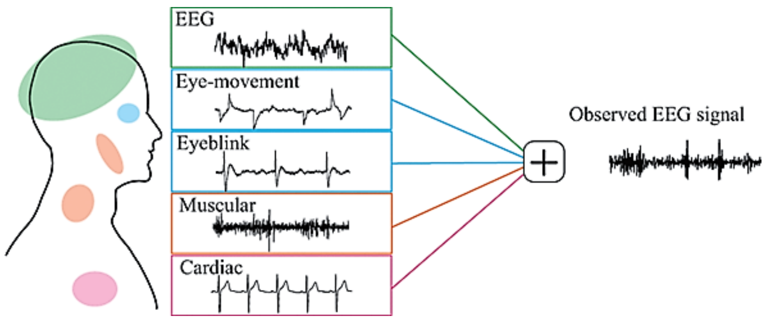


**Figure 2.11 EEG Artifacts [85]**

Many studies have shown that alpha and theta frequency bands correlate to mental workload and working memory activity. Cognitive load affects the power of these bands in opposite ways. When a subject has a strong cognitive load the theta power increases while the alpha power decreases. Therefore, the ratio of the power of these bands is a popular indicator of cognitive load [86].

In software engineering, EEG has been used to study the cognitive load of participants comprehending code. For example, Crk et al., used EEG to study the power in the alpha band across different programmer expertise levels. They found that participants with higher program experience have also a higher alpha band power during program comprehension tasks [87].

Yeh et al. used EEG to study the cognitive load while programmers comprehend confusing code patterns in C [88]. The authors found that confusing code patterns are related to a higher power in the theta band. [89].

Kosti et al. studied the cognitive load of participants during comprehension tasks and syntax-finding tasks. They could confirm that the former induced a higher activity in the theta bands and are therefore more demanding. Furthermore, they developed a method to predict the subjective difficulty of comprehension tasks based on the wave patterns collected by the EEG. [90].

Lee et al. studied the brain activity of programmers with EEG during program comprehension and found significant activation in the left frontal lobe [91]. In a follow-up study, Lee et al. used additionally eye tracking analysis. Using both techniques the authors could successfully predict task difficulty and programmer expertise [92].

Ishida et al. found a significant increase in the power of the alpha band when programmers finished a comprehension task [93]. Madeiros et al. used EEG to study the accuracy of code complexity metrics and found an insignificant correlation for all frequency bands [94].

## 2.2 Novice Programmers

The comprehension process in novice programmers has been studied since the 1970s. Soloway and Sopher examined different aspects of novice programming in their work titled: "Studying the Novice Programmer" [95]. Similarly, Hoc and Sheil reviewed methodological issues in introductory programming courses [96]. Additionally, Robins and Rountree [97] and Pears et al. [98] studied specific cognitive strategies, difficulties, and misconceptions in novices.

A common method found in the literature to study novice programmers is to compare them with experts. Soloway and Spohrer found deficits in the novices' understanding of language constructs such as loops, arrays, and recursion, as well as incapacity to plan and test code in comparison with experts [95]. Similarly, Winslow mentions that novices have superficial knowledge, lack appropriate mental models, and have an inefficient "line by line" approach to programming. The author also states that it takes around ten years for a novice to become an expert programmer [99].

According to Brown and Altadmrie, the most frequent mistakes made by novices are the method invocation with wrong arguments, unbalanced parentheses, and return statements [100]. Lishinki et al., state that the problem-solving abilities of students correlate with their performance in programming courses [101]. Similarly, Barlow-Jones and der Westhuizen found a strong correlation between logical and numerical reasoning and performance in introductory programming courses [102].

Ducasse et al. remark that novices spend most of their time debugging during programming, however, this activity is still difficult for them due to their limited knowledge [103]. Moreover, novices have difficulties understanding their programs when they do not follow the problem specification. Additionally, bug-finding and fixing activities take longer for novices as they cannot comprehend easily compiler error messages.

Despite the extensive research in this area, there is no consistent way to define novice programmers. Instead, different characterizations are used in the literature to describe and classify novice programmers. These methods are presented as follows:

The number of years has been used extensively to discern novices from experts. Lister, states that novices are programmers who have been actively learning programming for 3 or 4 years [104]. Sillito et al., consider instead the number of years an individual has been programming professionally. Subjects with less than 2 years of professional programming experience are considered novices [105].

Some researchers assess the educational background of subjects to categorize novices. This includes the education level i.e., undergraduates and graduates, grades of assignments or tests, number of programming languages they are familiar with, or how many programming courses they have taken. For example, Ricca et al., classified undergraduates as novices and graduates as experts [106].

The size of programs written by subjects is another metric used to classify novices. Subjects that have participated in projects writing more code have a higher programming ability than those who have written less code. For example, Muller used the number of lines of code of the largest program written to classify programmers. Programmers with the lowest expertise have written programs up to 500 lines of code [107].

Self-estimation has been proposed as an appropriate metric to measure programming experience. For example, Kleinschmager et al. [108], and Feigenspan et al. [109] found a correlation between the individuals' self-estimation and their performance in comprehension tasks and programming courses. Bunse used a five-point scale to classify programmers' expertise. Individuals in the first and second points of the scale were considered novices.

Some studies used a pretest to evaluate the subject's programming experience. For example, Biffl et al., [110] used a pretest to create three groups of different programming skill levels (novices, intermediates, and experts).
Finally, a supervisor is included in the study to estimate the experience of the subjects. Gallis Hans et al., used this technique to classify participants into novices, intermediates, and experts [111].

## 2.3 Related Work

Flowcharts have been used in computer programming since the 1940s. Many books are entirely dedicated to flowcharting and standards have been proposed due to their extensive usage. Flowcharts are a graphic representation through which the different operations that make up an algorithm or part of it are represented, establishing their chronological sequence. A flowchart is composed of boxes of different shapes to denote different types of operations. Arrows are used then to connect these shapes; the direction of the arrow denotes the flow of execution. The purpose of flowcharts is to facilitate the reading, understanding, processing, and memorizing of the information presented [15].

In software engineering, flowcharts have been extensively used due to their advantages. First, errors or omissions can be detected easier from a flowchart than from source code. Second, the flow of a flowchart can be understood easily and quickly. Third, it is a useful type of documentation, especially in the case of modifications [112].

Flowcharts could be effective tools for computer science students when writing and understanding algorithms. In the literature, there are studies that both support and contradict this statement. For instance, Schneiderman et al., conducted a controlled experiment to evaluate the advantages of flowcharts during program comprehension. In this investigation, one group of students examined code snippets, and another studied only flowcharts instead. The results indicated that there was no significant difference in their performance in terms of correctness [113].

Scanlan conducted a study to also evaluate the benefits of flowcharts. 82 students of computer science were separated into two groups. One group used pseudocode and the other only flowcharts to comprehend different algorithms. It was found that the group studying flowcharts: took less time to comprehend the algorithms, had fewer errors, had more confidence in their understanding, and referred fewer times to the algorithm [114].

Hendrix studied the effectiveness of control structure diagrams in code comprehension. In a similar setting as prior experiments, a group of 94 students were equally divided into two groups. One group studied source code only and the another one the source code rendered with control structure diagrams. The second group performed significantly better in terms of response time and correctness [115].

Cabo evaluated the effectiveness of flowcharting as a tool to learn Python. It was found that the ability of students to solve problems using flowcharts is a good predictor of their capacity to solve problems with Python. The majority of students found flowcharting easier than python and reported that it helped them to understand how to write programs in this programming language [116].

As computer science acquired significant popularity with technological developments, the need for more effective teaching tools also increased. From the 90s, some programming environments based on flowcharts and other types of visualization tools were developed. For example, Calloni and Bagert created BACCII and BACCII++ to support the teaching of procedural and object-oriented programming courses at Texas Tech University. Using these tools students can create flowcharts with a syntax-directed interface that is composed of a palette containing shapes representing different statements. The tool makes all necessary connections and allows the insertion of only correct statements. A study was performed at the end of an introductory programming course to evaluate empirically the tool. Students using BACCII and BACCII++ achieved significantly higher grades on their exams and assignments. Furthermore, more uniform learning was achieved as the variance of grades was reduced when the tool was employed [117].

Crews and Ziegler developed FLINT, also known as the flowchart interpreter. It was created at Western Kentucky University to support the introductory programming course. With this tool, students can create structured and syntactically correct flowcharts through the use of a builder interface. The program allows the execution of algorithms in a step by a step manner and saves the information for later review by the instructor. A study was performed to compare empirically flowcharts with programs written in Qbasic during code comprehension. It was found that students needed less time to comprehend structured flowcharts, performed fewer errors, had greater confidence, and referred fewer times to the flowcharts [118]. These results are in conformance with those of Scanlan.

Hooshyar et al, created FITS, an intelligent tutoring system based on flowcharts, where students can learn programming concepts and improve their problem-solving abilities. Students can navigate learning materials and create programs in a flowchart development environment. Using Bayesian Networks, the system can mimic a human's tutor action adapting itself to the needs and deficiencies of the students. In a controlled experiment, students using FITS experienced a significant improvement in their problem-solving abilities compared to the control group, which did not use the tool. Furthermore, the tool has shown to be considerably effective for students with different levels of prior knowledge [119].

Cilliers et al., created B#, a visual environment that was developed at Nelson Mandela Metropolitan University to support the introductory programming course. B# incorporates an iconic language and a programming environment. In a 9-week study, the performance of 59 students was evaluated. Half of the participants used PASCAL and the other half both textual and visual representations. The study found that the performance of students during the course improved significantly with the inclusion of the visual representation in addition to the text [120].

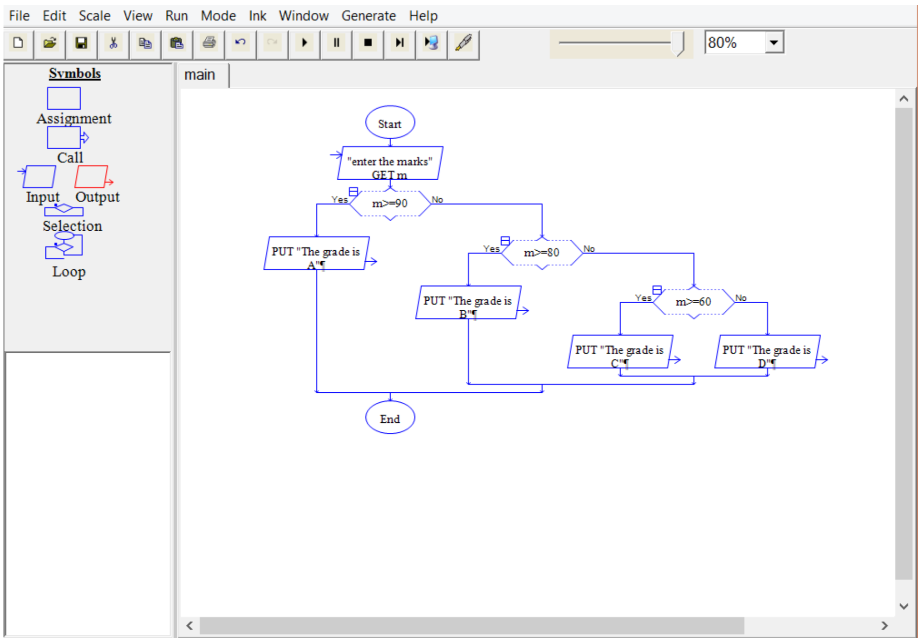


**Figure 2.12 The Flowchart Interpreter [118].**

Carlisle et al., developed RAPTOR. It is defined as a visual programming environment that supports students in understanding imperative and object-oriented programming techniques. Users choose from a palette the appropriate symbol to build the desired flowchart. RAPTOR alerts whenever an error is committed so the programs should be fixed right away. This tool generates automatically the corresponding source code in C#, C++ and Java. A study evaluated the benefits of this tool during the teaching of programming courses for 3 consecutive semesters. It was found that RAPTOR helps in developing problem-solving capabilities and understanding the flow of control of algorithms [121].

Marcelino et al., created SICAS, which is another teaching tool that was designed to support students in introductory programming courses. Based on constructivist theories, SICAS aims to enhance the problem-solving skills of students. The student can create and validate the created flowchart if the teacher provides a set of input/output data. The flowchart can be translated automatically into pseudo-code, C, or Java. The tool was further developed into a collaborative version called SICAS-COL and into an application for mobile devices. There was no empirical study associated [122].

Watts created SFC Editor (Structured Flow Chart Editor). This is a development environment for algorithms that was developed at Sonoma State University. Students can select different structures and create flowcharts with the necessary data. Additionally, the corresponding pseudo-code is automatically generated and flowcharts can be saved and shared across users [123].

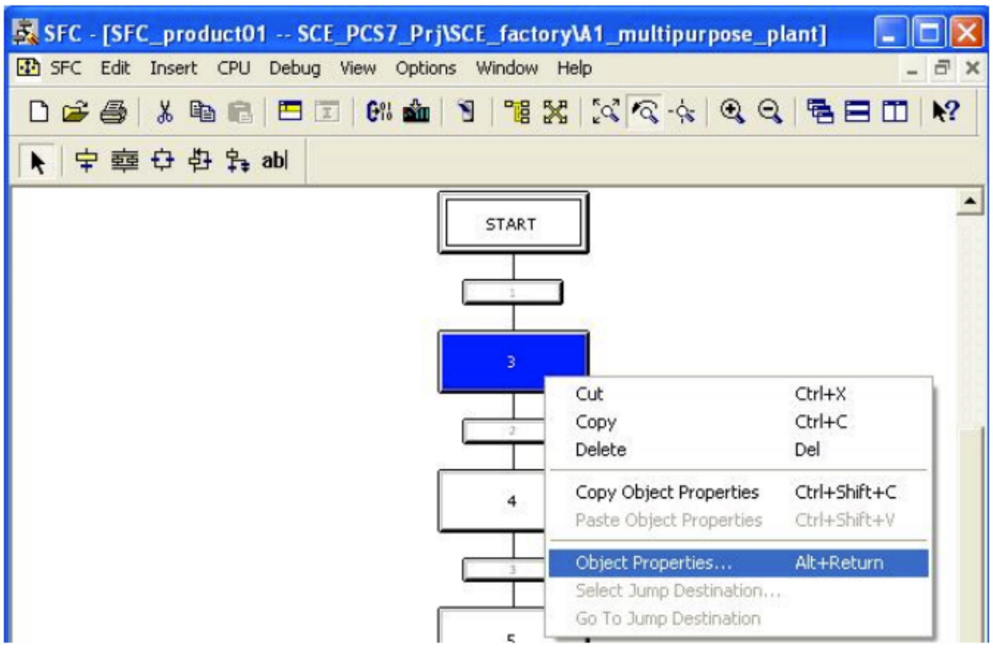

SFC - [SFC_product01 -- SCE_PCS7_Prj\SCE_factory\A1_multipurpose_plant]
SFC Edit Insert CPU Debug View Options Window Help
abl
START
3
4
5
Cut Ctrl+X
Copy Ctrl+C
Delete Del
Copy Object Properties Ctrl+Shift+C
Paste Object Properties Ctrl+Shift+V
Object Properties... Alt+Return
Select Jump Destination...
Go To Jump Destination

**Figure 2.13 Structured Flowchart Editor [123].**

Areias et al., developed ProGuide, another flowchart-based environment that guides students in solving programming problems. The student can create flowcharts for the proposed problem, the tool will present the student with warnings, hints, and examples whenever possible [124].

