Developing Eye Tracking Methods for Detecting Source Code Comprehension Strategies

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft
der Universität Duisburg-Essen

zur Erlangung des akademischen Grades

Doktor der Ingenieurswissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Fabian Deitelhoff
aus
Werl

Gutachter: Prof. Dr. rer. soc. Heinz Ulrich Hoppe
Gutachter: Prof. Dr. habil. Andrea Kienle
Gutachter: Prof. Dr.-Ing. Jürgen Ziegler

Tag der mündlichen Prüfung: 11. Dezember 2020

DOI: 10.17185/duepublico/73588
URN: urn:nbn:de:hbz:464-20201218-103917-4

Developing Eye Tracking Methods for Detecting Source Code Comprehension Strategies

ABSTRACT

Knowledge in STEM disciplines is becoming increasingly crucial to be able to use, understand, and create technology in the emerging society. Understanding technology, its importance, and its disadvantages are essential for participation in an emerging society. Adapting the education of children should be a key goal to prepare them for a hardly predictable future. As one domain of STEM, learning and teaching programming are considered difficult and pose great obstacles for both teachers and learners. Introductory programming courses are largely not as effective and successful as they could be, and according to extensive research in computer science education, one result is that students are not learning to program at all.

Therefore, learner support is a vital part of introductory courses to help them cope with the technical and semantic problems they face. Modified programming languages, instructional design, and intelligent tutoring systems (ITSs) are possible measures of positively changing the outcome of programming courses. Eye tracking as technology is suitable for analyzing the comprehension process of reading source code, and it is a technological foundation to interactively support learners.

Many research projects focus on the differences between novices and experts regarding their eye movement strategies and patterns. To date, specialized methods for analyzing and visualizing eye movement patterns, e.g., intending to detect useful data for a support system, are missing. Therefore, to move the knowledge in the source code comprehension community forward, this thesis focuses on developing and testing specialized visualization and analytical methods and tools. The research approaches and ideas necessary for this step are used in three case studies to test new visualizations, analysis methods, models, and learning hints. The results of these case studies serve as contributions to the source code comprehension research community and the vision of a dynamic learner support system. The contributions of this thesis primarily aim at better detecting source code comprehension strategies, as well as identifying methods and tools to support this detection.

Contents

Abstract i

Contents iii

List of Figures vii

List of Tables xi

List of Listings xiii

1 Introduction 1
- 1.1 Problem Statement . . . 2
- 1.2 The Vision – A Dynamic Learner Support System . . . 3
- 1.3 Aim and Research Agenda . . . 5
- 1.4 Outline . . . 7

2 Related Work 9
- 2.1 Spanning Research Areas . . . 9
- 2.2 Learning to Program . . . 12
  - 2.2.1 Programming as a Common Skill . . . 12
  - 2.2.2 Learning and Teaching Introductory Programming . . . 14
  - 2.2.3 Characterizing and Structuring Programming Knowledge . . . 17
  - 2.2.4 Novices Mistakes and Problems . . . 20
  - 2.2.5 (Intelligent) Learner Support . . . 23
- 2.3 Source Code Comprehension . . . 27
  - 2.3.1 Comprehension Strategies of Novices and Experts . . . 28
  - 2.3.2 An Overview of Models . . . 30
  - 2.3.3 Comprehension Strategies . . . 32
  - 2.3.4 Assessment of Source Code Comprehension . . . 34
  - 2.3.5 Foster Source Code Comprehension and Code Explaining . . . 35
- 2.4 Eye Movements in Source Code Comprehension . . . 36
  - 2.4.1 Mapping Eye Movement Data to Source Code . . . 37
  - 2.4.2 Eye Movement Metrics . . . 39
  - 2.4.3 Coding Schemes and Eye Movement Patterns . . . 40
  - 2.4.4 Patterns of Novices and Experts . . . 42
- 2.5 Learning Analytics Based on Eye Tracking . . . 44
- 2.6 Methods for Analyzing Eye Movement Patterns . . . 45
- 2.7 Methods for Visualizing Eye Movement Patterns . . . 46
- 2.8 Summary . . . 48

3 Research and Analytics Approach **51**
3.1 Investigations and Prototyping . . . 52
3.1.1 Preliminary Study – A Programming Environment Prototype . . . 53
3.1.2 Analyzing a Real-World Dataset . . . 54
3.1.3 Visualization and Analysis Ideas . . . 56
3.2 Analytics Approach – Methods and Tools . . . 57
3.3 Research Approach – Illustrating Case Studies . . . 58
3.3.1 Step 1: Finding Patterns – The EMIP Dataset Analysis . . . 58
3.3.2 Step 2: Provoking Patterns – The Code Smell Study . . . 59
3.3.3 Step 3: Testing Hints – The Learning Hints Study . . . 59
3.4 The Adapted Data Analytics Approach Based on CRISP-DM . . . 60
3.5 Summary . . . 62

4 Eye-Tracking-Based Analysis Methods and Tools **63**
4.1 Methodical Eye Tracking Challenges . . . 63
4.2 The Influence of Varying AOI Models . . . 66
4.3 Standard Analysis Techniques for Eye Tracking Data . . . 69
4.4 AOI Sequences (AOI-DNAs) . . . 70
4.4.1 Visualization . . . 71
4.4.2 Sequence Alignment . . . 73
4.4.3 Clustering . . . 74
4.4.4 Pattern Search with Regular Expressions . . . 74
4.5 AOI Transitions (AOI-STGs) . . . 76
4.6 CodeSight . . . 78
4.7 Consistent Data Analytics Pipeline . . . 78
4.8 Summary . . . 80

5 Case Study 1 – The EMIP Dataset Analysis **83**
5.1 Study Design . . . 84
5.1.1 Research Question and Hypotheses . . . 84
5.1.2 Methods . . . 86
5.1.3 Experimental Design . . . 86
5.1.4 Prerequisites and Environment . . . 87
5.1.5 Captured Data . . . 87
5.1.6 Participants, Material, and Procedure . . . 87
5.1.7 Analysis Design . . . 90
5.2 Study Results . . . 91
5.2.1 Structured Reading . . . 91
5.2.2 Early Fixation Hits . . . 93
5.2.3 Clustering Reading Patterns . . . 94
5.2.4 Patterns in AOI-STGs . . . 104
5.3 Threats to Validity . . . 107
5.4 Summary and Next Steps . . . 108

6 Case Study 2 – The Code Smell Study **111**
6.1 Study Design . . . 112
6.1.1 Research Question and Hypotheses . . . 112
6.1.2 Methods . . . 115

6.1.3 Experimental Design . . . 116
6.1.4 Prerequisites and Environment . . . 118
6.1.5 Captured Data . . . 121
6.1.6 Participants, Material, and Procedure . . . 122
6.1.7 Analysis Design . . . 128
6.2 Study Results . . . 132
6.2.1 Answer Quality . . . 132
6.2.2 Ratio for Duration and Fixations . . . 137
6.2.3 Pattern Reading Behavior . . . 140
6.2.4 Performance Experts vs. Novices . . . 150
6.2.5 Time of the Comprehension Question . . . 153
6.3 Threats to Validity . . . 156
6.4 Summary and Next Steps . . . 158

7 Case Study 3 – The Learning Hints Study 161
7.1 Study Design . . . 162
7.1.1 Research Question and Hypotheses . . . 162
7.1.2 Methods . . . 166
7.1.3 Experimental Design . . . 167
7.1.4 Prerequisites and Environment . . . 168
7.1.5 Captured Data . . . 171
7.1.6 Participants, Material, and Procedure . . . 173
7.1.7 Analysis Design . . . 178
7.2 Study Results . . . 180
7.2.1 Dynamic Hint Use . . . 182
7.2.2 Answer Quality . . . 185
7.2.3 Ratings for Learning Hints . . . 188
7.2.4 Ratings for Code Difficulty . . . 189
7.3 Threats to Validity . . . 191
7.4 Summary . . . 192

8 Contributions and Discussion 195
8.1 First Aim – Detect Source Code Comprehension Strategies . . . 195
8.2 Second Aim – Identify, Create, and Test Methods and Tools . . . 201
8.3 Third Aim – The Toolbox . . . 203
8.4 Retrospective of Source Code Complexity, Similarity Measures and Clustering 205
8.5 Retrospective of the Methodical Eye Tracking Challenges . . . 206
8.6 Future Work . . . 207

Bibliography 210

List of Figures

1.1 A brief overview of the three aims of this thesis. 6
1.2 A brief overview of the outline of this thesis. 7

2.1 The learning objectives for the cognitive domain of Bloom's taxonomy. Level names from (Anderson, Krathwohl, & Bloom, 2001). 15
2.2 Overview of the thirteen focus topics as primary scientific foundation of this thesis. 48

3.1 Screenshots of both the prototype (left) and the heat map of one participant superimposed on it (right). 53
3.2 Heat maps of all participants (left), and heat map of one participant (right). . 54
3.3 Gaze plot visualization of one participant. 55
3.4 Screenshots of the three iteration steps to visualize AOI-DNAs to highlight eye movement patterns. 56
3.5 A brief overview of the case studies and the timeline of working on them. . 58
3.6 The original (left) and modified (right) CRISP-DM processes on which the data analyses in this thesis are based. 62

4.1 Two sampling examples with a low and a high sampling rate to visualize the possible differences. 65
4.2 The AOI grid model dividing the stimulus into areas. 67
4.3 A domain-specific AOI model based on tokens, lines, and regions for a source code stimulus. 67
4.4 An example of a behavior-enriched AOI model, based on a heat map of the viewing behavior of participants. 68
4.5 AOI margin and padding visualized for a Java code snippet. 69
4.6 Two examples of AOI-DNAs visualized with a color code for sequential reading behavior (Story Order Reading, SOR). 72
4.7 The Rectangle source code example with visualized AOIs (line model), AOI-DNA colors for reference on the left, and an example of regular expression with matching AOIs. Image from (Deitelhoff, Harrer, & Kienle, 2019b). 75
4.8 Extracts from the *flicking* (left) and *retrace declaration* (right) visualizations. 76
4.9 *Flicking* (top) and *retrace declaration* (bottom) visualizations with pattern search matches (red borders). 76
4.10 Example of AOI-STG to visualize transitions between AOIs. The start (green) and end (red) nodes are visible, as well as fountains for a higher focus on specific AOIs. 77
4.11 Example of AOI-DNA in CodeSight to visualize reading behaviors with a gray-scale color code and red markers, which indicates a pattern search match. 78

4.12 An example of a Visual Studio analysis project (left) and the abstracted program flow for the generic sequential analysis (right). 79

5.1 The visualized EMIP experimental design with the two groups and the alternating stimuli order. 86
5.2 Distribution of answers to the questions on expertise in programming (overall) and expertise in the experiment language (Java). 88
5.3 A grayscale visualization for the defined AOIs (emip). 92
5.4 A grayscale color code to visualize the SOR pattern (participant IDs on the left). 92
5.5 Visualization of AOI-DNAs with a red, purple, and green color code to show the EOR pattern (participant IDs on the left). 93
5.6 AOI-DNAs (excerpt) with green and purple to visualize early fixations of the main method and calculation method(s) respectively. Every other AOI is marked in white to create gaps (participant IDs on the left). 94
5.7 Excerpts for the dendrograms for the Rectangle (left) and Vehicle (right) source code examples, both for the line self AOI model. 96
5.8 Dendrogram comparison results of the Rectangle and Vehicle source code examples (line and region AOI models) for 63 participants. 97
5.9 Dendrogram comparison correlations for 63 participants for the Rectangle and Vehicle source code examples (line and rectangle AOI models). 98
5.10 Comparison of two dendrograms for the line Rectangle and Vehicle source codes (“1 – 2”). An ideal matching participant group was automatically highlighted by the comparison algorithm. 99
5.11 Excerpt of the comparison of two dendrograms for the line Rectangle and Vehicle source codes (“1 – 2”) for a larger participant group. An optimal matching participant group was automatically highlighted by the comparison algorithm. 100
5.12 Eye movement data from Participants 8 and 151 for the Rectangle and Vehicle source code examples (java2, line, self). 101
5.13 The first cluster with 112 eye movement sequences (Rectangle). 101
5.14 The second cluster with 20 eye movement sequences (Rectangle). 102
5.15 The third cluster with 12 eye movement sequences (Rectangle). 103
5.16 AOI-STG for Participant 11, with the AOI line model on the Rectangle stimulus (correct answer). 105
5.17 AOI-STG for Participant 36, with the AOI line model on the Rectangle stimulus (incorrect answer). 106
5.18 A brief overview of the case study timeline (EMIP completed). 108

6.1 The visualized code smell experimental design with the four groups, the stimuli order, and the two different question times. 118
6.2 The layout of the setup in the eye tracking lab at the University of Applied Sciences and Arts, Dortmund. 119
6.3 The layout of the setup at the software development company in Lünen, Germany. 119
6.4 Screenshot of one example slide of the presentation for the study condition one: Rectangle non-smell, Vehicle smell, question after the code example. . . 120

6.5 Screenshot of one example slide, translated into English, of the presentation for the first study condition: Rectangle non-smell, Vehicle smell, question after the code example. . . . 121
6.6 Distributions for the question about the expertise in Java (all, novices, and experts). . . . 123
6.7 The study procedure for the code smell study. . . . 128
6.8 The AOI labeling for the Rectangle non-smelly source code example. . . . 129
6.9 The AOI labeling for the Rectangle smelly source code example. . . . 130
6.10 The AOI labeling for the Vehicle non-smelly source code example. . . . 130
6.11 The AOI labeling for the Vehicle smelly source code example. . . . 131
6.12 The legend for the mapping of AOIs to the grayscale color coding. . . . 142
6.13 Twenty-two clustered sequences for the Rectangle non-smelly source code example. . . . 143
6.14 The first cluster of the sequences for the Rectangle non-smell source code example. . . . 143
6.15 The second cluster of the sequences for the Rectangle non-smell source code example. . . . 144
6.16 The third cluster of the sequences for the Rectangle non-smell source code example. . . . 144
6.17 The AOI aggregation to combine consecutive AOIs to shorten the sequence. 146
6.18 The reading sequences used for the patterns of table 6.16 and the resultant regular expressions (1 → Rectangle non-smell, 2 → Rectangle smell, 3 → Vehicle non-smell, and 4 → Vehicle smell). . . . 147
6.19 The reading sequences used for the patterns of table 6.17 and the resultant regular expressions (1 → Rectangle non-smell, 2 → Rectangle smell, 3 → Vehicle non-smell, and 4 → Vehicle smell). . . . 148
6.20 The reading sequences used for the patterns of Table 6.18 and the resultant regular expressions (1 → Rectangle non-smell, 2 → Rectangle smell, 3 → Vehicle non-smell, and 4 → Vehicle smell). . . . 149
6.21 A brief overview of the case study timeline (code smell study completed). . . 158

7.1 The six study conditions (groups) formed by three source code examples and three learning hints – **(S)**yntax, **(D)**ynamic, **(P)**lain. The experimental gaze-enabled interface was used by every participant as the last step. . . . 168
7.2 The layout of the setup in the eye tracking lab at the University of Applied Sciences and Arts, Dortmund. . . . 169
7.3 The first question of the pre-test as an excerpt from the full pre-test. . . . 169
7.4 The Vehicle source code examples with the syntax highlighting learning hint. 170
7.5 The questions for the short UEQ (UEQ-S). . . . 170
7.6 The experimental interface with the gaze-enabled learning hints. . . . 171
7.7 Screenshot of the Eye Tracking Connector application to route the eye tracking data from Tobii Pro TX300 to the study prototype. . . . 172
7.8 Excerpt of the data from the MongoDB scheme for the learning hints study. 173
7.9 The distribution of answers to the question on expertise in Java in the learning hints study (all participants). . . . 174
7.10 The study procedure for the learning hints study. . . . 178
7.11 Mean, variance, and standard deviation per item of the UEQ-S. . . . 181
7.12 A brief overview of the case study timeline (learning hints study completed). 192

8.1 Overview of the 18 contributions of this thesis, distributed among eight main categories. 204

List of Tables

3.1 Overview of the two contextual layers for aligning the work of this thesis. . 57

4.1 Shortened example of a substitution matrix for the N-W algorithm. 73

5.1 Overview of the correct Rectangle answers (RC), the correct Vehicle answers (VC), and the average length and standard derivation for the sequences per cluster. 103

6.1 Overview of the four conditions for the code smell study, consisting of combinations of Rectangle and Vehicle source code examples, smell and non-smell, as well as questions before and after variations. 117
6.2 The distributions of the skill levels (self-assessment) per condition – lower values are better and emphasized. 133
6.3 Rated answers for various groups and the calculated ANOVA results. 134
6.4 Participants' answers for the first condition (Rectangle non-smell, Vehicle smell, question after). 135
6.5 Participants' answers for the second condition (Rectangle non-smell, Vehicle smell, question before). 135
6.6 Participants' answers for the third condition (Rectangle smell, Vehicle non-smell, question after). 136
6.7 Participants' answers for the fourth condition (Rectangle smell, Vehicle non-smell, question before). 136
6.8 The average and standard derivation for the duration and fixation of both source code examples (divided by smell and non-smell variants). 138
6.9 The distribution for the duration (overall) per group, as well as the calculated ANOVA results. 139
6.10 The distribution for the fixation (overall) per group, as well as the calculated ANOVA results. 139
6.11 The distribution for the duration ratio per group, as well as the calculated ANOVA results. 139
6.12 The distribution for the fixation ratio per group, as well as the calculated ANOVA results. 140
6.13 SOR and EOR eye movement patterns encoded as sequences of letters based on the AOIs per source code stimulus. 141
6.14 Visually identified SOR and EOR patterns for the source-code-varying examples. 142
6.15 Simplified patterns (based on the complete EOR patterns shown in table 6.13 on page 141) and corresponding matches per source code example. The black sub-strings were used for the regular expressions. 145

6.16 Short reading sequences and the corresponding pattern matches per source code example. . . . 146
6.17 Eye movement patterns and corresponding pattern matches based on the source code smells per source code example. . . . 148
6.18 Eye movement pattern definition for the flicking pattern with the corresponding pattern matches per source code example. . . . 149
6.19 The average answer quality of experts and novices, separated into different subgroups and compared using ANOVA tests. . . . 151
6.20 The average answer quality divided according to the question-before and question-after conditions, separated by different subgroups, and compared using ANOVA tests. . . . 155

7.1 Distribution of correct/incorrect answers across code examples for the two D_{use} groups with the average time used. . . . 183
7.2 The usage time of participants, divided into groups based on the pre-test or self-assessment (Java experience) results. . . . 184
7.3 Correct and incorrect answers for all three code examples and for all three learning hints (S → Syntax, D → Dynamic, and P → Plain). . . . 185
7.4 Overall fixation counts and fixation times per stimuli, additionally separated by learning hint (S → Syntax, D → Dynamic, and P → Plain). The highest values are emphasized. . . . 186
7.5 The difficulty rating for all three source code examples and for all three learning hints (S → Syntax, D → Dynamic, and P → Plain). The lowest values are emphasized. . . . 190

8.1 Overview of the summarized contributions of this thesis. . . . 204

List of Listings

4.1 Calculating AOI-Hits for AOI-DNAs from a list of metadata (participants) and loaded fixations. 80
5.1 The Rectangle source code example of the EMIP study. 88
5.2 The Vehicle source code example for the EMIP study. 89
6.1 The Rectangle source code example without a smell for the code smell study. 125
6.2 The Rectangle source code example with a smell for the code smell study. . . 125
6.3 The Vehicle source code example without a smell for the code smell study. . 126
6.4 The Vehicle source code example with a smell for the code smell study. . . . 127
7.1 The Bubble source code example for the learning hints study. 175
7.2 The GCD source code example for the learning hints study. 175
7.3 The Vehicle source code example for the learning hints study. 176

FOR MY DAUGHTER *LEA* – MY CONSTANT SOURCE OF INSPIRATION

"Things are only impossible until they're not."

– Captain Jean-Luc Picard (TNG, Season 01, Episode 17)

1 Introduction

December the 3rd, 2017 was the happiest day of my life: My daughter Lea was born. Watching her growing up in the past three and a half years and thinking about her future in this technological world, my sense of foreboding is becoming stronger. A constant stream of new and changing demands awaits her. Through founding my startup – called brickobotik – to help schools integrate science, technology, engineering, and mathematics (STEM) learning into their curriculum, I am increasingly aware of the fact that the educational system is not nearly as well prepared as it should be to empower young people for this future. Since digital representations and tools permeate society, educational institutes, workplaces, and everyday life are changing rapidly. Living in this world in transition means changing the way children are educated, as well as the way they perceive and interact with technology, which the COVID-19 pandemic has recently made clear, along with the necessity to homeschool children in many countries after closing schools.

STEM Knowledge

The desirable general knowledge of STEM is becoming increasingly critical. Apart from this, combining STEM skills with more meta-level skills, such as problem solving, creativity, and critical thinking is a worthwhile goal, regardless of whether a person will work directly in the technology sector. Understanding technology, its importance, its disadvantages, and its future evolution is essential for participation in an emerging society. Adapting the education of children should be a key goal to prepare them for a hardly predictable future. Education systems throughout the world are changing their curricula with more or less effort and success. Organizations for the *K-12* primary and secondary education in the USA are contact points for schools to integrate STEM into their daily teaching routine. Other

countries, for example Germany, are strengthening their effort[1] to integrate lessons such as computer science into their schools.

Computational Thinking as a Vital Ingredient

As argued by Wing, "computational thinking" (CT) is a vital ingredient in the context of learning STEM and the more meta-level skills (Wing, 2006). Although CT is not reducible to programming, programming is an activity that both builds on CT and can support the development of CT. Accordingly, it has been argued that there is an overall value in learning basic programming concepts and skills (Greiff et al., 2017).

Dynamic Learner Support System

To support the process of learning programming, e.g., specialized learning material, novice programming environments (NPEs), novice programming languages, and (intelligent) tutoring systems (ITSs) have been created over the past decades. Although these systems can be beneficial (Crow, Luxton-Reilly, & Wuensche, 2018; Kulik & Fletcher, 2016), a more dynamic system, tailored to the reading and comprehension processes of source code, is missing to date. Such a dynamic learner support system can use the reading behavior of a current user, match the available patterns to previously analyzed data, and provide hints when useful. Eye tracking as technology has proven to be a useful tool when measuring and analyzing reading behavior. However, to help learners understand source code, it is still unclear whether eye tracking can be combined with knowledge of source code comprehension tasks. This missing link raises the questions of whether eye movement data can be the foundation for more dynamic learner support systems and whether such eye movement data can be analyzed to gain knowledge about learners' comprehension strategies. These questions lead to the problem statement, including the focus on source code comprehension, learning hints, and the necessary research agenda to fill noticed research gaps.

1.1 PROBLEM STATEMENT

Three Different Modes for Creating Programs

Applying eye tracking as an objective source of information (Busjahn, Schulte, Sharif, et al., 2014) to the field of programming education raises an immediate question: Which part of learning programming should be augmented? Creating a computer program requires different modes: a) reading (source code comprehension), b) modifying (source code manipulation), and c) creating (source code production) source code. The colloquial term "programming" requires all three steps, mostly in an arbitrary sequence with many iterations. This thesis focuses on the *source code comprehension process* to reduce the inherent complexity of creating a dynamic learner support system for all three modes. Nevertheless, it must be noted that the other two steps for creating a computer program may provide additional sources of information, which can help one to understand learners' thought and decision processes and their hurdles regarding understanding the domain of programming.

[1] `https://www.land.nrw/de/pressemitteilung/ministerin-gebauer-landesregierung-bringt-einfuehrung-der-faecher-wirtschaft-und` – Last accessed on December 14, 2020.

Eye Movement Patterns

Analyzing eye movement data collected from source code comprehension tasks is a challenging endeavor. Questions arise regarding what the abstraction level of the eye movement data and the source code (i.e., based on tokens, lines, regions, or beacons) is, what an eye movement pattern of a comprehension hurdle looks like if such a pattern is detectable in real time, and how a learning hint should be presented to be helpful for a learner. An eye movement pattern is a recognizable exemplar of any regularly repeated eye movement within the data. Such patterns describe reoccurring eye movements linked to, for example, specific source code features or comprehension processes. See Section 2.4.3 for an explanation.

Research Gaps in the Source Code Comprehension Community

In summary, these questions need a well-thought research agenda to fill the gaps identified in the source code comprehension community. Missing methods and tools must be constructed and tested as an integral part of gathering the knowledge necessary to create a dynamic learner support system. Even with coding schemes (Busjahn, Bednarik, et al., 2015) developed in the source code comprehension community, attributing eye movement data to source code comprehension is a questionable approach because an observer has no access to the thought and decision processes that a participant has undergone during the reading process. Thus, because of the sophisticated process of analyzing eye movement data, as well as the complexity of designing and conducting eye tracking studies, a thorough analysis demands a mixed-method approach. The next section describes the ideas and challenges in creating a dynamic learner support system, which serves as the overall vision and surrounding context for this thesis.

1.2 THE VISION – A DYNAMIC LEARNER SUPPORT SYSTEM

The following section is partly based on the publication "Towards a Dynamic Help System: Support of Learners During Programming Tasks Based Upon Historical Eye-Tracking Data" by Deitelhoff and Harrer (2018).

Learning Programming is Difficult

Research regarding the difficulties of learning to program has a long history. In 2002, Tony Jenkins argued that "few students find learning to program easy" (Jenkins, 2002). He mentioned different factors for his observation, such as learning styles and motivation. While one can argue about learning styles, especially motivation, other research has focused on programming as a complex cognitive activity (Pea & Kurland, 1984). Conditional statements, syntactic structures such as nested brackets and blocks, and mismatches of argument types are, among others, common challenges that novices in particular must cope with (Brown & Altadmri, 2017).

Supporting Learners is Important

Therefore, supporting learners is an active field of research. Different concepts, methods, and tools have been developed and tested (e.g., specialized learning material, NPEs, visual languages, and ITSs). Block-based languages are extensively used, especially for children, and research covers how block-based, text-based, and hybrid environments influence novice

programming practices (Weintrop & Wilensky, 2018). Apart from this, first versions of ITSs are integrated into novice programming environments to combine a more interactive help system with programming environments that are already intentionally designed to support novices while learning. *iSNAP* (Price, Dong, & Lipovac, 2017) is an extension of the *Snap!* (Harvey & Mönig, 2010) programming environment and one example of such an ITS.

Eye Tracking Data for Improved Tutoring Systems

Tutoring systems can have the additional benefit of supporting students automatically. Even though clickstreams are widely used in such scenarios, multimodal data streams can be beneficial, for instance by including produced artifacts or based on live data from the learner, e.g., by using eye tracking technology. Incorporating eye tracking data is the overall goal of the dynamic learner support system. Eye tracking data is used as an analytical and interactive foundation. Moreover, eye movements are analyzed dynamically and checked against predefined patterns to be able to integrate learning hints into the source code (Deitelhoff & Harrer, 2018). Supporting students while learning to program is crucial for their success. Educators must thus balance when and how to help their students. However, finding this balance is often problematic. Educators may be accurate about a student's conceptions of errors but they lack knowledge about the frequency and *time-to-fix* (Brown & Altadmri, 2017). The error of "mismatched brackets," to name one example, has a median time-to-fix of *17* sec, analyzed in a study by Brown and Altadmri (2017). These situations can be detected by a dynamic learner support system to help students when such an error occurs – after waiting a while to allow them to fix the error without help. This approach can result in a more accurate feedback loop, tailored to the needs of the individual learner. However, such a feedback loop must also be well designed; simply "waiting a while" will not be sufficient.

Potential of Dynamic Learner Support System

A well-designed and technologically sophisticated dynamic learner support system, with the ability to find bugs in learners' code and suggest fixes, can be likened to a mentor or tutor in a class. The system is not intended to replace a teacher but to support the self-learning phases of learners, which is useful for e-learning, massive online courses, or everything taught without the possibility of immediate feedback from teachers.

Open Questions

There are more open questions than answers when analyzing state-of-the-art research, which results in identifying the following six questions and challenges for designing a dynamic learner support system based on eye tracking:

1. If a user struggles to understand source code, is this reflected within the eye tracking data in the form of eye movement patterns? This link is crucial to distinguish comprehension processes, and it results in or is based on problems in understanding the code from the non-problematic comprehension processes.

2. Can these eye movement patterns be detected in real time while a user is working with the system? Detecting and analyzing patterns after the source code comprehension tasks is only suitable in early-stage research projects. For a real application, the patterns must be analyzed while the learner is reading the code.

3. Are the eye movement events, based on the patterns, sufficiently differentiable for use in data-supported learning hints? Supporting learners will highly depend on different interventions and hints in the source code comprehension process. Different stages (patterns) of source code comprehension hurdles must thus be distinguishable.

4. How can eye movement patterns be compared fast and reliably? For a support system based on eye movement patterns, it is necessary to analyze an emerging pattern in the source code comprehension process and patterns that were detected in a previous session.

5. Which interventions or learning hints are useful and effective in the context of source code comprehension? Analyzing eye movement data to detect patterns is essential, but the learning hints with which learners are presented are equally important to get their attention and to be supportive in the learning process.

6. Which methods and tools are necessary to support addressing the questions mentioned above? The current state of source code comprehension research likely does not provide the required tools for analyzing eye movement data.

The aforementioned questions must be answered to build a dynamic learner support system. They lay out essential anchor points for the context of this thesis. Both the aims of this work and a research agenda to follow them are influenced by this overall vision.

1.3 AIM AND RESEARCH AGENDA

Methodical and Tooling Requirements

Keeping the six questions for building a dynamic learner support system in mind, specific methodical and tooling requirements can be specified to close the current gap found in the literature and the source code comprehension community. The process of designing a dynamic learner support system includes various theoretical and methodical steps to be followed before the system can be built. The goals of this thesis are to identify necessary steps in both categories and to develop and test promising theories and tools to detect source code comprehension strategies, as illustrated in Figure 1.1 on page 6. These aims are broken down as follows:

1. The **first aim** of this thesis is to lay out the steps that are needed to detect source code comprehension strategies in different datasets and study scenarios.

2. The **second aim** is to identify, create, and test the methods and tools that are needed to support the testing of the ideas.

3. The **third aim** is to describe the resulting toolbox for detecting source code comprehension strategies.

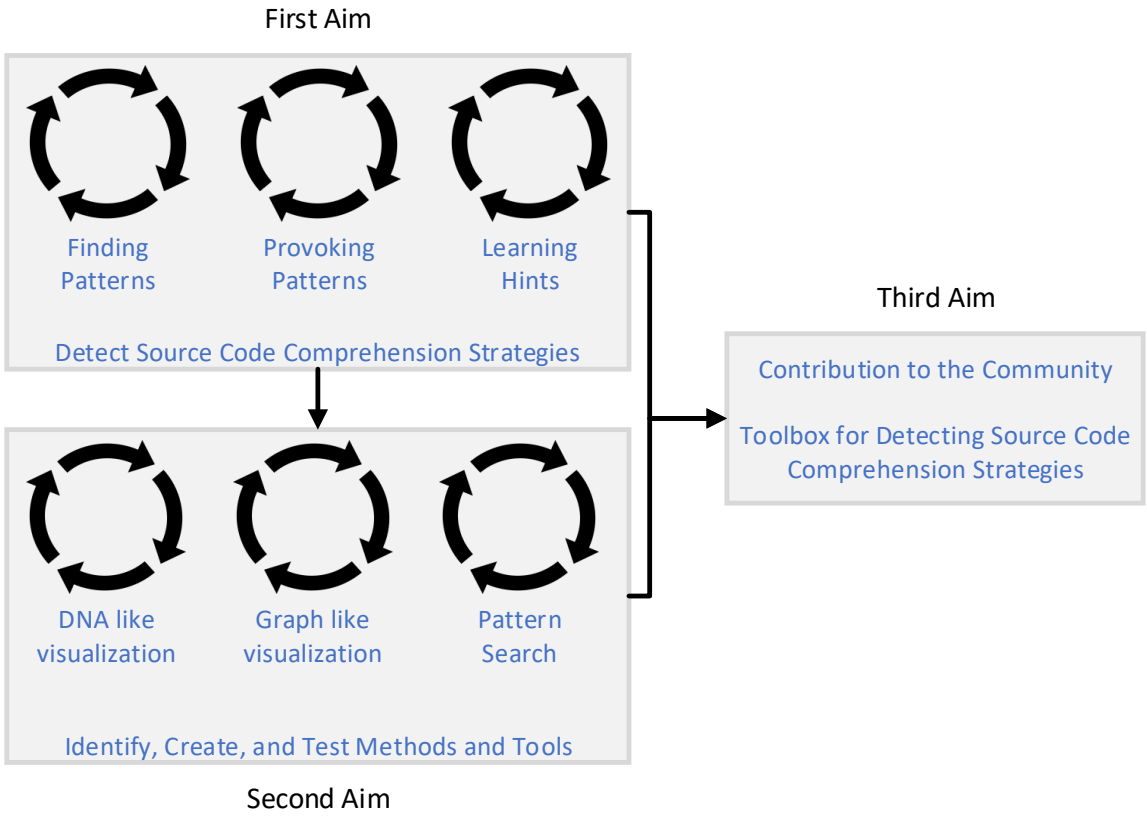


Figure 1.1: A brief overview of the three aims of this thesis.

Eye Movement as Focus of the Thesis

Building a dynamic learner support system requires many technical and organizational aspects to be considered. Such a support system likely requires additional input channels from learners, such as screen and typing activities, and errors from the underlying compiler of the programming language. This thesis focuses on eye movement-related aspects of the formulated aims. When necessary, other aspects are considered when evaluating the proposed solutions. To effectively follow the goal of closing the recognized gaps regarding the theory and methods for building a dynamic learner support system, a research agenda was prepared to lay out the research path for the three case studies. The research for the theoretical approach involves the following three steps:

1. Finding eye movement reading patterns linked to comprehension strategies. This first step is essential because detecting eye movement reading strategies and patterns is one thing, but knowing that the origin of a specific pattern is a comprehension hurdle is a necessity for a support system.

2. Using source code smells in a source code comprehension setting to provoke behavioral reading patterns, which can be categorized as comprehension strategies and are

detectable as characteristic eye movement patterns. Relying on data from other researchers and their empirical studies can be a prime step and a head start in the analysis process. However, for an optimal relationship between eye movement strategies and source code comprehension hurdles, a self-conducted study provides more control over the setting and the data collection.

3. Testing the effectiveness and efficiency of learning hints. Detecting source code comprehension hurdles is essential to help learners cope with problems; however, the question remains as to which method of helping them is the best to be active and have an effect on the learner. This is a way of designing an interaction mediating between the presentation of source code and the learners misconceptions about the source code, which manifest themselves as comprehension hurdles.

Knowledge Gain and Tests of Methods and Tools

All steps are necessary and equally crucial for designing a dynamic learner support system. In parallel, the research for the methodical approach also follows three steps: 1) using a DNA metaphor to visualize eye movement data, 2) using a modified state transition graph based on the DNA metaphor to visualize eye movement patterns, and 3) enhancing the DNA metaphor to search for eye movement patterns. The case studies are used to test the methods and to drive both research agendas forward; this defines the outline for this thesis. The general goal is the gain of knowledge about reading strategies and the effect of learning hints, as well as the identification, development, and testing of methods and tools to realize the vision of a dynamic learner support system.

1.4 OUTLINE

The Scope

The previous sections described the problem statement, the challenges regarding the overall vision of the dynamic learner support system, and the aim and research agenda. The remainder of this thesis is outlined as follows, with the general structure illustrated in Figure 1.2.

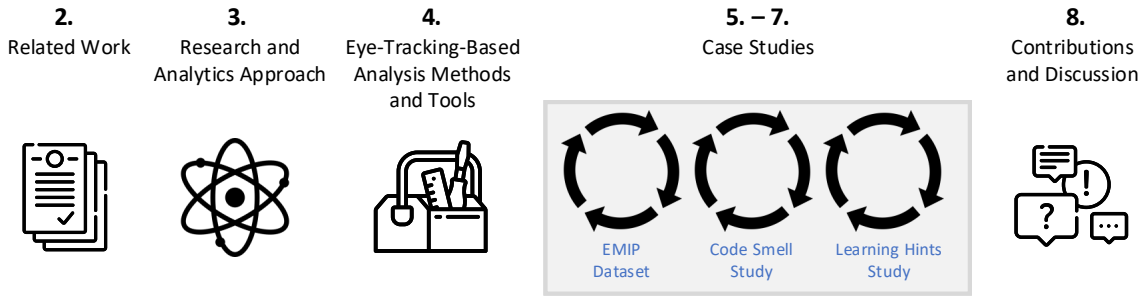


Figure 1.2: A brief overview of the outline of this thesis.

Related Work

Chapter 2 summarizes the related work for the main research areas, in detail learning to program, source code comprehension, eye movements in source code comprehension, methods for analyzing eye movement patterns, and learning analytics in combination with

eye tracking. The summary section recapitulates missing approaches, methods, and tools to fill some gaps on the way in which to create a dynamic learner support system.

Research Agenda and Data Analytics

Chapter 3 provides a detailed description of the approach from a research agenda and a data analytics perspective. It narrows down the missing points in theory and methods, and it lays out the three illustrating case studies. In addition, it visualizes the adapted data analytics approach that is used to design and conduct the data analysis of the three case studies.

Eye Tracking Based Methods and Tools

Chapter 4 outlines eye-tracking-based analysis methods and tools containing methodical eye tracking challenges, the influence of varying area-of-interest (AOI) models, standard analysis techniques for eye tracking, and the methods and tools developed in this thesis (e.g., AOI-DNAs, and AOI-STGs).

Three Case Studies

Chapters 5, 6, and 7 address the three case studies, specifically the analysis of a public eye tracking dataset, and the design and analysis of two new studies – a code smell study, and a learning hints study – both of which are essential to test theories and methods.

Discussion and Summary

Finally, Chapter 8 provides a summary of this thesis, aiming to describe the contributions regarding the three aims, including the limitations and open issues, a retrospective of the methodical eye tracking challenges, and future work ideas.

"The most elementary and valuable statement in science, the beginning of wisdom, is 'I do not know'."

– Lt. Commander Data (TNG, Season 02, Episode 02)

2 Related Work

This chapter outlines the main research areas of this thesis and the goal of designing the dynamic learner support system. Based on the problem statement (see Section 1.1), the six challenges identified when building such a system (see Section 1.2), and the aims of this thesis (see Section 1.3), this related work chapter contains a broad spectrum of important literature, designed as a top-down approach. A brief overview of the spanning research areas like computer science education and programming education is covered first. As a follow-up, an overview of the advantages, challenges, already available support systems, and mistakes while learning to program get discussed. Subsequently, an introduction to source code comprehension and its various sub-areas important for this thesis' aims is made. Afterward, learning analytics, and eye movements specifically for the area of source code comprehension, as well as methods for visualizing and analyzing eye movement patterns, are provided. In conclusion, the last section summarizes and clarifies, which theoretical aspects, methods, and tools are missing to create a dynamic learner support system for source code comprehension successfully.

2.1 Spanning Research Areas

STEM Education

George Forsythe is considered one of the pioneers of computer science, which can be seen as an invention to teach everyone about anything. Precisely, it is "the study of computers and all the phenomena associated with them" (Newell, Perlis, & Simon, 1967). Forsythe claimed in 1968 that the three most valuable tools in STEM education are 1) language, 2) mathematics,

and 3) computer science. Forsythe wrote about this new term for the first time in an article in the Journal of Engineering Education (1968). Donald Knuth told in 1972 a story about the journal article from Forsythe. Knuth wrote:

> "He [Forsythe] identified the 'computer sciences' as the theory of programming, numerical analysis, data processing, and the design of computer systems, and observed that the latter three were better understood than the theory of programming, and more available in courses." (Knuth, 1972, p. 722)

Programming as a Skill and in Educational Settings

Besides, Forsythe also stressed both experimental and theoretical computer science, with references to physics, where good experimental work can win many prizes. According to him, this should be possible for computer science, too. Since then, computing education, and the corresponding research, have a long history (Guzdial & du Boulay, 2019). Teachers are educating students about computing for years and in many different ways, e.g., for ever-changing programming languages. The first wave of computing education research targeted the theoretical and empirical studies of programming as a skill (Sime, Arblaster, & Green, 1977; Weinberg, 1985). Subsequently, another wave of research focuses on learning programming in educational settings (Feurzeig, Papert, Bloom, Grant, & Solomon, 1970; Papert, 1980). Seymour Papert and his colleagues were one driving force at this time. Papert wrote:

> "[...] that children can learn to use computers in a masterful way, and that learning to use computers can change the way they learn everything else have shaped my research agenda on computers and education." (Papert, 1980, p. 8)

Open Research Questions

In essence, computing education research is to "explore how humans come to understand computing and how to improve that understanding" (Guzdial, 2008), which is essential to make "computational thinking" (CT) accessible for everyone. To date, many important research questions from computer science education research are still open. Mark Guzdial publishes a new list nearly every year. The last one is from February 2019[2]. This comprehensive list, as well as other research (Denny, Becker, Craig, Wilson, & Banaszkiewicz, 2019), shows, that researchers and teachers have to find a better overlap between questions that get asked in computing education research and the questions that are considered necessary. Examine their assumptions could be a useful goal for both of them.