Progranimate is a web e-learning platform for creating and executing flowcharts. The platform allows the synchronized execution of the flowchart and the corresponding source code. The tool was evaluated with high school and university undergraduate students showing positive results [125].

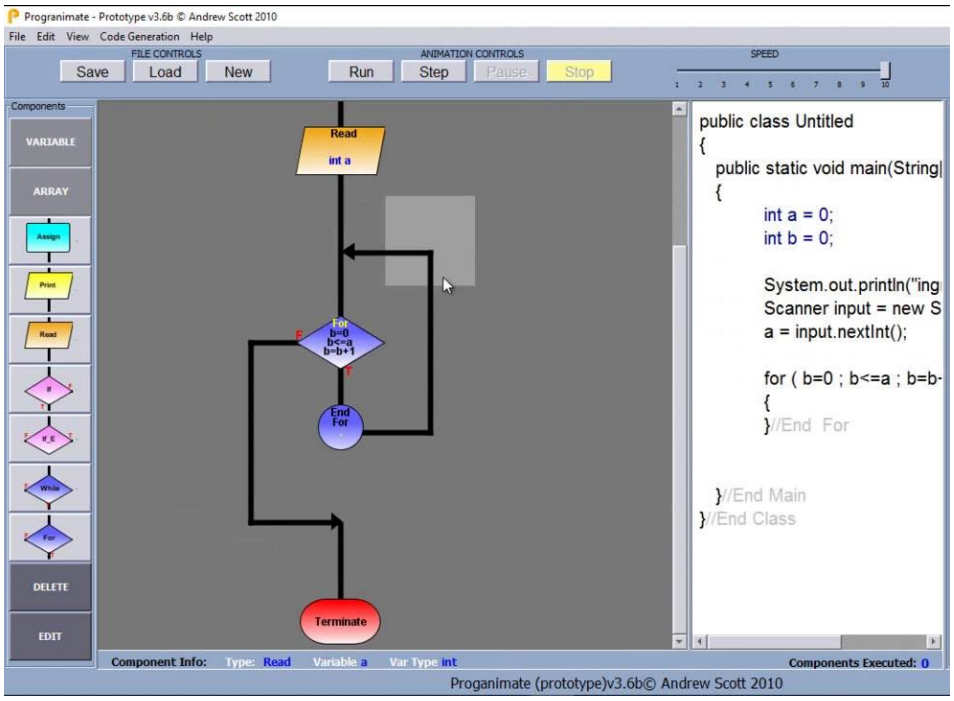
Iconic Programmer is another tool that allows the development of programs with flowcharts. Tracing, debugging, and automatic generation of code in Java and C are also supported [126].

Other forms of programming visualization like visual programming languages have appeared to facilitate the comprehension of algorithms. Green and Petre evaluated View, a popular visual programming language in a quantitative study. The study evaluated the code comprehension of experienced programmers with C and View programming languages. It was found that in most cases it was more difficult to understand algorithms when they were represented by the visual programming language [127].

Whitley et al. examined the comprehensibility of LabVIEW, a visual programming language, with a semantically equivalent textual language. Three types of tasks were used: Tracing, parallelism, and debugging. The results show that subjects using the visual programming language have a better performance in parallelism and debugging tasks. However, participants using the textual representations performed better with tracing tasks [128].

Algorithm animation is another technique where users can visualize the behavior of an algorithm based on its flow and operations. Crosby and Stelovsky studied algorithm animations in an educational context. The researchers conducted an investigation where a group of students interacted with algorithm animations and another group had a regular algorithm lecture. It was found that the students who studied algorithms through animation performed better in comprehension tasks afterward [129].}

Hansen et al. [130] found that students that learned an algorithm using algorithm animation had greater performance than students who learned the same algorithms using textual representations. Similarly, Stasko et al [131] performed a study where one group of students learned data structures with algorithm animation and another by studying standard textual materials. The animation group performed better in the post-test but the difference was not statistically significant. Lawrence [132] also performed a similar study where students learned different algorithms through both visual and text methods. No statistical significant difference was found between the groups.



```
public class Untitled
{
  public static void main(String|
  {
        int a = 0;
        int b = 0;

        System.out.println("ing
        Scanner input = new S
        a = input.nextInt();

        for ( b=0 ; b<=a ; b=b-
        {
        }//End  For


  }//End Main
}//End Class
```

Figure 2.14 Programimate [125].

Price et al., [133] studied the effect of algorithm animation in debugging. During an experiment, two groups of students debugged the same program. One group used algorithm animation and another standard code. The study did not find significant differences between the two groups in debugging tasks. Similarly, Mulholland [134] compared a graphical tracer with a textual tracer empirically, participants that used the graphical tracer were able to solve fewer tasks than the other group.

Heijstek et al., evaluated graphical representations in software architecture. During a study, 47 participants examined both visual and textual artifacts representing software architecture design. The study did not find that any representation is significantly more effective in this area. Additionally, experienced programmers prefer textual representations [135].

Dzidek et al., studied the costs and benefits of using UML in software engineering. The authors conducted a study with professional programmers. It was found that UML has a great impact on the maintenance of software systems. However, its use does not significantly increase the time required to perform maintenance tasks [136].

BPMN (Business process modeling notation) is used to represent service-oriented computing processes. Birkmeier et al. studied the effectiveness of BPMN in comparison with UML Activity diagrams. The results of the study indicate that the languages have no differences in effectiveness or subjective preference [137].

# 3 Methodology

A controlled experiment is the selected method to evaluate the effect of flowcharts on the program comprehension of novices. This is a systematic research approach, in which one or more independent variables are varied while keeping external factors constant. The effects of the variation over one or more dependent variables are observed and analyzed [138].

As a starting point, the goal, variables, and hypotheses of the study are defined. The hypotheses will state the possible relationships between variables. The definition of these parameters will influence the experiment design and subsequent phases of the study. The characterization of variables and hypotheses is also known as operationalization as a set of suitable operations is chosen to collect, measure and analyze the data and test the hypotheses [138].

## 3.1 Goal

Section 1.1 highlighted that problem-solving is one of the key skills that novices lack during introductory programming courses. This refers to the ability to transition from a problem specification to a working algorithm. In this process, mental models are crucial and students who cannot develop appropriate mental models are significantly disadvantaged. These students will find introductory programming courses tedious and difficult and are likely to drop out. Therefore, training in problem-solving skills and the development of mental models should be an important part of any introductory programming course [10].

Visualization techniques such as flowcharts could provide novices with an appropriate mental model of the programming concepts and structures that compose an algorithm. They could be used as an effective teaching tool due to their small learning curve and simplicity. Flowcharts focus on control structures such as selection and iteration and emphasize program composition and the flow of execution. This allows the novice to concentrate on the problem while minimizing the impact of complex programming syntaxes [25].

This thesis studies the effect of flowcharts on the program comprehension of novices. The results of the present thesis could be a useful starting point to find suitable flowchart-based teaching methods that potentially alleviate the difficulties of novices in introductory programming courses. The research question is stated as follows:

*RQ: What is the impact of flowcharts on the program comprehension of novices?*

## 3.2 Independent Variable

Independent variables are intentionally varied and their influence over dependent variables is studied and analyzed. The independent variables are altered according to some predefined levels which are also referred to as treatments [138]. In this study, the independent variable is the presence of flowcharts during the comprehension process. That is, whether a flowchart is present or not in conjunction with the semantically equivalent source code.

## 3.3 Dependent Variable

Dependent variables are the outcome of a study. Their behavior will depend on variations of the independent variable [138]. In this study, the dependent variable is the impact of program comprehension. For this, the following measuring factors have been selected:

- Visual Attention: Operationalized as fixation time over the stimulus.
- Correctness: The ratio of the number of correct answers and total answers.
- Response Time: Total time that a participant needs to study an algorithm to submit an answer.
- Mental Workload: Operationalized as the theta-to-alpha ratio of an EEG signal.
- Subjective Preference: Whether participants prefer flowcharts in addition to code snippets.

## 3.4 Hypotheses

In order to study the impact of flowcharts on the code comprehension of novices, the first step is to evaluate whether the participants actually refer to flowcharts. Next, the impact of flowcharts on the measuring factors of the dependent variable is analyzed. As found in the literature, flowcharts could be very effective for both writing and comprehending algorithms. Therefore, the following hypotheses are stated:

***H1:*** *Participants refer to flowcharts in addition to code snippets.*
***Ho1:*** *Participants do not refer to flowcharts in addition to code snippets.*

***H2:*** *Participants using flowcharts take less time to complete the comprehension tasks.*
***Ho2:*** *There is no significant difference in response time of comprehension tasks due to the use of flowcharts.*

***H3:** Participants using flowcharts answer with a higher correction rate.*
***Ho3:** There is no significant difference in correctness of comprehension tasks due to the use of flowcharts.*

***H4:** Participants using flowcharts have a lower cognitive load during the comprehension tasks.*
***Ho4:** There is no significant difference in cognitive load during comprehension tasks due to the use of flowcharts.*

***H5:** Participants prefer flowcharts in addition to code snippets.*
***Ho5:** Participants do not prefer flowcharts in addition to code snippets.*

Hypotheses 1-4 are evaluated quantitively. The fifth hypothesis is evaluated qualitatively through a post-interview as indicated in section 3.7.3.

## 3.5 Participants

This study focuses on the program comprehension process of novices. However, there is no consensus on the specific traits that novice programmers have. Instead, several definitions have been proposed in the literature considering different aspects such as level of education, number of years programming, size of programming projects, or grades as noted in section 2.2. This study considers novice programmers as a mixture of all these definitions in order to facilitate the recruitment process. A pre-questionnaire is used to evaluate the programming experience of the participants and to ensure that they can be considered novices according to at least one definition [139].

The information on academic programming experience, professional programming experience, and knowledge of programming languages and paradigms are collected along with demographic data. Self-estimation is also used to measure programming experience. The participants are asked to estimate their programming skills compared with their coursemates and professionals with 20 years of experience.

Basic programming knowledge is required to participate in the study. This includes control structures, arrays, variables, and the basic syntax of the java programming language. Participants should be older than 18 and have at least 1 hour of time availability.

To recruit participants, a description of the study is posted in Opal forums and social media groups of different modules held by the Computer Science Faculty of Chemnitz Technology University. The participants are offered 10 Euro compensation for their participation independent of their performance. The anonymity of the results is also ensured. The invitation includes a link to reserve a spot according to the participants' availability. The demographics of the participants are presented in section 4.1.

## 3.6 Confounding Factors

Confounding factors are external unwanted variables that could also influence dependent variables besides the independent variables [138]. These factors should be controlled using suitable methods to minimize bias. A comprehensive list of confounding factors for experiments in software engineering is defined in [140]. The relevant confounding factors for the present study and related control techniques are described as follows:

- Program experience is a major confounding factor in code comprehension studies. In [141], Falessi et al. stress that recent and relevant experience has a great impact on comprehension tasks. In the present experiment, only participants that comply with at least one definition of novice programmer in section 2.2 are considered. A pre-questionnaire is used to verify this requirement.

- Evaluation apprehension can be present in the experiment if the students consider that their performance may affect their grades [140]. To control this confounding factor, participants are told that the results are anonymous and their performance has no effect on course marks or the compensation money.

- Fatigue and Mortality are important factors that are related to the loss of concentration and the unwillingness to complete tasks [140]. To avoid their effect, a single short session is designed.

- Process conformance is needed to avoid biased results [140]. Participants are instructed to complete the tasks without any help and in an uninterrupted manner. Additionally, the time that participants needed to complete the tasks is tracked. If this time is too long or too short compared with the results of other participants due to distractions or other reasons, the corresponding results are discarded.

- Mono-method bias happens when only one measure is used to quantify the variable [140]. Both the response time and correctness of answers are considered for the analysis of data.

- Selection refers to the procedure used to select participants [140]. This procedure was explained in the previous section.

- Study Object Coverage confounding factor is present when participants do not complete the whole experiment [140]. Incomplete task results are excluded from the analyzed data.

- Learning and Ordering effects refer to how the order of tasks influences the results. Participants could learn in early tasks and show a different performance in final tasks [140]. To exclude this effect, the order of tasks is randomized. Moreover, mock tasks are presented at the beginning of the experiment so participants get used to the experimental setting.

- The size of the Study Object and Tasks refers to the material used in the study [140]. These are presented in following section.

- Interindividual differences are always present in controlled experiments. They refer to the variation between individuals in traits, behaviors, or characteristics [142]. This confounding factor is controlled by using a within-subject design.

- Intelligence refers to the ability to solve problems. As the age of a person is strongly related to the participants [143], a pre-questionnaire is designed to avoid big differences in the population age.

## 3.7 Experiment Material

The material for the present study consists of code snippets, pre-questionnaire, and post-questionnaire. This material is described in the following sections.

### *3.7.1 Code Snippet Selection*

Selected code snippets have a great influence on the participants' performance and behavior in the study. To establish appropriate selection criteria, it is necessary to have a deep understanding of the capabilities and knowledge of the participants and the goal of the study [144]. The following are the criteria considered for code snippet selection:

- The code snippets should not include domain knowledge. As the participants are novice programmers, they could skip code snippets that require specific knowledge or experience such as advanced methods or classes. This situation could increase greatly the response time, participants could lose interest and the experiment data would be insufficient to answer the research question [144].

- The code snippets should be challenging enough so that participants are likely to study the algorithm using both the source code and flowcharts. If code snippets are trivial participants could complete the comprehension tasks easily by looking only at the source code. Challenging code snippets are likely to generate useful data that allows the analysis of the usage of flowcharts by novices.

- The code snippets should enforce the bottom-up strategy. The presence of beacons in code snippets may facilitate significantly the comprehension, generating bias in the results [145].

- The code snippets should include the concepts that are taught in introductory programming courses i.e., control and conditional structures. Advanced programming concepts could confuse the participants.

- The code snippets should have already been used as material in other studies. This will ensure that the snippets are suitable for a program comprehension study. New code snippets require to be tested thoroughly in pilot studies requiring more time and participants.

Using these criteria, 14 code snippets are selected. These materials are suitable to find the impact of flowcharts in code comprehension of novices.

### *3.7.2 Pre-questionnaire*

A pre-questionnaire is created to measure programming experience and collect demographic information [139]. It is also necessary to control the experience of the participants with flowcharts, as individuals with more experience are likely to perform better compared with inexperienced subjects. The following questionnaire is then implemented:

- How long have you been programming for educational purposes? (In years):
- How long have you been programming professionally? (In years):
- How do you estimate your programming experience on a scale from 1 to 10?
- How do you estimate your programming experience compared to fellow students and experts with 20 years of a practical experience on a scale from 1 to 5?
- How do you estimate your experience using flowcharts to visualize or design a program on a scale from 1 to 5?
- How experienced are you with the following programming languages on a scale from 1 to 5? Java, C, Python, JavaScript.
- How many additional programming languages do you have moderate experience with?
- How experienced are you with the following programming paradigms on a scale from 1 to 5? Functional programming, Object-oriented programming, Imperative programming, Logical programming

### *3.7.3 Post-questionnaire*

Post questionnaires are an important part of the study as they give insight into how the participants solved the tasks. These questions help to discover patterns, preferences, or strategies regarding the use of flowcharts during the comprehension process of novices [138]. This short interview starts with questions about how the comprehension tasks were solved and whether any strategy was followed. After that, a discussion about the advantages or disadvantages of flowcharts takes place. The post-questionnaire questions are stated as follows:

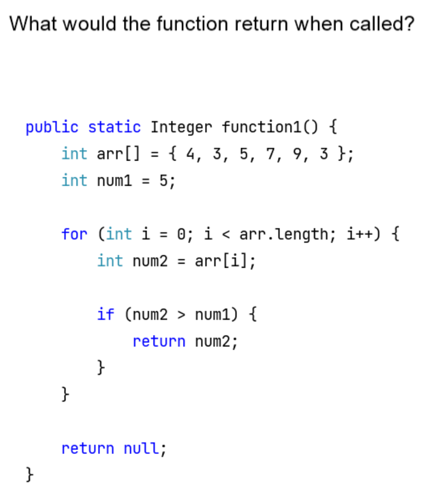
- How did you solve the tasks?
- Did you use a specific strategy to solve the tasks?
- How much did you refer to the flowcharts and how much to the code?
- For which task did you spend more time, when there was a flowchart present or not?
- Do you prefer tasks where the flowchart was present or not?
- Do you think there is an advantage or disadvantage when the flowchart is included along with the code? Why?

## 3.8 Tasks

During comprehension tasks, the participants are challenged to understand some code. If the subject claims to have achieved the necessary understanding, this should be proved by performing correctly some tasks. Different comprehension tasks, such as correction, debugging, and coding, have been used in literature. The comprehension task should be aligned with the aim and the resources available in the study [144].

An output task is selected as it allows a fast measurement and data collection without compromising internal validity [144]. In this study, the participants are given code snippets in conjunction with flowcharts. The participants are asked to determine the output of the presented algorithm. Variable obfuscation is used to minimize beacons [145]; thus, the bottom-up approach is enforced. The comprehension task can be summarized as follows:

- Each comprehension task is composed of two slides. On the first slide, the participant sees a code snippet and possibly a flowchart, and is asked to determine the output. The code snippet and the flowchart will contain the input and all information needed to solve the task as illustrated in Figure 3.1.


What would the function return when called?

```java
public static Integer function1() {
    int arr[] = { 4, 3, 5, 7, 9, 3 };
    int num1 = 5;

    for (int i = 0; i < arr.length; i++) {
        int num2 = arr[i];

        if (num2 > num1) {
            return num2;
        }
    }

    return null;
}
```

**Figure 3.1 First slide of the comprehension task.**

- On the second screen, the participant will see four options and has to select the correct answer. If the participant does not know the answer, a skip option is available as illustrated in Figure 3.2. This will guarantee unbiased results.

**Figure 3.2 Second slide of the comprehension task.**

As noted before, a flowchart may be present in conjunction with the code snippet. The design of flowcharts must not introduce additional bias. For this, the following criteria are considered regarding the flowchart aesthetics:

- Only the standard symbols should be used in program flowcharts. These are illustrated in Table 3.1.
- The program logic should depict the flow from top to bottom and from left to right.
- Each symbol used in a flowchart should contain only one entry point and one exit point, with the exception of the decision symbol.
- All decision branches should be well-labeled.
- The flowcharts should include the exact same text as the corresponding code snippet. This includes type annotations (int), semicolons (;), and methods (System.print.out).
- The syntax highlighting is kept in both the source code and the flowcharts. Syntax highlighting is used in major IDEs and participants would behave differently if this is removed from the code snippets.
- The borders of the shapes in flowcharts have the same color as the syntax highlighting in the source code. In this way, a color balance between flowcharts and code snippets is achieved.
- The font sizes and line spacing are chosen to facilitate the analysis of the visual attention data. The font size should be also comfortable to read.

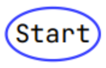
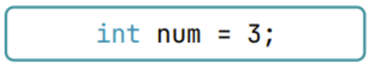
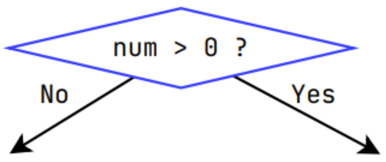
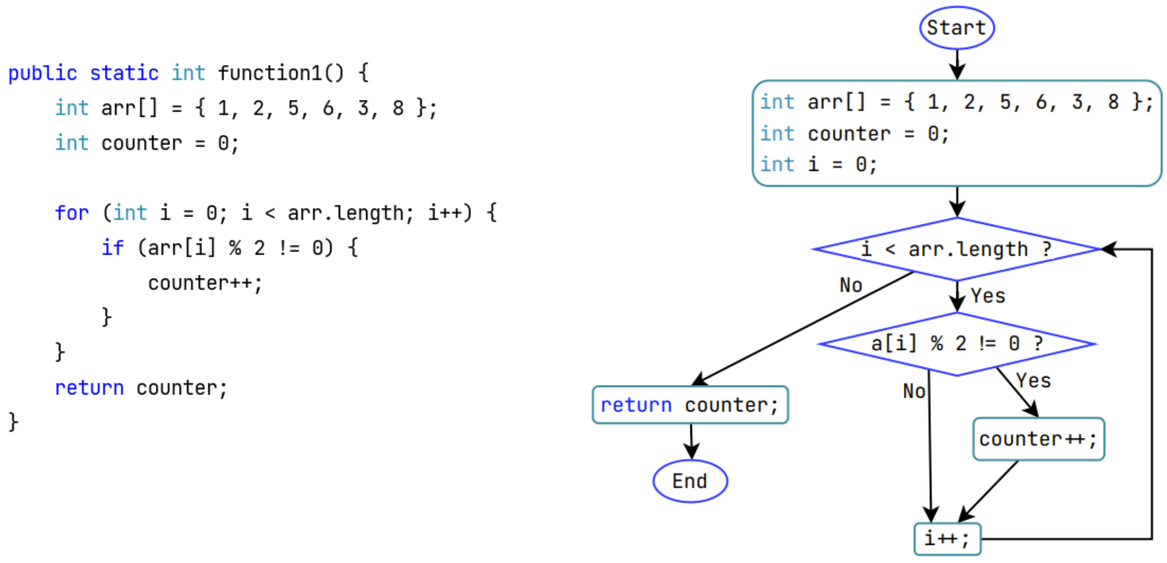
| Shape | Meaning |
|---|---|
|  | Oval: Represents the starts or the end of a flowchart |
|  | Rectangle: Used for arithmetic operations and data manipulations. |
|  | Rhombus: Used for decision making between two or more alternatives |

**Table 3.1 Flowchart Standard Symbols**

Following these criteria, the flowchart design selected is illustrated in Figure 3.3. This design will potentially minimize bias and allow the collection of useful visual attention data.



```
public static int function1() {
    int arr[] = { 1, 2, 5, 6, 3, 8 };
    int counter = 0;

    for (int i = 0; i < arr.length; i++) {
        if (arr[i] % 2 != 0) {
            counter++;
        }
    }
    return counter;
}
```

**Figure 3.3 Comprehension task with Flowchart.**

## 3.9 Experiment Design

The within-subjects design is used in the experiment for the following reasons: First, it requires smaller samples as compared with a between-subject design. Second, it is statistically powerful, which means it can detect small influences on the population. Finally, it excludes the effect of individual differences which represents an important confounding factor; this aspect is hard to control and large samples are required to mitigate its effect in other designs [146].

As the within-subject design is chosen, participants will perform both comprehension tasks with code snippets in conjunction with flowcharts and code snippets alone. The design is described as follows:

- There will be a total of 14 comprehension tasks, 8 comprehension tasks with code snippets in addition to flowcharts and 8 with code snippets alone.
- The participants will have 30 minutes of experiment time to complete as many tasks as possible.
- The comprehension tasks with and without flowcharts are alternated across the experiment.
- The order of the code snippets selected in section 3.7.1. is randomized across participants to avoid order and learning effects.
- After each comprehension task, participants have a 10-sec cross fixation task rest.
- A mock task is also used at the beginning of the session to mitigate learning effects.

## 3.10 Tools

The equipment and software selected to conduct the experiment are described in the following sections.

### *3.10.1 Eye Tracker*

Eye tracking refers to the process of measuring where participants are looking, also known as the point where the gaze is fixed. These measurements are carried out by an eye tracker, which records the position of the eyes and the movements participants make [147].

The selected Eye Tracker is the Tobii Pro X3-120 EPU. This device uses invisible infrared light and high-definition cameras to project light into the eye and record the direction it reflects off the cornea. Advanced algorithms are then used to calculate the position of the eye and determine exactly where to focus. This makes it possible to measure and study visual behavior and fine eye movements since the position of the eye can be mapped 120 times per second [148].

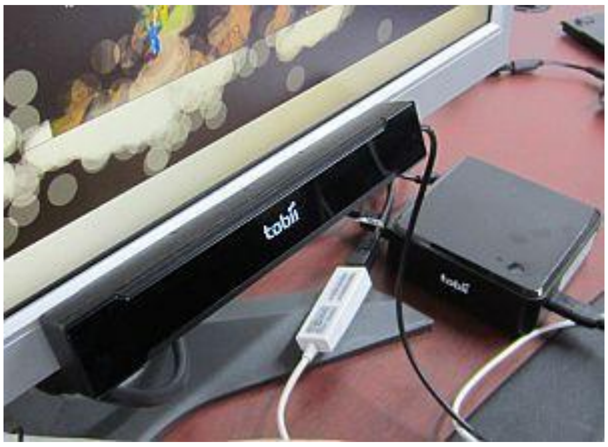


**Figure 3.4 Tobii eye tracker [148].**

The eye tracker is positioned below the stimulus screen as illustrated in Figure 3.4. The resolution of the monitor is 1920 × 1080 pixels, with a physical screen size of 51.1 × 28.7 cm. The distance between the participant and the screen is approximately 60 cm and the distance to the eye tracker is approximately 65 cm. Communication with the eye tracker is achieved using the Tobii SDK controlled through Python running on Windows 10.

### *3.10.2 EEG*

The device selected for the recording of EEG data is the CGX Quick-20r. This is a wireless EEG headset that uses dry sensors as illustrated in Figure 3.5. The channels are positioned according to a 10-20 montage. Wireless technology allows the subject to move freely while real-time data is collected. The headset obtains high-quality EEG data with no complicated preparation. The device uses different mechanisms and replaceable dry sensors to adjust to various scalp shapes and sizes, maintaining electrodes' position in a standard montage. The EEG signal is sampled with a frequency of 1000Hz and converted to digital data at 24 bits of resolution [149].

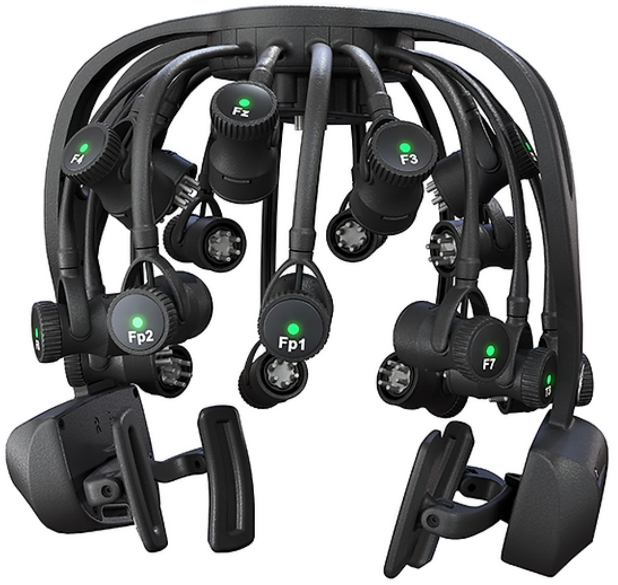


**Figure 3.5 CGX EEG Headset [149].**

### *3.10.3 PsychoPy*

Psychopy is an open-source software written in Python language to present stimuli and collect participants' input. This software uses OpenGL graphics to generate a stimulus of high visual precision. Psycophy is used to generate a full-screen presentation where the code snippets and flowcharts are presented. Responses are gathered via the keyboard. Participants can move between options with the arrow keys and select the answer by pressing the spacebar [150].

# 4 Conduct

The implementation of the present study is described in the following chapter. The individual steps that had to be carried out for execution are explained. In addition, the participants in the study and the collection of the data are described.

## 4.1 Participants Demographics

A total of 11 participants initiated and finished the test. The participants are students of Computer Science-related programs of the Technology University Chemnitz. Participants have 3 to 5 years of academic programming experience. Only two participants had professional programming experience of up to 1.5 years. Participants also self-estimated their programming experience according to classmates and experts with 20 years of experience. Most of the participants stated a moderate programming experience compared to classmates and insufficient programming experience compared to experts. Additionally, participants also estimated their java programming experience to be moderate. Additionally, the experience of the participants with flowcharts was mostly vague or non-existent. The participants' demographics are summarized in Table 4.1.

| | |
|---|---:|
| **Male** | 7 |
| **Female** | 4 |
| **Age (in years)** | 26 ± 3 |
| **Learning Programming (in years)** | 4 ± 1 |
| **Professional Programming (in years)** | 0.5 ± 1 |
| **Java Programming (in years)** | 2 ± 1 |
| **Flowchart Experience** | 0.5 ± 0.5 |

**Table 4.1 Participants Demographics.**

## 4.2 Procedure

In this section, the detailed process of the study from the recruitment to the post-interview is described:

1. An invitation message is posted in the opal course forums and social media groups of different modules held by the Computer Science Faculty of Technical University Chemnitz. The invitation states that the participation is voluntary, that it will not affect course grade, and that 10 euros are given as compensation.

2. The invitation contains a website link, where participants can choose a time slot according to their availability. The scheduling platform informs the participants that the study will last 1 hour and that is conducted on the university campus.

3. After participants have chosen a suitable time slot, an email is automatically sent containing a brief description of the experiment and general information. The email contains a link to the pre-questionnaire.

4. The participants fill in the pre-questionnaire used to collect demographics and programming experience-related information.