Computer Science Education Research

The research field important for this thesis is computer science education research (Busjahn, Schulte, Sharif, et al., 2014). In source code comprehension (also known as *program comprehension* or *code comprehension*), a more specialized field of research in computer science education, eye tracking is used in various ways. In early research, Crosby and Stelovsky detected distinct reading patterns for algorithms written in pascal (Crosby & Stelovsky, 1990).

[2] `https://computinged.wordpress.com/2019/03/11/research-questions-from-the-cs-education-research-class-february-2019/` – Last accessed on December 14, 2020.

Other examples are analyzing the expert's (Bednarik, Busjahn, & Schulte, 2013) and novice's gaze (Busjahn, Schulte, & Tamm, 2015), to find differences between these skill groups. Within these mentioned fields, this thesis has its focus in computing education → programming education → source code comprehension, as one dimension needed to learn to program and to develop a dynamic learner support system.

Externalize Mental Processes

One core aspect of source code comprehension, and thus of this thesis, is to analyze the comprehension of source code written in a programming language as a form of diagnostic understanding of the underlying processes. In order to understand involved processes while reading source code, mental activities present must be made visible. While there is no direct way to reveal these processes directly, researchers have used methods like post-tests, introspection, concurrent think-aloud, retrospective think-aloud, and analyses based on artifacts. The goal is to externalize what one is thinking with indirect methods. Consequential, source code comprehension, as a way to externalize mental processes while reading source code, is depending on eye tracking as a form of direct method to gather attention dynamics and sequences. This method is usable in source code comprehension only scenarios when diagnosing the reading behavior is the primary goal, and analyzing artifacts is not an option, due to missing create or edit cycles.

Eye Tracking as Technology

Fortunately, eye tracking as technology is becoming more and more affordable. Besides, the devices are getting more reliable, smaller, and data more accurate. Today, only a small sensor array is needed to track someone's eye movements. Eye tracking is distinguishable into two use cases: analytical and interactive. This thesis covers both aspects. The analytical approach to capture eye movement data to analyze source code comprehension processes, and the interactive approach, to analyze, trigger, and test the usage of dynamic learning hints. Eye tracking foundations such as fixations, and saccades are necessary for both use cases (Duchowski, 2017). Eye tracking as a data source adds an objective source of information about user behavior, which is an important reason why eye tracking as a methodology gets gradually applied to various fields of science. Keith Rayner has made major contributions in the area of text reading (Clifton et al., 2016). Additional non-computer science fields are, e.g., chess (Blignaut, Beelders, & So, 2008; Reingold & Sheridan, 2012; Van Der Maas & Wagenmakers, 2005), (text) reading (Rayner, 1998), piloting (A. et al., 2005), mammography (Sridharan, Bailey, McNamara, & Grimm, 2012), surgery (Vine, Masters, McGrath, Bright, & Wilson, 2012), and medical-dental images (Castner et al., 2018). Another research area is the teacher and learner support, e.g., for analyzing how the teacher and learner interaction is perceived visually (Dessus, Cosnefroy, & Luengo, 2016; Sharma, D'Angelo, Gergle, & Dillenbourg, 2016), gaze visualizations for remote collaborative work (D'Angelo & Gergle, 2018), to improve the communication between pair programming with a shared gaze awareness (D'Angelo & Begel, 2017), to improve instructional design for learners (Jarodzka, Holmqvist, & Gruber, 2017), or, more general, gaze-based interactions (Duchowski, 2018). Besides these

examples, eye tracking as a method gets used in settings like mobile eye-tracking experiments and applications (Prieto, Sharma, Wen, Dillenbourg, & Caballero, 2014), computer-supported collaborative learning (Schneider et al., 2018; Sharma, Chavez-Demoulin, & Dillenbourg, 2017), teacher students settings in classrooms (Dessus et al., 2016), and Massive Open Online Courses (MOOCs) (Sharma, Alavi, Jermann, & Dillenbourg, 2016; Sharma, Jermann, & Dillenbourg, 2015). The next section discusses learning to program from various perspectives, e.g., learning and teaching introductory programming, programming knowledge, novice mistakes, and learner support systems.

2.2 LEARNING TO PROGRAM

Learning Programming is Difficulty

The process of learning programming is a long and arduous journey. According to Winslow (1996), reaching the level of an expert programmer takes about ten years; a generally agreed circumstance by other researchers (Robins, Rountree, & Rountree, 2003). According to Dreyfus, Drey-fus, and Zadeh (2008), a commonly cited source, the continuum between the novice and expert levels can be separated into the five stages 1) *novice*, 2) *advanced beginner*, 3) *competence*, 4) *proficiency*, and 5) *expert*. This skill difference and the difficult learning process cause the need for a good computer science education.

Programming as an Essential Skill

This section provides an overview of the literature regarding programming as an essential skill and its importance, learning and teaching introductory programming, the characterization of programming knowledge, common novice mistakes and problems, and systems for an (intelligent) learner support are discussed. These topics form a body of knowledge of why programming is important and challenging to teach and learn, how programming knowledge is formed, which mistakes often occur in the learning process, and already tested (intelligent) systems to help learners.

2.2.1 PROGRAMMING AS A COMMON SKILL

21st-Century Literacy

Computing education itself, and learning to program as a characteristic embedded in it, is seen as a foundation for 21st-century literacy and is becoming more important in our technological world (Tabesh, 2017). In his SIGCSE 2019 keynote[3], Mark Guzdial describes computing is a medium for "Engineering Thinking", "Scientific Thinking" (Krajcik & Merritt, 2012), and "Historical Thinking." It asks questions (for science) and defines problems (for engineering). At the moment, educators use a variety of technological tools, programming environments, and educational concepts to teach computing. Many of them can provide challenging and dynamic coding experiences.

[3] https://computinged.wordpress.com/2019/03/02/a-crowdsourced-blog-post-about-my-sigcse-keynote-computing-education-as-a-foundation-for-21st-century-literacy/ – Last accessed on December 14, 2020.

Computational Thinking (CT)

In computing education, and therefore in programming as a basic skill for problem-solving, CT is an important ingredient. Seymour Papert (Papert, 1980, 1996) first used CT as a term. The phrase itself was made popular by Wing (2006), who describes it as a "universally applicable attitude and skill set everyone, not just computer scientist, would be eager to learn and use." (Wing, 2006, p. 33) For Wing, "computational methods and models gives us the courage to solve problems and design system that no one of us would be capable of tackling alone." (Wing, 2006, p. 35) Besides, "a central element of CT is abstraction in the specific sense it has in the context of computational principles" Ulrich Hoppe and Werneburg (2019, p. 19), which was highlighted by Wing (2008). Later, she describes CT as a shorthand for "thinking like a computer scientist" (Wing, 2014). Mastering these skills does not mean just thinking like a computer scientist, it means thinking at multiple levels of abstraction, which is a skill set, not only important for the desirable general knowledge of science, technology, engineering, and mathematics (STEM), but for everyone for everyday life.

End-User Programming

One often overlooked domain of utilizing knowledge in STEM and the mentioned meta-level skills, is the so-called *end-user development*, also called end-user programming. In 2006, this was called an emerging paradigm (Lieberman, Paternò, Klann, & Wulf, 2006). Estimated numbers from 2005, which rely on the improved calculation method of (Boehm et al., 1995), extended the original *55 million* people in America to *90 million*, who are using some sort of programming, spreadsheets, and databases (Scaffidi, Shaw, & Myers, 2005). This estimation seems way off from today's perspective, considering the rapid development in technology and the current digitization. In 2010, a CHI Special Interest Group Meeting about end-user software engineering picked the end-user programming topic up to move forward the research in software engineering methodologies and reduce the number of errors in end-user software (Myers et al., 2010).

Two Purposes in Learning Programming

Learning to program can serve two primary purposes: 1) learning through programming or 2) learning programming in its own right (Mendelsohn, Green, & Brna, 1990). Both directions are not distinct, because learners can switch between them in the learning process when their motivation, goals, and intentions changed or if a teacher moves from one topic and learning goal to another. These different learning intentions got taken into account in different research directions. Researchers like Feurzeig and Papert had the intention to teach mathematics and problem-solving through programming, which is the first purpose (Feurzeig et al., 1970; Papert, 1980). In contrast, researchers like Friend (1975) and Youngs (1974) studied the errors learners made in languages, and which programming constructs were the most difficult to learn, which tends towards the second purpose.

Overall, learning to program can serve different goals. Even if a learner does not strive to be a professional developer, a glimpse of computer science education and understanding can change the way the world is perceived and, therefore, a goal worthy of pursuing. However,

learning and teaching introductory programming has its challenges, as the next section shows.

2.2.2 LEARNING AND TEACHING INTRODUCTORY PROGRAMMING

Demanding Courses and High Failure Rates

Introductory programming courses, often referred to as CS1 and CS2 courses (Hertz, 2010), are demanding and cognitively challenging. Besides, a common conception is that it is challenging to learn to program (Hanks, McDowell, Draper, & Krnjajic, 2004; Jenkins, 2002; Robins et al., 2003). According to Bergin and Reilly (2005, p. 293) "it is well known in the Computer Science Education (CSE) community that students have difficulty with programming courses and this can result in high drop-out and failure rates." While it is agreed upon that learning to program is hard, the evidence for the immediately mentioned high drop-out and failure rates is, at best, an anecdotal evidence. Studies showed that the pass rate is highly dependable on the Country, course size, institutional-grade level, used programming language, and the year, the course was taught. The study from Bennedsen and Caspersen (2007), and the revised one from Watson and Li (2014), show an almost identical pass rate of CS1 courses: approximately 67 %. Beaubouef and Mason (2005) analyzed this and stated that drop-out and failure rates reaching as high as 30 %-40 % between the first and second years of programming courses. Also, it must be noted that such studies face low answer rates for their questionnaires. Thus, the generalizability of the results is at least debatable.

Improve Teaching and Learning Experiences

Despite the unclear data about drop-out and failure rates in introductory programming courses, decades of research targets teaching, technology, different approaches, and automated feedback systems. The overall goals are improving the teaching and learning experiences, and to lower the drop-out and failure rates (Yadin, 2011). A literature survey on teaching introductory programming revealed that there is only a limited effect on classroom teachings due to the active research (Pears et al., 2007). Improving introductory programming courses can, e.g., rely on pair programming, peer instruction, and media computation, which were reported to work, at least at universities (Porter, Guzdial, McDowell, & Simon, 2013). Others rely, e.g., on game first approaches (Leutenegger & Edgington, 2007), specialized learning environments (Singh, Gulwani, & Solar-Lezama, 2013), programming languages like Scratch (Resnick et al., 2009), Snap! (Harvey & Mönig, 2010) with a functional programming perspective, or syntactically simple languages and formative feedback on successful and erroneous aspects of code (Koulouri, Lauria, & Macredie, 2014). Extensive literature surveys point out many directions when planning introductory programming courses, but the vast amount of different approaches does not allow for canonical answers to how to teach programming (Pears et al., 2007). Others, like Brown and Wilson (2018), compiled a more pragmatic list of tips, in their article "Ten quick tips for teaching programming." He suggests, e.g., using peer introduction and live coding, pair programming, worked examples with sub-goals, and authentic tasks.

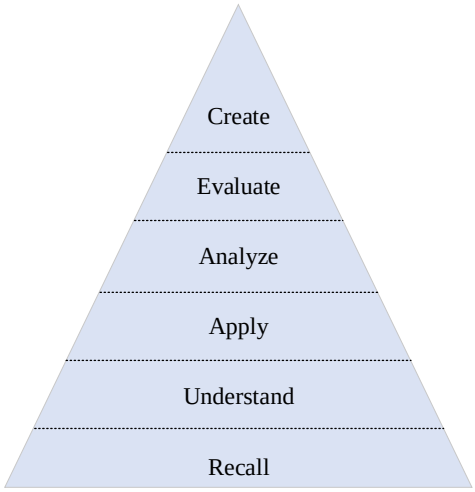


Figure 2.1: The learning objectives for the cognitive domain of Bloom's taxonomy. Level names from (Anderson et al., 2001).

Taxonomies for Learning Objectives or Outcomes

A vastly different attempt from the approaches mentioned above is the use of taxonomies, e.g., to sort learning objectives by cognitive complexity (Bloom) or to sort learning outcomes by structural complexity (SOLO). Both taxonomies allow the material of an introductory programming course to be adapted to the needs and skills of learners. Thus, the tasks are not implicitly designed for the hypothetical average student (Lister & Leaney, 2003).

Bloom's Taxonomy

The taxonomy by Bloom and others (1956) divides learning into the *cognitive*, *affective*, and *psychomotor* domains. For this thesis, the cognitive domain is the most important. "Bloom's taxonomy" is used in two ways: Referring to the taxonomy with the mentioned three domains or referring learning objectives within the cognitive domain. The latter one is meant in this thesis when mentioned "Bloom's taxonomy." Figure 2.1 shows the learning objectives for the cognitive domain (levels names from Anderson et al. (2001)). The original six levels, from bottom to top, are 1) knowledge, 2) comprehension, 3) application, 4) analysis, 5) synthesis, and 6) evaluation. Thus, the comprehension part is the second-lowest level and, therefore, important in learning to program. Applying this taxonomy to programming is challenging, but the taxonomy has motivated improvements to the instruction of introductory programming courses (Alaoutinen & Smolander, 2010; Khairuddin & Hashim, 2008). Besides these efforts and accomplishments, Bloom's taxonomy shows that introductory programming courses are challenging. With the taxonomy's help, one can link the goals of typical programming courses with elements of the taxonomy. For example, creating small programs in a course, for solving a problem described in non-programming terms, matches the level of synthesis or creation of the taxonomy.

SOLO Taxonomy

The SOLO taxonomy contains five levels of responses from learners, measured by the structural complexity of the result: 1) pre-structural, 2) uni-structural, 3) multi-structural, 4) relational, and 5) extended abstract (Biggs, John, Tang, & Catherine, 2011, pp. 77-78). This taxonomy is used to analyze programming assignments, e.g., code-reading and code-writing

tasks. From the viewpoint of these tasks, the levels of the SOLO taxonomy are interpreted differently as a code-reading (comprehension) task, e.g., pre-structural means, that a learner substantially lacks knowledge of programming constructs. For a code-writing task, this level means the inability to write the correct code. Thus, the taxonomy can be used to analyze both dimensions. E.g., Lister, Simon, Thompson, Whalley, and Prasad (2006); Whalley, Clear, Robbins, and Thompson (2011) have done this for introductory programming courses. SOLO teaches us, that the expected outcome of introductory programming courses is complex, for both code-reading as well as code-writing. For both skills, the successful completion of a programming task often requires a relational level of understanding, which is not easy to achieve in an introductory course.

No Programming Skill Hierarchy

Despite the efforts from researchers and educational staff, not all students reach a reasonable level of competence regarding programming education. Carter and Jenkins (1999) stated, that "few teachers of programming in higher education would claim that all their students reach a reasonable standard of competence by graduation. Indeed, most would confess that an alarmingly large proportion of graduates are unable to 'program' in any meaningful sense." (Carter & Jenkins, 1999, p. 1) Besides, research shows that "the average student does not make much progress in an introductory programming course," according to a CER literature review from the 1980s (Robins et al., 2003). Furthermore, many students struggle to get past learning basic language features (Dalbey & Linn, 1985). In summary, research shows, that students do not learn how to create working programs (Soloway, Lochhead, & Clement, 1982; Tew & Guzdial, 2011; Venables, Tan, & Lister, 2009), that they do not learn to read code properly (Adelson & Soloway, 1985; Simon, 2011; Teague, Corney, Ahadi, & Lister, 2012), and that many students poorly understand fundamental programming concepts (Samurçay, 1989; Sanders, Galpin, & Götschi, 2006; Sleeman, Putnam, Baxter, & Kuspa, 1986). The empirical work of the BRACElet project, which is based on the SOLO taxonomy (overview in Clear et al. (2011)), shows that a learning hierarchy of programming skills cannot be given at the moment.

Minimal Competence at Tracing and Explaining

In conclusion, all kinds of problems understanding source code while learning to program will cause more problems in the learning process. Therefore, these problems in understanding must be addressed. Regarding the hierarchy of programming skills and the role of source code reading and comprehension, this thesis targets the following conclusion:

> "In arguing for a hierarchy of programming skills, we merely argue that that some minimal competence at tracing and explaining precedes some minimal competence at systematically writing code. Any novice who cannot trace and/or explain code can only thrash around, making desperate and ill-considered changes to their code – a student behavior many computing educators have reported observing." (Venables et al., 2009, p. 128)

2.2.3 CHARACTERIZING AND STRUCTURING PROGRAMMING KNOWLEDGE

Categorizing Programming Knowledge

This section describes research about the questions, what learning to program involves when students succeed, and what characterizes programming knowledge. The latter is a broad field with lots of seminal theoretical work. This section focuses on categorizing programming knowledge associated with source code comprehension to narrow down this research field.

Influence Programming Knowledge

Linn and Dalbey (1985) describes a "chain of cognitive accomplishments", consisting of three elements: 1) single language features, 2) design skills, and 3) general problem-solving skills (Linn & Dalbey, 1985, pp. 58-60). These can be amended with more specific skills, like templates, and procedural skills (planning, testing, reformulating). Besides the chain, the work covers precursor stages, which can concern the learning of language features. Also, "instruction strongly influences outcomes in introductory programming classes" (Linn & Dalbey, 1985, p. 76), which even influences the ablest students, a group that "learned significantly more in exemplary classes than in typical classes" (Linn & Dalbey, 1985, p. 76). Instruction also influences the "medium ability students who did as well as high ability students in exemplary classes and were far more successful than medium ability students in typical classes" (Linn & Dalbey, 1985, p. 77). These findings are crucial because they show that proper instructions in introductory programming courses influence programming knowledge.

Framework for Knowledge Representation

In the 1990 published book, Rogalski and Samurçay (1990) presented a framework for the analysis of programming. In detail, the work shows a general framework for knowledge representation, which is tailored to the programming field. In this framework, the difficulty of novices are divided into four areas: 1) conceptual representations about the computer device, 2) control structures, conditional statements, iterations, and recursion, 3) variables, data structures and data representation, as well as 4) programming methods. The overall conclusion is that acquiring programming knowledge is a complex process, which involves many aspects like program design, understanding, modifying, and debugging from a more bird's-eye view, and the structuring of basic operations like loops, conditional statements on the computer literacy level, into underlying schemas and plans, which will be discussed in Section 2.3.2. Schema and plans are an essential aspect of source code reading and source code comprehension. The work of Rogalski and Samurçay (1990) also mentioned, that the strategies developed by students have to be adaptable enough, to "benefit from programming aids" Rogalski and Samurçay (1990, p. 170). This aspect is essential, because programming environments offer a wide range of supporting features, which a) are often not or not sufficiently used by novices, and b) should influence the choice of the programming environment, novices have to use in introductory programming courses (Kelleher & Pausch, 2005).

Distinct Types of (Programming) Knowledge

A different approach to structuring student's knowledge of programming was made by McGill and Volet (1997). The work identified three distinct types of programming knowledge: 1) syntactic, 2) conceptual, and 3) strategic. The source of these types is the educational com-

puting literature, especially Bayman and Mayer (1988), a work build on earlier research of Boysen and Keller (1980), Shneiderman and Mayer (1979), and Linn (1985). Besides, McGill and Volet (1997) identifies three distinct types of knowledge in the cognitive psychology literature: 1) declarative knowledge ("know that"), 2) procedural knowledge ("know how"), and 3) conditional knowledge ("know when, where, and why"). All types are then combined into a conceptual framework of the various components of programming knowledge. The resulting two-dimensional framework was used to re-analyze data from an experimental study of Volet (1991). The analysis showed that the model has "[...] potential for detecting deficiencies in the programming knowledge of novices programming during a course of instruction, and for designing appropriate instruction in introductory programming." (McGill & Volet, 1997, p. 276) The analysis of the programming knowledge consists of analyzing examination questions and a classification, which types of knowledge are required to answer these questions.

Four Learning Steps as Theoretical Framework

More recent research creates a theoretical framework, which distinguishes between four learning steps: 1) reading semantics to trace code and predict the effect of syntax on behavior (Xie et al., 2019, p. 10), 2) writing semantics, referring translating of unambiguous natural language descriptions into syntax (Xie et al., 2019, p. 10), 3) reading templates, to identify reusable abstractions of programming knowledge (templates) (Xie et al., 2019, 12), and 4) writing templates, to be able to use program templates to solve a problem (problem-solving) (Xie et al., 2019, p. 12). The theoretical framework can be summarized as "read before write and semantics before templates," highlighting the importance of code reading and source code comprehension as first, as well as distinct tasks and understand the semantics of the underlying programming language before trying to understand larger blocks of code combined as a template.

Programming Skill Acquisition

In contrast to the theoretical frameworks, which aim at describing programming knowledge and skill from a more teaching and technical perspective, other research focus on the relationship between programming skill acquisition and individual differences of learners. Shute (1991) investigated the effect of prior knowledge, general cognitive skills, problem solving abilities, and learning style measures. The analysis is based on 260 participants in a seven-day study, receiving Pascal instructions from an intelligent tutoring systems. No participant had prior knowledge of Pascal. One outcome of the study is, that the acquisition of programming skills is highly correlated with certain problem-solving abilities. Because these abilities are trainable (Bransford & Stein, 1993), introductory programming courses "may benefit from the inclusion of supplemental instruction on relevant problem-solving skills." (Shute, 1991, p. 16) Another result is, to alter the teaching process to include small chunks of knowledge for participants with a lower working memory. This is doable in an (intelligent) programming environment or tutoring system, but hardly achievable in day to day settings of introductory programming courses. Nevertheless, the working memory

capacity "[...] was shown to be an important predictor of programming skill acquisition." (Shute, 1991, p. 17) Overall, the work of Shute (1991) argues for adjusting the teaching according to the individual differences of learners, based on a study using an intelligent tutoring system (ITS), an approach, discussed in more detail in Section 2.2.5.

Chunking, Schemas, and Mental Models

Psychology provides an additional view of programming knowledge. This research field contains information about how complex information is processed, the link to the working memory, and how chunking and schemas have overcome the limits of the working memory. Besides, mental models help learners deal with complex environments. These topics are discussed in more detail in Section 2.3.2, because they play a crucial role in source code comprehension. These concepts clarify that learning to program requires mental representations from a) the problem domain, b) models of the problem and c) problem-solving patterns. Besides, one important aspect is often neglected in introductory programming courses: Students have to learn the characteristics of the underlying computer system.

Notional Machine as an Abstraction

In this context, Du Boulay (1986); Du Boulay, O'Shea, and Monk (1999) introduced the term notional machine, a description for the "general properties of the machine that one is learning to control." (Du Boulay, 1986, p. 57) It describes environmental parameters to a program and explains the essential characteristics of a programming language. Notional machines are tight to programming languages, which is why every programming language has a (slightly) different notional machine. Misconceptions about this notional machine can cause problems along the way of learning how to program (Perkins, Schwartz, & Simmons, 1988). The lack of a proper mental model for the underlying machine leads to all kinds of ideas from learners about how statements in a programming language are executed. Besides, mental model theory shows that a wrong mental model is hard to replace by a correct one. Novices needs to understand this notional machine because it is an abstraction of the computer that executes a program. The misconceptions novices have about a notional machine are one fundamental aspect and explanation for their lack of ability to write programs. Moreover, tracing the program execution, such as debugging, needs a good knowledge about the notional machine, because humans trace programs via executing the programming statements in a mental model. Tracing is furthermore vital for source code comprehension. Thus, knowledge about the notional machine and an excellent mental model can help in the process of source code comprehension.

No Problem Hierarchy

When summarizing findings from research and literature, the question arises, which of the mentioned problems is the most important one? As a conclusion of this section, all challenges are equally important and, to complicate things furthermore, not independent of each other. Tracing skills and understanding source code is one of the challenges and one of the primary topics of this thesis. As the next sections show, novices make various mistakes and have various problems understanding programming and language concepts.

2.2.4 NOVICES MISTAKES AND PROBLEMS

Programming Languages for Humans

In the past decades, extensive research on the topic of novices and experts programming skill were conducted. Du Boulay et al. (1999) provides an overview of the early work on empirical studies of programming. Besides, Shneiderman (1975) noted already in 1975 that researchers and language developers had begun to accept that humans are not just a marginal factor in the programming process. They are essential when designing programming languages and teaching how to program. It seems logical that many mistakes of novices could be avoided, or their frequency reduced if the design of programming languages would target beginners – a topic with recent research activity not covered in this thesis (Kaijanaho, 2015; Stefik & Siebert, 2013).

Novice and Expert Differences

In general, novices experience learning to program in a different way than experts experience the programming task itself. They have entirely different needs and face various challenges during the programming process. The study of Wiedenbeck (1985), in which novices and experts have to make timed decisions on short program segments, showed that "[...] experts were significantly faster and more accurate than novices." Wiedenbeck (1985, p. 383) This result is attributed to automation's effect, which is the ability to perform a task rapidly, smoothly, and correctly with little attention. Another vital area of research is the difference in educator beliefs and student data regarding novice programming mistakes. In introductory programming courses, educators form beliefs about the errors their students make regularly. Research on eighteen mistakes in Java shows that there is almost no consensus between educators regarding the frequency of novice errors and that the level of expertise of an educator has no effect on how closely their belief of an error frequency matched the black box data (Brown & Altadmri, 2014). A similar conclusion was drawn by Spohrer and Soloway (1986). The authors found out that the instruction in introductory programming courses can be improved when computer science instructors strive to "[...] familiarize themselves with specific high-frequency bugs, [...]" (Spohrer & Soloway, 1986, p. 632) In programming and computing education research, novices' difficulties with programming and the design of programming languages targeting novices are two main areas of interest (Guzdial & du Boulay, 2019).

Students do not Learn Programming

As mentioned in Section 2.2.2, many students do not learn how to write functioning programs. Not to mention good programs, or right source code concerning coding, formatting, and task-specific standards. "[...] an alarmingly large proportion of graduates are unable to 'program' in any meaningful sense." (Carter & Jenkins, 1999, p. 1) Not an unexpected result, but an outcome of many years with deep grounded problems in the understanding of computer science and programming concepts, taught by introductory programming courses.

Types of Novices and Problem Categories

Before this section dives into different aspects of problems and mistakes novices face while learning, it differentiates between different kinds of novices and different meta-level categories of problem areas. Perkins, Hancock, Hobbs, Martin, and Simmons (1986) differ-

entiates between the two categories, "stoppers" and "movers." Both are important for the goal of a dynamic learner support system. Stoppers simply stop to program, when facing a problem. "They appear to abandon all hope of solving the problem on their own" (Perkins et al., 1986, p. 41) These learners need more supervision or the help of an automatic system. On the contrary, movers keep trying, modifying their code to fix the problem they face. Movers can make sense of programming errors and error messages more often and use the feedback effectively (Robins et al., 2003). Regarding the meta-level categories of problem areas, it is necessary to distinguish between the categories 1) conceptual and 2) technical problems or misconceptions. McCall and Kölling (2014) created a categorization of novice programming errors. The extensive list can be found online[4]. The first two entries are "variable not declared" and "; missing", both combined, responsible for over 20 % of the errors analyzed in the study.

Tracing as a Skill for Novices

Besides, tracing abilities are necessary. In their research, Détienne and Soloway (1990) describe the differences between *concrete* and *symbolic tracing*. The former uses generic values to trace program execution, e.g., following the most basic steps of a program like "looping 10 times, adding a value if something is true, and printing the result." For experienced programmers, this is the default strategy when reading a new program. In contrast, symbolic tracing is used with real values to follow the program execution, e.g., "looping 10 times, adding the value 12 to the initial value zero, if the variable 'threshold' is below zero, adding the value 25, if the variable 'threshold' is above zero." Concrete tracing is an important skill when debugging programs. For novices, it is even more important, according to Vainio and Sajaniemi (2007). They argue that novices need to trace a program with real values to understand the relationships between source code elements like statements and loops. These relations are missing in symbolic tracing. Unfortunately, novices do not trace their programs if not prompted to do so (Perkins & Martin, 1986). According to Sleeman et al. (1986) "at least half of the students could not trace through programs systematically." (Sleeman et al., 1986, p. 22) Besides, Adelson and Soloway (1985) showed that novices cannot mentally trace interactions within a system, and Kaczmarczyk, Petrick, East, and Herman (2010) report an incapacity to "trace code linearly" as a major novice difficulty. Associated with problems in tracing programs, are difficulties with the program state. Sajaniemi, Kuittinen, and Tikansalo (2008) shows concrete examples of these difficulties, and Du Boulay (1986) stated that students are not always aware of the fact that "each instruction operates in the environment created by the previous instructions." (Du Boulay, 1986, p. 62) Besides, tracing a program means to run a mental model, which make tracing and the knowledge about models dependent on each other. Moreover, this knowledge about the model is based on the notional machine, which is another link of necessary concepts, which novices need

[4] `https://bluej.org/davmac/2014_novice_errors/categories.html` – Last accessed on December 14, 2020.

to understand. To summarize this area of problems, novices need a robust understanding at a low level of abstraction, for successfully tracing a program.

Misconceptions Based on Technical Issues

Another area of problems or misconceptions is based on technical issues while learning to program. Sleeman et al. (1986) analyzed errors during Pascal programming exercises in 1986, with the result that out of 35 students, "over half were classified as having major difficulties – less than 10 % had no difficulties." (Sleeman et al., 1986, p. 3) They also found out that "errors were made with essentially every Pascal construct." (Sleeman et al., 1986, p. 3) Pea and Kurland (1984) describes programming as a complex cognitive activity. In the learning process, novices have to face various errors on different levels of abstraction (Brown & Altadmri, 2017; Ebrahimi, 1994). Regarding the errors for language constructs, Ebrahimi (1994) summarizes the findings as "logical if", "input", "repeat until", "if with '='", "logical if and do while." Subsequent analysis showed that if statements and loops caused the most problems. According to Brown and Altadmri (2017), conditional statements, syntactic structures such as nested brackets and blocks, and mismatches of argument types are common challenges and cause many problems. In an early study of Youngs (1974), novice errors are categorized as "syntactic", "semantic", "logical", and "clerical." Novices have far more semantic errors, were for experts, errors were more evenly distributed. Another description for semantic error is an error in the logic of a program. After analyzing 15,000 code fragments in the C programming language, created by novices, Ettles, Luxton-Reilly, and Denny (2018) concluded that integer division, uninitialized variables, and indexing/iterating arrays are the most common misconceptions. A few kinds of logic errors occur very frequently. Research conducted by Izu, Weerasinghe, and Pope (2016) reveals that learners show a poor understanding of loops, loop ranges, and vector access. The results are backed up by an evaluation framework based on the SOLO taxonomy. The overall body of research targeting programming misconceptions is quite extensive. Madison and Gifford (1997) found problems in the concepts of parameter passing, Sleeman et al. (1986) analyzed code comprehension tests and interviews and found problems on various levels of abstraction like variables, assignment, print statements, and control flow.

Syntax Barrier

In addition to these technical issues, the syntax barrier, novices have to face, is an enormous problem. Sevella and Lee (2013) showed in their research that most of the barriers novices have to face are "[...] basic programming barriers, rather than conceptual barriers." (Sevella & Lee, 2013, p. 20) Others showed that in short drill and practice exercises, all students "were writing code that does not compile, and that weaker students were often unable to solve their syntax problems." (Denny, Luxton-Reilly, Tempero, & Hendrickx, 2011, p. 212) Overall, the study showed that students were struggling with syntax problems to a greater extent as the authors initially expected.

Compiler Error Messages

Another area of problems is compiler error messages or in general error messages from the programming environments. These errors are related to syntax errors, because such an

error inevitably leads to a compiler error message that needs to be understood. In Traver (2010), the poor design of error messages was analyzed. Besides, the work shows that a compiler error message is an interface between the computer and the human user. Therefore, methods of human-computer interaction (HCI) should be used to improve the messages. In studies where more precise and more understandable compiler error messages were used, the frequency of compiler errors is reduced (Becker et al., 2016). Error messages are especially problematic in situations where they are depended on each other. These are so-called multiple compiler error messages, where often only the first error is essential, and the rest can disappear after fixing the first one. This knowledge is not available for novice programmers. In an analysis of Becker et al. (2018), 21 million compiler error messages were examined. The results show that the frequency of errors stays consistent with other research when all error messages are considered. When only the first message is analyzed, these frequencies changed. Thus, it is worth focusing on the first error message, if multiple ones are present, and to teach novices this concept in introductory programming courses.

Helping Learners is a Necessity

In summary, programming languages designed for humans have improved many parts of learning programming. Overall, novices and experts have entirely different needs and face different challenges while programming. Moreover, analyzing the outcome of introductory programming courses showed that many students do not learn programming at all, which is a problematic result, given all the effort put into such courses, e.g., in universities. Novices are subject to many misconceptions while learning programming, for example, technical issues, syntax, and semantics. Current programming environments are not improving this situation much. Novices are making many mistakes and have numerous problems on the journey of learning programming. Hence, a way of helping them is necessary. One way of doing so is using an (intelligent) learner support, which is the topic of the following section.

2.2.5 (INTELLIGENT) LEARNER SUPPORT

Teaching Systems and Empowering Systems

Educators and researchers are well aware that supporting learners is a beneficial goal. Dividing the aspect of learning programming in two categories, "learning to program" or "programming to learn," according to Mendelsohn et al. (1990), can be used to categorize programming environments into two main categories 1) *teaching systems* and 2) *empowering systems*. This taxonomy was created by Kelleher and Pausch (2005), who wanted to lower the barrier to programming. The first category includes environments for improving programming abilities; the second category contains environments to help learners follow other learning goals.

Novice Programming Environments

Programming environments intentionally designed to help learners be categorized as novice programming environments (NPEs), and are included in the teaching systems category. Well-known examples are Greenfoot (Kölling, 2010), which uses structural highlighting, as well as Alice (Moskal, Lurie, & Cooper, 2004), Scratch (Moreno-Leon & Robles, 2016),

and Snap! (Harvey & Mönig, 2010), which avoid syntax errors with a drag-and-drop visual language, or *WIPE* (Efopoulos, Dagdilelis, Evangelidis, & Satratzemi, 2005), which is a web integrated programming environment to teach fundamentals of programming. NPEs are a commonly used to introduce programming, generally tailored to a specific domain-specific application domain. Empirical studies have shown that these environments improve test scores (Dann, Cosgrove, Slater, Culyba, & Cooper, 2012), and retention (Moskal et al., 2004). Besides, other NPEs combine programming with the interest of learners, like games, simulations (Utting, Cooper, Kölling, Maloney, & Resnick, 2010), and stories. These specialized environments are a good start in augmenting the learning process. However, these tools are still dependent on teachers available in an introductory programming course to help students when they get stuck and do not know how to proceed with a task.

Intelligent Tutoring Systems (ITS)

Intelligent Tutoring Systems (ITSs), which are environments specifically designed to help learners, are eager to close this gap. There are different types of ITS for various problem domains. Traditionally, ITS have been developed for well-defined domains, like algebra and programming (Anderson, Corbett, Koedinger, & Pelletier, 1995). ITS for ill-defined domains, such as reading and writing is often based on natural language processing (NLP) (Jackson & McNamara, 2013). According to Johnson, McCarthy, Kopp, Perret, and McNamara (2017), the combination of ITS and NLP "provide one-on-one training, practive, and feedback in ways that would be impossible in traditional reading and writing instruction given constraints of both time and resources in the classroom." (Johnson et al., 2017, p. 565)

ITS in Programming Education

The focus in this section is ITS in programming education scenarios. These environments can be categorized in the teaching systems category according to Kelleher and Pausch (2005). An ITS want to approximate a human in the role of a teacher or tutor to support learners (Price et al., 2017), to dynamically adapt the learning material to the individual learner (Murray, 1999), or to add complementary material directly into the programming environment to support learners (Brusilovsky, 1992). Such systems have proven to be helpful in programming education, where students perform two standard deviations higher than students who had no access to such an environment (Corbett, 2001). The discipline of bringing together NPEs and ITSs is relatively young. One pioneer is *iSnap*, which is an extension to the *Snap!* programming environment, adding, e.g., detailed logging and automatically generated hints, created by comparing current student's code to prior student solutions (Price et al., 2017). In Stamper, Eagle, Barnes, and Croy (2013), two two versions of the *Deep Thought* logic tutor were tested. The group with the tutor, tested over two semesters, completed significantly more tasks, then the group without the tutor. The ITSs mentioned above are designed to help learners in the process of programming. In contrast, methods like automated feedback generation are intended to help learners when submitting programming assignments. Singh et al. (2013) tested such an automated feedback generation, with a reference implementation of the assignment and common errors learners often make. When submitting the Python

code, the system generates a list of changes needed to correct the assignment. The result of the analysis is that the system can correct 64 % of incorrect solutions of a benchmark set. Another approach uses the *GradeIT* system, which uses automated grading and programming repairing in an introductory programming course (Parihar et al., 2017). According to the study and analysis, the systems graded similar to human teaching assistants (TAs) on 15,613 submissions. The authors stated that "the difference in marks was less than 20 % of the total marks for 88 % of the submissions." (Parihar et al., 2017, p. 97), and that the system was "more consistent than TAs when awarding marks for similar submissions." (Parihar et al., 2017, p. 97) The research of Shute (1991) used a Pascal tutor, to test, among other individual differences, if hints influence the learning outcome. Their results indicate that the task outcomes were not influenced by the hints, because "the ITS emphasized the higher, conceptual level of programming more than the lower level, syntactical aspect of Pascal coding." (Shute, 1991, p. 15) A meta-analytic review about the effectiveness of ITS, analyzing 107 effect sizes involving 14,321 participants from research found in major bibliographic databases, concluded, that "in some situations ITS can successfully complement and substitute for other instructional modes and that these situations exist at all educational levels and in many common academic subjects." (Kulik & Fletcher, 2016, p. 914) Overall, there was no significant difference in learning from a human tutor or an ITS. As a result, ITSs should no replace other modes completely. The meta-analytic review, and preview reviews by Steenbergen-Hu and Cooper (2013, 2014), examined evaluation research, in which the use of ITSs was compared to other modes of instruction. The work of Crow et al. (2018) focuses on a systematic review of ITS in programming education. These systems are called Intelligent Programming Tutors (IPTs). The overall result of the review and analysis is that IPTs benefits from including additional reference material, like "planning resources, reference material, worked solutions, and different types of questions." (Crow et al., 2018, p. 61)

ITS and Eye Tracking

In recent years, eye tracking is more and more used in analyzing how students learn and how they interact with NPEs and ITSs. Learning from examples is one area of research. The study of Najar, Mitrovic, and Neshatian (2014) showed how students are reading worked-examples in SQL in an SQL-Tutor called system. The analysis revealed that experts spent more time in some areas compared to novices, e.g., the database schema. Such information can later be integrated into the ITS, when combined with eye tracking, to help learners automatically. In the work of Belenky, Ringenberg, and Olsen (2014), the authors analyzed students' collaboration with an ITS for elementary-level fractions. They used a dual eye tracking setup and found out, that collaborating students showed learning gains, and that the dual eye tracking data shows, that learning gains were related to conceptual knowledge only in the procedural condition. Different approaches, regarding eye tracking and ITS, were conducted by D'Mello, Olney, Williams, and Hays (2012) as well as Hutt, Mills, White, Donnelly, and D'Mello (2016). In the previous work, the ITS was developed for dynamically

detecting and responding to students' boredom and disengagement, to be able to promote engagement and learning. The analysis revealed, that "in general just-in-time gaze-reactivity was quite effective in reorienting students' attention towards the tutor." (D'Mello et al., 2012, p. 23), and that "gaze-reactivity positively influenced learning gains, particularly at deeper levels of comprehension." (D'Mello et al., 2012, p. 23) However, one must remark, that off-screen gaze-behaviors persisted. Even if the system explicitly instructed to pay attention. Moreover, this eye tracking analysis, combined with an ITS, targets students' attention and not a direct intervention of the learning process. The latter mentioned research analyzed the gaze-based detection of mind wandering when learning with an ITS (Hutt et al., 2016). Their main result is that mind-wandering could be detected in around 50 % of the time.

Eye Movement Modeling Examples

Furthermore, eye tracking is used as a supporting technology in learning contexts, which refers to using a shared gaze in various ways. Thus, eye tracking is incorporated as a navigational mechanism, to guide others in different situations, like static workspaces, when interacting with provided objects, or when modifying within a given format. Schlösser (2020) described these different contexts and gave a classification scheme based on interdependence (work coupling), group size, and workspace interaction. Using a gaze can mean a *one-way* gaze sharing, like an expert-novices setting, in which the expert explains an algorithm (Bednarik & Shipilov, 2012). In a *two-way* gaze sharing setup, D'Angelo and Begel (2017) analyzed a collaborative pair programming refactoring tasks, in which the gaze was visualized as a 5 line high, 20 px wide rectangle within the source code, to highlight each participant gaze. A different approach for eye tracking in a learning context are eye movement modeling examples (EMME) (van Gog, Jarodzka, Scheiter, Gerjets, & Paas, 2009). An EMME is an example of solving a task, usually by an expert. The expert's gaze is recorded and then used as a cue for the learner to foster visual attention and gaze behavior. In combination with verbal explanations and comments from the expert, an EMME can impact learning and problem solving (Jarodzka, Van Gog, Dorr, Scheiter, & Gerjets, 2013). EMMEs use a human modeler (e.g., a teacher), to demonstrate a computer-based task, with eye movements superimposed on the task. A recent classroom study of Bednarik, Schulte, Budde, Heinemann, and Vrzakova (2018) used EMME in source code comprehension. The authors analyzed, if a combination of three modalities 1) dynamic visualization of the block model, 2) gaze replay of experts following this block model, and 3) a teacher's verbal commentary, can benefit a learner. The analysis found a positive impact on student's performance regarding the comprehension performance, and the correct answers for the source code examples. It is yet unclear if EMMEs are beneficial only for specific tasks or a generalizable approach (van Marlen, van Wermeskerken, Jarodzka, & van Gog, 2016). In this thesis, the gaze sharing approach is not pursued further. Instead, the analytical approach is being used, gathering knowledge about source code comprehension hurdles, and their effect on eye movement pat-

terns. The assumption is that this can pave the way for a support system based on source code comprehension characteristics.