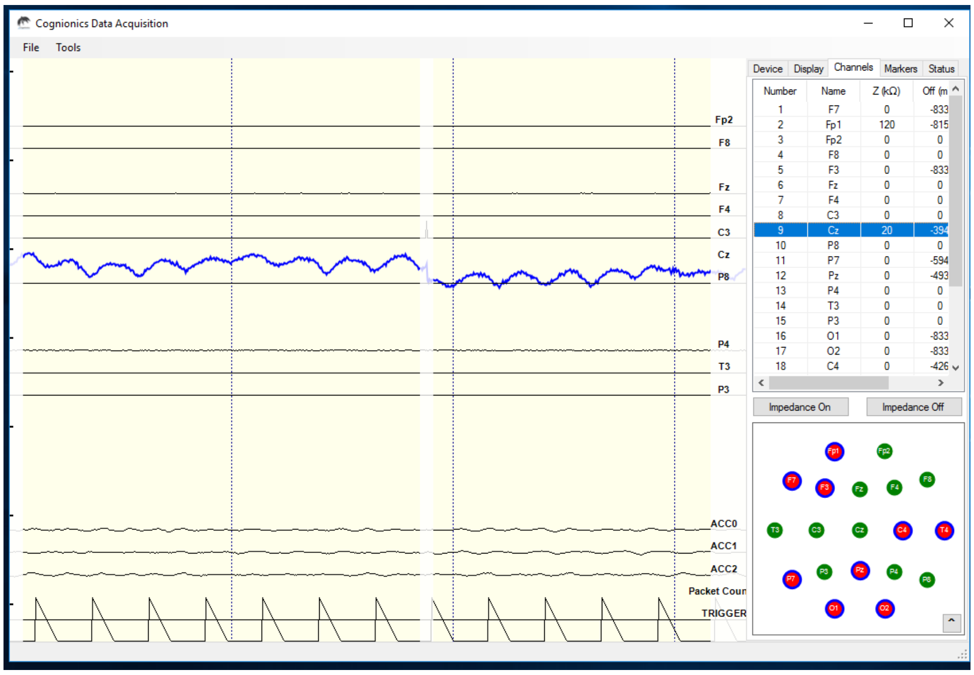
5. The student is received in the lab and the following instructions are given:

   - The study will last approximately 30 min, during this time it is important to avoid any distractions and focus solely on the tasks.
   - The participant will be presented with some small programs written in Java. The task is to state the output of the presented program.
   - The participant should try to answer all tasks if possible. But any task can be skipped if the participant is unsure. This is important to get unbiased results.
   - The visual gaze data is captured with an eye tracker. A calibration process is needed to configure correctly the device.
   - An EEG device is also used to collect brain activity during the experiment. a calibration process is also needed and it will last approximately 5-10 minutes.
   - It is important that the participant stays as relaxed as possible and minimizes body movements to collect useful data.

6. The EEG is calibrated, for this, every electrode in the EEG cap should be individually positioned. The duration of this process depends on the calp size and shape. The software CGIX is used to visualize the impedances of each electrode. if the impedance is acceptable a green light will be shown as illustrated in Figure 4.1. Every electrode should have a suitable impedance to conclude this step.

7. The eye tracker is calibrated. First, the participant sits approximately 60 cm apart from the screen. Then, the monitor height is adapted to the participant. The program Eye-tracker Manager will show a green screen as illustrated in Figure 4.2 if the participant is correctly positioned. Then the calibration process starts. For this, the participant has to watch 12 points distributed on the screen. In the

end, the accuracy of the calibration is calculated and visualized. This process is repeated if the accuracy is low.



| Number | Name | Z (kΩ) | Off (m |
|---|---|---|---|
| 1 | F7 | 0 | -833 |
| 2 | Fp1 | 120 | -815 |
| 3 | Fp2 | 0 | 0 |
| 4 | F8 | 0 | 0 |
| 5 | F3 | 0 | -833 |
| 6 | Fz | 0 | 0 |
| 7 | F4 | 0 | 0 |
| 8 | C3 | 0 | 0 |
| 9 | Cz | 20 | -394 |
| 10 | P8 | 0 | 0 |
| 11 | P7 | 0 | -594 |
| 12 | Pz | 0 | -493 |
| 13 | P4 | 0 | 0 |
| 14 | T3 | 0 | 0 |
| 15 | P3 | 0 | 0 |
| 16 | O1 | 0 | -833 |
| 17 | O2 | 0 | -833 |
| 18 | C4 | 0 | -426 |

**Figure 4.1 EEG Signals and Calibration [150].**

8. The participant is ready to begin the experiment. The study presentation is projected on the screen. The instructions and general information is repeated on the first slides.

9. Some training tasks are presented, these consist of a brief explanation of flowcharts followed by mock tasks that allow the participant to familiarize themself with the experimental setting. Two mock tasks are used. One containing a code snippet, and the second one with both the code snippet and flowchart.

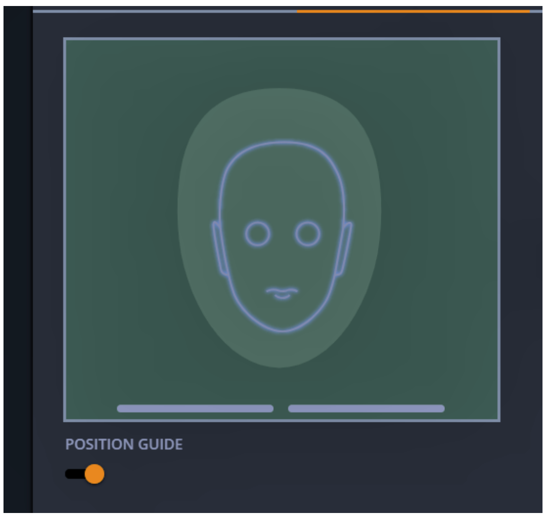


**Figure 4.2 Eyetracker Calibration [148].**

10. The participant starts the comprehension tasks. Between each comprehension task, a 10-sec cross-task is presented.

11. If the participant finishes the 14 tasks or if the 30 min time frame is over, the experiment ends. The EEG cap is removed from the participant.

12. The post-questionnaire interview is conducted. This questionnaire is necessary to gain better insights into the overall experience and to find out the preferences of the participants regarding the use of flowcharts.

13. The compensation is given to the participant and he/she leaves the lab.

## 4.3 Data Collection

After the study ends successfully, the data gathered has 3 formats:

1. The participants’ answers and response times are collected in CSV files. These files are generated by Psycophy software once the study ends.
2. The Visual attention data containing the x and y coordinates of both the left and right eyes are stored also in CSV files. These files are generated using the Tobii Eye Tracker SDK for Python.
3. The EEG data containing 19 channels that map to different regions of the brain is collected in the XDF format. The output stream of the CGX software is recorded and collected in real-time.

# 5 Data Analysis and Results

The present study's objective is to analyze the effect of flowcharts on the program comprehension of novices. Different measuring factors have been chosen for this purpose: visual attention, correctness, response time, cognitive load, and subjective preference. Data related to these measuring factors has been collected by conducting a within-subject controlled experiment. The preprocessing, characterization, and analysis of the data are presented in this chapter. Finally, the hypotheses are tested and the research question is answered.

## 5.1 Data Preparation

Data preparation includes the cleaning, processing, and transformation of data into a valuable format. This process will facilitate the evaluation of hypotheses by removing potential bias or irrelevant data that could interfere with the analysis.

### *5.1.1 Time and Correctness Data Processing*

The response time and answers of the participants are recorded in CSV files. The CSV files contain the columns participant, snippet, response time, and correctness. The correctness column can contain the values “yes” or “no” depending on whether the participant answered correctly or not. The participants had also the option to skip a certain comprehension task. In this case, the corresponding value in the column correctness will be “skip”. Python scripts are used to extract the data from the CSV files.

### *5.1.2 Eye Tracking Data Processing*

Eye Tracking data is recorded in CSV files containing the gaze position of participants during each comprehension task. The I2CM algorithm is used to identify the fixations of the participants from the raw stream of (x, y) coordinates [151]. AOIs are defined to relate the fixations to the source code region or flowchart region of the stimulus. The total fixation time over the code snippet and flowchart regions is calculated by summing up the respective values.

### *5.1.3 EEG Data Processing*

The EEG data processing is based on spectral analysis of the electrical brain activity. First, a 0.5-40 Hz bandpass filter is applied to remove noise and other unwanted signals. Next, an ICA process is performed to remove electrical components related to eye movements, muscular activity, or other unwanted artifacts [152]. Finally, the theta-to-alpha power ratio of the EEG signal is computed as it has a strong correlation with cognitive load [86].

### *5.1.4 Data Cleaning*

The data cleaning is performed over the results obtained in the previous steps. The observations containing a value of "skipped" in the correctness column are discarded. Additionally, values greater than 2 standard deviations from the mean are considered outliers and are removed. Comprehension tasks with response times shorter than 10 s are discarded as well. The final results are described in the following sections.

## 5.2 Descriptive Statistics

In this section different strategies such as tables, box-plots, bar charts, and percentages are used to describe the collected data. These methods summarize the most important features of the results and facilitate the evaluation of the hypotheses.

### *5.2.1 Eye Tracking Data Results*

The eye-tracking data is used to examine if participants referred to flowcharts during comprehension tasks. During the study, each participant completed comprehension tasks with code snippets only and with code snippets in conjunction with flowcharts. The total fixation time of the code snippet region and the flowchart part of the stimulus is calculated for each algorithm. Table 5.1 shows the distribution of the fixation times.

Participants had a mean fixation time of 104.95 s over code snippets alone. When flowcharts are present in addition to code snippets, participants fixated 90.65 s over code snippets and 30.98 s over flowcharts on average. The fixation time over code snippets is reduced by 14.3 s (13.62 %). Furthermore, the total fixation time over the stimulus increases by 16.67 s (15.88 %) when flowcharts are available.

| | *Code Alone* | | *Code in addition to Flowchart* | | | |
|---|---|---|---|---|---|---|
| Algorithm | Total Responses | Mean Fixation Total (s) | Total Responses | Mean Fixation Time Code (s) | Mean Fixation Time Flowchart (s) | Mean Fixation Total (s) |
| *concatlists* | 5 | 104.64 | 5 | 104.43 | 31.93 | 136.36 |
| *countEvenNumbers* | 5 | 80.81 | 5 | 71.2 | 23.95 | 95.15 |
| *crosssum* | 6 | 92.24 | 4 | 117.46 | 20.02 | 137.48 |
| *decimalToBinary* | 5 | 99.99 | 4 | 104.18 | 60.74 | 164.93 |
| *diffPosAndNegNumbers* | 4 | 102.09 | 5 | 94.89 | 16.2 | 111.08 |
| *dropNumber* | 4 | 174.51 | 6 | 101.8 | 58.49 | 160.29 |
| *findTheLargest* | 5 | 99.41 | 6 | 59.12 | 47.22 | 106.34 |
| *firstAboveThreshold* | 6 | 100.32 | 3 | 88.68 | 38.96 | 127.64 |
| *hindex* | 3 | 148.38 | 2 | 141.21 | 28.12 | 169.33 |
| *integertoString* | 3 | 127.9 | 5 | 122.15 | 33.42 | 155.58 |
| *isPrime* | 4 | 59.44 | 4 | 43.22 | 22.24 | 65.46 |
| *multiplicationByAdding* | 5 | 79.58 | 5 | 81.67 | 3.75 | 85.42 |
| *removeDoubleCharacters* | 4 | 114.02 | 4 | 100.44 | 28.43 | 128.87 |
| *sumOfIntervalNumbers* | 5 | 122.03 | 4 | 68.28 | 10.33 | 78.61 |
| *TOTAL* | 64 | 104.95 | 62 | 90.65 | 30.98 | 121.62 |

**Table 5.1 Fixation Time Results**

To further analyze the collected data, it is necessary to decide for each observation if the flowchart was actually used. For this, a minimum threshold of 10% of the total fixation time is defined. If a participant fixated more than the defined threshold over a flowchart, it is considered that the flowchart was used. Table 5.2 shows the number of comprehension tasks where flowcharts were used and the fixation time distribution for each participant.

| | *Code Alone* | | *Code with Flowcharts* | | | |
|---|---|---|---|---|---|---|
| Participant | Total Responses | Mean Fixation Time (s) | Total Responses | No. of tasks where Flowcharts where used | Mean Fixation Time Code (s) | Mean Fixation Time Flowchart (s) |
| *Participant 1* | 3 | 148.65 | 4 | 4 | 140.54 | 25.53 |
| *Participant 2* | 5 | 69.58 | 6 | 4 | 95.99 | 44.48 |
| *Participant 3* | 7 | 105.17 | 7 | 3 | 96.72 | 14.45 |
| *Participant 4* | 4 | 163.34 | 3 | 3 | 141.38 | 61.83 |
| *Participant 5* | 7 | 103.22 | 7 | 0 | 87.64 | 3.18 |
| *Participant 6* | 6 | 109.28 | 6 | 2 | 122.3 | 9.11 |
| *Participant 7* | 7 | 91.71 | 6 | 0 | 94.17 | 0.21 |
| *Participant 8* | 4 | 147.84 | 3 | 3 | 20.93 | 86.61 |
| *Participant 9* | 7 | 74.28 | 6 | 5 | 82.37 | 73.76 |
| *Participant 10* | 7 | 89.55 | 7 | 7 | 29.02 | 64.83 |
| *Participant 11* | 7 | 110.7 | 7 | 0 | 101.19 | 2.66 |

**Table 5.2 Distribution of Fixation Time across Participants.**

Table shows that two participants used flowcharts extensively (Participants 9 and 10). These participants referred to flowcharts in five and seven comprehension tasks respectively. Five participants used flowcharts in four or three comprehension tasks, one participant used them in 2 comprehension tasks, and three participants did not use flowcharts at all. Overall, flowcharts were used in 31 out of 62 comprehension tasks.

### *5.2.2 Response Time, Correctness and Cognitive Load Results*

The results for correctness, response time, and cognitive load are visualized in Table 5.3. The table includes the information for every algorithm and two task variants, i.e., when the code snippet is alone and when a flowchart is available in addition to the code snippet. However, only the 31 observations when flowcharts were used are considered for the analysis. The number of responses for each algorithm differs as not every participant saw all algorithms.

| | *Code Alone* | | | | *Code with Flowcharts* | | | |
|---|---|---|---|---|---|---|---|---|
| *Algorithm* | Total Responses | Correctness in % | Mean Response Time (s) | Cog. Load | Total Responses | Correctness in % | Mean Response (s) | Cog. Load |
| *concatlists* | 5 | 0.4 | 119.13 | 0.62 | 3 | 1 | 188.31 | 0.66 |
| *countEvenNumbers* | 5 | 0.6 | 96.39 | 0.78 | 2 | 1 | 148.87 | 0.61 |
| *crosssum* | 6 | 0.5 | 105.29 | 0.67 | 2 | 0.5 | 181.53 | 0.52 |
| *decimalToBinary* | 5 | 0.2 | 115.59 | 0.65 | 3 | 0.33 | 169.04 | 0.86 |
| *diffPosAndNegNumbers* | 4 | 0.75 | 116.29 | 0.82 | 1 | 1 | 122.43 | 0.95 |
| *dropNumber* | 4 | 0.25 | 188.32 | 0.68 | 5 | 0.6 | 189.31 | 0.69 |
| *findTheLargest* | 5 | 0.6 | 114.27 | 0.84 | 3 | 0.67 | 144.14 | 0.6 |
| *firstAboveThreshold* | 6 | 0.33 | 114.47 | 0.59 | 2 | 1 | 148.23 | 0.96 |
| *hindex* | 3 | 0.33 | 166.26 | 0.49 | 1 | 0 | 223.08 | 0.42 |
| *integerToString* | 3 | 0.33 | 140.82 | 1.12 | 2 | 0.5 | 230.55 | 0.27 |
| *isPrime* | 4 | 0.5 | 77.79 | 0.64 | 3 | 0.67 | 71.42 | 0.46 |
| *multiplicationByAdding* | 5 | 1 | 96.43 | 0.41 | 1 | 1 | 75.61 | |
| *removeDoubleCharacters* | 4 | 0.5 | 128.33 | 0.63 | 2 | 0.5 | 190.51 | 0.69 |
| *sumOfIntervalNumbers* | 5 | 0.6 | 132.39 | 0.48 | 1 | 1 | 91.84 | |
| *TOTAL* | 64 | 0.5 | 119.59 | 0.66 | 31 | 0.68 | 160.57 | 0.65 |

**Table 5.3 Correctness, Response Time and Cognitive Load Results.**

The results regarding response time are illustrated in Figure 5.1. Participants completed comprehension tasks with algorithms alone in 48.97 – 237.34 s (mean=119.59 s), and comprehension tasks using flowcharts in 73.91 - 257.19 s (mean =160.57 s). This represents an increase of 40 s (34.26 %) in the mean response time when flowcharts are used.

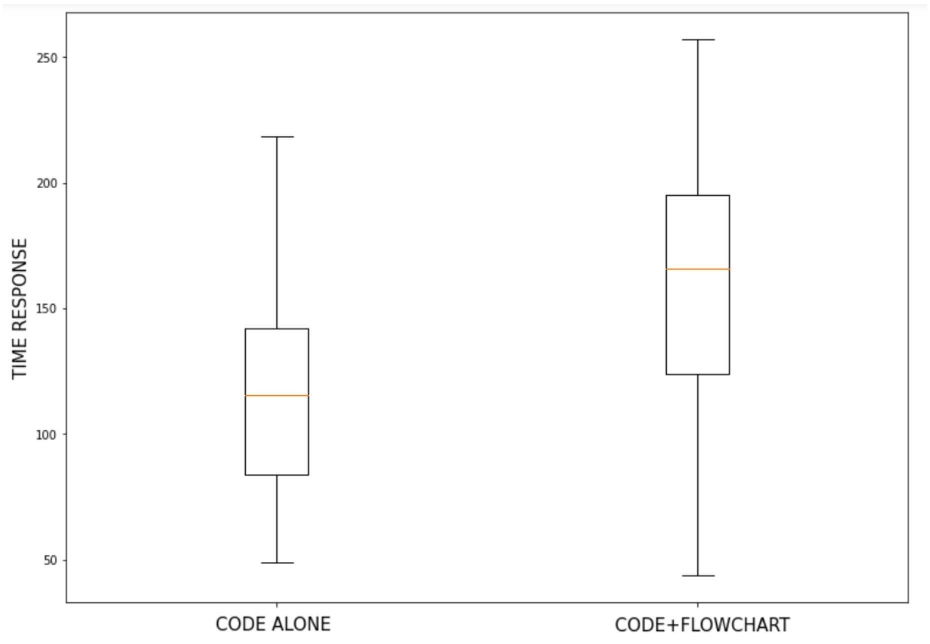


**Figure 5.1 Boxplot of Time Response Distributions.**

The correction rate for comprehension tasks with code snippets alone is 0.5%. For comprehension tasks when participants used flowcharts, the correction rate is 0.68%. It is important to note, however, that fewer observations were considered for the latter calculation as flowcharts were not used in every comprehension task where they were available. Figure 5.2 illustrates the correction rate for each algorithm. It can be seen that the correction rate of comprehension tasks when flowcharts are used is greater for 10 out of 14 algorithms, it was lower for 2 algorithms, and equal for one algorithm.

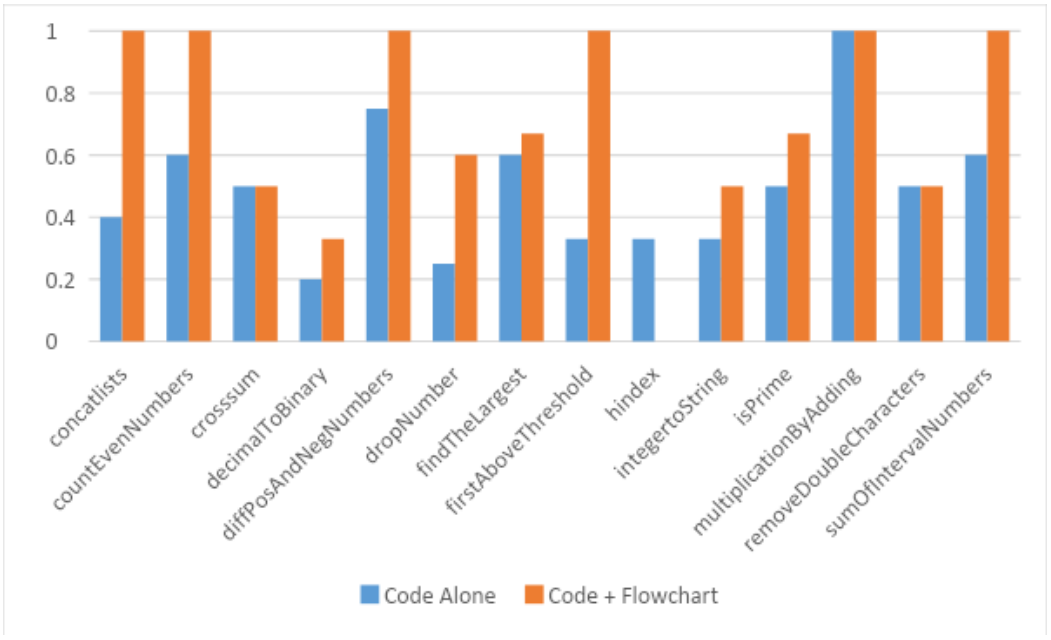


**Figure 5.2 Correctness across Algorithms.**

The results for cognitive load are visualized in Figure 5.3. Boxplots are used to represent the distributions of cognitive load for comprehension tasks with code snippets alone and comprehension tasks when flowcharts were used. The cognitive load is operationalized as the ratio of the relative power of the theta and alpha bands of the EEG signal. The mean cognitive load for comprehension tasks with code snippets alone is 0.66 and with code snippets in addition to flowcharts is 0.65.

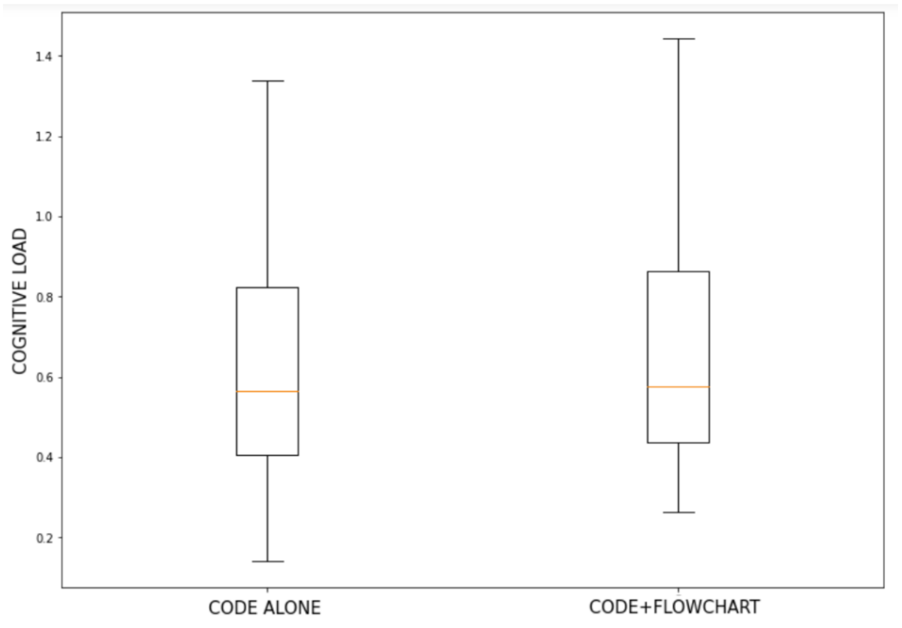


**Figure 5.3 Boxplot of Cognitive Load Distributions.**

### *5.2.3 Post Interview Results*

As mentioned in the methodology a short interview was conducted after the participants finished all the comprehension tasks. The data gathered in the post-interview is used to evaluate the fifth hypothesis: *Participants prefer flowcharts in addition to code snippets*. Below is the summary of the responses collected during this phase of the study:

● **Did you use a specific strategy to solve the tasks?** Most participants stated that they focused on the iteration part of the algorithm to solve the tasks. They first calculated the upper and lower limits of the iteration control structure and then went through each iteration till the desired result was found. Two participants stated that they tried to divide the algorithm into clusters especially when it was too long or difficult to follow. One participant stated that he recognized parts of some algorithms which facilitated the comprehension process.

● **How much did you refer to the flowcharts and how much to the code?** The responses were divided: Two participants stated that they referred extensively to the flowcharts while solving the tasks. Five participants stated that they used flowcharts sometimes. And four participants stated that they focused on the code snippets only.

● **For which task did you spend more time, when there was a flowchart present or not?** Most of the participants who claimed to have used flowcharts stated that they felt the flowcharts helped them to solve the comprehension tasks faster although that it would depend on the type of algorithm. Two participants stated that because they used flowcharts to verify the answer obtained, probably the time response was higher when flowcharts were used.

● **Do you prefer tasks where the flowcharts were present or not?** Seven participants stated that they preferred comprehension tasks when there were flowcharts available. Four participants stated that the code snippet was enough to solve the tasks.

● **Do you think there is an advantage or disadvantage when the flowchart is included along with the code?** Participants provided the following answers:

- “The flowcharts are helpful when I am confused about the code”.
- “When the code is long or has many nested conditions, I find flowcharts easier”.
- “I can see the flow of the algorithm better in flowcharts, it is easier to follow the flow there”.
- “For mathematical calculations, it is easier to follow the loops and to know in which iteration you are in a flowchart. With the code I often forgot in which iteration I am and the count”.
- “They are useful to test the result that you get in the code. I can be surer of my answer”.
- “I got tired of looking at the code so I switched to the flowcharts”.
- “I looked at the flowcharts when I didn’t understand the code or when I forgot the numbers”.
- “I didn’t use flowcharts because I’m used to the code”.
- “I think that the code is enough, I didn’t need flowcharts”.
- “I have been programming only with code and I just don’t like anything else”

## 5.3 Hypotheses Testing

After summarizing the data, the hypotheses can be evaluated with significance tests. These tests evaluate if there is a significant difference in the dependent variable after varying the levels in the independent variable, as the difference could also occur randomly. Significance tests, compute the conditional probability of having observed the result under the assumption that there is no difference in the dependent variable after varying the levels in the independent variable. If the conditional probability is below 5 %, we can reject the assumption that there is no difference in the dependent variable. This assumption is also known as the null hypothesis and the conditional probability is referred to as the significance level or p-value [138]. The null hypotheses for the study are presented and tested below:

**$H_o$1: Participants do not refer to flowcharts in addition to code snippets.**

Visual attention analysis indicates that participants referred to flowcharts for 30.98 s on average. This represents 25.47% of the total fixation time over the stimulus when a flowchart is present in addition to code snippets. Therefore, the null hypothesis $H_o$1 is rejected.

**$H_o$2: There is no significant difference in response time of comprehension tasks due to the use of flowcharts.**

Response time analysis indicates that participants completed comprehension tasks in 119.59 s with code snippets alone, and in 160.57 s when flowcharts were used. This represents an increase of 34.26 % over the mean response time. Therefore, the null hypothesis $H_o$2 is rejected.

**$H_o$3: There is no significant difference in correctness of comprehension tasks due to the use of flowcharts.**

The correction rate for comprehension tasks with code snippets alone is 0.5%. For comprehension tasks when flowcharts were used, the correction rate is 0.68%. A Chi-square test is applied to find out if there is a significant difference between these values. A significance level of 0.04819 is obtained (Chi2 = 3.9032). Using the Phi formula, the effect size is calculated as 0.4. As the significance level is greater than 0.05 and the effect size suggests a moderate relationship between the presence of flowcharts and the correctness ratio, the hypothesis $H_o$3 is rejected.

**$H_o4$: There is no significant difference in cognitive load during comprehension tasks due to the use of flowcharts.**

EEG data analysis indicates that participants had a mean cognitive load of 0.66 during comprehension tasks with code snippets alone, and 0.65 when flowcharts were used. Furthermore, boxplots in Figure 5.3 show that there is no significant difference between the treatments. Therefore, the null hypothesis $H_o4$ is accepted.

**$H_o5$: Participants do not prefer flowcharts in addition to code snippets.**

During the post-interview 7 out of 11 participants stated to prefer flowcharts in addition to code snippets. Therefore, the null hypothesis $H_o5$ is rejected.

## 5.4 Answer to the Research Question

The research question was formulated in the methodology as follows:

*RQ: What is the Effect of Flowcharts on Code Comprehension of Novices?*

After analyzing the data and testing the hypotheses, it is possible to answer the research question as: *Novices may use flowcharts in addition to code snippets as part of their code comprehension process and this could potentially improve the correctness ratio without incurring additional cognitive load burden, although the response time could increase. Furthermore, subjects' preferences are in favor of flowcharts in addition to code snippets.*

# 6 Discussion

Based on the results of the study as well as the confirmed and rejected statistical hypotheses, the corresponding interpretation is presented in this chapter. In the following subsections, the visual attention, response time, correctness, and cognitive load results are discussed and related to the research question: *What is the impact of flowcharts on the code comprehension of novices?* Lastly, the threads to construct, internal and external validities are presented.

## 6.1 Visual Attention

The hypothesis related to this measuring factor was formulated in the methodology as follows:

*H1. Participants refer to flowcharts in addition to code snippets.*

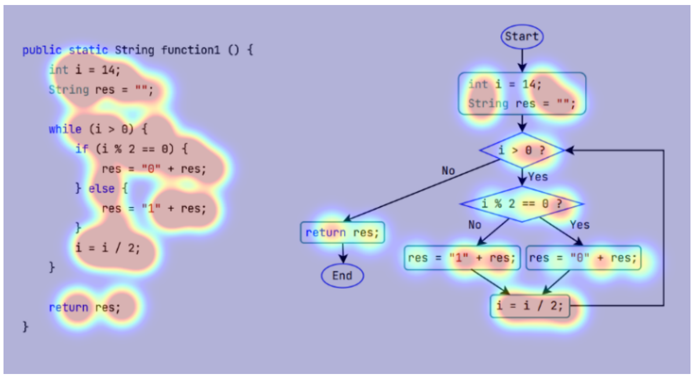
The Visual attention data showed that the fixation time over flowcharts was significant. Flowcharts were used by participants in 31 out of 62 comprehension tasks with a mean fixation time of 32 s. During these 31 comprehension tasks, participants also fixated less time on the code snippets. This suggests that novices would look for other resources besides the sole code snippets to understand the algorithms and complete the tasks. Such behavior is related to a lack of comprehension and problem-solving skills. If novices cannot understand the algorithms written in a certain programming language, they would look for alternatives or aids that could offer more information. The visual attention results suggest that flowcharts could be a suitable and practical resource for novices when they are confused about code snippets. Participants themselves stated that flowcharts were a useful aid to visualize the flow of execution, especially when algorithms were complex or long and that it was easier to perform calculations in each iteration with flowcharts.