Ambiguous Results for ITS

The role of ITS in learning programming is essential, yet the results are ambiguous. In some settings and domains, they work pretty well. In others, there is room for improvement (Crow et al., 2018). A topic with little research is the combination of eye tracking and ITS to match the reading behavior in source code comprehension tasks. Results in this area could help understand source code comprehension better and, in the overall vision, create a dynamic learner support system based on eye movement data. The next sections describe the research field source code comprehension, with all its challenges and opportunities.

2.3 SOURCE CODE COMPREHENSION

Reading Code is a Necessity

Writing source code often seems to be the only goal both in programming education and in industry. In contrast, to be able to write code, mastering reading is a necessity. Thus, it is advisable to separate the three tasks 1) source code comprehension, 2) source code manipulation, and 3) source code writing/creation. In some literature, comprehension is viewed or compared against code generation (Robins et al., 2003). All categories and tasks are equally important for developing software. However, source code comprehension is the baseline for the other two. Code reading, as a part of the source code comprehension process, is not often reflected on (Busjahn, Schulte, & Busjahn, 2011). This thesis focus on source code comprehension. In the research community, this task is predominantly known as *program comprehension* or *code comprehension*. This thesis deliberately uses the term source code comprehension, because it describes the research area in its entirety because in research source code is present in every form. Not only executable programs. Thus, the term source code is precise and covers every aspect of code.

High Amount of Software Maintenance

The importance of source code comprehension can be deduced from the overall effort of maintenance needed in the complete software life cycle. Maintenance has continuously risen over the last decades and is about 80-90 % of the overall effort in software projects (Piattini, Polo, & Ruiz, 2003). According to Foster (1993), developers spend 30-90 % of their time in software projects on reading source code. When learning to program, comprehending source code is a first necessity (Brooks, 1983), regardless of the specific programming task, like maintaining existing code, developing new software, or adding features (Shaft & Vessey, 1998). Additionally, a walk-through of source code is a crucial ability to locate bugs (Perkins & Martin, 1986). Reviewing code mainly relies on comprehending basic principles by "mental simulations" while reading (Détienne & Soloway, 1990).

Code-Tracing and Code-Explaining

Lopez, Whalley, Robbins, and Lister (2008) analyzed the relation between code-tracing and code-explaining. They report, that code-tracing or code-explaining alone only accounts for some of the observed variation in code-writing skills. However, both skills combined account for a substantial amount of the variation. Later results of Lister, Fidge, and Teague

(2009) are consistent with the earlier findings. The authors found a "statistically significant relationship between tracing code, explaining code, and writing code." (Lister et al., 2009, p. 164) Besides, interviews conducted by Busjahn and Schulte (2013) reveal that code reading is an integral part of comprehending programs and algorithms. Combined with comprehension, it is sometimes viewed as a learning goal itself, a goal worth pursuing. Research suggests that to comprehend source code, novices first analyzed programs regarding program models and later domain models. Thus, knowledge in terms of a particular program with specialized task knowledge embedded in it is formed earlier, and domain models form later with more expertise. Introductory programming courses should have the goal of taking novices on the road of schema-building. According to schema theory, expertise is a path of knowledge-building, and underlying schemas can form more complex ones. This result is an essential part of arguing for teaching source code comprehension as a dedicated process and learning goal.

Source Code Comprehension as Learning Goal

To summarize this introduction to source code comprehension, reading and successfully comprehending source code is essential in the learning process. Thus, it should be a dedicated learning goal (see Section 2.3.5). The following subsections describe and discuss the topic of source code comprehension from different perspectives. First, abstractions in comprehending source code and differences between novices and experts are explained. Followed by different models of source code comprehension, the difficult process and different strategies to assess source code comprehension tasks, as well as fostering it in the teaching process and comprehension strategies used by learners.

2.3.1 COMPREHENSION STRATEGIES OF NOVICES AND EXPERTS

Differences in Comprehending Code

Reading source code takes up a considerable amount of mental resources and time for developers. Many different aspects have to be taken into account to read and comprehend source code on a productive level: the quality of code, like the structure and identifier names, (Schankin et al., 2018), as well as the developers' ability. Brooks (1983) names three distinct sources of differences between developers comprehending source code: 1) programming knowledge, 2) domain knowledge, and 3) comprehension strategies. If someone has more knowledge about source code, syntax, and how to confirm hypotheses against code, this is a huge advantage. Likewise, the knowledge of the domain and how source code is read in terms of the comprehension strategies, like following the hierarchy of function calls, locating the data input/output first, or merely recognizing common forms of algorithm implementations.

Schemas for Complex Information

In the context of reading and, in this case, particular for source code reading, the process can be seen from different perspectives. According to psychologists, a learner forms *schemas* to process (complex) information. The working memory can only hold seven items ($\pm$ two items, "the magical number 7") (Miller, 1956). To achieve anything sophisticated with this

limited memory capacity, it is explainable with the mechanism of chunking, where information is grouped to form bigger chunks, e.g., memorizing a phone number (Miller, 1956). Using larger chunks of information or a more automatic chunking process is the key to the growth of expertise. Generic knowledge that can be used efficiently and automatically in everyday situations is important to solve problems. "Schemas represent knowledge as stable patterns of relationships between elements describing some classes of structures that are abstracted from specific instances and used to categorize such instances." (Kalyuga, 2010, p. 48) Building and using schemas is an important step for expertise, but new information needs to be integrated constantly, which can lead to misconceptions, e.g., for novices when learning to program.

Schema-Building, Plans, Chunking, and Beacons

Long-term memory is a central component of problem-solving (Sweller, 2010), and schemas reduce the complexity of problem-solving. Thus, schema-building is a crucial goal in introductory programming courses, because reading source code means to retrieve schemas to store general solution patterns. Several empirical research shows, that programmers, when working on a programming task, make use of plan schemas (Détienne, 1990; Rist, 1989). Experts form multiple layered models of source code to combine program and domain models. Besides, programmers use every strategy they can, like top-down, forward-developing, and breadth-first (Rist, 1989). Studies targeting experts show that they use chunking or abstractions for identifying critical components (Détienne & Soloway, 1990; Fix, Wiedenbeck, & Scholtz, 1993). In (Soloway, 1986) the author argues for explicitly teaching plans as abstractions, a result supported by Rist (1989). Another essential concept is beacons to identify design patterns in source code quickly (Astrachan, Berry, Cox, & Mitchener, 1998) or as significant roles in the code (Kuittinen & Sajaniemi, 2004), regarding variables and the operations on them. A beacon can be considered a pattern in the source code, which can be spotted quickly and activates knowledge schemas. A typical example is the source code needed to swap two variables in a loop, which can activate a sorting schema (Brooks, 1983).

Novices and Experts Reading Differences

Besides, research shows that novices and experts read source code differently. Novices have limited surface knowledge; they lack detailed mental models, have problems applying relevant knowledge, and approach programming line by line rather on a level of meaningful chunks (Winslow, 1996). Others conclude in their research, that "experts' better knowledge at an abstract level or a concrete level depends on what abstract or concrete knowledge is implied." (Ye & Salvendy, 2007, p. 473) From a more practical point of view, novices spent less time on planning and testing their programs (Husic, Linn, & Sloane, 1989), and struggle with language features and concepts, like initialization of variables and variables as a concept in general (Spohrer & Soloway, 1989). These are just some examples of the extensive research since the 1980s on the differences between novices and experts and their ability to read source code. An extensive overview can be found in (Robins et al., 2003, p. 151–155). Following the work of Höfer (2011), experts are not faster in every situation. The authors

found that "[...] it seems that experts had sacrificed speed for quality." (Höfer, 2011, p. 223) Experts were slower in implementing changes than novices but took one hour in average extra time to implement tests, to achieve a 2.6 % higher line coverage. The authors stated that the exact reasons could not be evaluated with the available data.

Different Abstractions

To summarize this section, novices and experts use different abstractions when reading source code, influenced by programming knowledge, domain knowledge, and comprehension strategies. For an efficient and productive source code comprehension process, it is essential to recognize beacons, plans, and variable roles. The next section introduces source code comprehension models, which explain strategies to understand source code.

2.3.2 AN OVERVIEW OF MODELS

Source Code Comprehension Models

Researchers created different types of source code comprehension models on various levels of detail, based on the maintenance tasks they were studied on respectively used for (Von Mayrhauser & Vans, 1995). Some models are aiming for understanding code in general, others targeting a particular purpose like debugging. Overall, these models are based on understanding source code programmers have not written themselves, which is a focus of this thesis. According to Von Mayrhauser and Vans (1995, p. 10), some static and dynamic components for mental models are always present, like strategies, actions, and processes for dynamic behaviors, as well as chunks, plans, and beacons for the static entities. These can be found in almost every source code comprehension model.

Comprehension Model as Mental Representation

Since the 1970s, different source code comprehension models have been proposed. The comprehension literature covers a broad spectrum of topics, for example, the distinction between program and domain models, and top-down vs. bottom-up strategies. The common understanding is that a source code comprehension model is a mental representation formed while reading and familiarizing with foreign source code. A comprehension model aims at describing these mental representations. The focus of this section is to describe different source code comprehension models. Explaining different reading strategies is the goal of Section 2.3.3. Over the last three decades, programming technologies and software development methodologies evolved, which influenced the research of source code comprehension models. Harth and Dugerdil (2017) identified three major source code comprehension periods:

1. The **Classical Period** (before 2000): Research was driven by psychologists, which identified all major strategies in this period.
2. The **Optimistic Period** (between 2000 and 2010): More interest in the software understanding process to enhance it with tools and techniques.
3. The **Pragmatic Period** (after 2010): Focus on authentic source code comprehension problem to develop techniques to facilitate maintenance.

In these three periods, different source code comprehension models were created through research. Due to the vast amount of literature, the following part of this section summarizes relevant models and their implications on the understanding of comprehension processes. An extensive overview can be found in the literature review of Harth and Dugerdil (2017).

The Classical Period

In the classical period, Brooks, Soloway, and Ehrlich, Letovsky, Pennington, as well as Mayrhauser and Vans formulated different models focusing on the "human behavior and the associated knowledge models" (Harth & Dugerdil, 2017, p. 404ff). For Brooks, the comprehension process involves matching all found beacons in the source code to high-level descriptions (hypothesis). Soloway and Ehrlich used text comprehension theories. Their model uses a linguistic point of view, with "plan knowledge" or programming plans to find essential parts in the code. Letovsky conducted experiments with professional programmers and novices and observed that the source code comprehension process is like an investigation.

The Optimistic Period

In the optimistic period, Rajlich and Wilde, Kelson, Murray, and Lethbridge, Rilling et al., as well as Ko et al., created comprehension models with the focus of tools and techniques to help with program comprehension (Harth & Dugerdil, 2017, p. 407ff). One reason for this shift in research interests may be the rise of object-oriented programming, which became widely used in this period. Rajlich and Wilde described the comprehension process from the viewpoint of the learning process. In contrast, Kelson formulated a general view of the comprehension process, based on operations, events, and properties. Another, more structural view, was suggested by Murray and Lethbridge, which based their model on design patterns, introduced by the Gang of Four, to form a micro-theory of understanding.

The Pragmatic Period

In the pragmatic period, Belmonte et al., Benomar et al., and Nosal et al. shifted the research community's attention to the question, what source code comprehension is, rather than how comprehension is reached in the two periods before (Harth & Dugerdil, 2017, p. 409ff). Belmonte et al. proposed a model based on the three layers 1) why? – Business Layer, 2) what? – Mapping Layer, and 3) how? – Implementation Layer, to answer questions about the program's purpose. Benomar et al. created a unified model to combine program design understanding and program evolution understanding, two major research directions in software comprehension, as they noted. The model of Nosal et al. is based on the understanding, formed by controlled experiments, that source code comprehension is hypothesis-based. When reading source code, participants tried to match elements found in the solution domain (source code) to the problem domain (requirements). The model consists of the four layers 1) abstract domain, 2) features and concepts, 3) plans and beacons, and 4) source code. In the work of Belmonte, Dugerdil, and Agrawal (2014), they "propose a rigorous source code comprehension model built as a hierarchy of three abstraction levels that link the source code elements to the purpose of the program." (Belmonte et al., 2014, p. 9)

Source Code Oriented Perspective

When comparing these different models, and the others not mentioned in this thesis but gracefully described in Harth and Dugerdil (2017), a clear shift from a psychology-centered

view, primarily based on mental models, to a more source code oriented perspective, based on source code layers and different domains, is observable. This shift is consistent with the current work in source code comprehension, where new research is targeted more on fundamental source code and programming language concepts, than on comprehension models. This thesis is based on the work of the third period, because the empirical studies and the subsequent data analysis is based on eye movement data with a concrete programming language (Java), with no deviations, and on research questions, targeting programming constructs or architectural concepts. Besides and, most importantly, designing and implementing a dynamic learner support system requires the relationship between different abstraction levels (problem and mental models) to concrete source code elements. These links are distinctive for the later research period and resulting source code comprehension models.

2.3.3 COMPREHENSION STRATEGIES

Comprehension Strategies and Eye Movement Patterns

Defining and detecting comprehension strategies is essential in source code comprehension research. Such a strategy defines how source code is read and how it leads to knowledge about the source code and semantics. Furthermore, without proper strategies, comprehending source code is unlikely to be successful, because the reading and comprehension process will tend to be unstructured and not goal-driven. This thesis separates between comprehension strategies and comprehension patterns, called eye movement patterns in subsequent sections and further discussed in Section 2.4.3. A strategy is a more generic, global approach. A pattern is localized and often highly coupled to specific language or source code features. A rough categorization of comprehension strategies was proposed by Exton (Exton, 2002). These five categories are derived from the "Constructivism Learning Theory" (Hein, 1991). It is important to note that this theory suggests that personal experience is highly essential in the learning process, which cannot be standardized as a result. Nevertheless, recurring strategies can be observed to handle abstractions in source code:

1. **Bottom-up Strategy**: A straightforward approach, where source code is read line by line, to group information (chunking). The abstraction process starts from the source code (low level) to the global description. Therefore, it is called "bottom-up."

2. **Top-down Strategy**: An approach used when prior knowledge of the domain is available. A starting assumption about the program is used, which gets refined by reading source code fragments, to create and validate more specific hypotheses about specific features.

3. **Hybrid Strategy**: This approach is also known as knowledge-based strategy (Exton, 2002), in which concepts of both the bottom-up and top-down approach are used. Familiar code is read top-down until unfamiliar code is encountered, where a bottom-up reading strategy is more appropriate.

4. **As-needed vs. Systematic Strategy**: In large software systems, understanding the system in its entirety is not a worthwhile goal. Developers will use an as-needed strategy, to familiarize them with essential parts of the code, e.g., for a maintenance task. For other tasks, e.g., code migrations, studying the whole codebase may be necessary, a systematic strategy.

5. **Integrated Strategy**: In this strategy, it is considered, that comprehending source code can involve all strategies mentioned above simultaneously when the code is read.

Meta-Level Approach to Reading Code

These five comprehension strategies provide a more meta-level overview of how readers approach source code. The strategies are related to comprehension models mentioned in the previous Section 2.3.2. The models of the first period defined by Brooks (1983), as well as Soloway and Ehrlich (1984), are considered to include a top-down strategy, according to the classification of Exton (2002). In contrast, the model of Mayrhauser and Vans (1995) includes an integrated strategy because it combines the models of Pennington (bottom-up) and Soloway and Ehrlich (top-down).

Scanning Phase

The research on comprehension strategies, apart from generic models, is extensive, too. A well-known publication is from Hidetake, Masahide, Akito, and Ken-Ichi (2007) which revealed that approximately 70 % of the source code is read in the first 30 % of the overall time spent reading the code. Since then, this is often referred to as the scanning phase. Sharif, Falcone, and Maletic (2012) replicated this result with a larger sample (15 programmers). The results show that less time spent to scan the code initially is related to more time needed to find defects.

Gender Differences in Source Code Comprehension

Obaidellah and Haek (2018) pursues a different research direction. The authors analyzed gender differences in source code comprehension. The results show that no significant differences can be found in the data, only "a slight difference in preference for information allocation, since female participants rely slightly more on the statement of the problem whereas male participants prefer examining the output example if provided." (Obaidellah & Haek, 2018, p. 8) Others reported differences in reading source code in the C programming language between male and female participants (Hou, Lin, Lin, Chang, & Yen, 2013).

Global Comprehension Strategies and Localized Patterns

Overall, differences in reading behavior and comprehension strategies were found over the last decades. Global comprehension strategies can explain generic reading and comprehension approaches. For a dynamic learner support system more fine-grained and specific reading patterns are necessary, to be able to conclude comprehension hurdles. Therefore, this thesis not only focuses on global comprehension strategies but also on local eye movement patterns.

2.3.4 ASSESSMENT OF SOURCE CODE COMPREHENSION

Approaches for Measuring Source Code Comprehension

The assessment of source code comprehension is a diversified research topic. Questions arise on how to measure source code comprehension, how to present source code (snippets), and which focus is essential from the perspective of participant groups. Different aspects of novice source code comprehension were conducted in the early 1980s (Du Boulay, 1986; Mayer, 1981). Lister et al. (2006) studied the ability to explain the purpose of a source code snippet in plain English, hence summarizing the purpose (Abid, Maletic, & Sharif, 2019). In general, source code comprehension is measurable in different ways (e.g., perception, attention, memory, and reasoning). Not every approach seems to be justifiable. Recalling the memory of participants, e.g., which variables were present in the source code, is only superficially involved in source code comprehension. It is necessary, but should not be the primary dependent variable when measuring source code comprehension. Measuring the outcome of a comprehension process is another story with different approaches. Researchers used comprehension tasks, think-aloud protocols, and memorization, e.g., variables or other essential aspects of a source code snippet.

The Presentation of Source Code

The presentation of a source code example can affect the source code comprehension process. Various research studied the difference of syntax highlighting and more vibrant visualizations of source code on source code comprehension (Asenov, Hilliges, & Müller, 2016). Previous research has shown the effects of syntax highlighting as a form of learner support. Some studies found effects for novices or in general (Asenov et al., 2016); some do not (Hannebauer, Hesenius, & Gruhn, 2018). The benefit can be that the highlighting is used as a visual cue for programmers to decrease the time required for mental execution (Sarkar, 2015). One explanation for the varying results is that novices tend not to use/ignore the highlighting or misinterpret the meaning completely.

Dynamic and Static Comprehension Questions

The correct way of measuring source code comprehension in a study is crucial, which is why many researchers list their way of measuring the comprehension in the study design section as dependent variables. The answer to a comprehension task is often used as the performance indicator of the source code comprehension task of learners. Besides, a behavioral comprehension question is often used, because one can assume, that a behavioral question cannot be answered without comprehending the source code first, even if a learner recognizes the code, it is necessary to comprehend it and to build a mental model of it with the given data. Such a question aims at the behavior of a program and asking for the result of an operation, e.g., running a loop three times. Siegmund (2016) gives an excellent overview of different ways of measuring source code comprehension and categorizes them into 1) think-aloud protocols, 2) memorization, and 3) comprehension tasks. The think-aloud protocols are an excellent way to gain information about a learner's strategy, like building a hypothesis for known source code or inference for unknown code examples. From today's perspective, memorization is an odd technique to measure the level of comprehension but

was used several times in earlier research (Rouse, 1982; Shneiderman, 1976). Shneiderman compared source code comprehension to musicians' ability, who can remember a thousand notes of a song (Siegmund, 2016, p. 2), which seems even more bizarre from today's perspective. Besides, new approaches to measure source code comprehension were discovered and used. Functional magnetic resonance imaging (fMRI) (Siegmund et al., 2014), and electroencephalography (EEG) (Kluthe, 2014), to name two examples, were used to measure source code comprehension of participants.

Controllable Studies and Generalizable Results

A big challenge for measuring source code comprehension is the balance between a highly controlled study, to control the influence of confounding factors (maximizing the internal validity), and the generalizability of the study results. The study of Hanenberg (2010) is an excellent example of how difficult it is to choose a good study design. The authors want to measure and evaluate the effects of static and dynamic type systems. What programming languages should be used in such a scenario, and how one can minimize the effect of two sets of different programming language features on, e.g., the development time? The author chose to develop its programming language and a small IDE with a minimal feature set. Thus, the only difference between the two participant groups is the type of system.

Focus on Dynamic Comprehension Questions

In conclusion, through the assessment of source code comprehension, researchers can gain various perspectives on learners' reading and comprehension processes. In this thesis, the focus is on source code comprehension assessed with comprehension questions, targeting the dynamic properties of a source code snippet. Thereby, participants must use the provided data to run the code snippet mentally after building a mental model via reading the code. This approach is much more relatable to real-world scenarios and avoids recalling parts of the source code via memorizing. Besides, in interviews with participants, a small summarizing task is included, to ask for a source code example, e.g., "this code implements a sorting algorithm."

2.3.5 FOSTER SOURCE CODE COMPREHENSION AND CODE EXPLAINING

Code Reading as a Dedicated Teaching Strategy

In the past, researchers have studied ways how to teach programming efficiently (Ludi, Natarajan, & Reichlmayr, 2005). Many approaches follow the software industry's goals, like not only teaching technical aspects but also experience of typical non-technical issues, which are essential in real-world software projects. Also, pair programming (McDowell, Werner, Bullock, & Fernald, 2003) is integrated and evaluated in learning situations, as well as introductory courses that facilitate active learning through project work. Code reading is analyzed in various studies to gain insight into source code comprehension processes or as an educational topic, e.g., with research regarding "reading before writing" or "teaching programming by immersion, reading, and writing" Campbell and Bolker (2002). However, code reading is rarely reflected on as a dedicated teaching strategy. In many research projects, source code comprehension is limited to measure how participants read source

code or for evaluating student's achievements. The learning process is left out and how to attain progress in the case of poor performance.

Source Code Comprehension as Teaching Goal

Source code comprehension should be viewed as a part of the learning process, not as a side effect of writing programs and learning to program in general. Reading (Busjahn & Schulte, 2013), tracing (Hou et al., 2013), and explaining (Murphy, McCauley, & Fitzgerald, 2012) has been seen or proposed as assessments. Sudol-DeLyser, Stehlik, and Carver (2012) suggests using questions targeting source code comprehension as learning events. More recent educational approaches address course design changes to move source code comprehension in focus. The PRIMM project (Predict-Run-Investigate-Modify-Make) aims to help teachers organize lessons for pairs of students. They are guided by reading and adjusting code before they write a new one. The PLTutor project shows the relationship between machine behavior and syntax. Students learn about programming language semantics through observation. In the CS POGIL project (Process Oriented Guided Inquiry Learning), groups of students use critical thinking questions, including reading, analyzing, and adjusting code.

Code Reading and Explaining as Essential Skills

Furthermore, code reading and explaining should be an essential skill and taught alongside source code comprehension skills. Reading and explaining skills can make a massive difference in transitioning from a novice to an expert skillset. As an example, code reading has a social element in the sense of social skills. Experts are using source code comments to, e.g., infer, a) how many and what kind of people were involved in writing the code, b) under what conditions the code was written, c) and if signs, e.g., of carelessness or haste, are visible. This information can be used to locate bugs and put the comments from the real code's perspective. Studies show that mainly experts are using the many aspects of code reading to their advantage (Riedl, Weitzenfeld, Freeman, Klein, & Musa, 1991). Novices have to learn these skills. Therefore, code reading and code explaining should be distinct parts of programming education. Overall, the focus on source code comprehension and code reading is vital in introductory programming courses and deserves more attention in computer science education research and teaching programming.

2.4 EYE MOVEMENTS IN SOURCE CODE COMPREHENSION

Eye Movement Data on Complex Stimuli

Analyzing eye tracking data in source code comprehension settings means analyzing eye movement data on structured but quite complex stimuli. These characteristics raise many questions regarding the mapping of eye movement data to source code elements and regions. Which eye movement metrics can be applied to source code examples to extract useful information, and which eye movement patterns may be present in the data? Incorrect assumptions about the nature of reading source code, such as reading natural language text, can lead to an inconclusive analysis of the available data. Busjahn, Bednarik, et al. (2015) indicate that "[...] there are specific differences between reading natural language and source code." (Busjahn, Bednarik, et al., 2015, p. 255) Besides, the results showed that "[...] non-linear

reading skills increase with expertise." (Busjahn, Bednarik, et al., 2015, p. 255) Such research and findings are necessary and need to be considered when designing empirical source code comprehension studies and analyzing the gathered data.

Programming Languages and Paradigms

Furthermore, the design of and available concepts in programming languages make it interesting but rather complex to record and analyze eye movement data of comprehension processes. Available eye movement patterns are dependent on the programming paradigms of a language, e.g., *procedural*, *object-oriented*, *functional*, *declarative*, or *logical*, type systems, data structures, and different concepts of variables (Ulrich Hoppe & Werneburg, 2019). The reading and comprehension processes vary when analyzing the data of an object-oriented or functional language, to name two examples. For the OOP code, often recurring jumps can be expected, follow the structure of the object-oriented program and the data flow. For the functional language, it is assumable, that regressions within a range of some consecutive lines are detectable. These effects are explainable with the different paradigms and with the resulting varying reading strategies, e.g., declarative reading and procedural reading. Functional code is written as a sequence or chain of functions, while procedural and object-oriented code, the latter to some extend, is written as a sequence of statements. Such differences must be taken into account. This thesis is based on Java as language for the source code examples. Thus, only procedural and object-oriented code, and resulting eye movement data, and patterns are considered.

Mapping Eye Movement Data to Source Code is Crucial

The following sections summarize important work regarding ideas and methods to map eye movement data to source code, essential and useful eye movement metrics as well as patterns, observable when reading source code, the development and use of coding schemes for analyzing eye movement data of source code comprehension processes, as well as differences of experts and novices. The presented work highlights essential aspects of analyzing and interpreting eye movement data that originated from source code examples as stimuli.

2.4.1 MAPPING EYE MOVEMENT DATA TO SOURCE CODE

Natural Language Text vs. Source Code

Reading natural language text and source code are quite different tasks (Busjahn, Bednarik, et al., 2015). A computer program is lexically and syntactically different from natural language text. Source code consists of a limited vocabulary and is organized differently, e.g., with formally defined structures and layout. The text comprehension model of Kintsch (1998) describes the understanding process of natural language text as two concurrent phases: *text* and *domain*, e.g., *how* it is written and *what* it means (Busjahn, Bednarik, et al., 2015). Additionally, source code has a third dimension: *execution*. Without this, source code loses an important aspect of its complexity. Thus, all three dimensions must be considered when mapping eye movement data to source code examples as stimuli.

Coding Conventions

Mapping eye movement data to source code elements and regions is crucial in every empirical eye tracking study. The exact process of doing so depends on the goal the analysis has.

Besides, presenting source code to participants must follow specific rules and must respect the technological and organizational culture of the programming language in use. Supporting developers reading source code has led to coding conventions, "which are meant to help them to communicate with other developers, including reviewers, maintainers, and testers of their code." (Bauer, Siegmund, Peitek, Hofmeister, & Apel, 2019) To neglect such conventions will affect source code reading and, therefore, the collected eye movement data, which may fail the mapping of eye movement data to source code and the subsequent analysis.

Mapping Eye Tracking and Object Attention

Eye tracking data, in its original form as coordinates with X and Y data points, can be applied with various visualization methods to source code examples, which can be useful in scenarios with small data sets or a few participants. A more detailed explanation of visualization techniques of eye movement patterns is available in Section 2.7. In most analyses cases, the eye tracking data needs to be mapped to source code regions and elements more directly to measure the participants' attention with greater detail. Object attention, in general, and the order of attention, play a crucial role in understanding the cognitive processes of participants doing source code comprehension (Costa-Gomes, Crawford, & Broseta, 2001). By applying such concepts, researchers may be able to verify hypotheses about decision processes (Schulte-Mecklenbeck, Kühberger, & Johnson, 2011). One common approach for object attention measurement is defining areas or regions of interest (AOIs; ROIs). The amount of attention gets calculated by matching fixations to these AOIs based on coordinates.

Non-Standardized AOI Methods

In the past, AOI methods got reinvented continuously (Holmqvist, 2011), which led to a lack of common terminology, strategies, and methods of defining AOIs for a particular set of stimuli. While domain-specific quasi-standards may exist, like in reading research (Rayner, 2009), these standards can most likely be used partially in code comprehension research. The lack of standards in defining and reporting AOIs in source code comprehension research may directly impact this field's advancement.

Margin and Padding Model

There are several recommendations regarding the definition of AOIs. In this thesis, margins and paddings are considered around an object. The padding and margin can be further enlarged if the accuracy of the used eye tracker is low (Holmqvist, 2011). In general, AOIs should be created with enough distance between each other, to minimize potential overlapping and to be able to increase the margin later in the analyzing process. The major problem is, that increased AOI sizes can lead to false positives, by assigning fixations to an object not related to it (Orquin, Ashby, & Clarke, 2016). However, defining AOI too small can lead to an inflation of false negatives, by not assigning a fixation to an AOI where it belongs in reality.

Reporting of AOI Parameters and the Signal Detection Problem

AOI definitions and the use of AOI margins are highly variable in source code comprehension studies, most likely based on source code lines. Unfortunately, the specifications of AOIs, especially about margins, are omitted and not reported in most studies, which should be changed in the future (Deitelhoff, Harrer, & Kienle, 2019a). Despite the generalized advice to keep AOIs maximal (Holmqvist, 2011), choosing an appropriate AOI model is a form

of a *signal detection problem*. It seems that little to no research has been conducted in the domain of source code comprehension on how AOIs should be defined.

Varying AOI Models

The majority of source code comprehension studies use some source code as stimuli, which constrains the definition of AOIs in this domain. Source code has some non-negotiable characteristics, like *tokens*, lines including *empty* ones, and, in most cases, a *maximum character length* for each line. However, even in such scenarios, one can imagine many different AOI models that can detect various kinds of eye movement patterns. An often-used approach is to use the AOI line model, where code lines define AOIs. Furthermore, an AOI region model is used to combine important lines to semantically important areas. Nevertheless, again, how to choose AOIs and AOI margins seems to be very variable between studies, and even within a study, one AOI around a line can be larger than another one, which leads to an implicit prioritization of the larger AOIs (Orquin et al., 2016). Besides, the question of larger or smaller AOIs influences the amount of AOI-Hits tremendously (Deitelhoff et al., 2019a). Tools like EyeCode[5] can automatically detect AOIs in source code stimuli, but features like automatic altering and testing AOI models are not included as of today. In general, a common ground for selecting and defining AOI models in source code comprehension, with the need for reporting the chosen AOI model in the publication, is necessary.

AOI to Source Code Mapping is Challenging

To summarize this introduction of the mapping challenges, AOIs are the predominant way of mapping eye movement data to source code elements. It is dependent on the situation, which elements and parts of a stimulus are captured by which AOIs. Manually defining AOIs, e.g., in another file, can benefit the analysis process, because the AOI definitions can be changed between analysis runs to compare different settings. Besides, AOIs not relying on source code elements alone add the benefit of capturing data like question and answer areas, which proved to be viable in the studies described later in this thesis.

2.4.2 EYE MOVEMENT METRICS

Eye Movement Metrics are not Standardized

Eye movement metrics are a common way to gain first insights into eye movement data (Sharafi, Shaffer, Sharif, & Gueheneuc, 2016). These metrics can be calculated for the overall code examples, for regions, code lines, or in general for every pre-defined AOI. Technically speaking, eye movement metrics are derived from eye tracking data like the number and duration of fixations, saccade length, and then mapped to domain-specific questions important for source code comprehension scenarios. In eye tracking studies, a wide variety of different metrics are proposed and used to measure and to interpret visual effort (Sharafi et al., 2016). Unfortunately, this led to many different eye movement metrics, sometimes only distinct in minor detail. Besides, many researchers define their metrics, making it harder to compare them on a meta-level and conclude different studies. Lastly, it is hard to define

[5] `https://github.com/synesthesiam/eyecode` – Last accessed on December 14, 2020.

standards regarding eye movement metrics in the source code comprehension community, even though such standards are much needed to compare and replicate eye tracking studies.

Fixation-, Saccade-, and Scanpath-Based Metrics

In many source code comprehension studies, eye movement metrics measure the visual effort of participants. Such metrics need to be chosen with the task and stimuli in mind to be representative and useful. A metric useful for analyzing eye movement data on UML models, like Fixation Count (FC), where a higher number indicates inadequate arrangements of elements in a diagram (Yusuf, Kagdi, & Maletic, 2007), may express something completely different on source code as a stimulus. An extensive list of metrics, based on a systematic literature review, can be found in the work of Sharafi et al. (2016). They divided the metrics into 1) fixation-based, 2) saccade-based, 3) scanpath-based, as well as 4) pupil-size- and blink-rate-based metrics. The empirical source code comprehension studies described later in this thesis makes extensive use of the first category, the fixation-based metrics, to explain participant strategies. Besides these mentioned metrics, like the overall time on the stimulus and the overall time until a question was answered, are also considered in many source code comprehension studies.

Fixation-Based Metrics

Metrics based on fixations can be divided into metrics based on the *number* or the *duration* of fixations. To name some examples from Sharafi et al. (2016), such metrics are, Fixation Count (FC), Fixation Rate (FR), Average Fixation Duration (AFD), the Ratio of ON-target:All-target Fixation Time (ROAFT), and Fixation Time (FT), also called "Dwell Time" (Busjahn, Bednarik, & Schulte, 2014). These metrics are often used to compare AOI-Hits between participants, and to verify hypotheses. Especially the ROAFT metric can be useful to test if important AOIs, e.g., for important aspects of a code example, are visually visited more often than others. As mentioned above, an extensive list of 20 metrics for visual effort with references to over 60 publications can be found in Sharafi et al. (2016).

Metrics Must be Standardized

This section briefly introduced eye movement metrics to measure the visual effort based on different eye tracking data points, focusing on metrics for visual effort based on fixations. This is the most often used category in the source code comprehension studies described in this thesis. Overall, standards for metrics are needed to make the results comparable. Concluding, the mentioned metrics are useful to find differences in participant's behavior on an AOI-based level. While this is unquestionable useful, more global relations between AOIs, e.g., to include the stimulus context, must be part of extensive analysis, which is the goal of eye movement patterns discussed in the next section.

2.4.3 CODING SCHEMES AND EYE MOVEMENT PATTERNS

Coding Schemes To Categorize Patterns

A coding scheme is a valuable tool to find and categorize common patterns in eye movement data. The elements in a coding scheme "reflect cognitive processes behind the observable visual behavior of programmers." (Busjahn, Schulte, & Kropp, 2014, p. 111) Coding schemes can simplify the process of detecting such eye movement patterns, especially across empirical

eye tracking studies with source code as a stimulus. The elements of such a coding scheme can serve as a basis not only for categorizing and detecting patterns but also for the discussion of analysis results. Coding schemes are necessary because there is no easy matching between eye movement data and cognitive processes. A scheme is often highly coupled to an AOI model because the elements of a scheme describe patterns that emerge due to eye movement transitions of specific source code areas.

Block Model, AFECS, and EMIP

Therefore, central for the finding and categorizing patterns in eye movement data is the development of coding schemes, which can be summarized as "to label and aggregate eye movement data of programmers understanding source code." (Busjahn, Schulte, & Kropp, 2014, p. 111) Throughout the years of research, various coding schemes with different intentions were proposed. Categorization schemes for patterns have different intentions and levels of detail. Like the *Block Model* Schulte (2008), which categorizes eye movement patterns from an educational point of view. There is not a one-size-fits-all coding scheme that's suitable for all use cases and across programming languages. According to Saldaña (2009), a code "is most often a word or short phrase that symbolically assigns a summative, salient, essence-capturing, and/or evocative attribute for a portion of language-based or visual data" (Saldaña, 2009, p. 3) Such codes will most-likely form patterns. Overall, a coding scheme organizes such codes into categories. The flexible expandable coding scheme (AFECS) was proposed by Von Mayrhauser and Lang (1999), and is based on the integrated comprehension model (von Mayrhauser & Vans, 1994). The AFECS scheme was later extended by O'brien, Shaft, and Buckley (2001). They compose an extensive analysis coding scheme for think-aloud protocols. A coding scheme to analyze free-form program summaries was presented by Good and Brna (2003, 2004). Programmers can summarize source code with their own words and own abstraction level, which is captured by this coding scheme. Others have demonstrated to create a coding scheme with an automated coding approach, which seems to work in most cases for non-strategy categories, i.e., eye movement patterns (Hansen, Goldstone, & Lumsdaine, 2014). Automated coding of strategies, especially when they should be agreeable with human programmers, is a completely different and complex topic. Throughout this thesis, the coding scheme of the EMIP workshop[6] is used to analyze eye movement data and find and classify patterns in it. These patterns were created in the course of multiple EMIP workshops held as co-located events during various scientific conferences. The creators of the EMIP coding scheme used a combination of quantitative and qualitative methods (Busjahn, Schulte, & Kropp, 2014). The coding scheme consists of various categories, codes for reference purpose, a textual description, and classification if the coding scheme element is observable or based on interpretation. An observable pattern is *Flicking* (moving back and forth between two related items), while an interpretable strategy is *Data Flow* (following a single object in memory). The complete

coding scheme can be accessed via the website of the EMIP workshop[6]. A more detailed description is available in Busjahn, Schulte, Sharif, et al. (2014).

Global and Local Eye Movement Patterns

As described above, a coding scheme is useful as a baseline for finding and categorizing eye movement patterns, which is a necessity and a common goal when analyzing eye movement data. The empirical source code comprehension studies described in this thesis are based on the two categories called *pattern* and *strategy* of the EMIP coding scheme. The first one describes observable patterns and the second one strategies based on interpretation, which is a common obstacle with every coding scheme and in every eye movement pattern analysis. While some patterns are objectively observable in the data, subjective patterns are more bound to the interpretation while analyzing. Eye movement patterns have the potential to explain comprehension strategies, which includes successful as well as unsuccessful strategies. The latter is named as comprehension problems throughout this thesis. Nevertheless, the EMIP workshop coding scheme serves as a baseline for every analysis in this thesis. For a more global view on a source code example, global patterns like the *Linear Scan* (Hidetake et al., 2007) and *Jump Control*, also known as *Story Order Reading (SOR)* and *Execution Order Reading (EOR)* are used to explain reading behavior Busjahn, Bednarik, et al. (2015). Besides, local patterns like *Flicking*, and *Retrace Declaration* are used to explain locally limited reading behavior. The EMIP coding scheme describes flicking as "the gaze moves back and forth between two related items, such as the formal and actual parameter list of a method call" and the retrace declaration pattern as "often-recurring jumps between places where a variable is used and where it had been declared." (Uwano, Nakamura, Monden, & Matsumoto, 2006)

Standards for Coding Schemes and Patterns are Essential

Both topics, coding schemes, and eye movement patterns, are essential in analyzing eye movement data, but also a very challenging part. A standard throughout the source code comprehension community is still missing, which is why this thesis uses the coding scheme and the embedded patterns from the EMIP workshop as a baseline, which prevents developing another coding scheme for this thesis. Overall, and as a summary of this section, coding schemes are an excellent way to use a defined set of eye movement patterns, even to analyze varying eye movement data from novices and experts.

2.4.4 PATTERNS OF NOVICES AND EXPERTS

Eye Movement Patterns of Novices and Experts

The difference between novice and expert programming skills is a subject of extensive research in the past decades. The additional source of information, eye tracking provides, is important to gain insight into comprehension strategies of novices and experts alike (Busjahn, Bednarik, et al., 2015) and to shed light on problem-solving and reading strategies of both groups (Crosby & Stelovsky, 1990). In Section 2.2.4 novice mistakes and problems were

[6] `http://emipws.org/sample-page/2013-analyzing-experts-gaze/coding-scheme/#scheme` – Last accessed on December 14, 2020.

introduced from a programming skill perspective. The literature, discussed in this section, is important and valid for this section, too, but the primary focus is the differences in the eye movement patterns between both skill groups. Besides, the eye movement data and emerging patterns are bound to Java source code used as stimulus material. Thus, the anticipated eye movement patterns are based on object-oriented and, in part, procedural source code.

Novices and Experts Differences

The skill gap novices and experts show is observable in the eye movement data, not only via the eye movement metrics but also in the eye movement patterns. Numerous empirical eye tracking studies were conducted to identify such differences. The difference between novices and experts, when debugging source code, was analyzed by Bednarik (2012). The results show that repetitive patterns, in this context between the two representations of the java source code and a program visualization system, are associated with less expertise. Members of the EMIP workshop analyzed the differences in the novice's gaze Busjahn, Schulte, and Tamm (2015) and respective expert's gaze Bednarik et al. (2013). A study from 1985 showed that novices tend to use a line-by-line reading approach (SOR pattern) to solve a problem (Anderson, 1985). Besides, novices tend to grasp programming and the domain problem via control structures (Winslow, 1996), which can be visible in eye movement patterns. Crosby, Scholtz, and Wiedenbeck (2002) stated that experts focus mainly on beacons, as discussed by Brooks (1983), but novices do not use them. In the work of Busjahn, Bednarik, et al. (2015), the authors showed that novices follow a linear reading pattern (SOR) in approximately 70 % of the analyzed cases, were experts used this pattern in 60 % of the time. Overall the authors stated, the execution order reader (EOR) pattern "better explains the experts' reading approach than Story Order does." (Busjahn, Bednarik, et al., 2015, p. 263) In Sharif et al. (2012) the authors showed, that novices spent more time scanning the source code than experts. Besides, experts focus more on beacons and novices and syntactical elements of source code (Aschwanden & Crosby, 2006). Thus, SOR seems to be a separating factor for the novice and expert skill groups. In Yenigalla, Sinha, Sharif, and Crosby (2016), a semester-long study was conducted to assess the output and summary tasks of students for Java programs. Overall, novices were less accurate than experts based on eye tracking characteristics like fixation count and fixation duration. Pattern analysis was not conducted within the study.

Eye Movement Pattern can show Differences

As summary of this section, differences in the novice and expert reading behavior are observable via eye movement patterns. In general, experts tend to use the execution order of source code more often than novices. Besides, experts use beacons in source code more often, which can affect eye movement patterns, which are analyzable via eye movement metrics. The analyses of the empirical source code comprehension studies of this thesis use a mix of eye movement metrics and patterns to capture the necessary data. Relying only on eye movement metrics is equally erroneous than relying solely on eye movement patterns for detecting differences between skill groups.

2.5 LEARNING ANALYTICS BASED ON EYE TRACKING

Eye Tracking Data for Actionable Results

Eye tracking had and will continue to have an important impact on the field of learning analytics. On the one hand, the advance and remarkable progress of eye tracking as technology open new possibilities to integrate eye movement data into user interfaces. This interaction can lead to new human-machine interfaces, combining analytical and interactive eye tracking with the ability to adjust the interface to the user's needs. On the other hand, with more and more available eye tracking technology, this technique is integrated into the process of analyzing learners and creating better systems to support them (Njeru & Paracha, 2017). Eye tracking and learning analytics play an essential role in explaining the teacher and student interaction (Dessus et al., 2016), the human-robot interaction with indirect measures like *with-me-ness* (Lemaignan, Garcia, Jacq, & Dillenbourg, 2016), borrowed from the field of Computer-Supported Collaborative Learning (CSCL), and in the vast area of learning analytics to enhance learning programming (Trætteberg, Mavroudi, Sharma, & Giannakos, 2018). This thesis focuses on the latter, trying to develop methods to improve source code comprehension of novices. Overall, "learning analytics has the claimed potential for 'evidence-based' decision taking." Greller and Hoppe (2017, p. 10). In this scenario, eye tracking has the potential to function as an additional source of data. The goal to gain actionable results from data analyses (Greller & Hoppe, 2017) can be achieved with eye tracking data in specific ways. Achieving this goal is possible either in the offline mode of analytics (e.g., re-design of material after the analysis) or in the on-the-fly mode of analytics (e.g., intervene within a learning environment). Furthermore, eye tracking data can be combined with click-logs, log files, or other interaction data in a learning environment.

Eye Tracking Data for Interventions

Recent work investigated the potential of multimodal data to analyze and understand the learning experience. Click-streams are an often used source of information to analyze and predict learning behavior. Multimodal data-streams include other data sources, e.g., eye tracking and electroencephalography (EEG), or physiological data in general. The authors Giannakos, Sharma, Pappas, Kostakos, and Velloso (2019) showed that "multimodal data can help us increase the prediction accuracy of users' learning performance in learner-computer interaction (LCI)." (Giannakos et al., 2019, p. 117) Others focus on the possibility of predicting the collaboration outcome from eye tracking data (Sharma et al., 2017). The work of Sharma, Alavi, et al. (2016) focuses on gaze-aware interventions in the learning process, based on MOOCs, with the result that "gaze-aware feedback emerged as an influencing tool for intervention." (Sharma, Alavi, et al., 2016, p. 421)

Various Types of Interventions

Learning analytics and eye tracking are also present in programming research or programming education research, as learning to program is another field of application for both areas. Not every time eye tracking is directly involved, as the work of Ihantola et al. (2015) showed, in which educational data mining is the primary topic of a literature survey for the years 2005–2015. Others showed that eye tracking and semistructured interviews are

beneficial data sources that should be incorporated in more studies and prototypes (Trætteberg et al., 2018). The same authors stated in their conclusion that "visual learning analytics cater to learners' variability and diversity." (Trætteberg et al., 2018, p. 17) New methods are needed, which "are personalized, can be easily understood by learners and are linked with ways of improving and optimizing their learning." (Ferguson, 2012, p. 313) According to a literature review of Mavroudi, Giannakos, and Krogstie (2018), interventions around adaptive learning and learning analytics focus on student competencies. A small number of interventions focus on skills, and none on learning in terms of attitude change, e.g., programming habit changes. Besides, Awasthi and Hsaio (2015) mentioned that fewer tools are focused on amplifying learning opportunities.

Eye Tracking has a Growing Influence

Overall, to summarize this section, eye tracking plays a crucial role in learning analytics, with growing influence. Especially in adapting to learners' needs, eye tracking provides a valuable data stream in addition to click-streams. The vision of a dynamic learner support system, which is the overall context for this thesis, can use eye tracking exactly for such opportunities. New methods must be developed to detect source code comprehension strategies, which is the primary focus of this thesis.

2.6 METHODS FOR ANALYZING EYE MOVEMENT PATTERNS

Raw Eye Tracking Data

The raw form of eye movement data is X and Y data points. Often with the fixation time per point. After using a fixation filter (e.g., I-VT), a data stream can be used with various methods. Newer methods include machine learning approaches, e.g., based on LSTMs (Long Short-Term Memory) and 1D-CNNs (1D Convolutional Neural Networks) (Castner et al., 2020). These methods are often in early stages, e.g., predicting human eye fixations (LSTM-based) (Cornia, Baraldi, Serra, & Cucchiara, 2018), drowsiness estimation (Sun et al., 2018), as well as the automated classification of fixations, saccades, and smooth pursuits (neural net, CNN) (Hoppe & Bulling, 2016; Startsev, Agtzidis, & Dorr, 2019). While these approaches are focused on fundamental eye tracking characteristics, the work of Chen and Sun (2018) has the objective of predicting scanpaths with a form of LSTMs. The authors showed that such a recurrent neural network (RNN) architecture could successfully model attention shift behavior.

Eye Movement Data as Sequence of Letters

As described in Section 2.4.1, AOIs are a commonly used method to label important regions of a stimulus. If such AOIs are labeled with letters from an alphabet, they can form strings as eye movement representation. Thus, a widespread representation of eye tracking data is a string. With this approach, the AOI transitions are represented as consecutive letters. One way of analyzing these strings is to analyze *n-grams*. They are continuously ordered subsequences of the viewing behavior of participants, i.e., scanpaths. The n-gram analysis is one of the main forms of scanpath analysis, besides string alignment methods described later. Thus, n-grams are qualified to perform transition analyses, based on the

transition frequency. An analysis of Reani, Peek, and Jay (2018) showed, that n-grams with a length of four or more AOIs are not useful for finding distinguishing patterns between two groups. The general usefulness of n-grams for comparing the distributions in different groups was showed by Kübler, Rothe, Schiefer, Rosenstiel, and Kasneci (2017). Another frequency-based method is Hidden Markov Modelling, especially for modeling visual perception in simple tasks (Chuk, Chan, & Hsiao, 2014).

Sequence Alignment Techniques

These strings can be used for different string alignment methods, extensively known from DNA sequencing. Important contributions in this field were made by the Multiple Sequence Alignment technique (Hembrooke, Feusner, & Gay, 2006), the ScanMatch algorithm (Cristino, Mathôt, Theeuwes, & Gilchrist, 2010), the eMINE algorithm (Eraslan, Yesilada, & Harper, 2016), and the String-edit algorithm (Heminghous & Duchowski, 2006). Other algorithms additionally use fixation durations like the Scanpath Trend Analysis algorithm (Eraslan et al., 2016). String-alignment algorithms are used to find subsequences between gazes, form groups of gazes with similar transitions, and, therefore, a similar reading behavior. Besides, the Needleman-Wunsch (N-W) algorithm was successfully used by, e.g., Busjahn, Bednarik, et al. (2015) to find SOR and EOR patterns and by Castner et al. (2018) to compare different scanpaths in medical image reading skills to distinguish stages of expertise. The approach with the N-W algorithm is initially explained in Chapter 4. The application of these methods can be found in the methods and analysis sections of the empirical source code comprehension studies (see Chapters 5, 6, and 7).

Vector-Based Methods

Other methods, not adequately addressed in this thesis, include vector-based methods. The difference to the previously mentioned methods is that they take the length and the direction of saccades into account. Examples are the Mannan distance (spatial properties of the scanpaths, ignoring temporal dimensions), and methods based on saliency map comparisons. An extensive review and comparison is available in (Le Meur & Baccino, 2013).

This section briefly introduced different methods, primarily frequency-based and string-based, for analyzing eye movement patterns. While conducting the empirical studies described in this thesis, methods for analyzing reading behavior and searching for eye movement patterns were missing. The development of these methods is explained in Chapter 4.

2.7 METHODS FOR VISUALIZING EYE MOVEMENT PATTERNS

The following section is based on the publication "An Intuitive Visualization for Rapid Data Analysis: Using the DNA Metaphor for Eye Movement Patterns" by Deitelhoff et al. (2019b).

Qualitative Analyses

Besides analyzing eye movement patterns directly with various methods and tools, visualizing eye movement data to see patterns is another way of finding them. A visualization, in contrast to the previous section, is, in most cases, no algorithmic approach. Thus, many

visualizations are meant for a qualitative rather than a quantitative analysis. This section and the visualizations of this thesis are focused on qualitative approaches.

Heat Maps and Gaze Plots

Different tools and techniques exist to visualize and analyze eye movement data. Eye tracking data visualizations became more and more important since 2010, where the number of publications about this topic sharply increased (Blascheck et al., 2017, figure 1, p. 261). Much-used visualizations for eye tracking data are heat maps and gaze plots. Heat maps visualize the data as a color scheme and visualize how looking is distributed over the stimulus. Gaze plots visualize the location, order, and time spent looking at locations on the stimulus. Both visualizations are essential, but often not applicable to source code stimuli.

Varying Visualization Goals and Trade-offs

In general, the common goal of a visualization is to identify relevant eye movement patterns, in the best case, tied to the participant's intention. Another important aspect of visualization eye movement data is that this can be the first step before the analysis. Visualizing eye movement data can help to understand the essence of the data. Blascheck et al. (2017) categorized eye tracking visualizations into point-based, AOI based, and hybrid visualizations. Point-based methods, e.g., timeline (Blascheck et al., 2014) and scanpath (Goldberg & Helfman, 2010) visualizations, represent horizontal and vertical coordinates of fixations ordered by time. Visualizing the exact locations above the stimulus is not necessarily the best solution, because it can cause vast amounts of visual clutter (Rosenholtz, Li, Mansfield, & Jin, 2005). Additionally, this technique is not able to reflect eye movement patterns (Andrienko, Andrienko, Burch, & Weiskopf, 2012). Therefore, these visualization methods are not useful for the goal of this thesis. Scanpath visualizations focus on showing the linear timeline of the data to compare complete scanpaths but are not able to visualize certain eye movement patterns (Duchowski et al., 2010; Raschke et al., 2014). In contrast, AOI-based methods, divided into the timeline and relational visualizations (Goldberg & Helfman, 2010), focus on AOI fixations and transitions between them. These types of visualizations can be problematic with many AOI targets and transitions between them. Thus, a trade-off between various parameters is necessary when visualization eye movement data to find patterns. In contrast, AOI-based methods, divided into timeline and relational visualizations (Goldberg & Helfman, 2010), focus on AOI fixations and transitions between them. While this can be problematic with many AOI targets and transitions, the methods described later in this thesis uses implicit transitions between AOIs. An extensive list of publications for point-based, AOI-based, and mixed visualization can be found in Blascheck et al. (2017).

New Visualizations

Finding a viable eye movement visualization is an undertaking in itself. New methods and directions in visualization eye tracking data are proposed constantly (Blascheck et al., 2014). While planning and conducting the empirical source code comprehension studies described in this thesis, an easy visualization of eye movement patterns was missing. Easy in the sense of entry-level usability and easy to create from the available data sources. The results of a self-development visualization are explained in Chapter 4.

2.8 SUMMARY

This related work chapter briefly introduced essential topics in learning to program, source code comprehension, eye movements in source code comprehension, and methods and tools for analyzing and visualization eye movement patterns. Due to the three interconnected empirical source code comprehension studies, the spectrum of this thesis covers many research areas. Figure 2.2 visualizes the thirteen primary scientific topics of this thesis.

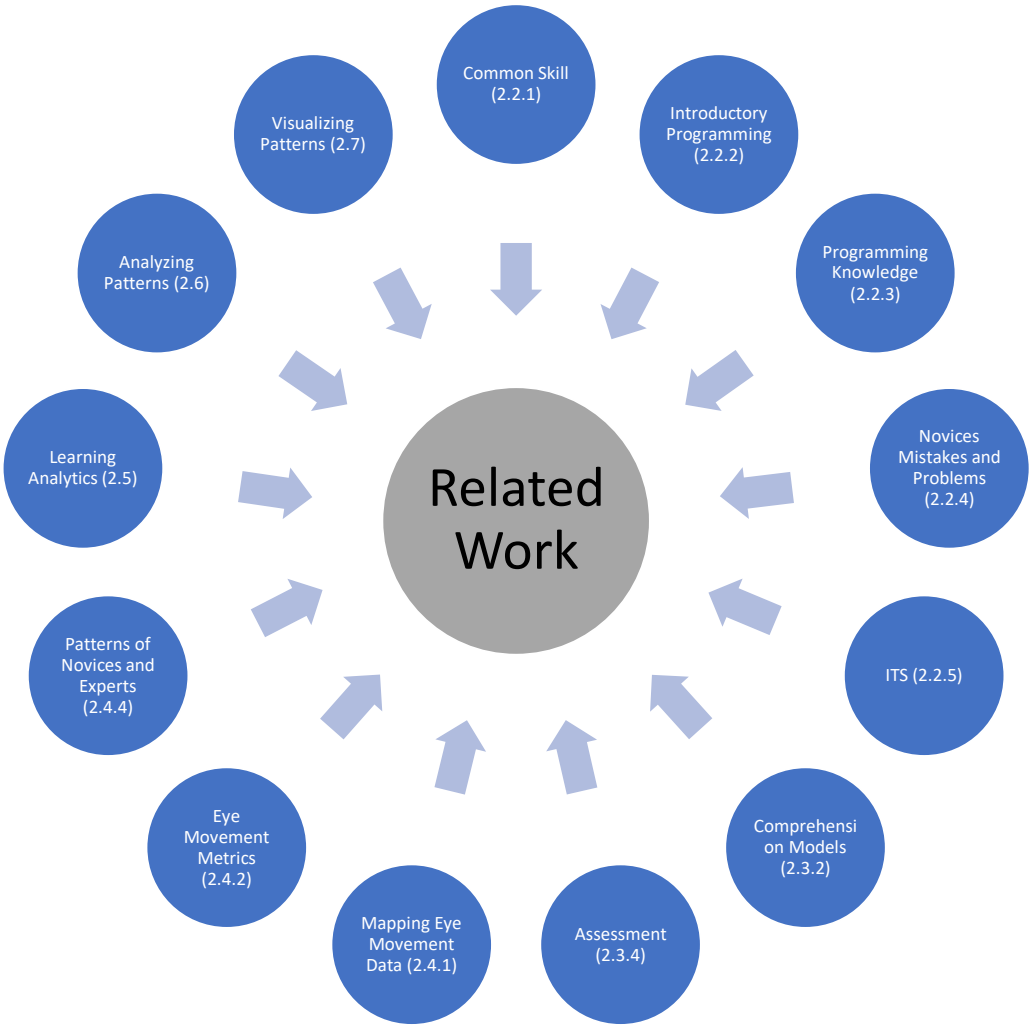


Figure 2.2: Overview of the thirteen focus topics as primary scientific foundation of this thesis.

Thirteen Focus Topics as Scientific Foundation

The scientific foundation is based on thirteen different areas reflected in various sections in this chapter. There is a considerable skill difference amongst learners because programming can be considered as difficulty, which relies on multiple levels of abstractions. However, programming must be viewed as an ordinary skill in the future, useful for everyone in the field of end-user programming, and is useful for learning through programming, if programming itself is not the goal (Section 2.2.1). Apart from this, learning and teaching introductory programming has some significant obstacles to overcome: High failure rates,

many methods, and tools to improve the experience overall, no programming skill hierarchy to prioritize learning steps, and often the result of no reasonable progress in learning to program (Section 2.2.2). Proper instructions in introductory programming courses can influence programming knowledge and learning outcomes (Section 2.2.3).

Novice Mistakes, ITS, and Comprehension Models

Regarding novices and experts, differences in error rates, problem categories, and the fact that most students do not learn programming at all (Section 2.2.4) are reasons to aim for the long term goal of developing a dynamic learner support system. Intelligent learner support (ITS) can help in some situations and with the support of eye tracking, but produced ambiguous results in the past (Section 2.2.5). Thus, the baseline of using novices and experts reading differences in the area of source code comprehension, in keeping with the pragmatic period of comprehension models with a focus on source code, is used to aim for the overall vision of a support system (Section 2.3.2).

Assessment, Mapping Data, Metrics, Patterns, and Learning Analytics

The current state of the assessment of source code comprehension in the scientific community led to a different set of empirical studies in this thesis, to use behavioral comprehension questions (Section 2.3.4). Moreover, the AOI approach for mapping the raw eye movement data was used, with fixation-based metrics (Sections 2.4.1 and 2.4.2). The differences in novice's and expert's eye movement patterns are a first step in differentiating these skill groups (Section 2.4.4). In combination with eye tracking, the reviewed learning analytics methods were the foundation to test the learning hints as a way of learner help or intervention (Section 2.5).

Analyzing and Visualizing Patterns

Specific methods to analyze AOI sequences (Section 2.6) and specific ways to visualize these sequences were developed to find eye movement patterns faster (Section 2.7) and to visualize and analyze the above-mentioned ideas and data sources.

Major Research Gaps

While planning the process and analysis steps of the three different case studies, two primary research gaps were recognized: 1) in visualizing eye movement data to get a fast and direct overview of reading patterns and 2) in finding eye movement patterns via a graph-based visualization and with a pattern search based on a user-definable pattern. These identified gaps were recognized as essential steps necessary to find eye movement patterns related to comprehension problems and move the state of research in the source code comprehension community forward.

Newly Developed Methods and Tools

Chapter 4 introduces the for this thesis newly developed methods and tools, as well as the used methods and tools. This thesis aims to propose methods and tools for visualizing and finding eye movement patterns to detect source code comprehension strategies. This work includes relevant contributions for the source code comprehension community, not only in the research area of eye movement patterns but also in achieving the long term vision of developing a dynamic learner support system to help novices in the source code comprehension process.

"You've got to be very careful if you don't know where you are going, because you might not get there."

– Yogi Berra, former New York Yankees catcher

3 Research and Analytics Approach

The Aims

Eye tracking studies usually accumulate a significant amount of data, which is challenging both for eye movement pattern and strategy identification and for methods and tools analyzing and visualizing such data. To fill the two research gaps identified in the related work chapter (see Section 2.8 for a condensed summary), well-thought-out research and analytics approach is needed. These approaches form the research and analytics outline for this thesis, matching the three aims described in Section 1.3. The following enumeration summarizes these aims and highlights the connection of the research and analytics approach:

First aim: Identify steps to detect source code comprehension patterns, summarized as "Pattern Identification." These steps are the content-related goals of the three case studies described in Section 3.3.

Second aim: Design and create methods and tools to test the ideas for pattern identification. These methods and tools are summarized as "Methods and Tools" and described in Section 3.2.

Third aim: Describe the resulting toolboxes for both pattern identification and the methods and tools to support detecting source code comprehension patterns. These contributions are described in Chapter 8.

Preliminary Ideas

Before planning and conducting empirical source code comprehension studies was even within the range of this thesis, preliminary ideas and directions had to be discussed,

tested, and at least partially verified. These early investigations involved analyzing available datasets and prototyping applications usable in studies. In addition to extensive literature research in the first month of this thesis, various ideas were established: The research topic was narrowed down to an analytical focus of eye movement patterns; ideas for empirical studies and analyses were created to detect and analyze eye movement patterns and strategies, which were defined as baseline for the source code comprehension analyses; a prototype for a preliminary study was constructed to test eye tracking within a programming environment with a visual and text-based programming language; the gathered data was used to create heat maps and gaze plots as visualization techniques; and a dataset from another researcher was analyzed to verify ideas for detecting eye movement patterns.

Specialized Visualizations Needed

As a result, the necessity for specialized visualizations became apparent to analyze eye movement patterns fast, easily, and successfully. Even though not every idea was a useful start for more investigations, and despite some ideas not being successful at all, this part of the thesis was essential for diving into both topics, namely eye tracking and source code comprehension, as well as for creating the prototype of a programming environment that works well in combination with eye tracking. The following sections and subsections summarize these steps, highlighting essential milestones in the beginning of this thesis. From a data analytics perspective, the CRISP-DM approach was adapted to plan the studies and subsequent analyses accordingly.

3.1 INVESTIGATIONS AND PROTOTYPING

Java as Stimuli Language

The initial ideas, influenced by the original research proposal, were discussed continuously and validated, first and foremost with the supervisors and additionally with external researchers. Participation at the doctoral consortium (DC) of the Psychology of Programming Interest Group (PPIG)[7] in July 2017 was essential to sort out ideas for the programming environment prototype and the programming language used in the source code comprehension studies. It became clear that when the primary goal is to detect eye movement patterns and strategies, the use of a self-developed language is not sufficient for source code comprehension studies. This is because the gathered results would not be comparable to other comprehension studies in the source code comprehension community. A shift towards Java as the programming language, due to its wide used in universities and other comprehension studies and its simple ways of presenting this code to participants, was the result.

EMIP Coding Scheme as Baseline

In February 2018, the seminar "Evidence About Programmers for Programming Language Design (18061)" was held in Dagstuhl[8] (Stefik, Sharif, Myers, & Hanenberg, 2018). The seminar was about knowledge in language design, eye tracking, source code comprehension,

[7] `http://www.ppig.org/workshops/ppig-2017-28th-annual-workshop` – Last accessed on December 14, 2020.

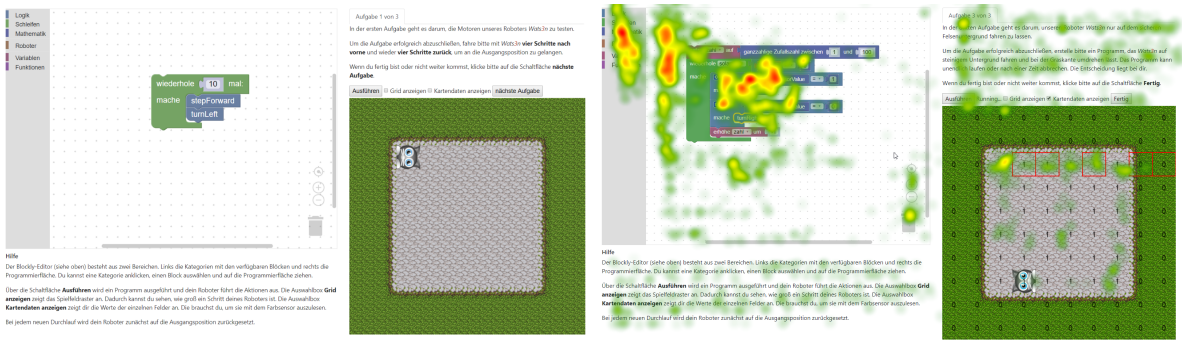
[8] `https://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=18061` – Last accessed on December 14, 2020.

programming education, and psychology. The on-site discussions about the main goals of this thesis were highly productive, successful, and essential in the retrospective for the ongoing work on the topics of this thesis. The feedback on the variance of eye tracking data, Java as a programming language in studies, and the idea of using pre-recorded eye movement data as the primary information source was necessary for the further development of this thesis. One essential outcome was the discussion about the EMIP coding scheme as a baseline for eye movement patterns and strategies, instead of creating own patterns to search for in the eye movement data.

3.1.1 PRELIMINARY STUDY – A PROGRAMMING ENVIRONMENT PROTOTYPE

Prototype for a Pre-Study

The investigations into the realms of eye tracking and source code comprehension analysis were made with a self-developed prototype and an eye movement dataset. The prototype included a programming area with Blockly, a task description, and a virtual environment for a robot (a cat in the early versions). Figure 3.1 presents a screenshot of the original programming environment (left) and a heat map superimposed on the stimulus (right). The prototype was planned as an environment for a pre-study, conducted with participants from a web-programming course at the University of Applied Sciences and Arts, Dortmund. Due to poor timing (the end of the semester) and a general lack of interest, the pre-study was never conducted.



(a) Original screenshot of the prototype.

(b) Screenshot of the prototype with a heat map.

Figure 3.1: Screenshots of both the prototype (left) and the heat map of one participant superimposed on it (right).

Text-Based Language and No Robot Emulation Preferable

Nevertheless, the prototype was used to record the eye movement data of two participants (colleagues) to test the general idea and design, the usability in combination with an eye tracker, the recorded data, quality, format, and potential for analyses. In addition, these tests were useful to become familiar with the eye tracking device itself and with conducting a source code comprehension study technically and organizationally. The results from the recordings were that a text-based language is preferable to find more participants at the university. Furthermore, some type of simulation or emulation, such as the robot, is not

necessary to gather eye movement data and can, in fact, distract from identifying eye movement patterns and comprehension strategies, which is also true for the lengthy explanations and task descriptions. The data suggested that these elements can distract participants and therefore alter the eye movement data significantly. Moreover, the heat maps were deemed to be a suitable visualization to aggregate the eye movement data; however, they are insufficient to obtain sequence information, which is necessary for analyzing eye movement patterns. Furthermore, gaze plots visualized the sequences of eye movement data, but in an insufficient form for large clusters of data, which are observable especially for the code block region and the selection of visual blocks (both in the upper left of Figure 3.1b). The gathered knowledge was thus valuable to develop specialized methods for analyzing and visualizing eye movement sequence data.

3.1.2 ANALYZING A REAL-WORLD DATASET

Real-World Dataset

To further create and validate ideas for analyzing and visualizing eye movement data, analyzing a real-world dataset[9] was the next step. This dataset served as the baseline to test heat maps and gaze plots as common visualizations for eye tracking data in the context of source code comprehension. Figure 3.2 depicts a heat map for all participants (left) and for one specific participant (right). The gaze plot for another participant is visible in Figure 3.3.

(a) Heat map visualization of all participants.

(b) Heat map visualization of one specific participant.

Figure 3.2: Heat maps of all participants (left), and heat map of one participant (right).

[9]Thanks to Teresa Busjahn, Freie Universität Berlin

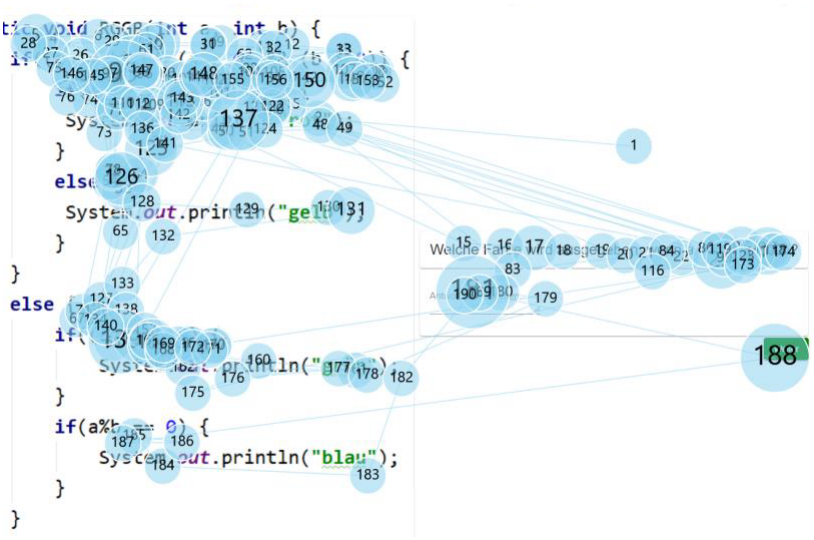


Figure 3.3: Gaze plot visualization of one participant.

Idea for Sequences of Letters

These results supported the initial analysis of eye tracking data from the self-developed prototype: that heat maps are a prime way in which to visualize aggregated data and that gaze plots can show sequences. However, the results also revealed that a method to accurately visualize eye movement patterns, without the problem of losing sequences at all (heat map) and without having trouble identifying sequences in large datasets (gaze plots), is missing. The idea of viewing eye movement data on a source code stimulus as a sequence of letters, when they matched against AOIs, arose while analyzing this dataset. This is an encoding of eye movement data because the eye tracking data (coordinates with x and y) is mapped against AOIs, which encode the eye tracking data as letters. The representation as a sequence of letters can then be used as the data basis for visualizations and analyses.

The following is an example of original eye movement data (excerpt):

```
CAAAAAABBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDCEDDDDDCCCC
DDDDEEEEEEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFCFFFFFFFFFFFFCCFEEEEF
FFFFFFFFFFFFFFCCDDCEEDEEEEEEEEEDDDDDFFFFFFFFFFCBACCCC
```

The aggregated eye movement data (based on the excerpt) is then as follows:

```
CABCDCEDCDEFCFCFEFCDCEDEDFCBAC
```

For this dataset, *Tableau*[10] was used to test new visualizations and to gain more in-depth insight into the dataset, which proved to be useful because it revealed that visualizing the encoding of AOI-Hits is useful for analyzing eye movement data.

[10] `https://www.tableau.com/` – Last accessed on December 14, 2020.

3.1.3 VISUALIZATION AND ANALYSIS IDEAS

Firsts Tests of AOI-DNAs

The initial idea of encoding and representing eye movement data as a sequence of letters was used as the basis for a visualization technique. These tests were important for later implementations of the AOI-DNAs and were organized as various iteration steps to approximate a solution that would be useful for analyzing eye movement patterns. Figure 3.4 presents screenshots of the first three iterations for designing and implementing AOI-DNAs. The first iteration (see Figure 3.4a) uses a grayscale color palette and letters to visualize linear reading patterns. The second iteration (see Figure 3.4b) uses a grayscale color palette and numbers to visualize patterns based on the source code execution. The third and last iteration (see Figure 3.4c) in this implementation cycle again uses a grayscale color palette, this time without letters or numbers, to visualize linear reading patterns. To help, an ideal linear reading pattern is visualized in the first row (not visible in Figure 3.4).

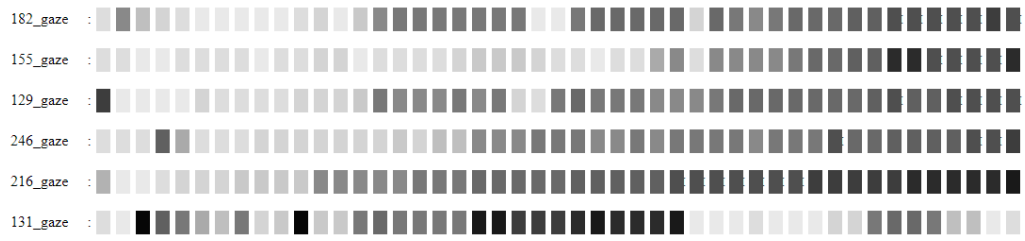
Color Coding and Sequence Alignment

Through these iterations, it became clear that color palettes can be useful to visualize the various reading patterns buried in eye movement data. In addition, the iterations demonstrated that using letters and color boxes with a small margin is the best solution for visualizing the data because visual clutter is reduced. Furthermore, these ideas included the use of the Needleman-Wunsch (N-W) and Levenshtein algorithms to investigate the possibility of sequence alignment and distance measures to compare eye movement data sequences. These analyses were conducted on the encoding of eye movement sequences. At this stage, these tests were promising enough to reuse for analyzing the data of the case studies.

Overall, these investigations were necessary for future planning of the thesis, including the analytics and research approaches, as well as the data analytics approach, both of which described in the next sections.

(a) 1. Iteration of visualizing eye movement data.

(b) 2. Iteration of visualizing eye movement data.



(c) 3. Iteration of visualizing eye movement data.

Figure 3.4: Screenshots of the three iteration steps to visualize AOI-DNAs to highlight eye movement patterns.

3.2 ANALYTICS APPROACH – METHODS AND TOOLS

Three Methods for Visualizing and Clustering

The literature research for this thesis revealed that methods and tools for a viable and straightforward overview of eye movement patterns in eye tracking data were missing. Visualizing eye movement patterns is a necessity and a common goal (see Section 2.7). However, especially for AOI-based methods, which focus on AOI fixations and transitions between them, an easy-to-use approach was missing. Moreover, an easy method of searching for (partially) available eye movement patterns, in the best cases embedded within a visualization, was also missing. This current state of research was highly influential regarding the analysis of the three case studies.

The first method focuses on encoding eye movement data with matching AOIs to visualize large strings of letters for the AOI-Hit sequences of multiple participants in a DNA-like style. With the resulting tool, viewing patterns can easily be visualized for a first and fast glimpse of the eye movement data.

The second method focuses on visualizing the AOI transitions, and therefore the viewing transitions, of a single participant in a graph-like structure. With the resulting visualizations, viewing patterns (e.g., clusters for specific AOIs or regions) are easily detectable.

The third method focuses on clustering AOI transitions, based on the first method, to show similarities and dissimilarities in the viewing behavior of participants. In addition, the tool, developed for using the first method, was extended to support a pattern search based on a pattern-matching algorithm.

The overview in Table 3.1 indicates the two contextual layers of this thesis, namely *pattern identification* and *methods and tools*. All empirical source code comprehension studies, analyses, and publications are aligned with one or both of these two layers.

	EMIP Dataset Analysis (see Chapter 5)	Code Smell Study (see Chapter 6)	Learning Hints Study (see Chapter 7)
Pattern Identification	Find Patterns	Provoke Patterns	Effect of Hints
Methods and Tools	DNA Visualization	Graph Visualization	Pattern Search

Table 3.1: Overview of the two contextual layers for aligning the work of this thesis.

3.3 RESEARCH APPROACH – ILLUSTRATING CASE STUDIES

Three Case Studies

The three case studies can be summarized as a story for finding patterns → provoking patterns → testing learning hints. To properly test the ideas for strategy identification, as well as the methods and tools, every case study was planned as an independent empirical source code comprehension study. No participant was required to take part in two or more studies, nor were they explicitly excluded from taking part in more than one study. Independent studies are overall less complex to organize in terms of participation and the recruiting of participants.

The following overview highlights the main parts and objectives of the three studies. Detailed insights into the experimental designs, the recorded data, hypotheses, analysis methods, and results are available in Chapter 5 (EMIP dataset analysis), Chapter 6 (code smell study), and Chapter 7 (learning hints study). Figure 3.5 briefly visualizes the process and time spans for the major work on the case studies.

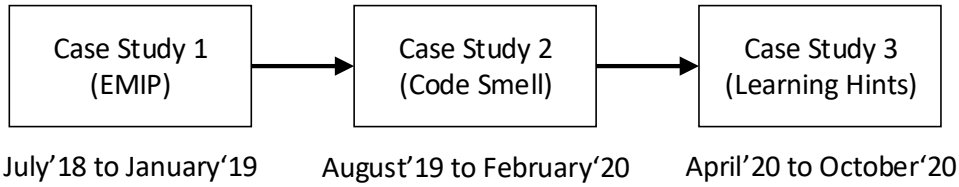


Figure 3.5: A brief overview of the case studies and the timeline of working on them.

3.3.1 STEP 1: FINDING PATTERNS – THE EMIP DATASET ANALYSIS

Problem: Missing Dataset

The early research for this thesis revealed that no dataset is available that contains information about eye movement patterns related to comprehension problems among different participants. Even for one specific source code example as a stimulus, this data is missing.

Solution: EMIP Dataset

The first study, in the form of an analysis of the existing EMIP dataset, thus aims to find specific eye movement patterns. The primary goal of this qualitative and quantitative analysis is to detect a) eye movement patterns and therefore b) comprehension strategies, both closely related to participants' source code comprehension problems in the best case patterns among different participants on the same source code stimulus.

Success: Analyzed Patterns

These eye movement patterns are a useful starting point for further analyses of what the source code comprehension problems are, how they occur, which subsequent patterns lead to a correct or incorrect comprehension result, what the first signs or ideas are, and how to intervene, to help participants with such eye movement patterns. The EMIP dataset analysis is described in Chapter 5.

3.3.2 STEP 2: PROVOKING PATTERNS – THE CODE SMELL STUDY

Problem: Semantic Meaning of Patterns

Preliminary to the second study, the EMIP dataset was analyzed to find comprehension strategies and patterns closely linked to comprehension problems. The results were promising for detecting reading strategies and patterns, but unsuccessful regarding the relation between participants' eye movement patterns and their comprehension problems. Therefore, a new study was designed and conducted to find such eye movement patterns.

Solution: New Study to Provoke Patterns

The second designed and conducted study, namely the code smell study, hence addresses the problems identified while analyzing the EMIP dataset. Specifically, eye movement patterns were found; however, the relation to comprehension problems was still missing. Thus, the second study is about a) using source code smells to provoke behavioral reading patterns, which can be categorized as comprehension strategies and are detectable as characteristic eye movement patterns linked to comprehension problems, and b) recording eye movement data not only from students but also from professionals working in the software development industry to compare both skill groups. The distractors, to provoke eye movement patterns for source code comprehension problems, are code smells, which are small regions in the source code examples that are not ideal to programming principles and conventions of the Java programming language.

Success: Better Understanding of Novices and Experts

Therefore, to obtain a better option for separating students and professionals in a novice and expert group, the primary goal of the code smell study was to provoke eye movement patterns not identifiable in the EMIP dataset and to record data from industry professionals. The study and analysis design are hypotheses-driven, with extended source code examples from the EMIP dataset, to make them as comparable to the previous EMIP analysis as possible. Provoking eye movement patterns solves the problem of not knowing whether a comprehension problem exists, because one can assume that a code smell will create uncertain situations in the comprehension process. The results of the code smell study are reported in Chapter 6.

3.3.3 STEP 3: TESTING HINTS – THE LEARNING HINTS STUDY

Problem: Learning Hint Comparison

After the code smell study, which was designed and conducted to provoke eye movement patterns in source code comprehension situations, the so-called learning hints study was created. The overall problem is that some learning hints, such as syntax highlighting, were tested in source code comprehension studies, but they were not compared to other learning hints.

Solution: Study with Varying Hints

The primary goal of this study was to visualize and analyze the way in which participants perceive source code examples in different study conditions. The questions are as follows: a) Is syntax highlighting more helpful than a dynamic scope visualization; b) will participants use the offered learning hints, and c) do they change their eye movement patterns? In conclusion, these analysis results were matched with the participants' answers to the

comprehension questions to analyze whether the different code representations and code interactions have an impact on these answers.

Success: Advantages for Some Hints

These results, described in Chapter 7, indicate that some code representations are more likely comprehensible and hence understandable than others, which affects learning environments and the overall vision of a dynamic learner support system.

3.4 THE ADAPTED DATA ANALYTICS APPROACH BASED ON CRISP-DM

CRISP-DM as Analytical Pipeline

Analyzing the collected eye tracking data from which to gain reasonable knowledge is a delicate process. Various steps must be followed to create a valid and useful data mining and data analysis pipeline. The studies were thus designed and the analyses performed with the cross-industry standard process for data mining (CRISP-DM) methodology (Wirth, 2000) in mind. This methodology was created in 1996 to shape data mining projects in the upcoming field of data science. The original CRISP-DM is described in the following six significant steps:

1. **Business understanding**: Understand the project objectives and requirements from a business perspective. The result is a data mining problem and a (preliminary) plan.
2. **Data understanding**: Collect the data and become familiar with it. Identify data quality (problems), first insights, and hidden information.
3. **Data preparation**: Construct a final dataset that meets data quality standards and the desired format.
4. **Modeling**: Choose appropriate data modeling techniques, and if they need a particular data format, then loop back to the data preparation step.
5. **Evaluation**: Evaluate the created model. If multiple models were created, then select appropriate quality criteria to compare them, and select the best model.
6. **Deployment**: Create deployable code from all previous steps, such as data preparation, modeling, and evaluation, which are necessary to run the data mining and analysis steps in an environment.