The visual attention data also shows that three participants did not use flowcharts at all. It is possible that for these participants the process of understanding the semantics of a new algorithm representation implied time or cognitive burden and therefore decided to stick with the Java representation. Another possible explanation is the categorization of novices by Felder and Silverman. According to [153], there are two types of novices: visual learners and verbal learners. Visual learners prefer pictures, diagrams, flowcharts, timelines, hands-on activities, and practical demonstrations. Verbal learners are more able to retain and recall more of what they read and hear. It is possible, that participants who did not refer to flowcharts are verbal learners who found text representations more suitable for their comprehension process and consequently did not refer to flowcharts.

Results also indicate that participants fixated differently on flowcharts depending on the algorithm they represent. For most algorithms, the fixation time over flowcharts is important and consequently, the total fixation time over the stimulus is significantly greater than the fixation time over code snippets alone. However, the flowcharts representing the algorithms multiplicationByAdding, sumOfIntervalNumbers, and diffPosAndNegNumbers were not extensively used, and thus, the total fixation time was not affected significantly. Interestingly the overall correctness rates for these algorithms were also the highest. This could suggest that novices do not refer to flowcharts when the algorithm complexity is low as the code snippet will be sufficient to complete the comprehension tasks. However, further research is needed to study the use of flowcharts considering code complexity.

During the analysis process of the eye-tracking data, three patterns of flowchart usage could be identified. In the first pattern, fixation times over flowcharts and code snippets are both significant. It is likely that the participants follow this pattern when they have problems understanding an algorithm. The participant will switch from the code snippet to the flowchart and vice versa looking for clues and information that could reveal the algorithm's purpose. During the second pattern, the participant focuses most of the time on the code snippet but still refers to the flowchart for a considerable time. It is probable that the participant obtained a possible answer for the comprehension task and used the flowchart to verify the answer or to confirm a possible hypothesis. Finally, during pattern 3, the participant focuses most of the time on flowcharts but some fixations occur also over code snippets. Likely, participants followed this pattern after they completed several tasks and tiredness was present. Flowcharts could be an alternative that alleviated fatigue or offered a more efficient approach to solving tasks. Heatmaps of some comprehension tasks related to these patterns are shown in Table 6.1.

**Pattern 1**

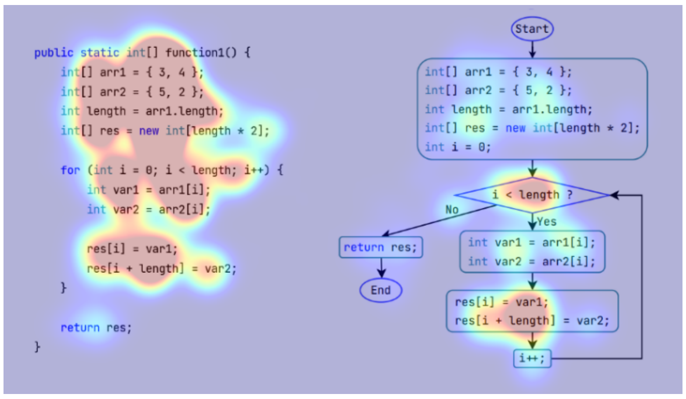


```
public static String function1 () {
    int i = 14;
    String res = "";

    while (i > 0) {
        if (i % 2 == 0) {
            res = "0" + res;
        } else {
            res = "1" + res;
        }
        i = i / 2;
    }

    return res;
}
```

**Pattern 2**



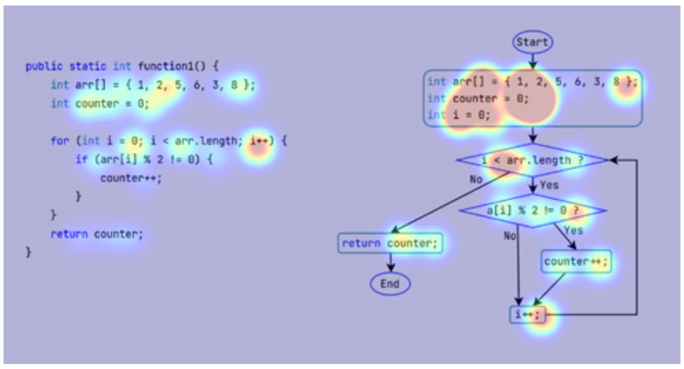
```
public static int[] function1() {
    int[] arr1 = { 3, 4 };
    int[] arr2 = { 5, 2 };
    int length = arr1.length;
    int[] res = new int[length * 2];

    for (int i = 0; i < length; i++) {
        int var1 = arr1[i];
        int var2 = arr2[i];

        res[i] = var1;
        res[i + length] = var2;
    }

    return res;
}
```

**Pattern 3**



```
public static int function1() {
    int arr[] = { 1, 2, 5, 6, 3, 8 };
    int counter = 0;

    for (int i = 0; i < arr.length; i++) {
        if (arr[i] % 2 != 0) {
            counter++;
        }
    }
    return counter;
}
```

**Table 6.1 Usage Patterns of Flowcharts.**

## 6.2 Response Time

The hypothesis related to this measuring factor was formulated in the methodology as follows:

*H2: Participants using flowcharts take less time to complete the tasks.*

Results indicate that the difference in response time of comprehension tasks with code snippets alone and comprehension tasks when flowcharts were used was significant. Participants spent on average 34.26 % more time completing tasks when flowcharts were used. This finding opposes the proposed hypothesis as a shorter response time was expected instead. However, it is important to note that overall participants had to process more information as two representations of the same algorithm were present in the stimulus. Visual attention results suggested that in most cases both representations were used in conjunction rather than individually. As noted in the previous section, this behavior likely occurs when novices have difficulties understanding the algorithm. Novices could initiate two comprehension processes in parallel with code snippets and flowcharts. In this case, they would try to follow the algorithm flow with both representations switching back and forth when they get confused or when they need more information. Eventually, some possible answers could be obtained and the additional algorithm representation could be used to verify the answer. If the verification fails, participants would continue this process till a suitable answer is found. This is a time-consuming usage pattern, especially for long or challenging algorithms. Additionally, participants stated to have a low experience with flowcharts. Under these conditions, participants needed to get used to the semantics of flowcharts and learn how to use them during the study. This could introduce an additional time burden during the initial comprehension tasks with flowcharts. Nevertheless, the additional time that novices spend with flowcharts in addition to code snippets constitutes a fair tradeoff, if during this time they acquire the appropriate mental models that lead to a more appropriate comprehension process.

## 6.3 Correctness

The hypothesis related to this measuring factor was formulated in the methodology as follows:

*H3: Participants using flowcharts answer with a higher correction rate.*

The Chi-squared test showed that the difference in the correctness ratio of comprehension tasks with code snippets alone and comprehension tasks when flowcharts were used was significant. Participants answered with a correctness ratio of 68% using flowcharts and 50% with code snippets alone. The 18% improvement could be explained by different factors. First, as stated by participants, flowcharts facilitated program comprehension by clarifying the flow of the algorithm, especially in the selection or iteration blocks. Second, flowcharts were used to verify possible answers, although this may introduce additional time burden, participants could correct themselves in case a first wrong guess is made and refine the comprehension process that could ultimately lead to a correct answer. Third, as participants struggle to understand an algorithm, they refer to the flowchart looking for clues or relevant information. Participants then could refer back to the code snippets with additional insights and a clearer picture of the algorithm flow. This process could be repeated till a suitable answer is found. Although this is a time-consuming pattern, it could prevent participants to skip the comprehension task or submitting an incorrect response. These findings support the use of flowcharts as a tool to enhance the weak comprehension skills of novices. The inclusion of flowcharts in the comprehension process increases the opportunities and resources that a novice has to understand an algorithm appropriately leading to a higher correction rate.

## 6.4 Cognitive Load

The hypothesis related to this measuring factor was formulated in the methodology as follows:

*H4: Participants using flowcharts have a lower cognitive load when completing the tasks.*

The results suggest that the difference in the cognitive load during comprehension tasks with code snippets alone and comprehension tasks when flowcharts were used was not significant. This indicates that the increase in the comprehension process time in novices due to the use of flowcharts does not induce a higher cognitive load. This is especially important as participants used flowcharts when they struggled to comprehend an algorithm. Using different strategies novices could answer comprehension tasks with a higher correctness ratio without incurring additional cognitive load. Moreover, although most of the subjects were not familiar with flowcharts, they learned how to use them during the experiment. This is a clear advantage of visual representations, as novices could understand the semantics of different symbols and how their composition depicts an algorithm without fatigue. The small learning curve of flowcharts makes them a practical alternative to communicate the flow of an algorithm.

## 6.5 Flowchart Preferences

The hypothesis related to this measuring factor was formulated in the methodology as follows:

*H5: Participants prefer to use flowcharts in addition to code.*

Results showed that participants prefer flowcharts in addition to code snippets. During the post-interview, seven out of eleven participants stated that they preferred comprehension tasks when flowcharts are present. These claims have been confirmed during the analysis of the visual attention data. Although, it was found that each participant used flowcharts to different degrees during the experiment. Some participants used them extensively, while others only in a few tasks. Moreover, participants stated to have solved comprehension tasks faster when flowcharts were available. After analyzing response times, data suggests that participants spent more time solving comprehension tasks with flowcharts. The subjective complexity of the algorithms probably decreased when flowcharts were used, which could explain their statements regarding time response. Participants also mentioned various advantages that flowcharts offered during the experiment, i.e., they illustrate easily the flow of the algorithm, they could be used to verify possible answers, and they assist in the understanding of confusing code. These statements relate to the results, as participants achieved a higher correction rate when flowcharts were used. The acceptance and approval of flowcharts by novices was significant, and generally constitute a valuable tool to mitigate the problems related to weak comprehension skills in novices.

## 6.6 Threads to Validity

Threats to validity are circumstances or problems that may challenge the veracity of the conclusions. They are identified as threats due to their possible consequences, since they could lead to reach partially or totally wrong conclusions.

### *6.6.1 Construct Validity*

Construct validity refers to the degree to which we can be sure we are measuring what we set out to measure [138]. After processing the eye-tracking data it was observed that for some participants the gaze position was slightly shifted to one direction. This situation is related to the use of glasses which could not be controlled during the experiment. However, during the preprocessing the gaze data was reviewed and adjusted to fit the correct offset, though some fixations could have been lost during this process. Similarly, EEG signals are sensitive to noise and other electrical signals from the body. This is why filters and further processing is necessary to perform a suitable analysis. Unfortunately, the removal of unwanted artifacts also distorts signals of interest [155]. This could affect the measurement of the cognitive load of subjects. However, ICA processing is known to be one of the most effective methods to clean EEG signals and valid results can be assumed [152].

### 6.6.2 Internal Validity

Internal validity is concerned with causation, i.e., whether the results can indeed be attributed to differences between treatments [138]. An important effect of stress and fatigue could be present at the time of the experiment. This is especially important if students have a high academic workload. However, most of the data were collected before and after the exam period to mitigate this effect. Furthermore, room temperature could also have influenced participant performance as data collection happened during summer. Some participants stated to be uncomfortable due to the high temperatures. Unfortunately, it was not possible to control room temperature as there was no air conditioning available in the experiment room. Similarly, the age of participants was used to control the intelligence and problem-solving ability differences between participants. As the age of the participants was similar, probably a more robust control method should have been used to mitigate this confounding factor. Nevertheless, all participants were undertaking a computer science-related degree that has similar requirements for admission, such as performance in prior studies.

### 6.6.3 External Validity

External validity refers to the generalizability of the results, namely whether they can be expected to hold beyond the specific experimental conditions that were used [138]. Short code snippets were considered as these are the type of code novice programs would deal with in introductory programming courses. Nevertheless, some specific types of applications could require longer programs. Additionally, when real code is considered, a certain level of domain knowledge is often required. In this experiment, it was assumed that novice programmers would not have a large amount of domain knowledge which is acquired mostly with experience. Still, in some cases, domain knowledge is strictly needed and it could be taken into account in future work. Additionally, only Java programming language was considered, as this is one of the most common languages taught in universities [140]. However, as the code snippets included mostly simple control structures, the effect of changing program languages would not be significant.

# 7 Conclusion and Future Work

Computer science students face different difficulties during introductory programming courses. In particular, a lack of problem-solving skills is an evident weakness in first-year students [9]. Research has shown, that novices do not have the appropriate mental models that allow them to translate problem specifications into working programs. Additionally, novices have difficulties describing the flow of the algorithms, its purpose and output [10]. The motivation of conducting this research was precisely to investigate ways to improve the problem-solving skills of novices by providing an accurate mental model through algorithm visual representations.

Many studies have demonstrated the potential that structured flowcharts have in helping novices develop conceptual knowledge and problem-solving skills [15]. Following this line of research, a controlled experiment was conducted including Eye-tracker and EEG devices. These technologies provided additional insights into the effect of algorithm visual representations on program comprehension. During the study, novices completed comprehension tasks with code snippets alone and with flowcharts in conjunction with code snippets. Novices' fixation time, cognitive load, response time, correctness, and subjective preference of flowcharts was evaluated accordingly.

Visual attention results indicated that flowcharts were indeed used by novices during the program comprehension process. This suggests that computer science students may accept and adopt flowcharts as a visual aid in program comprehension. Moreover, the inclusion of flowcharts extended the overall fixation time and consequently the response time of tasks. It is very likely, that during the additional time, novices could develop appropriate mental models that led them ultimately to a correct understanding of the program. It was found that the correctness ratio improved by 18 % when novices used flowcharts. This corroborates the advantages and opportunities that novices have studying algorithm visual representations in addition to code snippets. Furthermore, participants' subjective preferences are largely in favor of flowcharts. During the post interview, participants expressed that flowcharts facilitate the visualization of the algorithm flow and the comprehension of confusing algorithms. Hence, this study concludes that flowcharts could be an effective visual assistance during program comprehension of novices. Nevertheless, additional research is necessary to further study the benefits of algorithm visual representations in introductory programming courses:

- Study of flowcharts on code comprehension with different levels of code complexity and programming experience.
- Study of flowcharts on code comprehension in a pre a post studying design with EEG and Eye-tracker.  The treatment could consist in the inclusion of flowcharts in introductory programming courses.
- Development and empirical evaluation of programming environments based on flowcharts that support the generation of correct source code in modern programming languages.
- Study of the effect of flowcharts on other concepts of software engineering such as software architecture design.
- Study of the effect of different UML diagrams on code comprehension.
- Inclusion of different tools and methodologies to measure program comprehension: FMRI, recall, think-aloud protocols, debug-tasks.

# Bibliography

[1] Radermacher and G. Walia, “Gaps between industry expectations and the abilities of graduates,” Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13, 2013.

[2] N. Urbach, F. Ahlemann, T. Böhmann, P. Drews, W. Brenner, F. Schaudel, and R. Schütte, “The impact of digitalization on the IT department,” Business &amp; Information Systems Engineering, vol. 61, no. 1, pp. 123–131, 2018.

[3] “Europe needs high-tech talent:” Foundation for European Progressive Studies, 14-Jul-2022. [Online]. Available: https://feps-europe.eu/publication/europe-needs-high-tech/. [Accessed: 21-Oct-2022].