CRISP-DM for Scientific Approaches

The enumeration above shows many parallels to a successfully conducted and analyzed source code comprehension study. Some aspects are different due to exact circumstances; therefore, a slightly different process was used to design the empirical source code comprehension studies. This thesis proposes the following process with seven significant steps, which are followed in the three case studies:

1. **Scientific understanding**: Understand the study objectives and requirements from a scientific perspective with regard to the current state of scientific knowledge. The result is a (preliminary) plan to conduct the study, and a plan to validate hypotheses is raised. Furthermore, have possible publications in mind to report the findings in the last step.

2. **Data understanding**: Collect the data and become familiar with it. Identify data quality (problems), first insights, and hidden information. Alter the hypotheses if interesting subsets appear in the data. In this step, the self-developed visualizations AOI-DNA and AOI-STG already become essential and useful.

3. **Data preparation**: Construct a final dataset that meets data quality standards and the desired format to analyze and visualize the data.

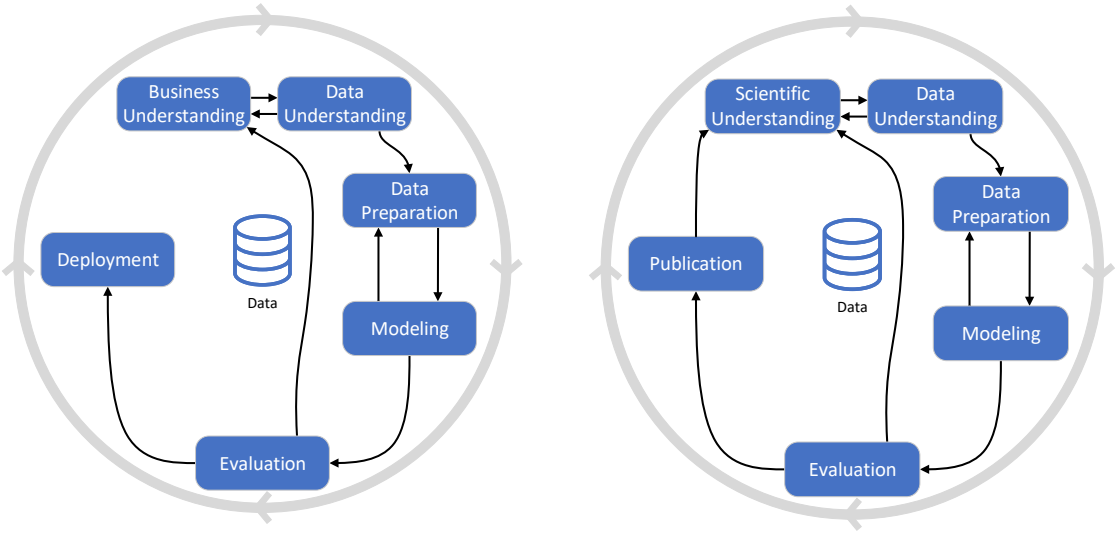
4. **Modeling**: Choose appropriate data modeling techniques, and if they need a particular data format, then loop back to the data preparation step.

5. **Evaluation**: Evaluate the created model. If multiple models were created, then select appropriate quality criteria to compare them, and select the best model.

6. **Publication**: Create a publication of some sort to report the findings to the community. Include all necessary steps to run the data mining and analysis steps (replication), such as data preparation, modeling, and evaluation.

7. **Reevaluation of scientific understanding**: Reevaluate the preliminary scientific understanding, formulate new and changed hypotheses, and plan further actions (e.g., for new studies), and begin at the first step.

Figure 3.6 illustrates both CRISP-DM processes side by side. The changes in the modified version on the right are small but important for the scientific direction.



(a) The original CRISP-DM process.

(b) The modified CRISP-DM process.

Figure 3.6: The original (left) and modified (right) CRISP-DM processes on which the data analyses in this thesis are based.

3.5 SUMMARY

Early Tests, Research Gaps, and CRISP-DM

To summarize this chapter, the investigation and prototyping was crucial to test ideas, examine a real-world eye tracking dataset with source code comprehension data, and find tangible research directions and approaches to follow in this thesis. Analytics and research approaches were clarified based on research gaps and the adopted direction of the overall thesis. It became apparent that heat maps and gaze plots are insufficient to visualize large eye movement datasets with sequential data, which is necessary to analyze eye movement patterns. As a result, own encodings and representations for eye movement sequence data were developed, and, from an organizational and project management perspective, the three case studies were outlined and planned. The CRISP-DM process was an important part of designing, conducting, and analyzing these three source code comprehension analyses. The next chapter explains the methods and tools developed or used throughout this thesis.

"What's measured improves"

– Peter F. Drucker

4

Eye-Tracking-Based Analysis Methods and Tools

Constructive Contributions

The previous chapter described encodings and representations of AOI-Hits, which led to implementations of new visualizations and analysis ideas. These were further developed as new eye-tracking-based analysis methods and tools. When designing and implementing such methods and tools, and when designing and conducting source code comprehension studies, methodical eye tracking challenges merit consideration, as the next section describes. This chapter summarizes the constructive contributions of this thesis, and Chapters 5, 6, and 7 report the results of the three case studies.

4.1 Methodical Eye Tracking Challenges

Eye Tracking is Biased

Working with eye tracking and the resulting data presents different challenges. Some of them are based on technology choices, while others are based on the choice of methods, and some challenges occur while choosing participants. In general, eye tracking is not immune to systematic biases. On the contrary, technology and directly involved methods can introduce biases at various stages of recording and analyzing the data.

Gender and Sampling Biases

The process of avoiding biases in eye-tracking-based studies starts when the experiment is designed. If the repetition of treatments should be avoided, between-subject designs are the concept of choice. It is crucial "to avoid accidental homogeneity of groups, e.g., testing two groups where one group is entirely male, the other entirely female (gender bias)"

(Duchowski, 2017, p. 206). While it is vital for other studies too, another gender bias can be introduced unintentionally when participants cannot be adequately recorded with the eye tracking setup in use and are thus excluded early in the process. This disqualification from the study often results in recruiting other participants. The problem with eye tracking is that some groups can be excluded inherently due to secondary characteristics, such as makeup and glasses, both of which can result in reflections that an eye tracker is not capable of correcting. Such circumstances can lead to sampling bias because a subgroup of a population has a lower sampling probability due to the mentioned constraints. In the end, this can lead to a non-random (biased) sample. This situation worsens if the participant group is recruited from a population in which some of these characteristics are present in higher numbers than usual or when some participant groups are structured unevenly – for example, related to specific faculties where the number of female volunteers can be lower. While experiments can be intentionally planned to use these circumstances, in most source code comprehension studies, they occur unintentionally and can cause problems.

Recognizable Software Biases

Another study-design-related bias that must be avoided is the bias toward any recognizable software products or technology in itself if that is not the primary goal of the experiment. The creation of mock-ups or prototypes of software environments, such as programming environments (integrated development environments [IDEs]), can be necessary to prevent positive or negative biases towards some products, companies, or features. The study by Hanenberg (2010), described in more detail in the related work Section 2.3.4, is a prime example of this.

Comprehension Questions

Regarding the design of source code comprehension studies, the structure of comprehension questions is crucial. An ill-defined question can move participants' focus to specific elements of a source code example, which might not be the original intention and can introduce a significant bias.

Eye Tracking Setup

Technical aspects of eye tracking setups must also be considered. These aspects concern the sampling rate of eye trackers – the Tobii TX 300 with 300Hz was used throughout this thesis, along with filters and other detection algorithms.

Sampling Rate

The sampling rate has a significant effect on the collected data, the accuracy, and the analyses that can subsequently be carried out with the data. A higher sampling rate results in a shorter sampling interval, which allows for the detection of "sharper" saccades in the means so that they seem to not occur so suddenly in the data due to an approximation. If the eye tracker samples are at 60Hz, the sampling interval is 16.67 milliseconds (ms), and with 120Hz, the interval is 8.33 ms. Moreover, eye trackers at 250Hz or 300Hz are frequently used for source code comprehension studies; the sampling intervals are 4 ms and 3.33 ms respectively for these sampling rates. Figure 4.1 presents two examples: a lower sampling frequency on the left and a higher one on the right. The difference can explain the fact

that the saccade on the left seems to occur abruptly, while the higher sampling frequency illustrates that this is not the actual case.

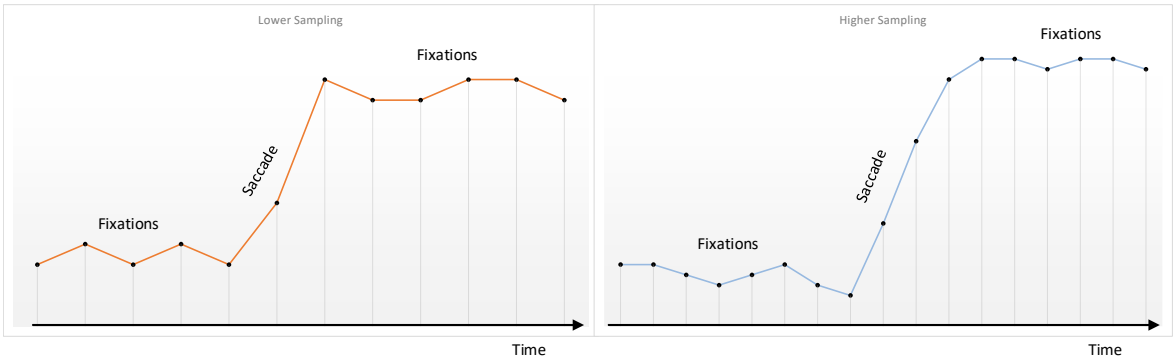


Figure 4.1: Two sampling examples with a low and a high sampling rate to visualize the possible differences (following a Tobii example[11]).

Quality of Fixations and Saccades

The practical effects are that not every eye movement characteristic can be measured with every sampling rate. At 120Hz, only fixations and pupil dilatation are measurable effectively, whereas at 600Hz and above, fixations, saccades, smooth pursuits, microsaccades, and pupil dilation are available in the data. Thus, with the used 300Hz eye tracking throughout this thesis, fixation and saccades are available with sufficient precision (see another Tobii example[11]). Both are the main eye movement characteristics needed. In addition, the sampling rate affects duration measures (i.e., fixation durations). For this eye movement characteristic, fixation start and endpoints are necessary, depending on the fixation filter's calculations. The real start and endpoints are somewhere in the sampling interval; they can be just before another sample is collected or shortly afterward. The sampling error can thus be between zero or as large as the sampling interval. On average, it will be half of one sampling interval, which can be influenced by a shorter sampling interval or higher sampling rate. Overall, there is no bias in the measurements because the start and the end of fixation are overestimated. Both errors cancel each other out on average.

Eye Tracking Filter

The way in which a filter is parametrized greatly affects the outcome of the transformation of raw data to fixation data. "There is always a risk that the bias or variability of an algorithm can drown out the effects you are looking for" (Holmqvist, 2011, p. 8). Otherwise, fixations and saccades are most often necessary for subsequent analysis steps. In this thesis, the built-in *I-VT* filter of the Tobii Pro Studio software was used with the default values. The exact values are additionally reported in the case studies.

AOI-Induced Biases

Another bias can be introduced when choosing the design of the AOIs, to mark essential regions of a stimulus. In source code comprehension studies, this is usually source code tokens, lines, or regions. The question is, how should AOIs be specified to detect as many relevant fixations as possible, and how does one ignore the noisy part of the data? This

[11] `https://www.tobiipro.com/learn-and-support/learn/eye-tracking-essentials/eye-tracker-sampling-frequency/` – Last accessed on December 14, 2020.

connection was analyzed in (Deitelhoff et al., 2019a). The goal was to assess how many fixations are left out or unnecessarily detected for smaller and larger AOI definitions. The next section describes the results.

Many Influences in Eye Tracking Studies

To summarize the methodical eye tracking challenges, the technological and methodical choices are highly influential and can significantly change both the interpretation of eye movement data and the outcome of studies. Thus, the choices that were made during a study and why they were made need to be made transparent when reporting data. When planning experiments, the mentioned biases must be considered to increase the chances of creating valid studies.

4.2 THE INFLUENCE OF VARYING AOI MODELS

The following section is based on the publication "The Influence of Different AOI Models in Source Code Comprehension Analysis" by Deitelhoff et al. (2019a).

AOI Model Categories

Defining AOIs is a common approach to map eye tracking data to a stimulus, which also applies for source code examples. There are different ways of defining AOIs, as described in general in Section 2.4.1. Some definitions ignore the underlying stimulus, whereas others completely depend on it. Moreover, some AOI models are based on the shape of the stimulus or objects within it, and others use domain knowledge in the definition process. Not every category is equally important for source code comprehension studies, but the categorization helps in identifying the ones that are.

GRID AOI MODEL

AOIs as Grid

A grid (or gridded) AOI model consists of AOIs separating the stimulus without any specific knowledge or without respect to the underlying domain. The stimulus is divided into AOIs of equal size so that there are no margins and usually no space left at the borders of the stimulus. Variants of grid AOI models exist, for example with unequally distributed AOI sizes or with a grid that respects the underlying stimulus. The AOI sizes are normally based on assumptions of the stimulus or domain-specific knowledge. Figure 4.2 portrays an example with source code as the underlying stimulus, visualizing why this approach is rarely used in source code comprehension.

DOMAIN-SPECIFIC AOI MODEL

AOIs Respecting the Stimulus

AOI models based on knowledge about the research domain are categorized as domain-specific AOI models. There are all sorts of AOI definitions in such models. In source code comprehension, domain-specific AOIs often model tokens, lines, or regions in the source code. In text reading, sentences and paragraphs may be important. In medical image reading skills, for example in the domain of dental students (Castner et al., 2018), a domain-specific

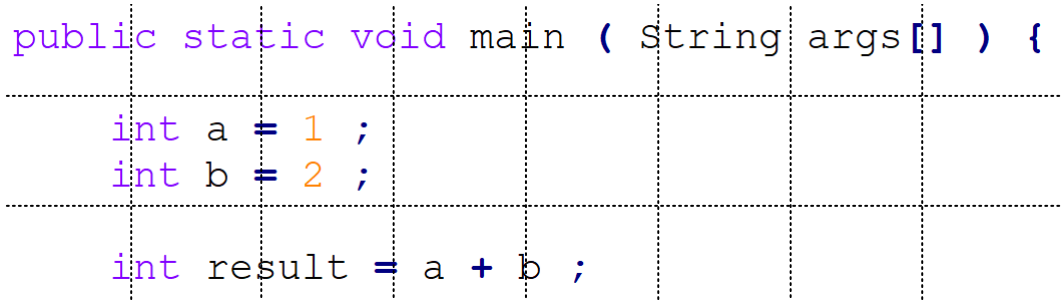


Figure 4.2: The AOI grid model dividing the stimulus into areas.

AOI model separates the stimulus into four equally distributed regions, similar to a grid. This is a variant of a grid model and in fact a combination of a domain-specific and a grid AOI model. Apart from this, some models are abstractions of one another. A region AOI model, which combines line AOIs without using a bounding rectangle, is an abstraction of the line AOI model and provides transitions between these AOI regions. The same connection can be made between the token and the line AOI models, whereas the line AOI is the abstraction from the tokens. Figure 4.3 illustrates a domain-specific AOI model with visualizations for tokens, lines, and regions.

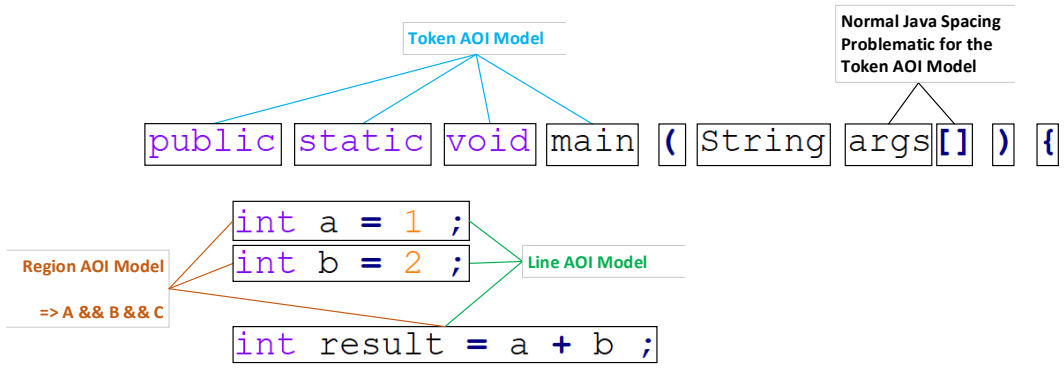


Figure 4.3: A domain-specific AOI model based on tokens, lines, and regions for a source code stimulus.

BEHAVIOR-ENRICHED AOI MODEL

User Behavior to AOIs

While there are arguments that every domain-specific AOI model is knowledge-driven, a specialized categorization for this kind of AOI model was created to categorize AOI definitions, directly induced or created by eye movements. For example, the eye movements of an expert can be used to create a heat map over a stimulus. The most viewed regions can then be ordered and used for AOI definitions. This allows for an AOI-Hit detection based on the eye movements of an expert to compare other eye movement data with that of one of the experts. Figure 4.4 displays an example of the behavior-enriched AOI model. This example is based on the heat map of the viewing behavior of participants.

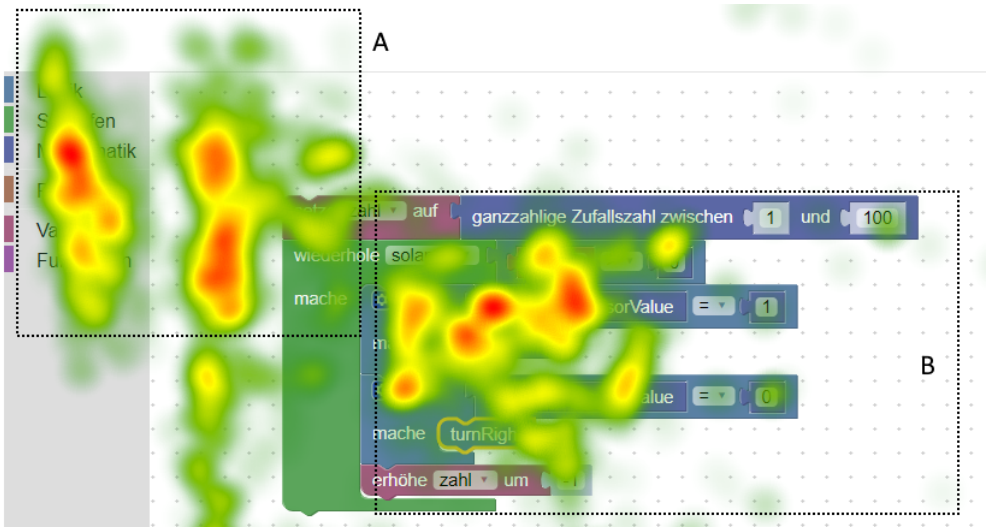


Figure 4.4: An example of a behavior-enriched AOI model, based on a heat map of the viewing behavior of participants.

AOI MARGINS AND PADDINGS

Cascading Style Sheet (CSS) Similar Definitions

Every AOI model needs an additional definition of AOI margins because the AOI model itself describes the data basis on which the AOIs are created. Holmqvist (2011) recommends AOI margins with a *buffer space* of 1° to 1.5° of a visual angle, depending on the accuracy of the eye tracking hardware. This buffer space is similar to padding, known from cascading style sheets (CSSs). In source code comprehension research, the question is whether AOI models should have gaps, and if yes, then how wide or narrow they should be. These definitions are limited by the available space between tokens, lines, or regions within a source code stimulus. All possible combinations can make sense in a particular scenario; however, they can significantly influence the subsequent data analysis.

AOI Model Influence Analyzed

If and how varying AOI models influence the analysis of eye movement data was investigated by comparing two AOI models called $Model_{emip}$ and $Model_{no-gap}$ (Deitelhoff et al., 2019a). The result was that for the source code example – Rectangle of the EMIP dataset – important eye movement sequences could be found, and they were altered between both models. The summary of this investigation is as follows: "The overall problem with missing AOI transitions is that they can have an effect on the validity of scan paths and therefore on the ability to explain a certain code comprehension behavior. At worst, a missing transition can break up a pattern search algorithm, because a certain pattern isn't existent in the data any more or is covered" (Deitelhoff et al., 2019a, p. 8).

Line and Region AOIs

Therefore, the following are important: how AOIs are defined on a source code stimulus and with which AOI margins and paddings. Overall, to avoid analyzing too much noise, the use of AOI definitions that fill up the whole space between each AOI (each source code line) is not recommended. In this thesis, the data analysis follows the AOI line and region definitions, based on source code lines and combinations thereof as regions. Both definitions

are categorized as domain-specific AOI models. For the paddings, the rule is to define the AOIs two to three pixels larger than the lines, to account for a small reduction in eye tracking data quality in the recording process. Furthermore, the margins are defined as maximally as possible. Figure 4.5 provides an example of padding and margins. The inner gray rectangles with dotted lines are present only for visualization purposes.

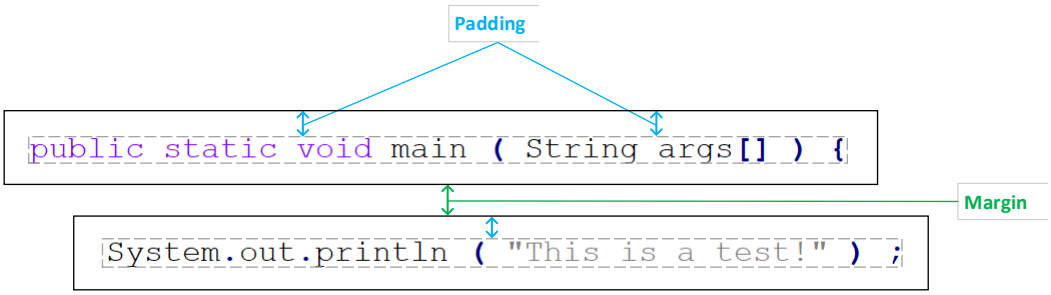


Figure 4.5: AOI margin and padding visualized for a Java code snippet.

4.3 STANDARD ANALYSIS TECHNIQUES FOR EYE TRACKING DATA

Screen Coordinates for Stationary Eye Tracker

Eye tracking data, which was recorded on a stimulus presented on a screen, is usually encoded as screen coordinates. Such representation is not necessarily used for other eye tracking devices. However, this thesis focuses on source code comprehension with a stationary eye tracker, and other scenarios are not covered in this work. Encoding as screen coordinates is a basic form of representing eye tracking data for a screen recording. To calculate fixations and saccades, this raw data is transformed with the help of a fixation filter; the transformation does not change the encoding as screen coordinates. The two most basic and common visualizations operate on top of this screen coordinate encoding: heat maps and gaze plots. Examples of these visualizations can be found in Figure 3.2, and 3.3 in Section 3.1.2.

Heat Maps for Aggregated Data

A **heat map** visualizes eye tracking data as aggregated data points with a color scheme informing about the visual attention. Heat maps can be constructed based on different eye tracking metrics and are usually superimposed on the stimulus to add the necessary information about the location back into the data. In source code comprehension studies, the number of fixations and the necessary effort on the stimulus are common metrics from which to construct heat maps. The number of fixations indicate how often a participant or a group of participants look at a region on the stimulus. Moreover, the fixation duration visualizes how long someone or a group of participants looks at regions of the stimulus. Heat maps are useful for visualizing an aggregated form of eye tracking data, but information about the location on the stimulus, the sequence, options for quantifying the data, and automated analyses are lost in the visualization. Analyses are based on the underlying encoded data.

Gaze Plots for Sequence Data

A **gaze plot** visualizes eye tracking data as a sequence of gaze points, consisting of the location, order, and time spent looking at a location. Gaze plots are usually superimposed on the stimulus to connect the location of a gaze point with the content information of the stimulus. The primary goal is to visualize the time sequence of looking at a stimulus material. In source code comprehension studies, gaze plots are often used to visualize the source code lines or regions that a participant is looking at. Gaze plots can visualize a single participant or a large group of participants. However, the visualization is prone to visual clutter because many different gaze plots will likely overlap in the visualization, making it difficult to follow a single gaze plot to analyze the sequence. Gaze plots are useful for visualizing a sequence of gaze points, but data quantification and automated analyses are lost in the visualization and done with the underlying encoded data.

AOI-DNA and AOI-STG

Section 3.1 explained that specific encodings, representations, and visualizations of sequence data to effectively analyze eye movement data were missing. Since finding eye movement patterns and strategies is the main goal of this thesis, such methods and tools had to be constructed. The resulting two methods, *AOI-DNA* and *AOI-STG*, are used throughout this thesis and are explained in the following two subsections.

4.4 AOI SEQUENCES (AOI-DNAS)

AOI-DNA as Sequence of AOI-Hits

Eye tracking raw data, transformed to fixations and saccades with a filter, can be used to determine the reading behavior on a stimulus – in this case, on source code examples. The AOIs defined for these examples (source code lines or regions) were matched against the fixations to obtain a list of AOI-Hits. Every AOI-Hit is labeled with a symbol, and a sequence of AOI-Hits is thus an encoding of eye movement data on a source code example, which is called *AOI-DNA*, because it represents the essence of an eye movement example in a source code. For readability purposes, unique letters were used to label each AOI and thus every AOI-Hit. This set of letters consists of distinguishable symbols related to the AOIs of a source code stimulus. These AOI-DNAs represent a unique reading behavior of one participant. Moreover, such a sequence is an abstraction of a gaze plot, which is often superimposed on the stimulus to add the necessary information about where a fixation occurs. Due to the labeling, this information is implicitly encoded within the AOI-Hits and the sequence.

Reasons for AOI-DNA as Method

The reasons for choosing AOI-DNAs over other methods and visualizations are manifold. First of all, it should be noted that the possibilities for visualizing AOI-DNAs, as seen in the following sections, are to be considered secondary. The primary reason for AOI-DNAs and the development of it is the representation of eye movement data on a stimulus. This representation is the basis for all downstream use cases described in the following. Using a sequence of letters for representing AOI-based eye movement data is not common in the source code comprehension community. Until this work, such a sequence was only an intermediate step, mostly hidden in the preparation process of the data. Using them directly and

naming the method as AOI-DNAs makes it clear, that these sequences as representations are more useful. A subsequent and new use case are 1D-CNNs. These Convolutional Neural Networks can use the one-dimensional input as basis for identifying and finding eye movement patterns, a promising way of detecting reading strategies encoded in a AOI-DNA.

Multiple Use Cases for AOI-DNAs

The representation of eye movement data on a source code example as sequence of letters can be used in various ways: visualizing the sequences, using sequence alignment and clustering techniques, and pattern matching with regular expressions to find subsequences. These use cases are explained hereafter.

4.4.1 VISUALIZATION

The following section is partly based on the publication "An Intuitive Visualization for Rapid Data Analysis: Using the DNA Metaphor for Eye Movement Patterns" by Deitelhoff et al. (2019b).

Visualizations for AOI-DNAs

Visualizing AOI-DNAs can be achieved in various ways, highlighting different aspects and patterns in the data. A common method is to represent each letter in the sequence as a unique box with a unique color. These visualizations are independent of any representation of the stimulus material. A visualization for the AOI-DNAs can use, for example, different color codes, wherein every symbol is assigned to a color to visualize reading behavior. Another aspect is rendering certain symbols such as the background color, which is equal to render them blank, to highlight different AOIs for a visual focus, for example important regions in the code or the number of non-hits on AOIs (blanks). Moreover, an aggregated sequence can be used to visualize and highlight different transitions and hide the dwell time in which an AOI is viewed more than once consecutively. AOI-DNAs are created by the consistent data analytics pipeline, described in Section 4.7. In the first versions, the analysis script created AOI-DNAs and visualized them as static HTML pages. The first iterations of AOI-DNA visualizations are available in Chapter 3.4. In later analyses, Tableau was used to visualize the AOI-DNAs, and, as the last iteration, the analysis tool CodeSight was created to gain more control over the visualizations and to integrate a pattern search. Figure 4.6 depicts two AOI-DNAs as examples with a color code for sequential reading.

The example shows two AOI-DNAs. The AOI-Hits in the sequence are visualized with a small gap to emphasize every AOI-Hit. In addition, the examples are color-coded with a grayscale color gradient to demonstrate sequential reading behavior. The green AOI-Hits depict eye movements to a specially designated area on the stimulus.

Color Coding and Color Gradients

To visualize patterns, they can be encoded with a specific color gradient, which must be as intuitive as possible to spot the main characteristics of the pattern without any further in-depth analysis. For example, color gradients from gray $\rightarrow$ black or based on red or blue colors are visually intuitive to spot linear patterns. To highlight certain aspects of a pattern, fixed colors for specific AOIs, while the rest is rendered in white, can be used to clearly visualize these AOIs. An important aspect is that most patterns are coupled to a context,

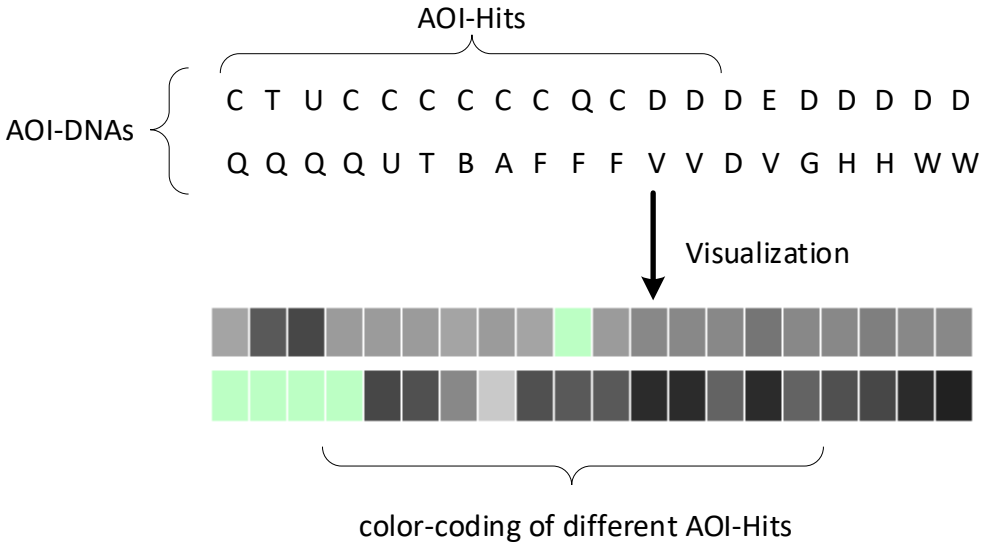


Figure 4.6: Two examples of AOI-DNAs visualized with a color code for sequential reading behavior (Story Order Reading, SOR).

which specifies the usefulness of a specific visualization. The same pattern on two different source code examples can result in entirely different visualizations with different assertions.

Layout Options

A visualization of AOI-DNAs can use different layouts to depict AOI-Hit sequences. a) The *horizontal layout* is suitable for the visualization of many data rows with a short or medium symbol count, and b) the *vertical layout* is appropriate for an average number of participants with a high symbol count. In the horizontal layout, analyzing patterns means scrolling horizontally, while in the vertical layout, scrolling vertically is necessary to view the high number of symbols. Furthermore, visualizing AOI-Hits with small rectangles has proved to be beneficial. Even with an ideal color code (e.g., with lines), the problem is that frequent transitions will result in visual clutter, which can be reduced by using rectangles with a small margin on both sides. As a result, every symbol is recognizable, while the big picture remains intact if a proper color code is used.

Long AOI-DNAs as a Drawback

AOI-based visualization methods often have problems with many AOIs and transitions. Nevertheless, one advantage of AOI-DNAs is that, theoretically, they can be used to visualize hundreds of AOIs, and since transitions are modeled implicitly by the symbol order, such sequences stay readable. Practically, it is a matter of color coding and visualizing eye movement patterns. With an increased number of AOIs, more colors are necessary to encode these symbols. Therefore, finding a sufficient number of distinct human colors to read and interpret visualized AOI-DNAs accurately could be a problem. In general, the advantages of AOI-DNAs are that transitions are modeled implicitly and that they are independent of the source code stimulus to be interpretable. One drawback is that many different AOIs and AOI-Hits produce long AOI-DNAs, which can then be difficult to analyze visually.

4.4.2 SEQUENCE ALIGNMENT

Similarity Measures

Sequence alignment, which is part of the sequence analysis approach, is used in this thesis to analyze similar reading patterns of participants. These algorithms are based on the similarity score, calculated with the N-W algorithm, which initially calculates the alignment between two protein or nucleotide sequences. The similarity is pairwise calculated for all AOI sequences of every participant and every code example. These scores provide insight into how similar the viewing behavior is. The N-W algorithm has been applied to eye movement patterns before, for example by Cristino et al. (2010). In the settings in this thesis, both sequences can have mutations. If two participants P_x and P_y compared with the algorithm, each sequence can be changed to create a better match to the other one. Another algorithm used in this thesis is the optimal match, which is similar to the Levenshtein distance. In addition, there are methods for alignment-free sequence analysis, which are not used and thus not covered in this thesis.

Substitution Matrix for Needleman-Wunsch

For the N-W sequence alignment, a substitution matrix was created. A shortened example is available in Table 4.1 because the size of these matrices depends on the number of different AOIs on a source code stimulus. These matrices can thus become relatively large and vary between the analyses in this thesis; however, the principle remains the same. The following substitution matrix (similarity matrix) operationalizes a basic *edit distance* between both sequences: $0 = match$, $1 = gap$ directions ($P_x \iff P_y$), $1 = insertion/deletion$, and furthermore $2 = gapopening$. In this matrix or model, which is used throughout this thesis, emphasizing the differences between specific AOIs is not necessary. Every substitution, deletion, or insertion operation has the same costs.

Table 4.1: Shortened example of a substitution matrix for the N-W algorithm.

	A	B	C	D	E	F	G	H	I	J
A	0	2	2	2	2	2	2	2	2	2
B	2	0	1	1	1	1	1	1	1	1
C	2	1	0	1	1	1	1	1	1	1
D	2	1	1	0	1	1	1	1	1	1
E	2	1	1	1	0	1	1	1	1	1
F	2	1	1	1	1	0	1	1	1	1
G	2	1	1	1	1	1	0	1	1	1
H	2	1	1	1	1	1	1	0	1	1
I	2	1	1	1	1	1	1	1	0	1
J	2	1	1	1	1	1	1	1	1	0

Optimal Matching

Apart from the N-W algorithm, the optimal matching (OM) algorithm is used. It is the basis of analyzing the similarities of sequences to cluster them afterward. The N-W algorithm was already used to compare eye movement data (Busjahn, Bednarik, et al., 2015), and the OM has proven to be successful for clustering sequence patterns in small group work (Doberstein, Hecking, & Hoppe, 2017).

4.4.3 CLUSTERING

Hierarchical Clustering with Dendrograms

After calculating the pairwise edit distances with the N-W algorithm for every participant, the *Euclidean distances* of the resulting matrices were calculated for use as input for a hierarchical clustering to find similar groups. *Dendrograms* were used to visualize the resulting *hierarchical clustering*, which uses Ward's minimum variance methods to find compact, spherical clusters. This algorithm has been implemented by the *ward.D2* method in R (Murtagh & Legendre, 2014). Beforehand, the *elbow*, *silhouette*, and *gap statistic* methods were used to determine the optimal number of clusters in the case studies.

TraMineR for AOI-DNAs

The R package *TraMineR*[12] is used "for mining, describing and visualizing sequences of states." The package can handle various forms of sequence data, for example to calculate distances and to discover and plot representative sequences and other aggregated characteristics (e.g., transition rates and average duration in each state), among other features. In this thesis, TraMineR is used to analyze AOI-DNAs with the optimal match algorithm and for clustering and visualizing these results. Clusters with similar sequences can thus be found (e.g., to detect similar AOI-DNAs and thus similar eye movement patterns). These results are valuable to detect groups within the eye movement data based on typical eye movement patterns, which then perhaps explain groups of participants with other related attributes, such as answer quality or the overall duration on a source code example.

4.4.4 PATTERN SEARCH WITH REGULAR EXPRESSIONS

The following section is partly based on the publication "An Intuitive Visualization for Rapid Data Analysis: Using the DNA Metaphor for Eye Movement Patterns" by Deitelhoff et al. (2019b).

Pattern Search for AOI-DNAs

In addition to pattern visualizations, the first version of a pattern search was implemented (Deitelhoff et al., 2019b). This search method is based on regular expressions usable to search for patterns in the AOI-DNAs. Every pattern match will be highlighted (red border) in the overview of AOI-DNAs to spot the relevant sections easily. In contrast to pattern visualizations, a search pattern can search for more complex patterns, for example one that specifies some AOIs in the beginning, followed by a limited number of other AOIs. In addition, the AOIs between two important AOIs can be specified. However, both examples cannot be achieved with visualizations alone.

Matching Based on Regular Expressions

The pattern search directly uses regular expressions without any simplifications or additions to the syntax. Therefore, the search patterns are quite chatty and sometimes complex, which is okay for a prototype and can be simplified in future versions. In the regular expression, labels for the AOIs are used, with a regular expression group in between. In the example `A(?:.1,)B`, the AOIs are `A` and `B`, and due to the characters *?:*, the expression in parentheses is a non-capturing group because capturing the space between AOIs for use as a capturing

[12] `http://traminer.unige.ch` – Last accessed on December 14, 2020.

result is not the intention. The dot matches any character (except for line terminators), and the curly brackets define a regular expression quantifier, matching characters between one and unlimited times. Figure 4.7 visualizes the connection between regular expressions and the AOIs of a source code example.

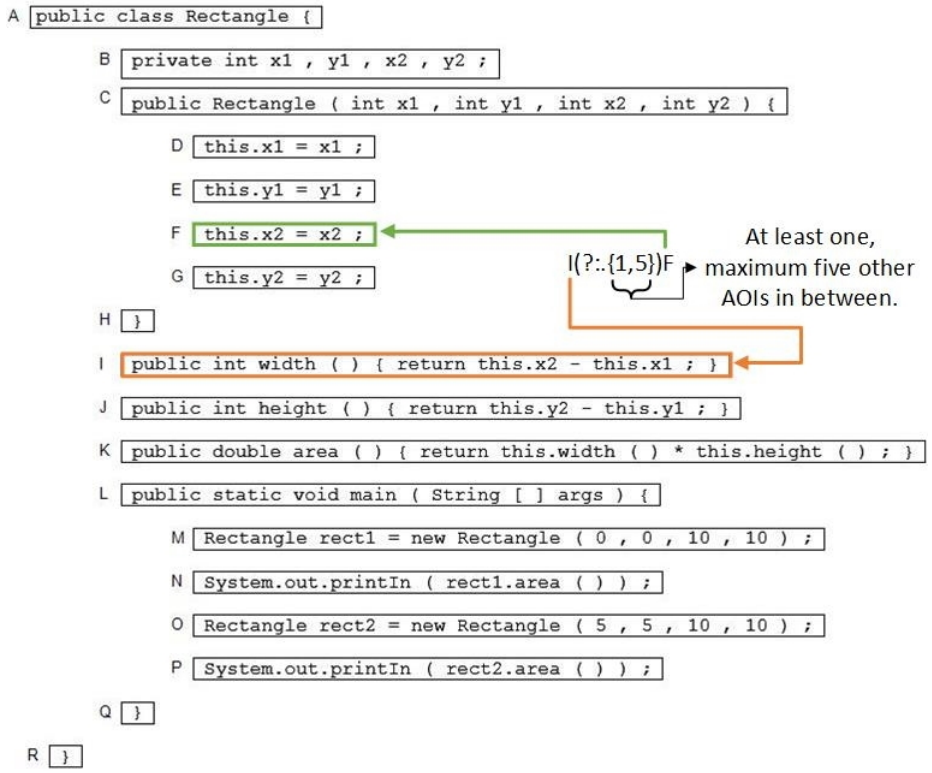


Figure 4.7: The Rectangle source code example with visualized AOIs (line model), AOI-DNA colors for reference on the left, and an example of regular expression with matching AOIs. Image from (Deitelhoff et al., 2019b).

Example of Execution Order Reading (EOR)

This method is useful to define and search for various eye movement patterns. For the execution order reading (EOR) pattern as an example, the AOIs L → M → C → N → K → I → K → J are important. The AOIs are colored with a gradient from light cyan to blue (T → ■). As a search pattern the regular expression `L(?:.1,)M(?:.1,)C(?:.1,)N(?:.1,)K(?:.1,)I(?:.1,)K(?:.1,)J` can be used, which will find the AOIs in the described sequence with a minimum of one AOI and no limit of AOIs between the specified ones.

Examples for Flicking and Retrace Declaration

Other examples are pattern searches for the eye movement patterns flicking and the retrace declaration. If the two colors dark violet and light green (■ and ■) with high contrast are chosen, they are highly distinguishable and highlight the AOIs M → C. This sequence describes the moving gaze between two related source code elements (e.g., a constructor call with real arguments and the constructor definition with parameter definitions). For the pattern search, the regular expression M(?:.1,5)C searches for this pattern, which will find the AOIs M and C with 1…5 other AOIs in between. The retrace declaration pattern describes jumps between a variable declaration and its use. Again, two colors with a high contrast

were chosen, in this case, light blue and red (and). They highlight the AOIs I → F. The sequence can describe the moving gaze between the usage of a variable in AOI I and the declaration and initialization in AOI F. The pattern search is based on the pattern `I(?:.1,5)F`, which is content-wise equivalent to that of the *flicking* pattern, but with different AOIs. Figures 4.8 and 4.9 visualize these patterns as one example of eye movement data.