[4] “Impacts of the COVID-19 pandemic on EU Industries,” CEPS, 29-Mar-2021. [Online]. Available: https://www.ceps.eu/ceps-publications/impacts-of-the-covid-19-pandemic-on-eu-industries/. [Accessed: 21-Oct-2022].

[5] “Degree programmes in Germany - DAAD,” www.daad.de. [Online]. Available: https://www.daad.de/en/study-and-research-in-germany/courses-of-study-in-germany/all-study-programmes-in-germany/. [Accessed: 21-Oct-2022].

[6] R. Burbaite, V. Drasute, and V. Stuikys, “Integration of computational thinking skills in stem-driven computer science education,” 2018 IEEE Global Engineering Education Conference (EDUCON), 2018.

[7] ICT education - a statistical overview. Statistics Explained. [Online]. Available: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=ICT_education_-_a_statistical_overview&amp;oldid=575052. [Accessed: 21-Oct-2022].

[8] J. Figueiredo and F. Garcia-Penalvo, “Teaching and learning tools for introductory programming in University Courses,” 2021 International Symposium on Computers in Education (SIIE), 2021.

[9] N. Lázaro Alvarez, Z. Callejas, and D. Griol, “Factores que inciden en la Deserción Estudiantil en Carreras de Perfil Ingeniería Informática.,” Revista Fuentes, vol. 1, no. 22, pp. 105–126, 2020.

[10] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, “A study of the difficulties of novice programmers,” Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '05, 2005.

[11] J. Bennedsen and M. E. Caspersen, “Failure rates in introductory programming,” ACM SIGCSE Bulletin, vol. 39, no. 2, pp. 32–36, 2007.

[12] "TouchDevelop," Microsoft Research, 03-Apr-2018. [Online]. Available: https://www.microsoft.com/en-us/research/project/touchdevelop/. [Accessed: 21-Oct-2022].

[13] Z. Ullah, A. Lajis, M. Jamjoom, A. Altalhi, A. Al-Ghamdi, and F. Saleem, "The effect of automatic assessment on Novice Programming: Strengths and limitations of existing systems," Computer Applications in Engineering Education, vol. 26, no. 6, pp. 2328–2341, 2018.

[14] T. Beaubouef and J. Mason, "Why the high attrition rate for computer science students," ACM SIGCSE Bulletin, vol. 37, no. 2, pp. 103–106, 2005.

[15] Westphal, B.T., Harris, F.C. and Fadali, M.S. (no date) "Graphical Programming: A vehicle for teaching computer problem solving," 33rd Annual Frontiers in Education, 2003. FIE 2003. [Preprint]. Available at: https://doi.org/10.1109/fie.2003.1264759.

[16] M. McCracken, T. Wilusz, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, and I. Utting, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS Students," Working group reports from ITiCSE on Innovation and technology in computer science education - ITiCSE-WGR '01, 2001.

[17] L. McIver and D. Conway, "Seven deadly sins of introductory programming language design," Proceedings 1996 International Conference Software Engineering: Education and Practice.

[18] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons, "Conditions of learning in Novice programmers," Journal of Educational Computing Research, vol. 2, no. 1, pp. 37–55, 1986.

[19] Z. Ullah, A. Lajis, M. Jamjoom, A. Altalhi, A. Al-Ghamdi, and F. Saleem, "The effect of automatic assessment on Novice Programming: Strengths and limitations of existing systems," Computer Applications in Engineering Education, vol. 26, no. 6, pp. 2328–2341, 2018.

[20] Y.-F. Shih and S. M. Alessi, "Mental models and transfer of learning in Computer Programming," Journal of Research on Computing in Education, vol. 26, no. 2, pp. 154–175, 1993.

[21] V. Ramalingam, D. LaBelle, and S. Wiedenbeck, "Self-efficacy and mental models in learning to program," ACM SIGCSE Bulletin, vol. 36, no. 3, pp. 171–175, 2004.

[22] S. Iqbal and O. K. Harsh, "A self review and external review model for teaching and assessing novice programmers," International Journal of Information and Education Technology, pp. 120–123, 2013.

[23] M. E. Tudoreanu, “Designing effective program visualization tools for reducing user's cognitive effort,” Proceedings of the 2003 ACM symposium on Software visualization - SoftVis '03, 2003.

[24] G. Ebel and M. Ben-Ari, “Affective effects of program visualization,” Proceedings of the 2006 international workshop on Computing education research - ICER '06, 2006.

[25] L. P. Baldwin and J. Kuljis, “Learning programming using program visualization techniques,” Proceedings of the 34th Annual Hawaii International Conference on System Sciences.

[26] B. T. Westphal, F. C. Harris, and M. S. Fadali, “Graphical Programming: A vehicle for teaching computer problem solving,” 33rd Annual Frontiers in Education, 2003. FIE 2003.

[27] C. Schulte, T. Clear, A. Taherkhani, T. Busjahn, and J. H. Paterson, “An introduction to program comprehension for computer science educators,” Proceedings of the 2010 ITiCSE working group reports on Working group reports - ITiCSE-WGR '10, 2010.

[28] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, “Understanding source code with functional magnetic resonance imaging,” Proceedings of the 36th International Conference on Software Engineering, 2014.

[29] A. Von Mayrhauser and A. M. Vans, “Program comprehension during software maintenance and evolution,” Computer, vol. 28, no. 8, pp. 44–55, 1995.

[30] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta, “On the impact of refactoring operations on Code Quality Metrics,” 2014 IEEE International Conference on Software Maintenance and Evolution, 2014.

[31] C. Chen, R. Alfayez, K. Srisopha, B. Boehm and L. Shi, "Why Is It Important to Measure Maintainability and What Are the Best Ways to Do It?", 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 2017.

[32] M.-A. Storey, “Theories, methods and tools in program comprehension: Past, present and future,” 13th International Workshop on Program Comprehension (IWPC'05), 2005.

[33] R. Brooks, “Towards a theory of the comprehension of computer programs,” International Journal of Man-Machine Studies, vol. 18, no. 6, pp. 543–554, 1983.

[34] M. P. O'Brien, "Software Comprehension - A Review & Research Direction", Technical Report UL-CSIS-03-3, University of Limerick, November 2003. Daniel Wagner.

[35] Soloway, E., Adelson, B., & Ehrlich, K. (1988). Knowledge and processes in the comprehension of computer programs. In M. T. H. Chi, R. Glaser, & M. J. Farr (Eds.), The nature of expertise (pp. 129–152). Lawrence Erlbaum Associates, Inc.

[36] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kastner, Andrew Begel, Anja Bethmann, and Andr ̈ e Brechmann. “Measuring Neural Efficiency of Program Comprehension.” In: Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE). ACM, 2017, pp. 140–150.

[37] Susan Wiedenbeck. “Processes in Computer Program Comprehension.” In: Workshop on Empirical Studies of Programmers. 1986, pp. 48–57.

[38] Susan Wiedenbeck. “The Initial Stage of Program Comprehension.” In: International Journal of Man-Machine Studies 35.4 (1991), pp. 517–540.

[39] Teresa Shaft and Iris Vessey. “The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension.” In: Information Systems Research 6.3 (1995), pp. 286–299.

[40] B. Shneiderman and R. Mayer, “Syntactic/semantic interactions in programmer behavior: A model and experimental results,” International Journal of Computer; Information Sciences, vol. 8, no. 3, pp. 219–238, 1979.

[41] N. Pennington, “Stimulus structures and mental representations in expert comprehension of computer programs,” Cognitive Psychology, vol. 19, no. 3, pp. 295–341, 1987.

[42] Basili, V.R. and H.D. Mills. Understanding and documenting programs. IEEE Transactions on Software Engineering, 8(3):270–283, May 1982.

[43] A. Fekete and Z. Porkoláb, “A comprehensive review on software comprehension models,” Annales Mathematicae et Informaticae, vol. 51, pp. 103–111, 2020.

[44] S. Letovsky, “Cognitive processes in program comprehension,” Journal of Systems and Software, vol. 7, no. 4, pp. 325–339, 1987.

[45] D. C. Littman et al. Mental models and software maintenance. J. Syst. Software, 7(4):341-355, 1987.

[46] J. Koenemann and S. P. Robertson, “Expert problem-solving strategies for program comprehension,” Proceedings of the SIGCHI conference on Human factors in computing systems Reaching through technology - CHI '91, 1991.

[47] S. Xu, Z. Cui, and Y. Gui, “Cognitive process during incremental software development,” 6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007), 2007.

[48] A. Gogus, “Bloom’s Taxonomy of Learning Objectives,” Encyclopedia of the Sciences of Learning, pp. 469–473, 2012.

[49] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh, and M. Zulkernine, “Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults,” Empirical Software Engineering, vol. 21, no. 3, pp. 896–931, 2015.

[50] A. Dunsmore and M. Roper. A Comparative Evaluation of Program Comprehension Measures. Tech. rep. EFoCS 35-2000. Department of Computer Science, University of Strathclyde, 2000.

[51] E. Soloway and K. Ehrlich, “Empirical studies of Programming Knowledge,” IEEE Transactions on Software Engineering, vol. SE-10, no. 5, pp. 595–609, 1984.

[52] A. Fontana and J. Frey. “Interviewing: The Art of Science.” In: The Handbook of Qualitative Research 361376 (1994)

[53] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed Hassan, and Shanping Li. “Measuring Program Comprehension: A Large-Scale Field Study with Professionals.” In: IEEE Trans. Softw. Eng. 44.10 (2017), pp. 951–976.

[54] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. “How Do Professional Developers Comprehend Software?” In: Proc. Int’l Conf. Software Engineering (ICSE). IEEE, 2012, pp. 255–265.

[55] Barbara DiCicco-Bloom and Benjamin Crabtree. “The Qualitative Research Interview.” In: Medical education 40.4 (2006), pp. 314–321.

[56] Rensis Likert. “A Technique for the Measurement of Attitudes.” In: Archives of Psychology 22.140 (1932), pp. 1–55.

[57] Charles Egerton Osgood, George Suci, and Percy Tannenbaum. The Measurement of Meaning. 47. University of Illinois press, 1957.

[58] Richard Miara, Joyce Musselman, Juan Navarro, and Ben Shneiderman. “Program Indentation and Comprehensibility.” In: Communications of the ACM 26.11 (1983), pp. 861–867.

[59] Andrew Duchowski. Eye Tracking Methodology – Theory and Practice, Third Edition. Springer, 2017.

[60] Keith Rayner. “Eye Movements in Reading and Information Processing: 20 Years of Research.” In: Psychological Bulletin 124.3 (1998), p. 372.

[61] Zohreh Sharafi, Bonita Sharif, Yann-Gael Guehneuc, Andrew Begel, Roman Bednarik, and Martha Crosby. “A practical guide on conducting eye tracking studies in software engineering.” In: Empirical Software Engineering (2020), pp. 1–47.

[62] Z. Sharafi, T. Shaffer, B. Sharif, and Y.-G. Gueheneuc, “Eye-tracking metrics in software engineering,” 2015 Asia-Pacific Software Engineering Conference (APSEC), 2015.

[63] B. Davies, “Precision and accuracy in glacial geology,” AntarcticGlaciers.org, 22-Jun-2020. [Online]. Available: https://www.antarcticglaciers.org/glacial-geology/dating-glacial-sediments-2/precision-and-accuracy-glacial-geology/. [Accessed: 27-Oct-2022].

[64] Martha Crosby and Jan Stelovsky. “How Do We Read Algorithms? A Case Study.” In: *Computer* 23.1 (1990), pp. 25–35.

[65] Bonita Sharif and Johnathon Maletic. “An Eye Tracking Study on camelCase and under score Identifier Styles.” In: Proc. Int’l Conf. Program Comprehension (ICPC). IEEE, 2010, pp. 196–205.

[66] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. “Eye Movements in Code Reading: Relaxing the Linear Order.” In: Proc. Int’l Conf. Program Comprehension (ICPC). IEEE, May 2015, pp. 255–265.

[67] Curtis Ikehara and Martha Crosby. “Assessing Cognitive Load with Physiological Sensors.” In: Proc. Hawaii Int’l Conf. on System Sciences. IEEE. 2005, 295a–295a.

[68] Jackson Beatty and Brennis Lucero-Wagoner. The Pupillary System, Handbook of Psychophysiology, Cacioppo, Tassinary & Berntson. 2000.

[69] Jackson Beatty and Daniel Kahneman. “Pupillary Changes in Two Memory Tasks.” In: Psychonomic Science 5.10 (1966), pp. 371–372.

[70] Eckhard Hess and James Polt. “Pupil Size in Relation to Mental Activity during Simple Problem-Solving.” In: Science 143.3611 (1964), pp. 1190–1192.

[71] Bruno Laeng, Sylvain Sirois, and Gustaf Gredeback. “Pupillometry: A Window to the Preconscious?” In: Perspectives on Psychological Science 7.1 (2012), pp. 18–27.

[72] Mariska Kret and Elio Sjak-Shie. “Preprocessing Pupil Size Data: Guidelines and Code.” In: Behavior Research Methods 51.3 (2019), pp. 1336–1342.

[73] Denae Ford, Titus Barik, and Chris Parnin. “Studying Sustained Attention and Cognitive States with Eye Tracking in Remote Technical Interviews.” In: Eye Movements in Programming: Models to Data (2015), 5 pages.

[74] Mahnaz Behroozi, Shivani Shirolkar, Titus Barik, and Chris Parnin. “Does Stress Impact Technical Interview Performance?” In: Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE). 2020, pp. 481–492.

[75] Maria Eckstein, Belen Guerra-Carrillo, Alison Miller Singley, and Silvia Bunge. “Beyond Eye Gaze: What Else Can Eyetracking Reveal about Cognition and Cognitive Development?” In: Developmental Cognitive Neuroscience 25 (2017), pp. 69–91.

[76] Michael Doughty. “Consideration of Three Types of Spontaneous Eyeblink Activity in Normal Humans: During Reading and Video Display Terminal Use, in Primary Gaze, and while in Conversation.” In: Optometry and Vision Science 78.10 (2001), pp. 712–725.

[77] Denae Ford, Titus Barik, and Chris Parnin. “Studying Sustained Attention and Cognitive States with Eye Tracking in Remote Technical Interviews.” In: Eye Movements in Programming: Models to Data (2015), 5 pages.

[78] Siyuan Chen, Julien Epps, Natalie Ruiz, and Fang Chen. “Eye Activity as a Measure of Human Mental Effort in HCI.” In: Proc. Int’l Conf. Intelligent User Interfaces. ACM. 2011, pp. 315–318.

[79] James Phillips, Richard Leahy, John Mosher, and Bijan Timsari. “Imaging neural activity using MEG and EEG.” In: Engineering in Medicine and Biology Magazine 16.3 (1997), pp. 34–42.

[80] “EEG caps leading manufacturer and supplier,” BESDATA, 21-Apr-2022. [Online]. Available: https://besdatatech.com/eeg-caps/. [Accessed: 27-Oct-2022].

[81] W. Mansor, M. S. A. Rani, and N. Wahy, “Integrating Neural Signal and embedded system for controlling small motor,” IntechOpen, 29-Aug-2011. [Online]. Available: https://www.intechopen.com/chapters/18890. [Accessed: 27-Oct-2022].