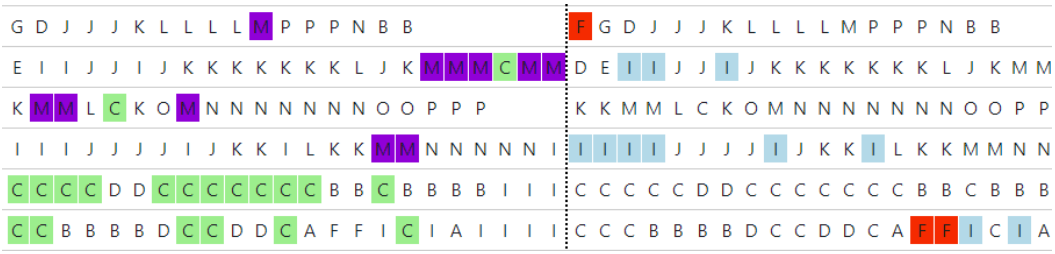


Figure 4.8: Extracts from the *flicking* (left) and *retrace declaration* (right) visualizations.

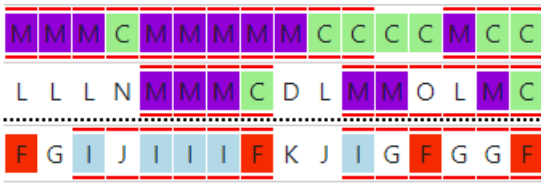


Figure 4.9: *Flicking* (top) and *retrace declaration* (bottom) visualizations with pattern search matches (red borders).

4.5 AOI TRANSITIONS (AOI-STGS)

The following section is partly based on the publication "An Intuitive Visualization for Rapid Data Analysis: Using the DNA Metaphor for Eye Movement Patterns" by Deitelhoff et al. (2019b).

AOI-STG as Model

While sequences of AOI-Hits implicitly contain transitions with the order of each label within the sequence, the AOI-STG model developed in this thesis models distinct AOIs and the changing transitions between them. The abbreviation STG stands for *state transition graph* because the visualization looks like a transition graph, with minor adjustments. An AOI-STG is comparable to Markov models, where every AOI is a node, and every transition is translated to an edge. Instead of using probabilities for the transitions, the index of the fixation is employed to be able to follow the flow of eye movements. AOI-STGs are constructed from AOI-DNAs with the *Dot* tool, which is part of the *Graph Visualization Software* (Graphviz) software package. This tool requires the definition of a graph in the Dot language. The files were created in parallel by the analysis pipeline when the AOI-DNAs were generated. The alphabet describes the nodes, and the order of symbols implicitly describes the transitions in the graph. An example of an AOI-STG is visible in Figure 4.10.

Fountains as Characteristic

This type of visualization has some advantages. The start and end nodes (green and red) are visible, and regions with accumulated transitions are visualized closer together.

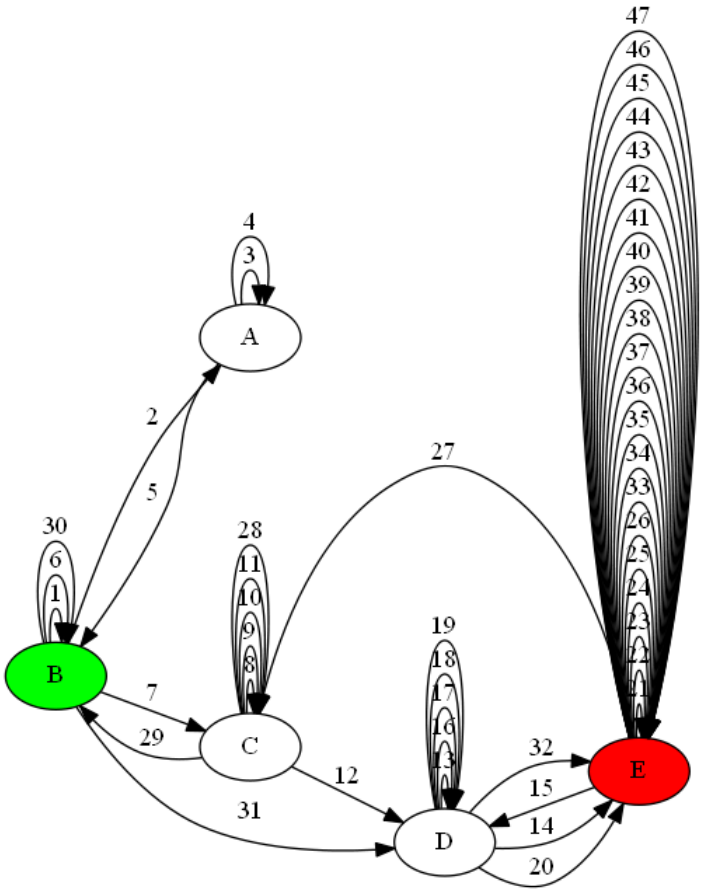


Figure 4.10: Example of AOI-STG to visualize transitions between AOIs. The start (green) and end (red) nodes are visible, as well as fountains for a higher focus on specific AOIs.

The Dot layout algorithm does this layout. If nodes are linked together with many edges, the algorithm tries to place them closer together. This layout minimizes transitions all over the graph. In addition, the so-called fountains visualize edges that have the same start and end node. These loops model the dwell time in eye movement data. High fountains indicate that an AOI was focused on many times. Furthermore, the index on every edge can be used to keep track of the eye movement in the data from one AOI to another. The drawback of AOI-DNAs is that many transitions increase the number of visualized AOIs, and the length of AOI-DNAs is a strength for AOI-STGs. Every AOI is only visualized once per graph. The drawback of AOI-STGs is that too many transitions can render the graph unreadable. This effect occurred throughout this thesis when many eye movements had to be visualized. For these scenarios, a static AOI-STG is thus not sufficient.

AOI-STGs to Visualize Transitions

Throughout this thesis, AOI-STGs are used to visualize transitions, especially for the AOI region model, where source code lines are combined into regions. For such data, the transitions are minimized because the regions cover more elements of the source code examples. For the AOI line model, AOI-STGs are well suited for finding clusters of transitions because of the layout algorithm.

4.6 CODESIGHT

Dedicated Application

The data analytics pipeline, summarized in Section 4.7, was created for every dataset-relevant eye movement metric and is the basis for AOI-DNAs and AOI-STGs. In the early stages of this thesis, the visualizations were done with static HTML pages or with Tableau. While this is sufficient for some scenarios, a dedicated application was missing to integrate various color codes and the newly developed pattern search based on regular expressions.

Data Bases are AOI-DNAs

The analyses' scripts still created the sequences of AOI-Hits, but the visualization of AOI-DNAs and the pattern matching were moved to CodeSight, which provides the features to use different color codes and to search for eye movement patterns (pattern search). Figure 4.11 presents an example of an AOI-DNA within the analysis tool.

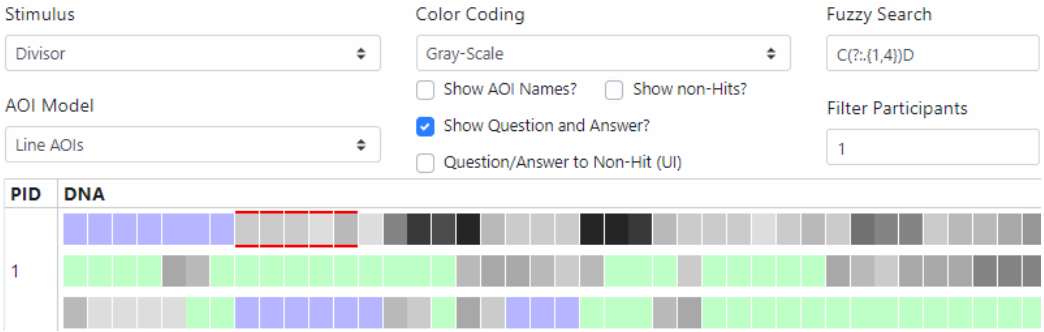


Figure 4.11: Example of AOI-DNA in CodeSight to visualize reading behaviors with a gray-scale color code and red markers, which indicates a pattern search match.

Focus on AOI-DNA Visualization and Pattern Search

CodeSight is capable of visualizing different AOI-DNAs in a horizontal layout. The tool also provides various options to switch between the stimulus (data basis); the AOI model; and other options, for example displaying the AOI names, non-hints (blanks), and specialized AOIs such as question and answer areas. Thus, switching between different studies and perspectives becomes easy. Furthermore, using the pattern search is possible, as is filtering participants by their IDs. If the pattern search finds matches, the areas of the AOI-DNAs are visualized with red borders.

Implemented as Web-App

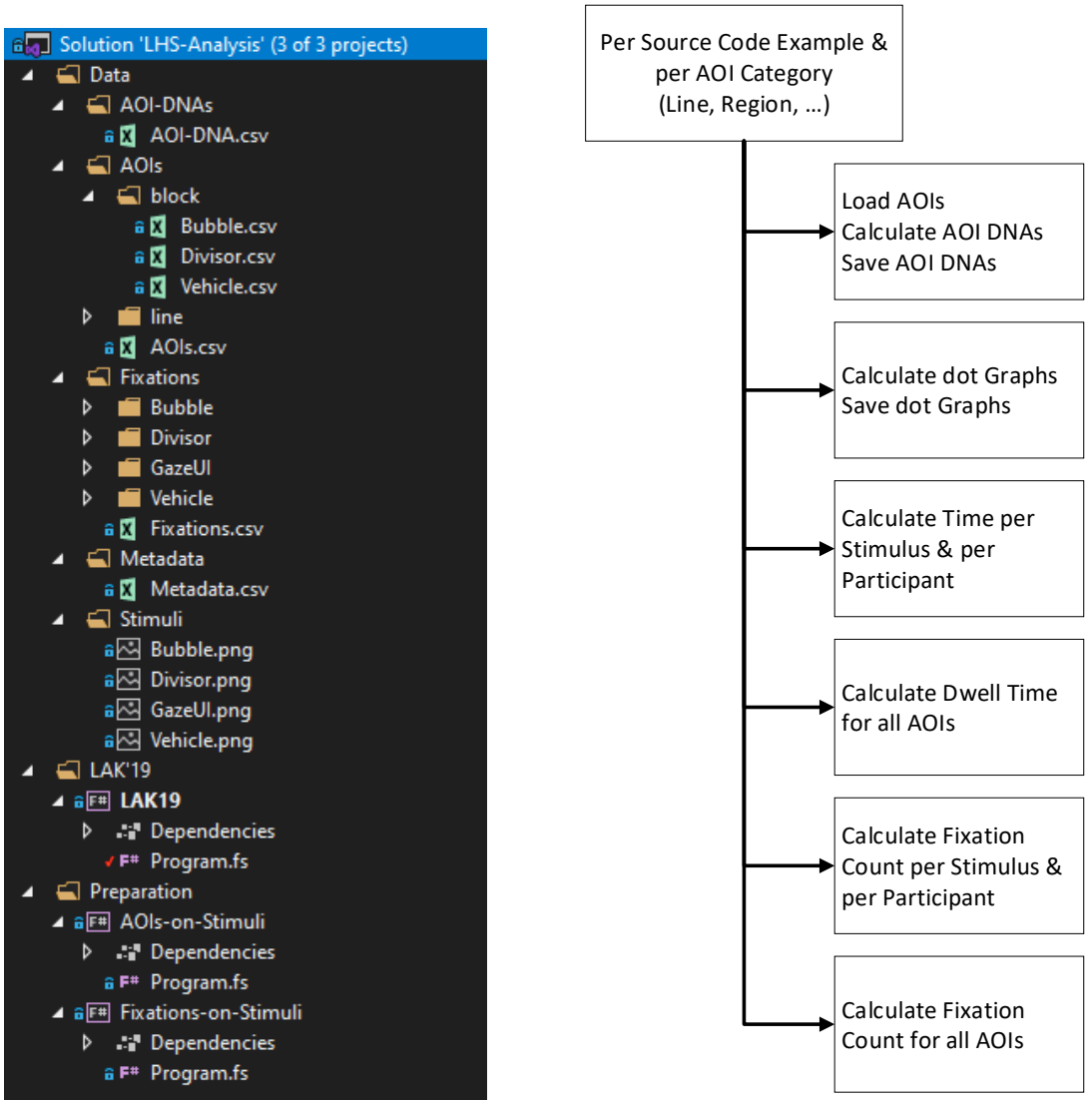
CodeSight is implemented as a web app with *ASP.NET Core 3* and *F#* in the back-end, which proved to be an advantage because some of the F# code of the analysis scripts could be reused in CodeSight. On the front-end, *Vue.js* and *TypeScript* are used to realize the user interface. Overall, CodeSight is in an early prototype stage, which works for the datasets used in this thesis. The application is not currently generalizable for every eye movement dataset.

4.7 CONSISTENT DATA ANALYTICS PIPELINE

Function Programming with F#

To make the data preparation and analysis process for the empirical studies easier, the data understanding and preparation steps were coded into *F#* projects for the .NET platform.

Therefore, the analysis scripts were used throughout all studies, and only small modifications were needed between the analysis steps of each study and analysis. These scripts made the analysis more reliable, which is a direct result of implementing the CRISP-DM process in every analysis process (refer to Section 3.4). Figure 4.12a depicts the Visual Studio project structure for the learning hints study analysis. Figure 4.12b illustrates the actual generic analysis pipeline for an analysis project, which is embedded in the *LAK19* project on the left side. The sequential data flow can be made parallel in future versions.



(a) Screenshot of the Visual Studio project structure for the learning hints study analysis.

(b) Generic data analysis pipeline for the analysis project. The sequential analysis can be parallelized in future versions.

Figure 4.12: An example of a Visual Studio analysis project (left) and the abstracted program flow for the generic sequential analysis (right).

Dedicated Analyses Projects

The implementation of multiple .NET projects, written in *F#*[13], which is a type-safe, multi-paradigm programming language, was essential for fast and reliable analysis. F# started as a research project at Microsoft Research and is currently developed by Microsoft and the F# Software Foundation. Such an analysis project usually consists of three different subprojects: 1) a visualization of the fixations on the source code examples; 2) a visualization of the AOIs on the source code examples; and 3) the principal analysis project, usually named after the primary analysis, as the name of the study indicates. If necessary, additional projects are added to the overall solution (e.g., for implementing specific analyzing methods for publications). Listing 4.1 presents an example of using F# to calculate AOI-DNAs from a list of metadata, loaded fixations, and AOIs.

F# Code for Calculating AOI-Hits

```
let calculateAOIDNAs stimulus aois aoiCategory =
  let data = CSS.Load "Data/Metadata/Metadata.csv"
  data.Rows
    |> Seq.map (fun elem -> (elem.PID, { ExpJava=elem.Exp_Java;
       ExpJavaYears=elem.Exp_Years_Java; S1R=elem.S1R; S2R=elem.S2R; S3R=
       elem.S3R }, loadFixations stimulus elem.PID))
    |> Seq.map (fun (id, metadata, items) -> (id, metadata, Seq.map (fun
       (x, y, duration) -> (checkAOIs aois x y duration)) items))
    |> Seq.map (fun (id, metadata, items) -> (id, metadata, Seq.filter (
       Seq.isEmpty >> not) items))
    |> Seq.map (fun (id, metadata, items) -> (id, metadata, Seq.concat
       items))
    |> Seq.iter (fun (id, metadata, elem) -> saveAOIDNAs stimulus
       aoiCategory id metadata elem)
```

Listing 4.1: Calculating AOI-Hits for AOI-DNAs from a list of metadata (participants) and loaded fixations.

These projects were used and continued with every empirical source code comprehension study and for every analysis idea. Therefore, the necessary time and effort for implementing the analysis pipeline were reduced to a minimum.

4.8 SUMMARY

This chapter introduced and explained the newly developed methods and tools, as well as the already existing tools, both of which were successfully used throughout this thesis.

AOI-DNAs and AOI-STGs

The AOI-DNAs and AOI-STGs are meaningful visualizations to highlight eye movement patterns in the fairly noisy eye movement data of source code comprehension studies. The pattern match made it easier to search for specific eye movement patterns related to a source code comprehension context. In addition, the CodeSight analysis application combined the AOI-DNAs and the pattern matching to allow for faster analyses of eye movement patterns.

[13] `https://fsharp.org/` – Last accessed on December 14, 2020.

These developed methods and tools are supported by known methods and tools, such as the ANOVA test and the N-W and OM algorithms, as well as the TraMineR R package.

Current Development as First Step

All of these methods and tools are important for analyzing eye movement patterns. Nonetheless, the developed methods and tools are only the first step. Throughout this thesis, many ideas had to be put on hold for various reasons, most often because of time constraints. These ideas, such as fully dynamic and animated AOI-DNAs and AOI-STGs, a query language for eye movement data, or a pattern language for the pattern matching, could further improve the analysis process of eye movement patterns. However, this is material for future projects (see Section 8.6).

"If I have seen further it is by standing on the shoulders of Giants."

– Isaac Newton in 1675

5

Case Study 1 – The EMIP Dataset Analysis

Primary Contributions

The results regarding the possibilities of AOI sequence visualizations and analyses formed the foundation for the idea to transfer this analysis to a larger dataset. The primary goal of the EMIP analysis was to detect comprehension strategies and patterns, both closely linked to the comprehension problems of participants. Such findings would allow for the further planning of subsequent studies and analyses. In summary, the study was conducted to make three primary contributions to the body of work of this thesis:

1. Apply the visualization of AOI-DNAs for a quick visual inspection of frequent patterns, such as story order reading (SOR) and EOR, as well as an overall visual summary of eye movements.
2. Apply the visualization of AOI-STGs for a more detailed inspection of individual eye movement patterns, based on node clustering, transition frequencies, and transition sequences.
3. Quantitatively analyze the clustering of sequence alignments to find common viewing strategies among the participants.

These ideas and planned contributions were the basis for the EMIP dataset analysis. The following sections explain the study design and the results of the analyses in detail.

5.1 STUDY DESIGN

High Diversity in the Dataset

The recording of participants was not involved in this case study, because the EMIP dataset consists of already recorded participants in source code comprehension settings. The exceptional characteristic of the EMIP dataset is that different universities all around the world recorded the data with the same hardware setup. This procedure should minimize differences in the experimental setup and design. These various involved participants from around the world make this dataset particularly interesting because they increase the diversity of, for example, prior knowledge and educational backgrounds.

5.1.1 RESEARCH QUESTION AND HYPOTHESES

Analyses with AOI-DNA and AOI-STG

Finding similarities or dissimilarities in gaze data between novices and experts, for example through clustering, repetitive patterns (Bednarik, 2012), and changed fixation durations (Busjahn et al., 2011), has been a primary focus of previous research (see also Section 2.4.4). This analysis uses two methods of quantitative analysis through 1) visualizations with AOI-DNAs and AOI-STGs and 2) a qualitative approach in a large dataset via sample checking. The assumptions are that the first signs of comprehension patterns regarding comprehension problems are visible and that visualizing and analyzing comprehension processes are possible with the mentioned visualization methods. Previous research has demonstrated that such patterns must be present (Busjahn, Bednarik, et al., 2015).

MEASURING SOURCE CODE COMPREHENSION (DEPENDENT VARIABLES)

Dependent Variables: Task Answer, Task Duration, and Visual Focus

As dependent variables, various sources to measure the outcome or effort of the source code comprehension tasks were used. The following variables specifically were used to measure the quality of the source code comprehension per participant:

Task answer. Participants' performance of source code comprehension task was determined by measuring the answers given for each source code example.

Task duration. The assumption was that structure-related comprehension questions cannot be answered without comprehending the source code first, even if a participant recognizes the code (partially). Thus, the time duration needed to answer a question reveals, to some extent, the effort required by a participant for the comprehension.

Visual focus. Eye tracking data was used to track the process of reading the source code. In combination with AOI models, the reading sequences of participants can be created, while they comprehend a code example. This process allows for not only the recording of the order and frequency of source code parts that a participant is looking at, but also the measurement of the duration of fixations.

EXPERIMENTAL VARIATION (INDEPENDENT VARIABLES)

As the experimental rationale, source code examples with fixed order were selected for the participants. The following variable was used to create the study conditions.

Independent Variables: Source Code Examples

Source code example. The study was built to compare two different code examples, namely *Rectangle* and *Vehicle*, with different levels of complexity and different comprehension questions. These source code examples where randomized to avoid carryover effects.

HYPOTHESES

Finally, the following four hypotheses were tested.

$H_{Structured-Reading}$ – For this hypothesis, an investigation was undertaken to determine whether the visualization of AOI-DNAs is useful to qualitatively analyze the reading patterns of participants. This analysis was based on the SOR and EOR eye movement patterns. The assumption was that with distinctive color coding, both patterns are visualizable and hence distinguishable. The results can be found in Section 5.2.1.

$H_{Early-Fixation-Hits}$ – For this hypothesis, an analysis was conducted to determine whether the visualization of AOI-DNAs is capable of visualizing eye movement data for participants such that a faster focus on important AOIs is emphasized. For this analysis, "important" was defined as AOIs representing the main method, calculations, or overall domain implementations, and "faster" was defined with an earlier hit on the specified AOI, indicated with an earlier color visualization of the respective AOI-Hit in the AOI-DNA. The assumption was that a specialized color scheme will highlight earlier fixation in the AOI-DNA visualizations. The general idea was that a faster focus on important AOIs signals deeper general knowledge about how the source code is organized, which is an important aspect when analyzing eye movement data. Please refer to Section 5.2.2 for the results.

$H_{Clustering-Reading-Patterns}$ – The question related to this hypothesis asked whether participants, grouped into the same clusters based on their eye movement data, share common attributes such as answer quality (correct or incorrect answers) and eye movement patterns. The clusters were created based on sequence analysis methods and visualized with dendrograms and TraMineR. The assumption was that at least some clusters contain related participants based on their eye movement patterns, who will share other attributes such as answer quality. The results are described in Section 5.2.3.

$H_{AOI-STG-Patterns-Edges}$ – For this hypothesis, an analysis was performed to determine whether AOI-STGs can indicate eye movement patterns through graph visualizations, which are based on node clustering, transition frequencies, and transition sequences. The assumption was that AOI-STGs provide a fast and reliable overview of AOI transitions and hence eye movement patterns via features of the visualization (e.g., layout and so-called fountains). Moreover, the question was, can AOI-STGs indicate important transitions via single or clustered edges, which are based on the layout algorithm of AOI-STGs? The assumption was that AOI-STGs allow for the spotting of frequently occurring transitions via visual clusters (edges to or from nodes). The results pertaining to both questions can be found in Section 5.2.4.

5.1.2 METHODS

Bottom-Up and Top-Down Approaches

In the analysis of the EMIP dataset, AOI-DNAs and AOI-STGs were applied in both a bottom-up and top-down approach. Predefined patterns, such as SOR and EOR, were used and visualized within the AOI-DNAs. In contrast, the AOI-STGs were used for the bottom-up approach. They visualize which clusters and patterns are visible in the graphs, without predefining them.

Furthermore, sequence alignment was used in the form of the N-W and OM algorithms. They were utilized in combination with different visualizations to analyze eye movement data as sequences, cluster them, and find eye movement patterns in the process.

5.1.3 EXPERIMENTAL DESIGN

Comprehension Questions Targeting the Implementation

The experimental design was predefined due to the fixed EMIP dataset and the already collected data. The source code examples (stimulus material) were alternated between the participants to avoid carryover effects: 1) Rectangle first, Vehicle second; 2) Vehicle first, Rectangle second; 3) Rectangle first, Vehicle second, and so on. The comprehension questions targeted the source code implementation or specification; they were not designed as behavioral questions. Furthermore, the EMIP study was structured as a *randomized within-subjects* design. Thus, fewer participants were required to target all conditions, and the random noise was reduced. Figure 5.1 visualizes the experimental design of the EMIP dataset.

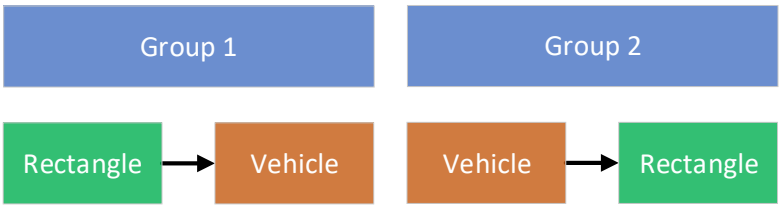


Figure 5.1: The visualized EMIP experimental design with the two groups and the alternating stimuli order.

5.1.4 PREREQUISITES AND ENVIRONMENT

I-VT Filter

The analysis had little prerequisites or demands regarding the environment. The EMIP dataset in a readable format was necessary, and the implementation of the *I-VT* filter to calculate fixation points rather than using the raw data. Which data was captured for the dataset and in which process, is described in the next section.

5.1.5 CAPTURED DATA

SMI RED250 Eye Tracker

The data of the EMIP dataset was collected in different participating labs across the world. All data was recorded with an *SMI RED250* mobile eye tracking system, and the screen resolution was 1920×1080 px for every recording. The dataset consisted of the stimulus material (source code examples, see the next section); the *emip_metadata.csv* file with the participants' background information (see the next section), in which the order of the source code examples was shown; and data about the comprehension questions. Moreover, the dataset contained *tsv* files for every participant with the raw eye movement data, along with metadata information about, for example, the IDF converter used, the number of samples, and the coordinates of calibration points. Tabs separated the data columns.

Demographic Data

The metadata for participants contains, among other things, the age, gender, mother tongue, English level, overall programming expertise, expertise in Java, and information about how long each participant is programming in different programming languages. No interview, think-aloud, or other method was used during data collection for the EMIP dataset.

5.1.6 PARTICIPANTS, MATERIAL, AND PROCEDURE

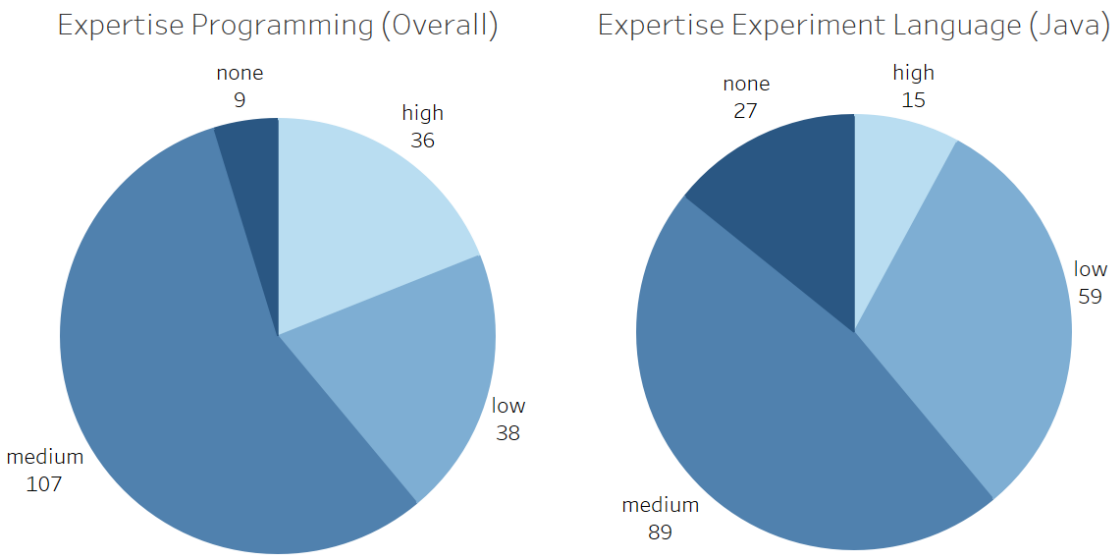
$n = 190$

The EMIP dataset consisted of 269 participants, 15 of which had to be excluded by the curating members of the EMIP workshop due to quality concerns about the recorded data. Furthermore, 15 % of the remaining 254 participants were not published. Of the remaining 216 participants, two hundred and seven (207) participants worked with Java as the language for the source code examples, five with Python, and four with Scala. Due to the small data samples, both the Python and Scala groups were excluded. Of the 207 participants working with Java, an additional 11 were excluded for poor overall data quality on both source code examples. Another six participants had a partially poor data quality on one of the source code examples, and they were consequently excluded from the analysis. The remaining 190 participants were the basis for the analyses of eye movement data ($n = 190$), 36 of whom were female and 154 were male, with a mean age of 26.52 ($SD = 9.07$).

Expertise in Java and Programming

All participants filled in a questionnaire with questions pertaining to, inter alia, their overall *expertise in programming (EP)* and their expertise in the *experiment language (EEL)*, in both cases with the options *high*, *medium*, *low*, and *none*. Based on the question about expertise in programming, the distribution was 36 (high), 107 (medium), 38 (low), and 9 (none).

The distribution of expertise in the experiment language (Java) was 15 (high), 89 (medium), 59 (low), and 27 (none). These results are based on the 190 participants who worked with Java. Figure 5.2 illustrates these mentioned distributions, which formed the basis for novice and expert assessment and grouping used in the analysis.



(a) Distribution of answers to the question on expertise in programming (overall).

(b) Distribution of answers to the question on the expertise in the experiment language (Java).

Figure 5.2: Distribution of answers to the questions on expertise in programming (overall) and expertise in the experiment language (Java).

Listings 5.1 and 5.2 contain the source code examples used for recording the EMIP data.

The Rectangle Code Example

```
1  public class Rectangle {
2    private int x1 , y1 , x2 , y2 ;
3
4    public Rectangle ( int x1 , int y1 , int x2 , int y2 ) {
5      this.x1 = x1 ;
6      this.y1 = y1 ;
7      this.x2 = x2 ;
8      this.y2 = y2 ;
9    }
10
11   public int width ( ) { return this.x2 - this.x1 ; }
12
13   public int height ( ) { return this.y2 - this.y1 ; }
14
15   public double area ( ) { return this.width ( ) * this.height ( ) ; }
16
17   public static void main ( String [ ] args ) {
18     Rectangle rect1 = new Rectangle ( 0 , 0 , 10 , 10 ) ;
19     System.out.println ( rect1.area ( ) ) ;
20     Rectangle rect2 = new Rectangle ( 5 , 5 , 10 , 10 ) ;
```

```
    System.out.println ( rect2.area ( ) ) ;
  }
}
```

Listing 5.1: The Rectangle source code example of the EMIP study.

The questions for the Rectangle source code were as follows:

1. "The program computes the area of rectangles by multiplying their width (x1-x2) and height (y1-y2)."
2. "The program computes the area of rectangles by multiplying their width (x2-x1) and height (y2-y1)."
3. "The program computes the area of rectangles by multiplying their width (x1-y1) and height (x2-y2)."
4. "I'm not sure."

The Vehicle Code Example

```
public class Vehicle {
  String producer , type ;
  int topSpeed , currentSpeed ;

  public Vehicle ( String p , String t , int tp ) {
    this.producer = p ;
    this.type = t ;
    this.topSpeed = tp ;
    this.currentSpeed = 0 ;
  }

  public int accelerate ( int kmh ) {
    if ( ( this.currentSpeed + kmh ) > this.topSpeed ) {
      this.currentSpeed = this.topSpeed ;
    } else {
      this.currentSpeed = this.currentSpeed + kmh ;
    }
    return this.currentSpeed ;
  }

  public static void main ( String args [ ] ) {
    Vehicle v = new Vehicle ( "Audi" , "A6" , 200 ) ;
    v.accelerate ( 10 ) ;
  }
}
```

Listing 5.2: The Vehicle source code example for the EMIP study.

The questions for the Vehicle source code were as follows:

1. "The program defines a vehicle by a producer, that has a type and can reduce its speed."
2. "The program defines a vehicle by a producer, that has a type and can accelerate its speed."
3. "The program defines a vehicle by a producer, that has a type and can accelerate and reduce the speed."
4. "I'm not sure."

5.1.7 ANALYSIS DESIGN

Rectangle and Vehicle Source Codes

Every participant had to comprehend the two source code examples – Rectangle and Vehicle. Both stimuli have two different scaling factors; one of each is slightly larger, similar to an image that is zoomed in. Since the raw data is available and fixations can be calculated with respect to this scaling factor, the different versions of the stimuli are combined in the analysis. *rectangle_java.jpg* and *rectangle_java2.jpg* are hereafter referred to as "Rectangle," and *vehicle_java.jpg* and *vehicle_java2.jpg* as "Vehicle."

I-VT Filter Parameters

As data preparation, fixations based on the raw data were calculated for every participant and saved in csv files. These files are the basis for the AOI matching. The calculation was done using a *I-VT* filter with a maximum radius of *60 px*, a minimum fixation duration of *60 ms*, and a maximum of *55* missing gaze samples to count as a fixation.

EMIP AOI Model, Lines and Regions

Various methods exist for defining AOIs on stimuli, and they have a significant influence on fixation hits and the resulting AOI patterns (refer to Section 4.2). For this analysis, the predefined AOIs, distributed with the EMIP dataset, were used. This model is called *emip* (large margins), in contrast to self-defined AOIs, which are called *self* (no margins). The difference is that self-AOIs are defined without a margin between each AOI, whereas emip AOI definitions utilize a margin. This distinction was used for the analysis reported in (Deitelhoff et al., 2019a). The definitions, saved in csv files, consist of rectangles and information pertaining to whether the AOI definition refers to a line AOI or an element AOI – the latter describes an AOI around one specific token. *Line AOIs* were chosen because they were sufficient as a basis for the analysis, and to obtain different views of the data, additionally defined region AOIs were added. These additions are based on essential areas of the source code, such as class variables, the constructor, or methods with the domain logic. Afterward, a programming expert checked the AOIs for consistency regarding the stimuli and the Java programming language. In this case, a region is a logical combination of 1 to n different line AOIs, instead of adding a bounding rectangle for AOIs in one region.

Letters for AOI Labels

Both stimuli are marked with the AOIs $A \ldots E$, with the main method labeled as E and the calculations as D. AOI A contains the class declaration, B the class attributes, and C the

constructor. These labels are used in all further analyses to refer to the specific regions in the source code.

Possible Sampling Bias

According to the methodical eye tracking challenges reported in Section 4.1, the EMIP dataset was pre-recorded so that not all aspects are addressable later on. The involved eye tracker was the *SMI RED250* mobile eye tracker with a sampling rate of 250Hz, which is sufficient to record fixations and saccades, both of which are important in source code comprehension. As the filter, the default I-VT filter proposed by Tobii was used with the default values reported above. For the AOIs, the proposed method by the EMIP workshop was utilized, with AOIs defined around every source code line and with a maximum margin between the AOIs. Furthermore, the imbalance between male and female participants with a factor of approximately 4:1 seems to follow the typical distribution of student male and female gender in computer science and engineering faculties where the data was collected. Another sampling bias based on aggravating recording conditions, such as makeup and glasses, cannot be controlled for afterward. It has been reported that some participants need to be excluded after recording them, and for the analysis in this thesis, additional participants were excluded due to a low eye tracking data quality; however, the reasons can be diverse. In addition, a bias towards programming environments was avoided because the source code examples were presented as images without a surrounding or supporting environment. Finally, the comprehension questions target the specification of source code, which can be improved with behavioral questions, which are employed in the following studies.

5.2 STUDY RESULTS

The following subsections report the results of the EMIP dataset analysis. The sections are organized according to the hypotheses described in Section 5.1.1.

5.2.1 STRUCTURED READING ($H_{Structured-Reading}$)

SOR for Novices

Busjahn, Bednarik, et al. (2015) found evidence that novices follow the linear SOR model more often compared to experts, which leads to the assumption that experts can use their skills to incorporate the duality of the code, consisting of static text and dynamic behavior, into their reading. Therefore, a participant who follows the EOR tends to be more on the expert side of source code comprehension.

AOI-DNA Visualizations

Finding such patterns is thus a crucial part of analyzing eye movement data. The analysis of the data in this hypothesis is based on a visualization of AOI-DNAs. Every calculated AOI-Hit, resulting in a unique character, was encoded with a different grayscale value, which is helpful to detect the SOR pattern because of the color-coding results in a color gradient from

light gray to black. Figure 5.3 illustrates this principle with an example (Rectangle source code example).

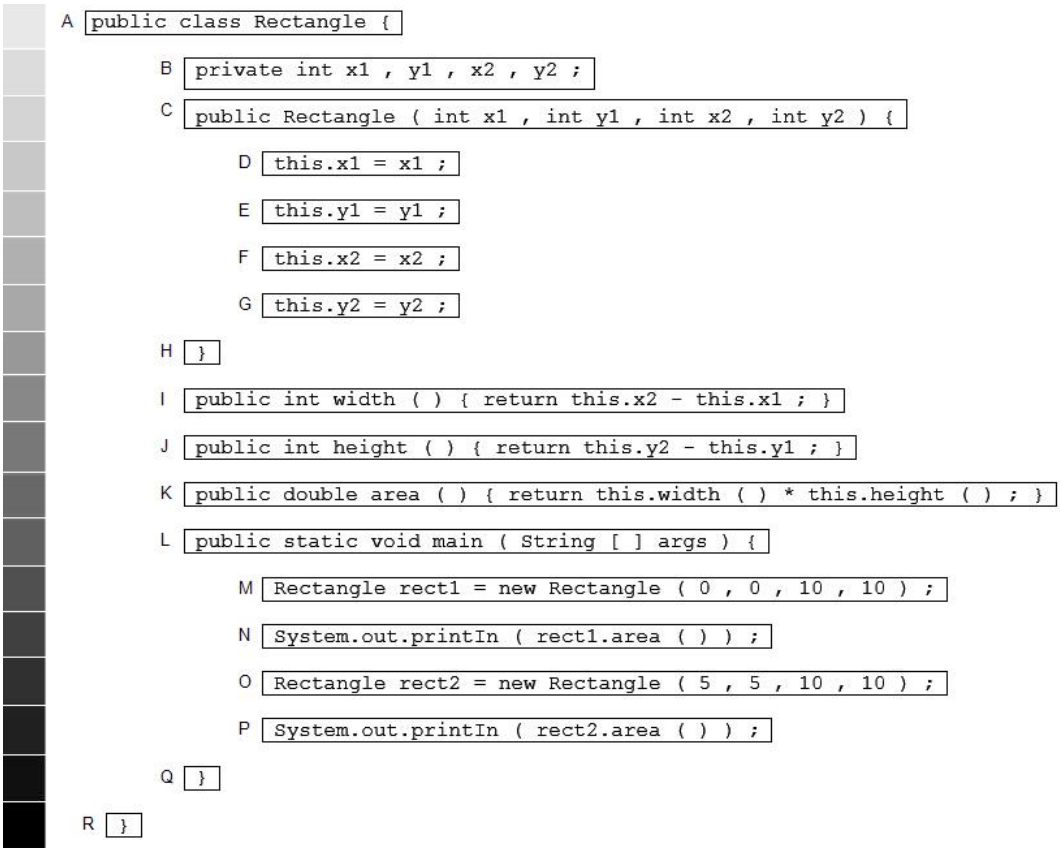


Figure 5.3: A grayscale visualization for the defined AOIs (emip).

In this way, a participant with SOR can be spotted instantly, even when numerous participants are visualized vertically (see Figure 5.4). These visualizations work in the same way for AOIs defined as lines and regions; even multiple SOR sequences can be spotted easily. Detecting SOR in the visualized AOI-DNAs works best for AOIs defined for lines. In contrast, there are only six different AOIs for the region AOI model on both the rectangle and vehicle stimuli. Therefore, spotting the reading behavior is somewhat more complicated; however, it is also possible.

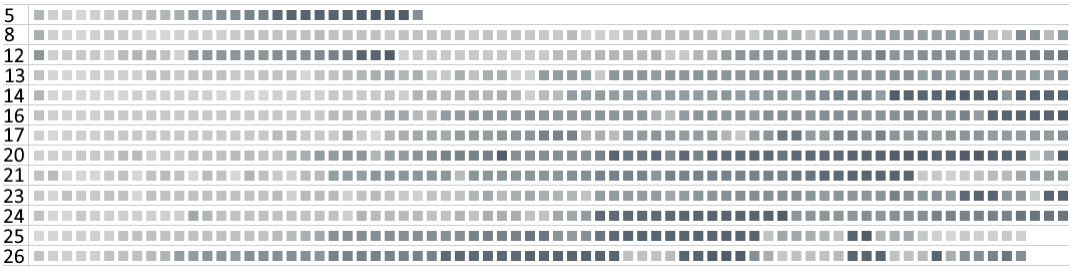


Figure 5.4: A grayscale color code to visualize the SOR pattern (participant IDs on the left).

EOR Detection Based on Regions

Detecting EOR works best for AOIs defined as regions, which reduces the number of AOI-Hits and hence different grayscale colors. Spotting the EOR patterns works even better if only the necessary AOI-Hits remain in the color code and if the color code itself is adjusted. Figure 5.5 illustrates this principle on some participants from the Rectangle stimuli with AOI regions.