[82] Igor Crk, Timothy Kluthe, and Andreas Stefik. “Understanding Programming Expertise: An Empirical Study of Phasic Brain Wave Changes.” In: ACM Trans. Comput Hum. Interact. 23.1 (2015), 2:1–2:29.

[83] P. D. Raphael Vallat, “Raphael Vallat,” Bandpower of an EEG signal. [Online]. Available: https://raphaelvallat.com/bandpower.html. [Accessed: 27-Oct-2022].

[84] J. A. Urigüen and B. Garcia-Zapirain, “EEG artifact removal—state-of-the-art and Guidelines,” Journal of Neural Engineering, vol. 12, no. 3, p. 031001, 2015.

[85] S. Kanoga and Y. Mitsukura, “Review of Artifact Rejection Methods for Electroencephalographic Systems,” IntechOpen, 29-Nov-2017. [Online]. Available: https://www.intechopen.com/chapters/54606. [Accessed: 27-Oct-2022].

[86] Wolfgang Klimesch. “EEG Alpha and Theta Oscillations Reflect Cognitive and Memory Performance: A Review and Analysis.” In: Brain Research Reviews 29.2-3 (1999), pp. 169–195.

[87] Igor Crk and Timothy Kluthe. “Toward Using Alpha and Theta Brain Waves to Quantify Programmer Expertise.” In: Int’l Conf. Engineering in Medicine and Biology Society. IEEE. 2014, pp. 5373–5376.

[88] Martin Yeh, Dan Gopstein, Yu Yan, and Yanyan Zhuang. “Detecting and Comparing Brain Activity in Short Program Comprehension Using EEG.” In: Frontiers in Education Conference. IEEE, 2017, pp. 1–5.

[89] Martin Yeh, Dan Gopstein, Yu Yan, and Yanyan Zhuang. “Detecting and Comparing Brain Activity in Short Program Comprehension Using EEG.” In: Frontiers in Education Conference. IEEE, 2017, pp. 1–5.

[90] Makrina Kosti, Kostas Georgiadis, Dimitrios Adamos, Nikos Laskaris, Diomidis Spinellis, and Lefteris Angelis. “Towards an Affordable Brain Computer Interface for the Assessment of Programmers’ Mental Workload.” In: Int.l J. Human-Computer Studies 115 (2018), pp. 52–66.

[91] Seolhwa Lee, Andrew Matteson, Danial Hooshyar, SongHyun Kim, JaeBum Jung, GiChun Nam, and Heuiseok Lim. “Comparing Programming Language Comprehension between Novice and Expert Programmers Using EEG Analysis.” In: Int’l Conf. on Bioinformatics and Bioengineering (BIBE). IEEE, 2016, pp. 350–355.

[92] Seolhwa Lee, Danial Hooshyar, Hyesung Ji, Kichun Nam, and Heuiseok Lim. “Mining Biometric Data to Predict Programmer Expertise and Task Difficulty.” In: Cluster Computing (2017), pp. 1–11.

[93] Toyomi Ishida and Hidetake Uwano. “Synchronized Analysis of Eye Movement and EEG during Program Comprehension.” In: Int’l Workshop on Eye Movements in Programming (EMIP). IEEE. 2019, pp. 26–32.

[94] Julio Medeiros, Ricardo Couceiro, Joao Castelhano, M Castelo Branco, Gon ˜ c¸ alo Duarte, Catarina Duarte, Joao Dur ˜ aes, Henrique 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). IEEE. 2019, pp. 1413–1416.

[95] E. Soloway and James C. Spohrer. 1988. Studying the Novice Programmer. L. Erlbaum Associates Inc., USA.

[96] Sheil B. A. (1981): The psychological study of programming. Computing Surveys 13:101-120.

[97] Robins, A., Rountree, J. and Rountree, N. (2003): Learning and teaching programming: A review and discussion. Computer Science Education 13(2): 137-172.

[98] Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., & Paterson, J. (2007). A survey of literature on the teaching of introductory programming. ACM SIGCSE Bulletin, 39(4), 204–223.

[99] Winslow, L. E. (1996) Programming pedagogy – A psychological overview. ACM SIGCSE Bulletin, 28(3), 17–22.

[100] Altadmri, A., & Brown, N. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (pp. 522–527). New York: ACM.

[101] A. Lishinski, A. Yadav, R. Enbody, and J. Good, “The influence of problem solving abilities on students' performance on different assessment tasks in CS1,” Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE '16, 2016.

[102] G. Barlow-Jones and D. van der Westhuizen, “Problem solving as a predictor of programming performance,” Communications in Computer and Information Science, pp. 209–216, 2017.

[103] M. Decasse and A.-M. Emde, “A review of Automated Debugging Systems: Knowledge, strategies and Techniques,” Proceedings. [1989] 11th International Conference on Software Engineering.

[104] R. Lister, On the cognitive development of the novice programmer: and the development of a computing education researcher, the 9th Computer Science Education Research Conference, Leiden, Netherlands, 2020, pp. 1-15

[105] J. Sillito, G. C. Murphy, and K. De Volder, “Asking and answering questions during a programming change task,” IEEE Transactions on Software Engineering, vol. 34, no. 4, pp. 434–451, 2008.

[106] F. Ricca et al. The Role of Experience and Ability in Comprehension Tasks Supported by UML Stereotypes. In Proc. Int’l Conf. Software Engineering (ICSE), pages 375–384. IEEE CS, 2007.

[107] M. Muller. Are Reviews an Alternative to Pair Programming? Empirical Softw. Eng., 9(4):335–351, 2004.

[108] S. Kleinschmager and S. Hanenberg. How to Rate Programming Skills in Programming Experiments? A Preliminary, Exploratory, Study Based on University Marks, Pretests, and Self-Estimation. In Proc. ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, pages 15–24. ACM Press, 2011.

[109] C. Bunse. Using Patterns for the Refinement and Translation of UML Models: A Controlled Experiment. Empirical Softw. Eng., 11(2):227– 267, 2006.

[110] S. Biffl and W. Grossmann. Evaluating the Accuracy of Defect Estimation Models Based on Inspection Data from Two Inspection Cycles. In Proc. Int'l Conf. Software Engineering (ICSE), pages 145–154. IEEE CS, 2001.

[111] J. Hannay et al. Effects of Personality on Pair Programming. IEEE Trans. Softw. Eng., 36(1):61–80, 2010.

[112] S. Xinogalos, "Using flowchart-based programming environments for simplifying programming and software engineering processes," 2013 IEEE Global Engineering Education Conference (EDUCON), 2013, pp. 1313-1322, doi: 10.1109/EduCon.2013.6530276.

[113] Ben Shneiderman, Richard Mayer, Don McKay, and Peter Heller. 1977. Experimental investigations of the utility of detailed flowcharts in programming. Commun. ACM 20, 6 (June 1977), 373–381.

[114] David A. Scanlan. 1989. Structured Flowcharts Outperform Pseudocode: An Experimental Comparison. IEEE Softw. 6, 5 (September 1989), 28–36.

[115] D. Hendrix, J. H. Cross and S. Maghsoodloo, "The effectiveness of control structure diagrams in source code comprehension activities," in IEEE Transactions on Software Engineering, vol. 28, no. 5, pp. 463-477, May 2002, doi: 10.1109/TSE.2002.1000450.

[116] Candido Cabo. 2018. Effectiveness of Flowcharting as a Scaffolding Tool to Learn Python. In 2018 IEEE Frontiers in Education Conference (FIE). IEEE Press, 1–7.

[117] B. Calloni. and D. Bagert, “Iconic Programming in BACCII vs. Textual Programming: which is a better learning environment?”, ACM, SIGSCE’ 94 3/94, Phoenix AZ, 1994, pp. 188-192.

[118] T. Crews and U. Ziegler, “The Flowchart Interpreter for Introductory Programming Courses.”. Proc. FIE ’98 Conference, 1998, pp. 307-312.

[119] Hooshyar, D., Ahmad, R., Yousefi, M., Yusop, F. and Horng, S.-J. (2015), Flowchart-based Intelligent Tutoring System. Journal of Computer Assisted Learning, 31: 345-361.

[120] A. Cilliers et al., “The effect of integrating an Iconic Programming Notation into CS1”, Proc. ACM ITiCSE 2005, 2005, pp. 108-112.

[121] M. S. Hall, “Raptor: nifty tools”, J. Comput. Sci. Coll., vol. 23(1) pp.110-111, Oct. 2007.

[122] M. Marcelino et al., “H-SICAS, a handheld algorithm animation and simulation tool to support initial programming learning”, 38th Annual Frontiers in Education Conf., 2008, pp. T4A-7 - T4A-12.

[123] T. Watts, “The SFC Editor a graphical tool for algorithm development”, J. Computing Sciences in Colleges, vol. 20, Issue 2, pp. 74-85, 2004.

[124] C. Areias and A. Mendes, “A tool to help students to develop programming skills”, Proc. 2007 Int. Conf. Computer systems and technologies, ACM, New York, NY, USA, 2007, Article 89, 7 pages.

[125] A. Scott et al., “Progranimate - A Web Enabled Problem Solving Application”, CSREA EEE 2008, 2008, pp. 498-508.

[126] S. Chen and S. Morris, “Iconic programming for flowcharts, java, turing, etc.”, Proc. 10th SIGCSE Conf. Innovation and technology in computer science education , ACM, New York, NY, USA, 2005, pp. 104-107.

[127] Green, T.R., & Petre, M. (1992). When Visual Programs are Harder to Read than Textual Programs.

[128] K.N. WHITLEY, Visual Programming Languages and the Empirical Evidence For and Against, Journal of Visual Languages & Computing, Volume 8, Issue 1, 1997, Pages 109-142, ISSN 1045-926X.

[129] M. E. Crosby, J. Stelovsky(1995) From multimedia instruction to multimedia evaluation. Journalof Educational Multimedia and Hypermedia 4, 162.

[130] STEVEN HANSEN, N.HARI NARAYANAN, MARY HEGARTY, Designing Educationally Effective Algorithm Visualizations, Journal of Visual Languages & Computing, Volume 13, Issue 3, 2002, Pages 291-317, ISSN 1045-926X.

[131] J.T. Stasko (1997) Using student-built animations as learning aids. In: Proceedings of the ACM Technical Symposium on Computer Science Education. ACM Press, New York, pp. 25-29.

[132] A.W. Lawrence, A. N. Badre &J.T. Stasko (1994) Empirically evaluating the use of animations to teach algorithms. In: Proceedings of the 1994 IEEE Symposium on Visual Languages. IEEE Computer Society Press, Los Alamitos, CA, pp. 48-54.

[133] B. Price (1990) A framework for the automatic animation of concurrent programs. Unpublished M.S. thesis, Department of Computer Science, University of Toronto.

[134] P. Mulholland (1998) A principled approach to the evaluation of SV: a case studying Prolog. In: Software visualization: Programming as a Multimedia Experience (M. Brown, J. Domingue, B. Price &J. Stasko, eds). The MIT Press, Cambridge, MA, pp. 439-452.

[135] W. Heijstek, T. Kühne and M. R. V. Chaudron, "Experimental Analysis of Textual and Graphical Representations for Software Architecture Design," 2011 International Symposium on Empirical Software Engineering and Measurement, 2011, pp. 167-176.

[136] W. J. Dzidek, E. Arisholm and L. C. Briand, "A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance," in IEEE Transactions on Software Engineering, vol. 34, no. 3, pp. 407-432, May-June 2008, doi: 10.1109/TSE.2008.15.

[137] Birkmeier, D., Klöckner, S., & Overhage, S. (2010). An Empirical Comparison of the Usability of BPMN and UML Activity Diagrams for Business Users. ECIS.

[138] C. Wohlin, Experimentation in software engineering. Berlin: Springer, 2012.

[139] Siegmund, J., Kästner, C., Liebig, J., Apel, S., & Hanenberg, S. (2012). Measuring programming experience. 2012 20th IEEE International Conference on Program Comprehension (ICPC), 73-82.

[140] J. Siegmund and J. Schumann, "Confounding parameters on program comprehension: a literature survey", Empirical Software Engineering, vol. 20, no. 4, pp. 1159-1192, 2014. Available: 10.1007/s10664-014- 9318-8.

[141] D. Falessi et al., "Empirical software engineering experts on the use of students and professionals in experiments", Empirical Software Engineering, vol. 23, no. 1, pp. 452-489, 2017.

[142] Siegmund, J., (2013). The human factor in computer science. In: Horbach, M. (Hrsg.), INFORMATIK 2013 – Informatik angepasst an Mensch, Organisation und Umwelt. Bonn: Gesellschaft für Informatik e.V.. (S. 1184-1185).

[143] Vernon, P. and Strudensky, S., 1988. Relationships between problem solving and intelligence. Intelligence, 12(4), pp.435-453.

[144] D. Feitelson, "Considerations and Pitfalls in Controlled Experiments on Code Comprehension", 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), 2021.

[145] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. 2017. Measuring neural efficiency of program comprehension. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). Association for Computing Machinery, New York, NY, USA, 140–150.

[146] G. Charness, U. Gneezy and M. Kuhn, "Experimental methods: Between-subject and within-subject design", Journal of Economic Behavior & Organization, vol. 81, no. 1, pp. 1-8, 2012.

[147] Zubair Ahsan, Unaizah Obaidellah, Visual behavior on problem comprehension among novice programmers with prior knowledge, Procedia Computer Science, Volume 192, 2021, Pages 2347-2354, ISSN 1877-0509.

[148] “Eye tracking software for Behavior Research - Tobii Pro Lab,” Tobii. [Online]. Available: https://www.tobii.com/products/software/data-analysis-tools/tobii-pro-lab. [Accessed: 30-Oct-2022].

[149] “Dry Eeg headset: Quick-20: CGX,” CGX 2021. [Online]. Available: https://www.cgxsystems.com/quick-20r-v2. [Accessed: 30-Oct-2022].

[150] “Home - psychopy®,” Home - PsychoPy®. [Online]. Available: https://www.psychopy.org/. [Accessed: 30-Oct-2022].

[151] Hessels RS, Niehorster DC, Kemner C, Hooge ITC. Noise-robust fixation detection in eye movement data: Identification by two-means clustering (I2MC). Behav Res Methods. 2017 Oct;49(5):1802-1823. doi: 10.3758/s13428-016-0822-1. PMID: 27800582; PMCID: PMC5628191.

[152] M. Ullsperger and S. Debener, Simultaneous EEG and fmri: Recording, analysis, and application. New York: Oxford University Press, 2010.

[153] FELDER, R. & SILVERMAN, L.,(1988),Learning and Teaching Styles in Engineering Education, Engineering Education, 78,7, pp 674-681.

[154] S. Davies, J. Polack-Wahl and K. Anewalt, "A snapshot of current practices in teaching the introductory programming sequence", Proceedings of the 42nd ACM technical symposium on Computer science education - SIGCSE '11, 2011.

[155] Widmann, A., Schröger, E. & Maess, B. (2015). Digital filter design for electrophysiological data--a practical approach. Journal of Neuroscience Methods, 250, 34–46.