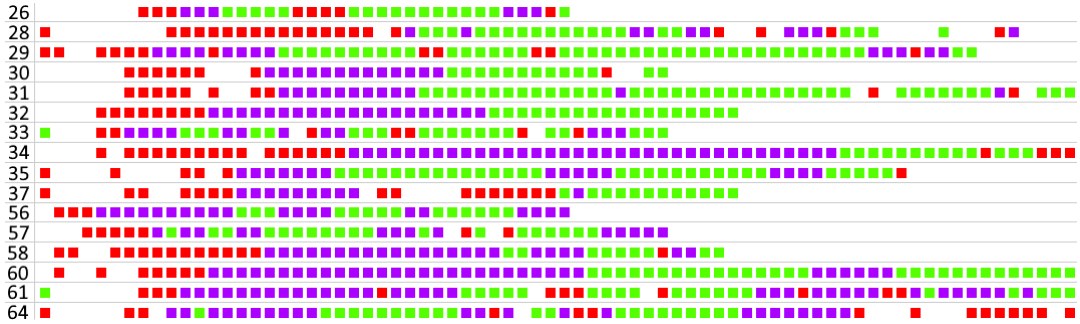


Figure 5.5: Visualization of AOI-DNAs with a red, purple, and green color code to show the EOR pattern (participant IDs on the left).

Color-Coding for EOR

The definition of an ideal pattern for EOR is $E \rightarrow C \rightarrow E \rightarrow D$. This color code results in green $\rightarrow$ red $\rightarrow$ green $\rightarrow$ purple. Participant 26, in the first row in Figure 5.5, displays the ideal EOR pattern. Participants 29 and 64 also have an EOR pattern in their gaze, but less ideally. This suggests that the AOI-DNA visualization is a useful way in which to obtain a fast first impression on the SOR and EOR patterns.

AOI-DNA can Visualize SOR and EOR

This hypothesis is thus confirmed: Visualizations based on color codes for the AOI-DNAs can reveal the SOR and EOR patterns within eye movement data, and spotting the results is easy, even for multiple vertically visualized AOI-DNAs at once.

5.2.2 EARLY FIXATION HITS ($H_{Early-Fixation-Hits}$)

Fixation Count Metric

The idea behind using the metric *fixation count* (see Section 2.4.2 for the eye movement metric) is to measure which AOI had more fixation hits over a defined period of time. This result can be used to analyze the number of fixations a participant needed to reach an important AOI, which is defined as an AOI that is crucial for the source code comprehension process (i.e., the main method, calculations).

Region Model

Both source code examples have a *main* method and one or more methods with calculations. Since the question of whether early fixations are present is more interesting, the AOI region model was chosen for this analysis. For every stimulus, there are the five AOIs: *A*, *B*, *C*, *D*, and *E*. The main method in the source code is *E*, and the calculations are in *D*. The assumption was that a participant will check at least these AOIs to comprehend the source code accordingly; however, this is not necessarily true because the questions are related to the specification of the source code (static source code comprehension questions).

Specialized Color Coding

The main method in the code is marked with *green* and the calculation method(s) with *purple*, to visualize the assumption about the source code comprehension. All other AOIs are removed from the visualization by marking them with the color *white*. Thus, the visualization shows clearly, which participant is starting where and if there are white gaps in the beginning. These gaps mean that the participant is not starting with the main method or with the calculation(s).

Few Results

The result for the Rectangle stimulus with regions is visible in Figure 5.6. It is observable that some participants started with the main method (*green*) and some with the calculation method(s) (*purple*). Others, such as Participant 183, began with the calculations and reading them, switching to the main method and back to the calculations. This behavior is noteworthy yet rare. At this point, most of the visualized participants had just one or two fixations of the relevant AOIs in the beginning. Even if attention is stretched to the first 10 fixations, the number of participants only increases slightly. One or two fixations, at the beginning, are no sign of intentional viewing and, above all, no sign of optimal source code comprehension. The data points are explainable by the starting point of a participant's view when the stimulus is visible on the screen for the first time.

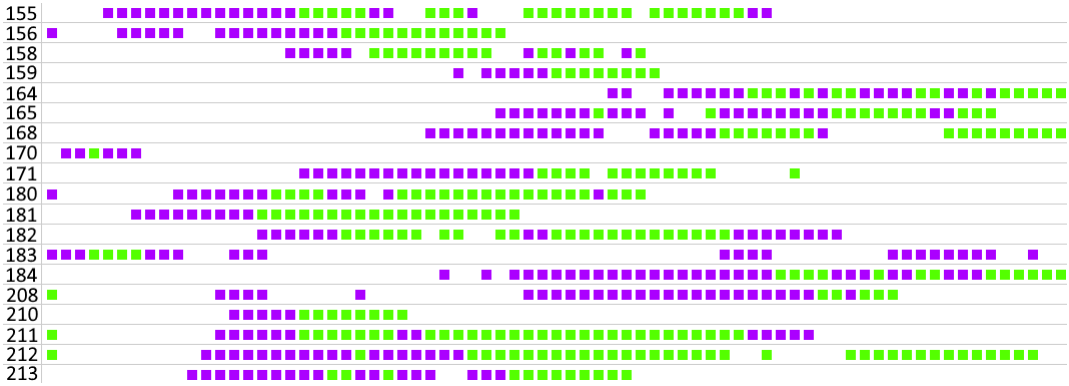


Figure 5.6: AOI-DNAs (excerpt) with green and purple to visualize early fixations of the main method and calculation method(s) respectively. Every other AOI is marked in white to create gaps (participant IDs on the left).

AOI-DNAs for Pattern Visualization

The summary regarding this hypothesis is that visualized AOI-DNAs, with the correct color code, can highlight essential viewing and reading strategies of the eye movement data. Early fixation hits on the main method (green) and calculations methods (purple) can be spotted fast and reliably if the remaining letters in the AOI-DNA are rendered blank. Therefore, AOI-DNAs are useful for visualizing eye movement patterns in combination with color schemes, which confirms the initial hypothesis.

5.2.3 CLUSTERING READING PATTERNS ($H_{Clustering-Reading-Patterns}$)

Clustering Based on AOI-DNAs

The primary goal of this hypothesis is to find new clusters via clustering methods. These clusters should provide quantitative insight into common attributes of the participants

within the distinct clusters. The data basis for this hypothesis is $n = 207$, because the analysis followed an explorative approach over all participants with Java stimuli, to examine, if the clustering methods can find clusters different from visualizing AOI-DNAs or AOI-STGs, even for not optimal eye tracking data. Besides, both AOI models *emip* and *self* (see Section 5.1.7), were used in this analysis, which is why results from the *self* AOI model are reported in this section. The data basis for the clusters are the non-aggregated AOI-Hits per participant, which are calculated and saved as, for instance, the following example string:

```
BCCEDFJJJKKJJLKLKLMMMNNNMOOPOOONMMMN
```

Sequence Alignment

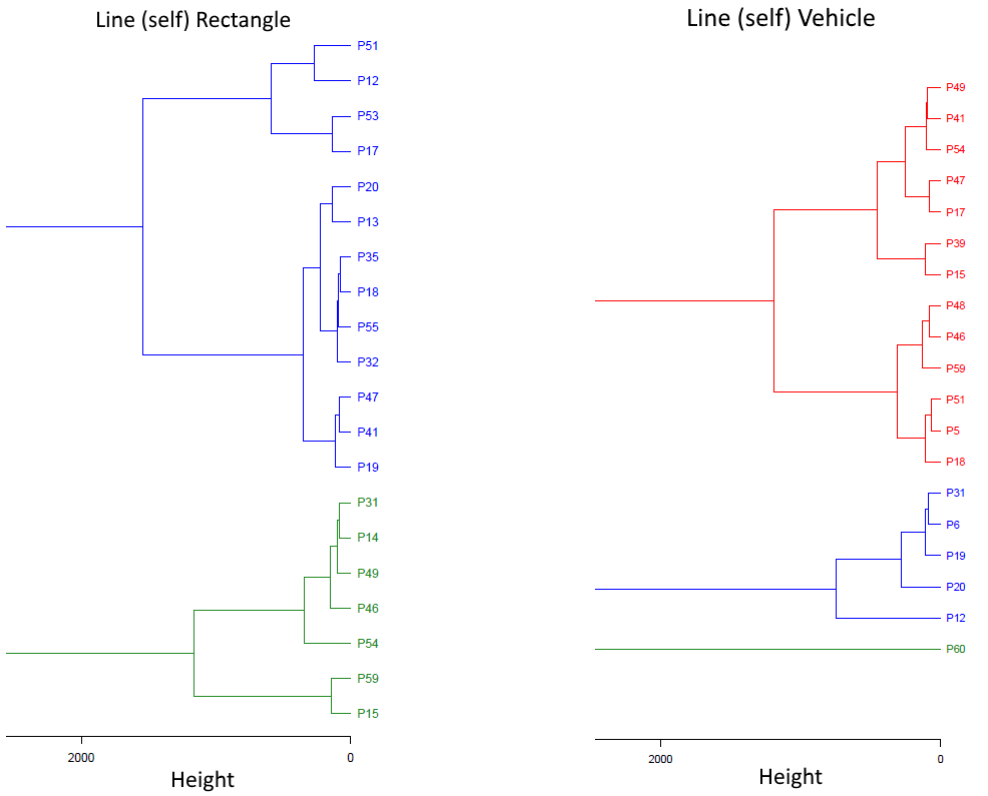
Various methods exist for calculating the distances between different strings. For this hypothesis, the two sequence alignment methods N-W and OM were tested. The data files with the eye movement data sequences were loaded to use the N-W algorithm, and every sequence was aligned with every other sequence per file to avoid mixing sequences generated based on entirely different source code examples. Moreover, per file, every participant was aligned to every other participant, and the calculated similarity results were saved in a matrix structure.

Hierarchical Clustering

To find similar participants, based on the similarities mentioned above saved in matrices per source code example, the *Euclidean distances* of these resulting matrices were calculated for use as input for a *hierarchical clustering* to find similar groups. *Dendrograms* were used to visualize the resulting clusters, which utilize *Ward's minimum variance* methods to find compact, spherical clusters. This is implemented by the *ward.D2* method in R (Murtagh & Legendre, 2014). Beforehand, the *elbow*, *silhouette*, and *gap statistic* methods were used to determine the optimal number of clusters. Both the first and second methods determined four clusters, whereas the third method determined eight clusters. The hierarchical clustering ward.D2 was compared to the *partitioning around medoids with an estimation of the number of clusters (PAMK)* (Frey & Dueck, 2007). This method creates six medoids and hence clusters. Since the initial tests for an optimal cluster count detected four clusters, the PAMK method six clusters, and the other methods eight clusters, six clusters were chosen. After creating and analyzing the first dendrograms, this number of clusters seemed to be a prime choice for the moment.

Dendrograms

The complete dendrograms are too large to display fully because of the large number of participants in the EMIP dataset. Figure 5.7 thus presents the cropped dendrograms for the Rectangle and Vehicle source code examples for the line self-AOI model. Figure 5.7a illustrates the rectangle dendrogram and Figure 5.7b the dendrogram for the Vehicle source code. Participants with similar eye movement patterns are clustered close to one another. Similarly clustered participants were searched on both Rectangle and Vehicle dendrograms to support this proposition, which increases the chance of finding participants with a better similarity based on the sequence analysis of the eye movement patterns.



(a) Excerpt of the dendrogram for the Rectangle source code example (line self AOI model).

(b) Excerpt of the dendrogram for the Vehicle source code example (line self AOI model).

Figure 5.7: Excerpts for the dendrograms for the Rectangle (left) and Vehicle (right) source code examples, both for the line self AOI model.

Manual Comparisons

Matches of groups of two participant between the two source code examples – Rectangle and Vehicle – can be found manually, which was done as a test by two researchers separately. This test revealed that not only is it possible to find matching participants, but also that this process is slow and error-prone. Matching participant groups were found by comparing the dendrograms automatically, resulting in correlations for all compared dendrograms and specific visualizations, showing the matching participant IDs with lines.

Automatic Comparisons

Figure 5.8 as an example, displays images for some of the dendrogram comparisons. The overview presents the results for the Rectangle and Vehicle source code examples, with the line and region model, for 63 participants. The other 144 participants produced dense dendrogram comparison images, where not much is visible. Furthermore, the vaguely identifiable colors show matching (color) and non-matching (black) groups of participants, and this highlighting will be useful in the later analysis. The straight lines in the diagonal images indicate the self-comparisons, which are equal, and the numbers in Figure 5.8 indicate the source code examples and AOI models. The names for these indices are 1) Rectangle line, 2) Vehicle line, 3) Rectangle region, and 4) Vehicle region. They were omitted to render the figure more readable.

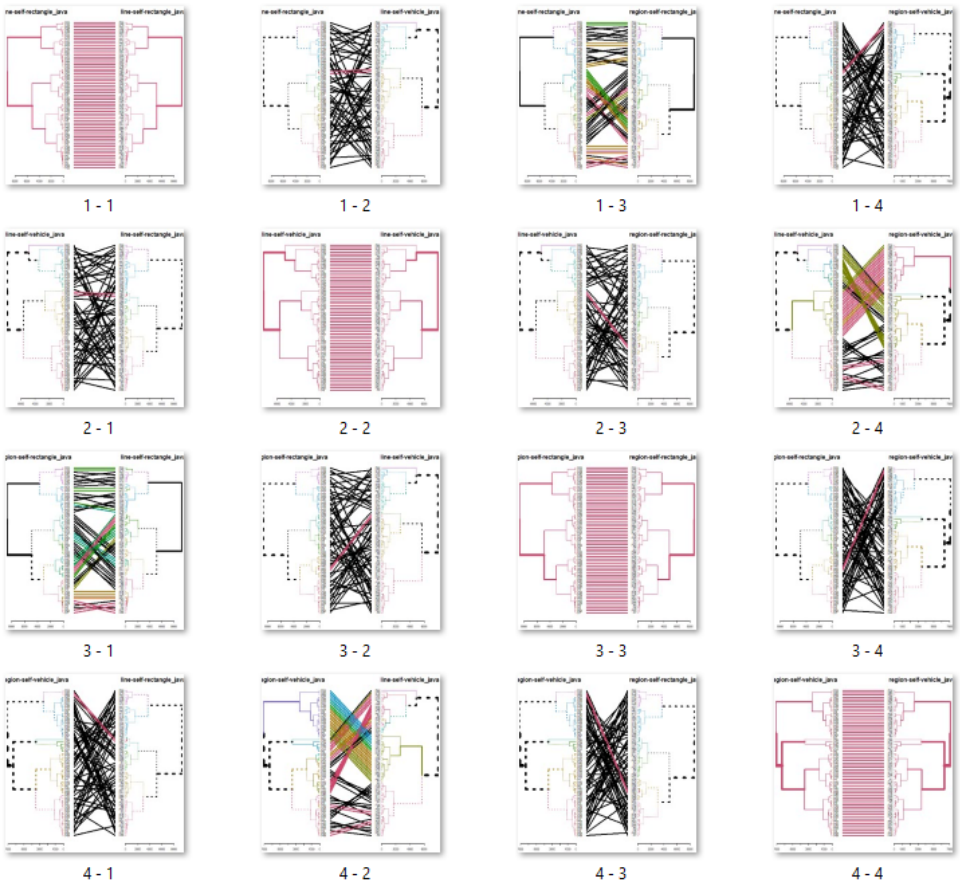


Figure 5.8: Dendrogram comparison results of the Rectangle and Vehicle source code examples (line and region AOI models) for 63 participants.

Interesting Correlation Results

The visualized correlations between every dendrogram comparison confirm the first assumptions but also shows some interesting facts. Figure 5.9 visualizes these correlations, where the diagonal shows the self-comparisons. As mentioned in section 4.2 on page 66, the region AOI model is simply an abstraction of the line AOI model, because no bounding rectangle was used to encapsulate the source code lines. The calculation is based on or-combinations between all line AOIs combined into a region. Thus, the region AOIs should be another view on the line AOI data.

Comparing Participant Groups

The Rectangle line and region comparison seem to show this. However, the Vehicle line and Rectangle comparison exhibit lower results, which is interesting but explainable with altered sequences through both AOI models (line and region). These models are similar because the region model is an abstraction of the line model. However, the sequences are altered in terms of AOI-Hits, which can alter the similarity measures. The participant groups, which need to be compared to find matching pairs between the source code examples, were selected from the Rectangle ↔ Vehicle line dendrograms. The region AOI model for this comparison indicates a higher correlation, which is explainable due to the coarser AOI model with fewer AOIs. The AOI line model thus seems to be a better choice, which results in using the dendrogram "1 – 2" (line Rectangle – line Vehicle) from Figure 5.8. In addition, the den-

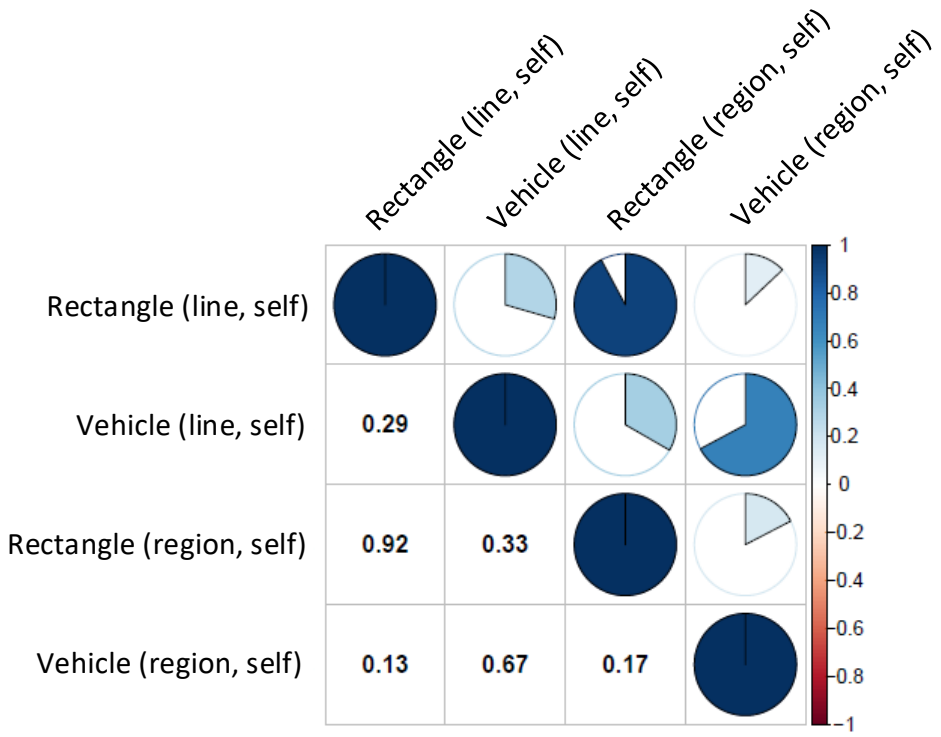


Figure 5.9: Dendrogram comparison correlations for 63 participants for the Rectangle and Vehicle source code examples (line and rectangle AOI models).

drogram "3 – 4" (region Rectangle – region Vehicle) was chosen to analyze the participant groups from the region AOI model. The same matching participant group was highlighted.

Parallel Lines for Matching Groups

These individual dendrograms were then used to identify matching participant groups. This matching was already done via the dendrogram comparison. As an example, Figure 5.10 displays the comparisons between the line Rectangle and line Vehicle dendrograms ("1 – 2"). The parallel lines indicate the moving positions of one participant from one dendrogram to another. To find a group of two participants who change positions between the dendrograms but, in the best case, not their immediate neighborhood, parallel lines need to be found. The selected visualization type for the dendrogram comparisons (`common_subtrees_color_branches = TRUE`) already highlights the most parallel lines and hence the most stable participant group. The best match is the group with Participants 63 and 52 (other matches are not highlighted). This participant group must be compared to determine whether common eye movement patterns can be found.

Few AOI-Hits

A detailed look at the data revealed that these participants were two examples of participants with a meager amount of eye tracking data present. Thus, insufficient eye tracking data could be matched against the available AOIs, resulting in limited eye movement data, and it is therefore possible to visualize the data here wholly. For the Rectangle source code example this meager amount of eye tracking data were as follows:

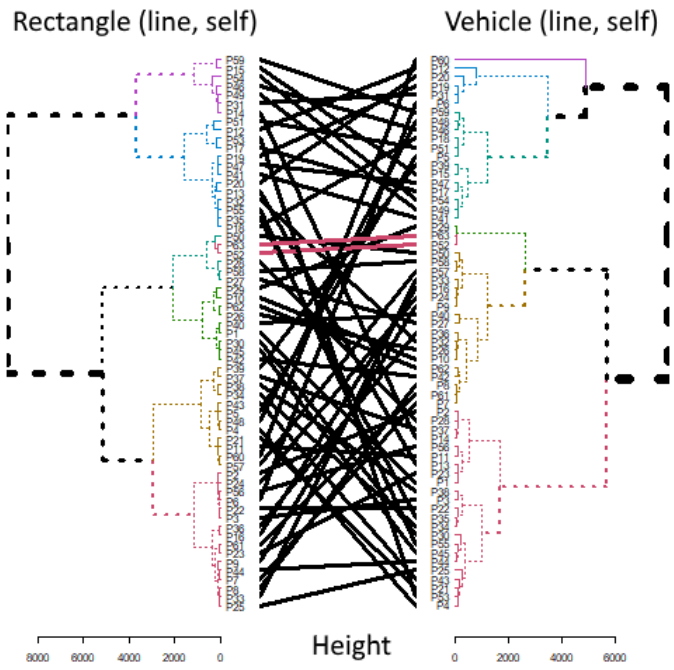


Figure 5.10: Comparison of two dendrograms for the line Rectangle and Vehicle source codes ("1 – 2"). An ideal matching participant group was automatically highlighted by the comparison algorithm.

52: DDEDAG

63: KC

For the Vehicle source code example the eye tracking data were as follows:

52: JGCAE

63: LFDBA

It is explainable that these eye movement data sequences were matched closer together with a sequence analysis than the other participants with much larger eye movement data sequences. Both Participant 52 and Participant 63 were also highlighted in every other dendrogram comparison.

Figure 5.11 presents another comparison of two dendrograms, this time for the line Rectangle and Vehicle source codes ("1 – 2"), but for a larger participant group, based on the "java2" stimulus of the EMIP dataset. The figure displays an excerpt because the data basis is too large to show the complete dendrograms.

Dendrogram Comparisons is Possible but Error-Prone

The highlighted lines detected the pair with Participants 8 and 151 (participant IDs). In the figure, other numbers are highlighted because the dendrograms show the IDs in ascending order and not the original participant IDs. They were matched for this analysis. The highlighted participants have similar and comparable eye movement patterns, as Figure 5.12 illustrates. The data also revealed that although both participants are in different skill groups, they answered the question for the rectangle source code correctly and the question for the vehicle source code incorrectly. Therefore, comparing eye movement data

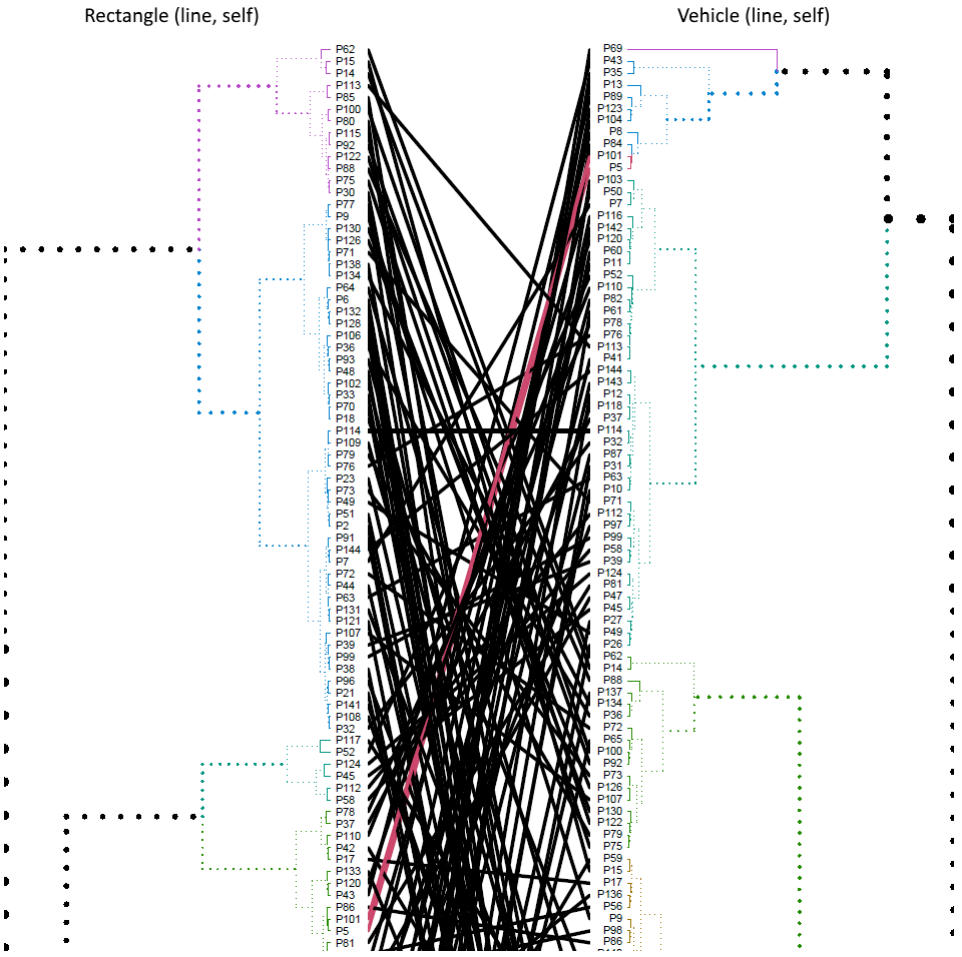


Figure 5.11: Excerpt of the comparison of two dendrograms for the line Rectangle and Vehicle source codes ("1 – 2") for a larger participant group. An optimal matching participant group was automatically highlighted by the comparison algorithm.

with dendrograms is possible, and similarities based on the eye movement patterns can be found. However, the process is tedious and error-prone.

Strength and Weaknesses

The analysis based on dendrograms and the comparisons between them revealed weaknesses and disadvantages of such analysis techniques: a) Smaller clusters, such as Participant 60 in the Vehicle (line) source code example, are less apparent, and b) clustering based on dendrograms concealed important attributes of eye movement data, such as the length of the sequences and the detailed structures. These had to be looked up after the clusters were created. An analysis method is thus needed for clustering the sequences based on sequence analysis, with the ability to display important aspects of the sequences within the clusters at once.

TraMineR and Optimal Matching

For this reason, an analysis and visualization with the TraMineR R package was conducted to investigate the sequence alignments with the OM algorithm. The data basis remains the same as for the analysis with the N-W algorithm: the calculated AOI-Hits based on the eye tracking data. This analysis is based on the OM algorithm, with a clustering based on the Ward algorithm, and three clusters. The dendrogram visualizations indicate that six clus-

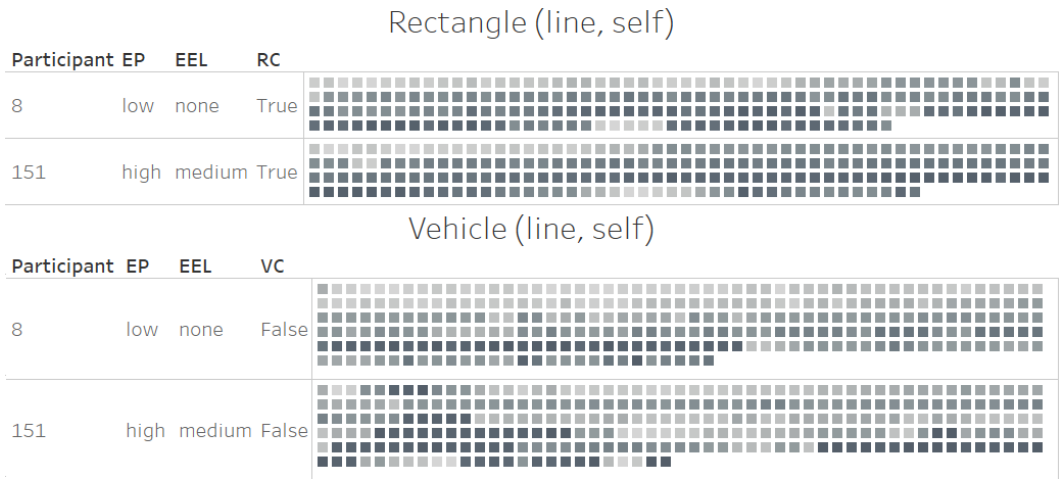


Figure 5.12: Eye movement data from Participants 8 and 151 for the Rectangle and Vehicle source code examples (java2, line, self).

ters are too much. This result was confirmed by visualizing six clusters with the TraMineR package because three clusters consist of only one eye movement sequence each.

Figures 5.13 (page 101), 5.14 (page 102), and 5.15 (page 103) visualize these three clusters as an example of the Rectangle source code stimulus. The grayscale color scheme was chosen deliberately because it depicts existing SOR reading patterns clearly.

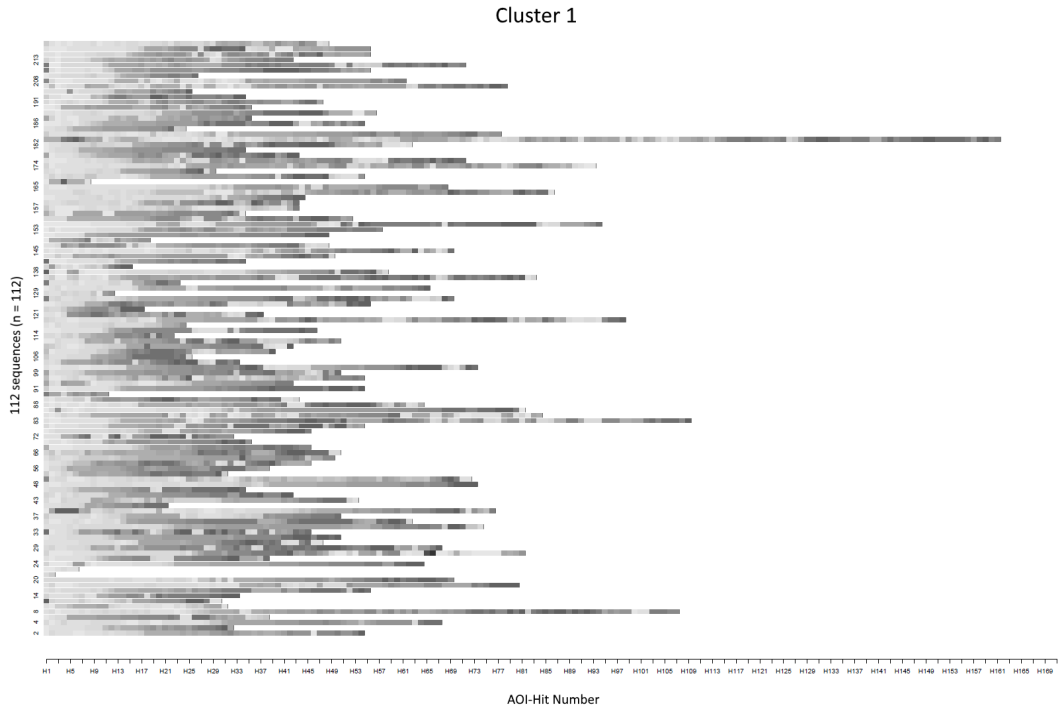


Figure 5.13: The first cluster with 112 eye movement sequences (Rectangle).

The first cluster contains 112 different eye movement sequences of varying lengths, and it is the largest cluster. Many of the sequences start with an upper AOI in the top of the source code examples; however, after a few AOI-Hits, many sequences bounce between different

First Cluster: 112 Sequences

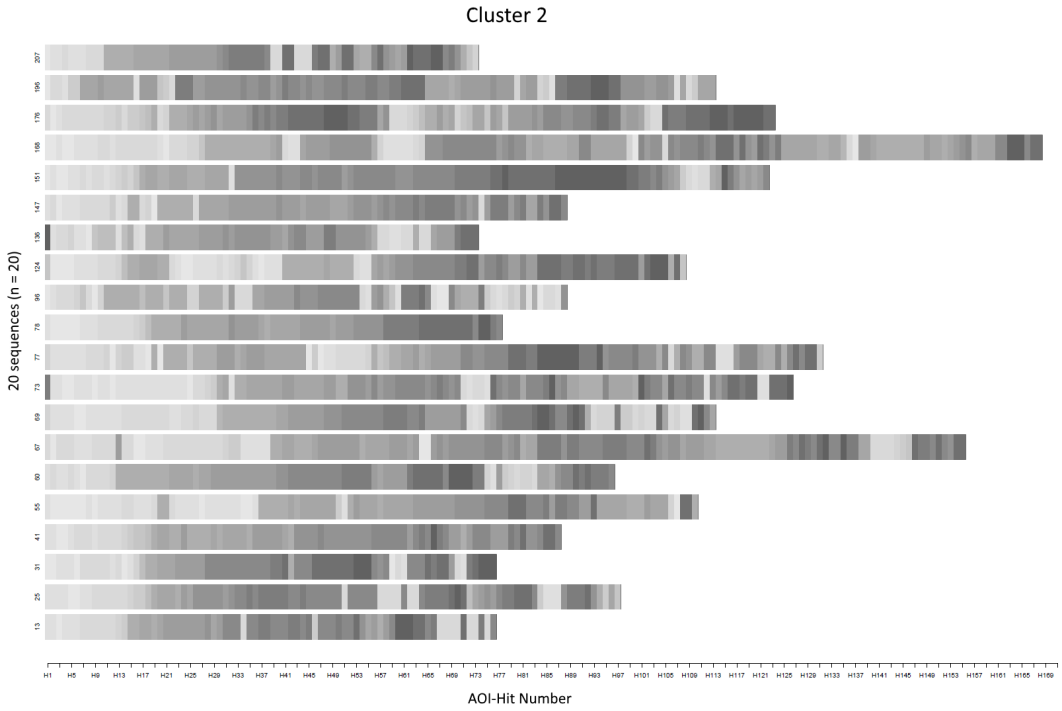


Figure 5.14: The second cluster with 20 eye movement sequences (Rectangle).

AOIs in the upper and lower segments of the source code examples. These variances and differences are the common attributes of eye movement sequences in the first cluster.

Second Cluster: 20 Sequences

The second cluster contains 20 eye movement sequences with more stable sequence lengths compared to the first cluster. This second cluster portrays appealing common attributes between the sequences. Except for 3 of the 20 eye movement sequences, the rest start in the upper region of the source code examples, followed by a reasonably straight SOR eye movement pattern. Based on the visualization, this second cluster is the most promising candidate to find participants with similar eye movement patterns. This finding will be analyzed further shortly.

Third Cluster: 12 Sequences

The third cluster contains 12 eye movement sequences, again with more stable sequence lengths than the first cluster. This third cluster is comparable to the second cluster. The eye movement patterns seem to be shared between the 12 clustered eye movement sequences. Most of them start in the top region of the source code example, with less straightforward SOR eye movement patterns. Overall, the second cluster seems to be a promising candidate, even after considering the third cluster.

Many Correct Answers in the Second Cluster

A consecutive analysis with another part of the dataset, this time based on the Rectangle and Vehicle source code examples and the self-AOI model, allowed no further in-depth analysis. The clusters provided no meaningful information. Analyzing the sequences from the bottom of the second cluster, it becomes clear that these eye movement sequences and participants share, to a large extent, a correct answer to the source code comprehension question: Participants 13, 25, 31, 41, 55, 60, and 67 have correct answers. Furthermore, the

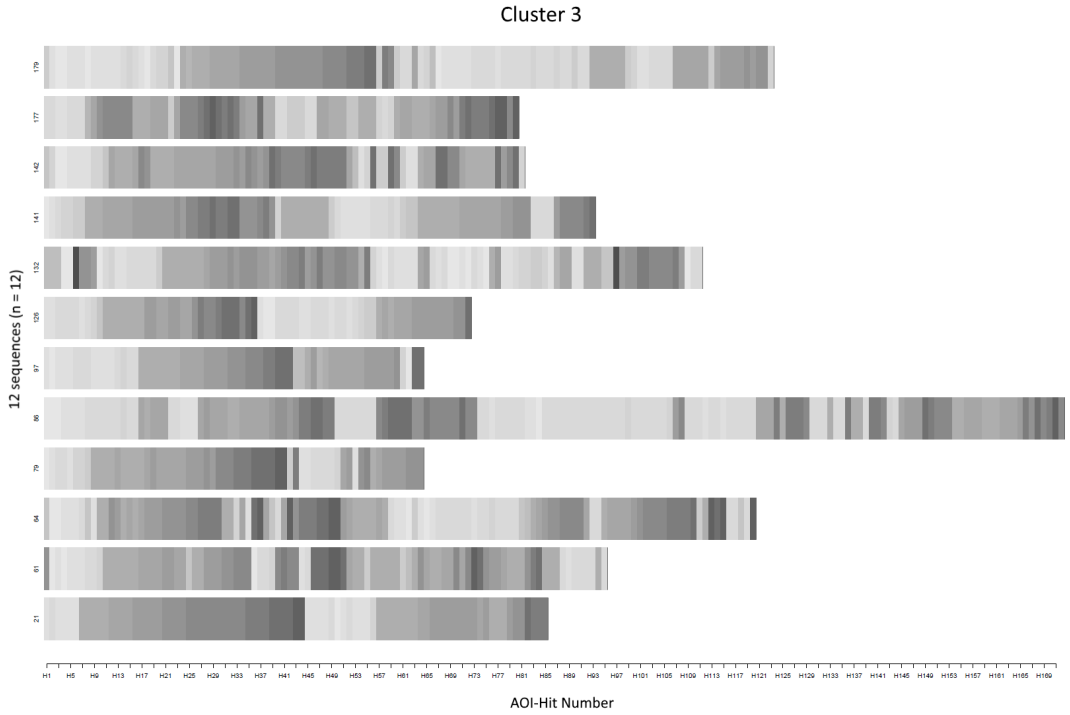


Figure 5.15: The third cluster with 12 eye movement sequences (Rectangle).

first one with a wrong answer is Participant 69. The overall distribution of correct answers, along with the average length and standard derivation for all clusters is provided in Table 5.1. The self-assessment of the participants in the second cluster revealed that these are not particularly unambiguous. In fact, programming experience and the experience with the experiment language (Java) are mixed. These participants are overall neither novices nor experts.

Table 5.1: Overview of the correct Rectangle answers (RC), the correct Vehicle answers (VC), and the average length and standard derivation for the sequences per cluster.

Cluster	# Sequences	RC	VC	AVG Length	SD Length
1	112	78	30	50.67	24.01
2	20	16	2	99.5	59.21
3	12	11	3	96.67	30.95

Clustering Reveals Shared Characteristics

Overall, this analysis revealed that clustering based on the sequence alignment and similarities is a useful method for finding and visualizing clusters. In addition, the TraMineR visualizations provide a better view on the overall sequences and clusters, where the dendrograms clearly indicate the matching participant groups. Both methods are thus useful. Moreover, the second cluster showed similar reading patterns (SOR) of all grouped participants, which is an additional benefit of clustering and this type of visualization. However, other characteristics of the clustered participants, such as answer quality, vary within the clusters.

Clustering for Similar Eye Movement Patterns

To summarize this analysis, clustering can show similar eye movement patterns in the eye movement data. The three clusters from the OM algorithm have proven to be useful for analyzing similar participants based on patterns, but not based on other characteristics such as answer quality. These results confirm the initial hypothesis.

5.2.4 PATTERNS IN AOI-STGS ($H_{AOI-STG-Patterns}$)

AOI-STGs for Transition Visualization

In addition to AOI-DNAs, AOI-STGs were used to visualize aspects of the eye movement data. First and foremost, they visualize transitions respecting the AOI model. In the AOI line model, more transitions between graph nodes are drawn compared to the AOI region model, and these transitions can be used to generate a different level of detail. Apart from this, eye movement patterns can indicate a certain behavior while reading and comprehending source code. These patterns can be described and categorized by coding schemes. A coding scheme highly depends on the underlying domain and data (see Section 2.4.3).

Figure 5.16 depicts the AOI-STG for Participant 11, generated from the AOI line model of the Rectangle stimulus. An AOI-STG was created for every participant of every stimuli, and specific participants, regarding the correct and incorrect answers for the stimuli, were selected to detect similar reading patterns via AOI-STGs within these groups.

Large Fountain

The loop on node C is highly salient. This phenomenon is known as a *fountain*, and it can be observed in most of the AOI-STGs. A fountain, generated by the AOI line model, indicates that a participant spends many fixations and therefore much time on the source code line, respectively on one region, if the underlying data is based on the AOI region model. Fountains can thus directly indicate where a participant spends most of their comprehension time. Linked with the AOI model data, one can easily spot fountains on important or less important AOIs. In addition, Figure 5.16 illustrates important clusters (e.g., between the AOIs D $\leftrightarrow$ F, I $\leftrightarrow$ J, and N $\leftrightarrow$ O $\leftrightarrow$ P). These clusters are not only visually important. The assumption was that a participant needs to comprehend AOIs D and F to know how the class attributes are set; J and K for the calculations in the Rectangle class; and N, O, and P for the main method. The assumption was that a participant with fewer connections within these clusters tends to offer fewer correct answers.

Figure 5.17 illustrates the AOI-STG for Participant 36, generated form the AOI line model of the Rectangle stimulus. This participant was not able to give the correct answer.

Fountains and Visual Clusters

Again, fountains can provide a useful first impression of where the participant spends many fixations. Compared to Figure 5.16, this AOI-STG reveals a more linear reading behavior. Participant 11 in Figure 5.16 had more regressions in total and more visual clusters. The visual cluster K $\leftrightarrow$ J for Participant 36, which is called a separated cluster, is important here. This cluster is connected to the rest of the graph by only a few transitions. This means that there is one transition to AOI K and then on K and between K $\leftrightarrow$ J, with a fast jump to AOI L. This is important because it describes a visual pattern: After the first linear

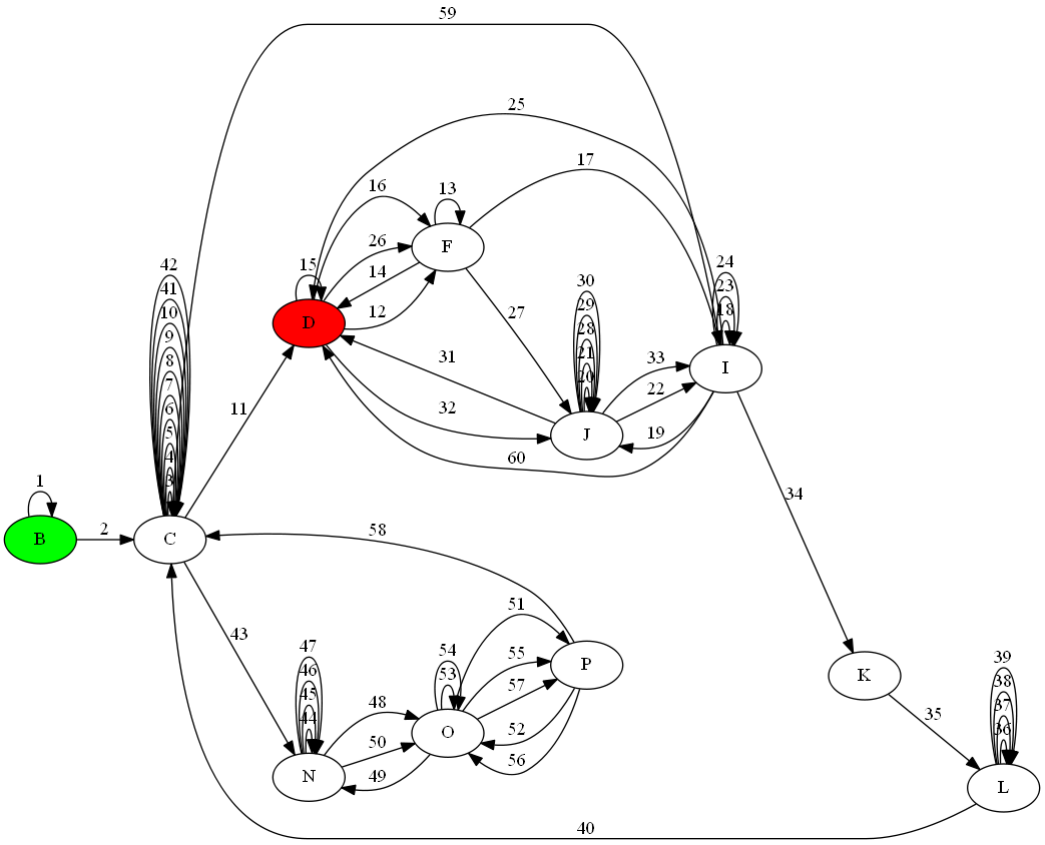


Figure 5.16: AOI-STG for Participant 11, with the AOI line model on the Rectangle stimulus (correct answer).

reading, the participant reads the important AOIs K and J only for a short period of time, without ever coming back to them. The analysis demonstrates that, first, this is not an ideal reading behavior for an optimal comprehension result. In addition, some evidence suggests that a mostly linear reading pattern with less regressions is also not an ideal reading behavior. The analysis of some randomly selected AOI-STGs substantiate these findings, which is noteworthy.

AOI-STGs for Node Clustering, Transition Frequencies, and Transition Sequences

The findings in the AOI-STGs clearly provide an answer for the hypothesis $H_{AOI-STG-Patterns}$. The hypothesis includes three assumptions on which data or analysis of an answer can be based. These are *node clustering*, *transition frequencies*, and *transition sequences*. Node clustering sheds light on how participants read through the text. A visual cluster in the AOI-STG, which reflects important AOIs, is a positive sign of an important reading behavior in this stimulus area. The other way around, the reading behavior tends to be inefficient if there are no clusters or just some with no connection to any important AOI cluster. Furthermore, transition frequencies provide insights into how strongly AOIs are connected to one another. Especially if those AOIs are not adjacent and not highly important for the source code comprehension, high transition frequencies will provide a first estimation of comprehension quality. With the index on every edge, assertions about the transition sequences can be made. However, it is important to know the order in which

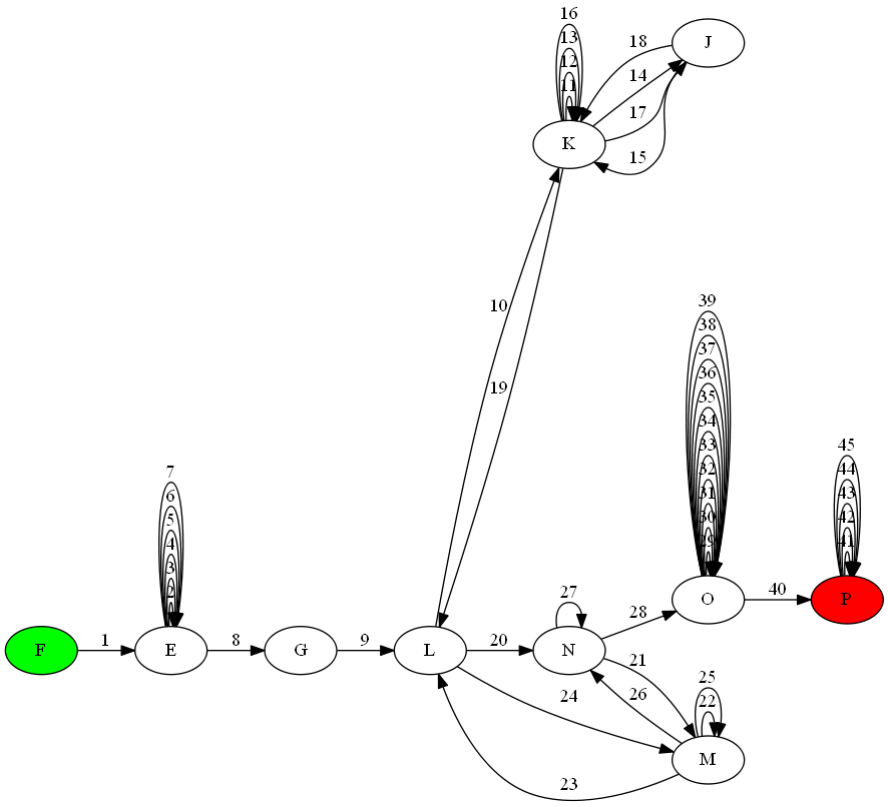


Figure 5.17: AOI-STG for Participant 36, with the AOI line model on the Rectangle stimulus (incorrect answer).

AOI-STGs were created; without the sequence order, assumptions about eye movement patterns cannot be proved later.

AOI-STGs for Pattern Visualizations

Regarding the coding scheme mentioned in (Busjahn, Schulte, & Kropp, 2014; Busjahn, Schulte, Sharif, et al., 2014), the *flicking*, *linear horizontal*, *linear vertical*, and *retrace declaration* patterns are observable. First, the flicking pattern is visualized in the AOI-STG with high transition frequencies between two AOIs. Second, an initial sign of the linear horizontal pattern is depicted by fountains for the line AOI model. The AOI-STG can be built on the token AOI model to gain a further understanding and better visualization of the pattern. Third, the linear vertical pattern is shown by a linear reading behavior, which results in nodes within the AOI-STG only connected with single transitions in one direction. Finally, the retrace declaration pattern can be found by analyzing AOIs with high transition frequencies. If these AOIs, based on the line AOI model, represent important AOIs (e.g., variable declarations), then this pattern can be analyzed visually.

AOI-STGs for Important Transitions

Given the way in which AOI-STGs are generated and visualized, important transitions attract almost instantly attention (e.g., as seen in Figure 5.17 between the AOIs L $\leftrightarrow$ K). These types of transitions indicate a separated cluster, which can then be analyzed further. Moreover, if AOIs are connected, this is visualized with clusters, which can then also be analyzed further.

To summarize the findings related to this hypothesis, eye movement patterns can be found through graph visualizations, based on characteristics such as node clustering, transition frequencies, and transition sequences. In addition, AOI-STGs indicate important transitions, especially between separated clusters because of the layout of nodes. These results confirm the initial hypothesis.

5.3 THREATS TO VALIDITY

When analyzing the EMIP dataset, several threats to the *internal validity* and *external validity* were noted.

Internal validity. The following limitations for causal conclusions based on the original EMIP study apply here. a) Both code examples use object-oriented programming (OOP), which has to be known by the participants. Even though this is taught in early courses in many universities and can thus be considered as a basic concept, many participants have problems with object-oriented concepts. It seems that this was not controlled when collecting eye movement data for the EMIP dataset. Furthermore, the source codes were formatted to have the option for tracking single tokens (e.g., spaces between keywords and semicolons) to obtain better fixation matches on single elements. These changes may influence readability. b) Although the source code examples were carefully selected to match code snippets from the empirical world of the selected participants, it is possible that some participants had recently worked on similar examples in an exam or course, which is advantageous. Therefore, these participants would have had an advantage in answering the source code comprehension questions. c) The expertise is based on a self-assessment. Participants answered questions related to their general programming knowledge and knowledge with Java. However, the judgment is likely to be biased because self-assessments are unreliable. This is a common problem in source code comprehension studies, without a prime solution yet to be known. Therefore, measurements based on the skill level cannot be guaranteed to be precise enough. d) The data was collected in different labs with different cultures and environments. These differences can have an influence on the data, even if the settings and comprehension tasks are close to real-world scenarios. e) The comprehension questions targeted the implementation or specification of the source code examples. The assumption was that behavioral questions, targeting the dynamic characteristics of source code, are more likely to test the comprehension performance of participants.

Internal Validity: OOP Code, Known Examples, Self-Assessment, Data Recording in Varying Labs, and Implementation-Based Questions

External validity. The following limitations for generalizable results were noted. a) The source codes were not representative of large systems or code bases. The code examples had 18 (Rectangle) and 22 (Vehicle) code lines (without blanks); these numbers are relatively small but necessary for an eye tracking study without specialized tools. b) No time limit per source code example was used to reduce the total amount of eye movement data collected per participant. This would have limited the "wandering around" effect, where participants

External Validity: Short Source Codes, No Time Limit, and Java Focused

look around the source code even if they already know the answer, which is not distinguishable from the other source code reading and makes eye movement data analysis more complex overall. The absence of a time limit can influence the eye movement data recording and is unusual for real-world scenarios. c) The analyzed eye movement patterns are based on the Java source code used in the study, and source code different from Java will likely produce other eye movement patterns.

5.4 SUMMARY AND NEXT STEPS

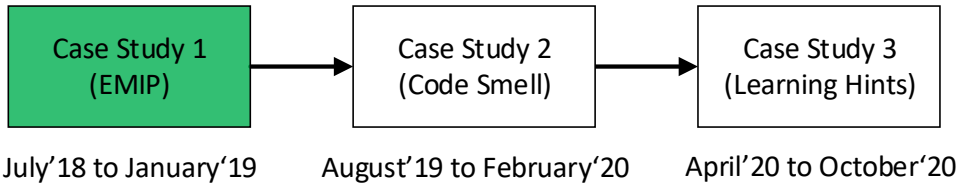


Figure 5.18: A brief overview of the case study timeline (EMIP completed).

The primary objective of the EMIP analysis was to detect common eye movement patterns that are linked to the outcome of a source code comprehension session. The first case study is thus complete (see Figure 5.18).

AOI-DNA and AOI-STG are Useful Visualizations…

The proposed methods, namely AOI-DNAs and AOI-STGs, provide a fast and reliable overview of eye movement data. The visualizations are particularly useful for a first impression and for a qualitative search for eye movement patterns, such as the ones mentioned in the hypothesis $H_{AOI-STG-Patterns}$ (see Section 5.2.4).

…With Limitations

The visualizations also revealed the limitations of such methods. The AOI-DNAs were used as a visualization tool to explore patterns with different color codes to highlight certain AOIs or to hide others. While this is useful for a first impression, a detailed analysis is only possible with an implemented visualization application that allows for searching, filtering, and switching colors. Therefore, it would be possible to detect more patterns and to make a detailed statement about eye movement patterns. It became apparent that these requirements must be bundled into a prototype to analyze AOI-DNAs even further.

Specialized Study Necessary

In general, detecting patterns only via visualizations is challenging because at some point, the amount of data will make the process too difficult. Therefore, no new eye movement patterns were found, only already known patterns, for example those described in the EMIP coding scheme (Busjahn, Schulte, & Kropp, 2014; Busjahn, Schulte, Sharif, et al., 2014). However, this is a success because it proves that a complex visualization is not necessarily needed to demonstrate the existence of eye movement patterns. A specifically designed study could help to detect eye movement patterns in the context of comprehension hurdles. Thus, a

slightly different approach is proposed to capture patterns indicating comprehension hurdles. The assumption was that this will make it easier to detect eye movement patterns tied to comprehension hurdles of participants.

Good and Bad Source Code

Every participant sees two kinds of stimuli: one with source code considered as *good*, and one with source code considered as *bad*. There are many definitions of good and bad code, for example source code with defects (bugs) or source code with a smell. This highly depends on the pattern one tries to observe, the intended recording time, the amount of data, and the complexity of the source code. The idea is that the bad source code will result in different eye movement patterns compared to the good source code.

Provoke Eye Movement Patterns

Another idea is to provoke eye movement patterns with focused questions. In most studies, comprehension questions are asked after the participant saw the stimulus. While this can be suitable for general research questions in source code comprehension, the assumption is that a focused question presented before the stimulus will produce focused eye movement patterns. It is still uncertain whether a question asked beforehand will indeed improve the data quality.

Behavioral Comprehension Questions

In addition, *behavioral questions* are proposed that target the source code as a measure for source code comprehension. The EMIP study design uses structural multiple-choice questions such as "The program computes the area of rectangles by multiplying their width (x1-x2) and height (y1-y2)." This can induce the problem of guessing the answer or simply memorizing some aspects or regions of the source code. A behavioral question, such as "Which value is returned by calling method *XYZ*, considering the state of the object?", forces the participant to build a mental model of the source code to execute the program, rather than simply guessing. This is in line with the source code execution, the third dimension of source code comprehension (Busjahn, Bednarik, et al., 2015).

Time Limit per Stimulus

Moreover, a *time limit* is proposed for every source code example, and a participant must thus focus on the task. In combination with the focused question, the assumption is that a participant will try to solve the comprehension task in the time limit, with a focus on the parts within the stimulus that are important for the participant. The time limit should be announced before the stimulus is presented, and there should be no visualization of the time limit as this could distract the participant and add additional noise to the eye movement data.

Retrospective Think-Aloud (RTA)

Furthermore, a *retrospective think-aloud (RTA)* (Busjahn et al., 2011; Holmqvist, 2011) is proposed after a participant has finished the task in order to explain the intention. This adds a new level and perspective to the eye movement data. Therefore, the assumptions are that the interpretation of the eye movement data is easier and that it would be possible to identify comprehension hurdles in the form of eye movement patterns.

A source code comprehension study with these goals and design changes in mind is described in Chapter 6.

"Your eyes can deceive you. Don't trust them."

– Obi-Wan Kenobi (Star Wars: Episode IV – A New Hope)

6

Case Study 2 – The Code Smell Study

Provoking Eye Movement Patterns

The results of the EMIP analysis (see Section 5.2) were promising in the sense of detecting reading strategies and patterns, but unsuccessful regarding the link to the comprehension problems of participants. After this analysis, the idea of conducting a specialized study for provoking eye movement patterns solidified (see Section 5.4). The study includes different ideas and concepts – such as a distractor, based on source code smells, that uses non-idiomatic source code – behavioral source code comprehension questions, a time limit per source code example, and an RTA to obtain more information about participants' cognitive processes.

Idiomatic Source Code Smells

This so-called code smell study aims at identifying the eye movement patterns of different participant groups. One group was asked to comprehend unmodified source code, while the other group received source code with one idiomatic source code smell. The assumption was that the source code smells are able to provoke behavioral reading patterns, which can be categorized as comprehension strategies and are detectable as characteristic eye movement patterns. This study and the subsequent analysis reported in this chapter close the link between source code smells → comprehension hurdles → eye movement patterns. The participant groups consisted not only of students from the local university campus, but also experts from one software development company. This allowed for a separation of the participant groups into a novice group (students) and an expert group (software developers). The data analysis focuses on a) whether code smells influence the answer quality of comprehension questions, the fixations, and the durations spent on the source code examples

and b) whether the expert group provides better answers than the novice group of participants. In summary, the study was conducted to make the following three contributions to the body of work of this thesis:

1. Using a distractor, based on idiomatic source code smells, to provoke behavioral reading patterns, categorizable as comprehension strategies and detectable as specific eye movement patterns with a connection to source code comprehension hurdles.
2. The design of the code smell study, with modifications to a commonly used source code comprehension study design in the community. The study uses a) behavioral comprehension questions, b) a time limit per source code example, and c) an RTA.
3. The analysis of two different skill groups (novices and experts), based on students and professional software developers.

6.1 STUDY DESIGN

The following sections describe the design of the code smell study, in which students and professional software developers were recorded.

6.1.1 RESEARCH QUESTION AND HYPOTHESES

Link Between Patterns and Comprehension Problems

Detecting eye movement patterns in the gaze data of participants, especially between novices and experts, is a common goal in source code comprehension studies (Bednarik et al., 2013; Busjahn, Bednarik, et al., 2015; Busjahn, Schulte, & Tamm, 2015). However, the overwhelming majority of studies do not search specifically for comprehension problems. Apart from this, the study design of the EMIP dataset raises some questions: Should there not be a time limit for the comprehension process of source code examples, to reduce the "wandering-around" effect of participants? Can behavioral questions be a better predictor of the source code comprehension process because they require the participant to execute the source code in a mental model, which was built by reading and comprehending the source code? To favor or encourage intentional reading, should the comprehension question be available before reading the source code? These questions and research gaps are the origins of the idea for the code smell study, which was primarily designed to establish the connection between eye movement patterns and comprehension problems and to address the aforementioned problems that were identified while analyzing the EMIP dataset.

MEASURING SOURCE CODE COMPREHENSION (DEPENDENT VARIABLES)

Dependent Variables: Task Answer, Task Duration, Time Limit, and Visual Focus

As dependent variables, various sources to measure the outcome or effort of the source code comprehension tasks were used. The following variables specifically were used to measure the quality of the source code comprehension per participant:

Task answer. Participants' performance of the source code comprehension task was determined by measuring the answers given for each source code example.

Task duration. The behavioral comprehension questions cannot be answered without comprehending the source code first. Therefore, the time duration needed to answer a question reveals, to some extent, the effort required by a participant, since the code smell study uses a time limit for every source code stimulus.

Time limit. The time limit per source code example ensured that a small amount of pressure was added to the comprehension task. This limit minimized the "wandering around" supports other measures such as the task answer and the task duration.

Visual focus. Eye tracking data was used to track the process of reading the source code. In combination with AOI models, participants' reading sequences can be obtained while they comprehend a code example. This process allows for not only the recording of the order and frequency of source code parts that a participant is looking at, but also the measurement of the duration of fixations.

EXPERIMENTAL VARIATION (INDEPENDENT VARIABLES)

Independent Variables: Source Code Examples, Code Smells, and Time of Question

As the experimental rationale, source code examples with fixed order and pre-defined source code smells were selected for the participants. Every code example was available in a smell and non-smell variant. The following specific variables were used to create the study conditions:

Source code example. The study was built to compare two different code examples, namely *Rectangle* and *Vehicle*, with different levels of complexity and different comprehension questions. To reduce the complexity of the study, the source code was *not* randomized (see Section 6.1.3).

Idiomatic code smell. Both source code examples were varied with a source code smell, based on idiomatic programming. The Rectangle code contained an identifier smell, where the name of one method and every variable in it were named with serially numbered names (i.e., `intVariable1`, `intVariable2`, ...). The Vehicle source code contained a structural smell, where the condition of the "if" is implemented so that the true branch remains empty, and the business logic was moved to the "else" branch.

Time of question. The comprehension question was posed at different times: For the first group, it was asked before the source code example was visible (and while the question was visible), and for the second group, it was asked after the source code example was shown, similarly to many other source code comprehension studies.

HYPOTHESES

Hypotheses About Code Smells and Skill Groups

The analysis investigated whether code smells have an impact on the quality of answers (correct or incorrect), whether a code smell is distinguishable depending on the overall comprehension time and ratio between code areas with and without a smell, and whether a code smell alters the reading and source code comprehension behavior overall, observable with different eye movement patterns. To answer these questions, different hypotheses were tested regarding the performance of novices and experts and whether the time of the comprehension question has an impact on the performance of a participant (correct or incorrect answers). Previous research has demonstrated that eye movement patterns must be present for different skill level groups (Busjahn, Schulte, & Tamm, 2015), but not if it is possible to provoke them explicitly. In summary, the following five hypotheses were tested. The first three hypotheses contain assumptions about the influence of the source code smells. The remaining two hypotheses relate to the effects of the novice and expert groups, as well as the time of the comprehension question.

$H_{Answer-Quality}$ – For this hypothesis, an investigation was undertaken to determine whether a code smell influences participants' response quality, which was measured by the quality of their answers (correct or incorrect). The results were analyzed to find significant differences between participant groups (e.g., based on non-smelly and smelly source code examples). The assumption was as follows: A source code with a smell accumulates more incorrect answers than a source code without a smell. The legitimately interesting considerations – if experts issue more correct answers compared to novices, if experts have a lower task duration overall, and if the time of the comprehension question alters the answer quality – were analyzed in the hypotheses $H_{Performance-Experts-Novices}$ and $H_{Comprehension-Question-Time}$ in detail. The results for this hypothesis are described in Section 6.2.1.

$H_{Ratio-Duration-Fixations}$ – The question pertaining to this hypothesis is whether a code smell influences the number of fixations and the overall duration a participant spends on a stimulus. The ratio between smelly and non-smelly AOIs (code lines) in terms of the duration and the number of fixations were analyzed. The calculations were based on simple metrics (e.g., the overall duration of a stimulus and the fixations on AOIs) to find significant differences between various groups. The assumption was that a code smell that is visually perceived, and where the reading process is recorded with eye tracking, should account for a higher duration and fixations on the source code overall and for the ratios between smelly and non-smelly AOIs in particular. This would support this study's assumption that a

source code smell interferes with the reading process of participants. Please refer to Section 6.2.2 for the results.

$H_{Pattern-Reading-Behavior}$ – For this hypothesis, a search was conducted for differences in the reading behavior of participants based on eye movement patterns. The analysis was based on the SOR, EOR, flicking, and retrace declaration patterns, and it used the developed tool, CodeSight (Deitelhoff et al., 2019b), to visualize eye movement patterns. The assumption was that a smelly source code example interferes with both novices and experts' ideal-typical reading behavior. This interference is measurable by the existence or absence of specific eye movement patterns. The results are described in Section 6.2.3.

$H_{Performance-Experts-Novices}$ – For this hypothesis, an investigation was undertaken to determine whether experts have better results compared to novices. The assumption was that experts can use their experience and knowledge to answer the comprehension questions faster, with less visual effort, and with more correct answers. The analysis was based on the quality of the answers (correct or incorrect), which were compared to find significant differences between various groups (e.g., the Rectangle source code vs. the Vehicle code or smelly vs. non-smelly code). This approach is different to the one related to the hypothesis $H_{Answer-Quality}$, in which an expert vs. novice analysis is not considered. The results are available in Section 6.2.4.

$H_{Comprehension-Question-Time}$ – The question related to this hypothesis is whether the time of the comprehension question has an impact on the quality of participants' answers (correct or incorrect). The assumption was that knowing the question before reading the source code examples is beneficial, regardless of the skill group and the presence of a source code smell. The results were analyzed to find significant differences between various groups. For this hypothesis, the question time was used (before and after the source code example) from different perspectives (e.g., the Rectangle source code vs. the Vehicle code or smelly vs. non-smelly code). Please refer to Section 6.2.5 for the results.

6.1.2 METHODS

AOI-DNAs, TraMineR, and ANOVA

The visualizations for eye movement patterns are based on the already described AOI-DNAs (Deitelhoff et al., 2019b), which were integrated into the CodeSight application to provide a dynamic interface for the static AOI-DNAs used in the previously described EMIP analysis. The search for and investigation of eye movement patterns is based on the top-down and bottom-up approaches. Both sequence alignment algorithms (the N-W and OM algorithms) were used in combination with dendrogram and TraMineR visualizations. They are crucial to identify and visualize important eye movement patterns and thus differences between

study conditions and participant groups. The data basis for this analysis consists of eye movement data and eye movement metrics. The differences among groups were analyzed with ANOVA tests, and downstream unpaired *t-tests* were carried out to verify the ANOVA results to analyze whether different results are detectable. This was not the case, and the ANOVA results are correct, which is why the ANOVA data and results are reported in this study.

Retrospective Think-Aloud (RTA)

Furthermore, an RTA was conducted with every participant at the end of the study. In such an RTA, participants see their eye movements superimposed onto the source code examples. This was done in the Tobii Pro Studio software, which has the RTA feature included. After the recording of the source code comprehension parts, the RTA mode was started, where a recording was selected. In this recording, participants' eye movements were visible on the source code stimulus. The eye movement data was calculated with the same fixation filter settings as the overall analysis. Participants then had to explain the individual eye movements while the recording was played as a video. They had the ability to play/pause the recording with the *space bar* for situations where details had to be explained. The recording displayed every saccade or new fixation one after another (step by step), and participants were urged to explain what they saw in their own terms. If a participant was too shy or had not said much at all, they were prompted to explain situations that seem to be important based on the judgement of the study leader. The goal was to obtain information about the intention of the participants' reading strategy, which is a dimension not available via eye tracking alone. The analysis of the RTA was conducted by two researchers independently who used the transcribed audio recordings to mark specific *justifications* and *references* in the RTAs. Every time a participant justified their viewing, it was marked with a green color, and each time some source code parts, variables, or code structure was referred to, it was marked in blue. In the end, the marks of both experts were discussed, matched, and merged.

Demographic Data

Furthermore, some demographic data of every participant, such as age, gender, degree course or job title, and their self-assessment regarding the overall programming knowledge and already used programming languages, was requested.

6.1.3 EXPERIMENTAL DESIGN

Smell and Non-Smell Source Code Examples

The code smell study was designed according to two different conditions: a) non-smelly vs. smelly source code and b) comprehension questions before the task vs. questions after the task. With the first condition, the goal was to test whether a code smell can provoke eye movement patterns. The goal of the second condition was to test whether participants' knowledge of the comprehension question before the stimulus is visible makes a difference. In many source code comprehension studies, the question is asked after a participant sees the stimulus. This study's assumption was that this has a substantial impact on participants'

performance (correct vs. incorrect answers). This could consequently lead to false conclusions of the comprehension process, wrong participant groups based on this performance, and hence false conclusions. The combination of non-smelly and smelly source code examples, as well as the time of the comprehension question, leads to four different conditions, visualized in Table 6.1.

Table 6.1: Overview of the four conditions for the code smell study, consisting of combinations of Rectangle and Vehicle source code examples, smell and non-smell, as well as questions before and after variations.

	Rectangle non-smell and Vehicle smell	Rectangle smell and Vehicle non-smell
Question after	**Condition 1** 8 CS Students 3 Professionals	**Condition 3** 8 CS Students 3 Professionals
Question before	**Condition 2** 8 CS Students 3 Professionals	**Condition 4** 8 CS Students 3 Professionals

Randomly Assigned Participant Groups and Stable Stimulus Order

With these conditions, every participant was required to answer one question for the two source code examples with which they were presented. The first stimulus was always a Rectangle task, followed by the stimulus from the Vehicle task. Per participant, it was randomly decided whether the Rectangle stimulus has the code smell. Aligned to this decision, the smell in the Vehicle stimulus was added or removed accordingly. Therefore, a participant received the Rectangle stimulus first, either with or without a code smell, followed by the Vehicle stimulus, also with or without a code smell, aligned with the presence of a smell in the Rectangle stimulus. No random selection was made regarding whether a participant received a stimulus with the Rectangle or the Vehicle task first. While this is indeed possible and can reduce carryover effects, it makes the participant recruitment and overall study more complex. To reduce this complexity, trade-offs in the study design are necessary (Siegmund, 2016). Every participant was randomly assigned to the group with the comprehension question either before or after the first stimulus was presented. The time of comprehension questions did not change within the study. If participants were assigned to the group with the question before presenting the source code example, they saw the question for the first and second stimulus beforehand, and vice versa. Figure 6.1 visualizes the experimental design according to the order of stimuli and the different groups.

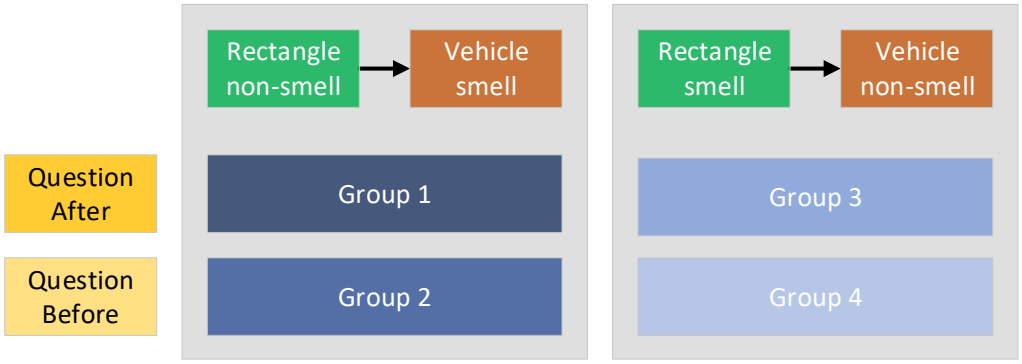


Figure 6.1: The visualized code smell experimental design with the four groups, the stimuli order, and the two different question times.

Four Conditions

To summarize the experimental design for the code smell study, four conditions were created, and participants were randomly assigned to one of them. The conditions had a stable sequence of source code examples (stimuli) with one of the Rectangle tasks first, followed by one stimulus of the Vehicle task, but with a different sequence in terms of smell and non-smell, as well as the time of the comprehension question.

6.1.4 PREREQUISITES AND ENVIRONMENT

Laboratory Study Setup

The prerequisites for the code smell study primarily affect the eye tracking setup. Figure 6.2 presents the layout of the setup in the eye tracking lab, where the student group was recorded. In contrast, Figure 6.3 depicts the layout of the setup at the company where the experts were recorded. Both setups were similar, with the main difference being the side table with the documents – this difference is only important for how the briefing of the study was conducted; it had no effect on the recording. Both groups were recorded with the *Tobii Pro TX300* and the *Tobii Pro Studio* software.

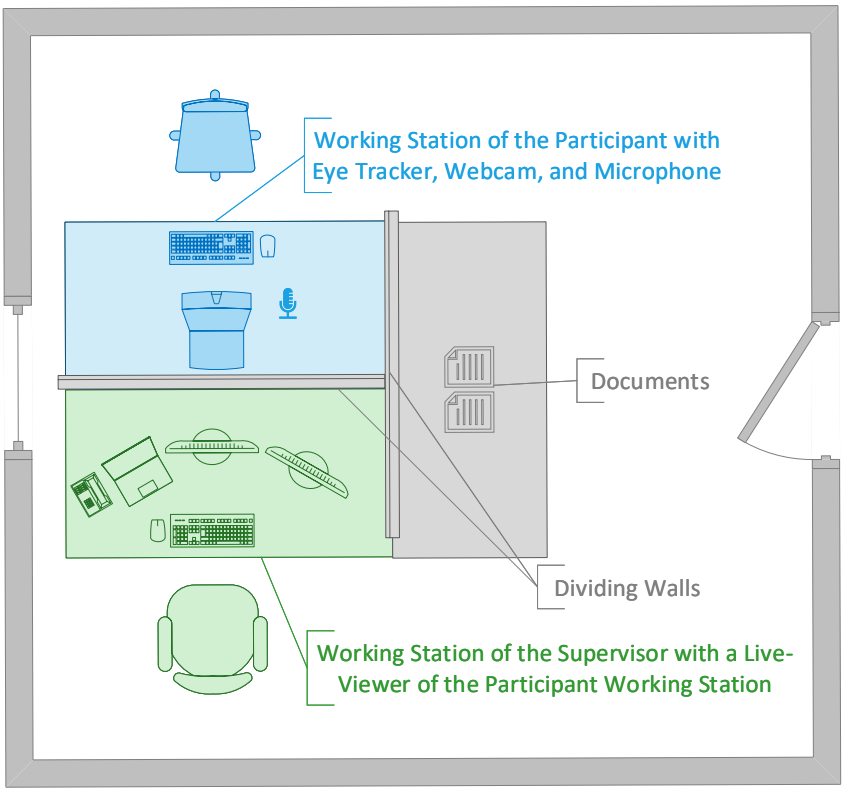


Figure 6.2: The layout of the setup in the eye tracking lab at the University of Applied Sciences and Arts, Dortmund.

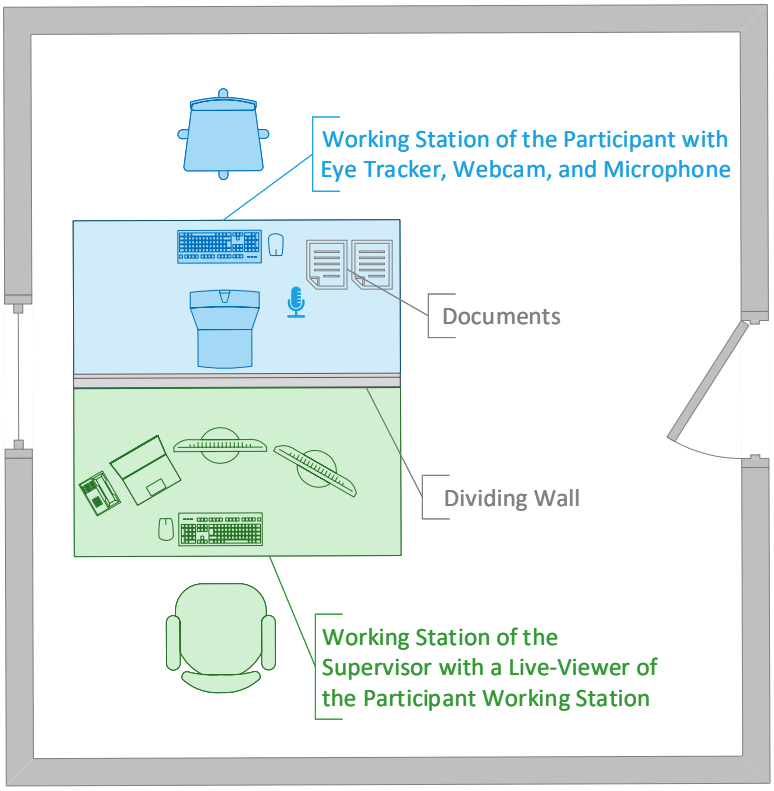


Figure 6.3: The layout of the setup at the software development company in Lünen, Germany.

PowerPoint Presentation for the Stimuli

No specialized programming environment or study prototype was needed to conduct the study. The four conditions were encoded into PowerPoint presentations, which were used per participant. The conditions could thus be balanced. Every time a participant had to be excluded for poor eye movement data quality (i.e., too few gaze examples, low overall quality of the recording, or insufficient eye tracking calibration), the condition was filled up with the next participant. Figure 6.4 portrays one example page of the study condition with the Rectangle source code without a smell, the Vehicle code with a smell, and the comprehension question after the source code stimuli were presented (condition 1).

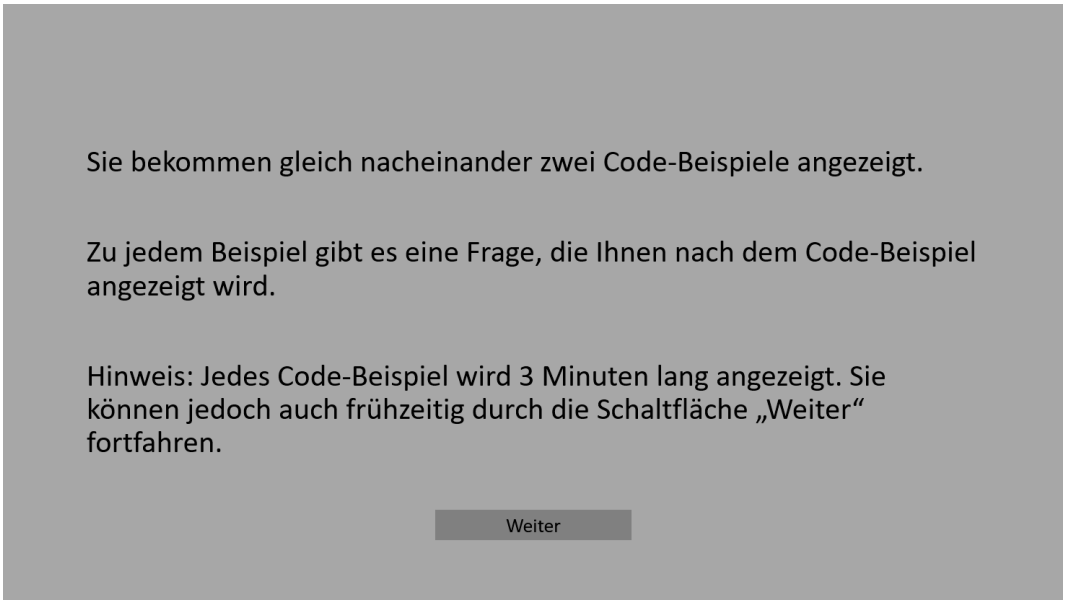


Figure 6.4: Screenshot of one example slide of the presentation for the study condition one: Rectangle non-smell, Vehicle smell, question after the code example.

The study was conducted in German. The text on the example slide translates as seen in Figure 6.5.

The PowerPoint files also included a short explanation of what to expect (i.e., a briefing) and a notice that the data is recorded with an eye tracking device. The presentation files were encoded as follows (OK = non-smell):

1. **RectOK-VehSmell-Frage nachher (RO-VS-FN)** – Rectangle without smell, Vehicle with smell, question after the source code was presented.
2. **RectOk-VehSmell-Frage vorher (RO-VS-FV)** – Rectangle without smell, Vehicle with smell, question before the source code was presented.
3. **RectSmell-VehOk-Frage nachher (RS-VO-FN)** – Rectangle with smell, Vehicle without smell, question after the source code was presented.
4. **RectSmell-VehOk-Frage vorher (RS-VO-FV)** – Rectangle with smell, Vehicle without smell, question before the source code was presented.

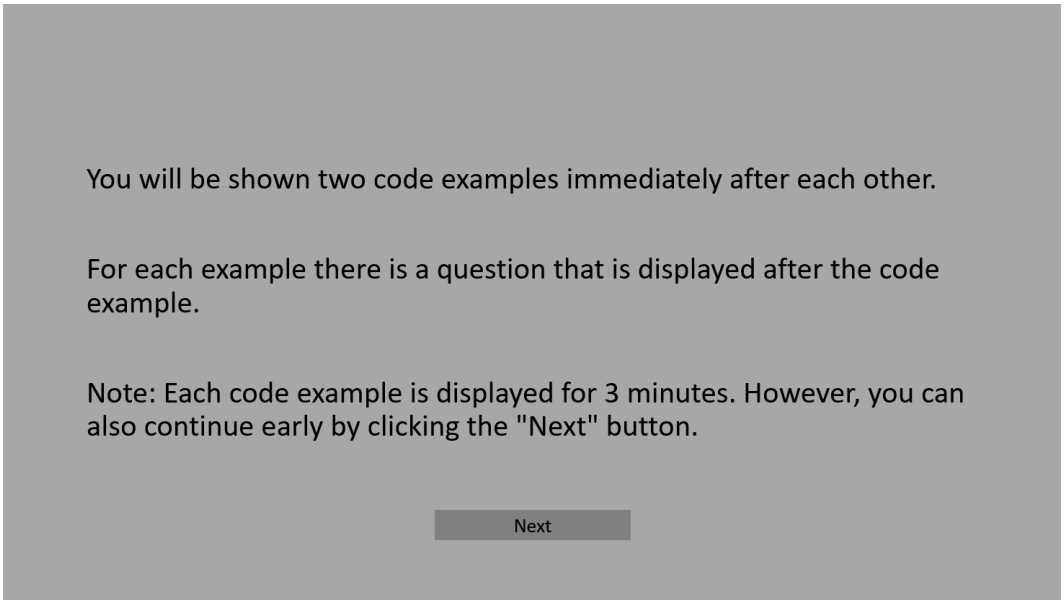


Figure 6.5: Screenshot of one example slide, translated into English, of the presentation for the first study condition: Rectangle non-smell, Vehicle smell, question after the code example.

To assign participants to the correct conditions with missing recordings, the following was manually noted during the recordings: which condition was already balanced and which was not.

6.1.5 CAPTURED DATA

Tobii Pro TX300 Eye Tracker

The eye tracking data for the code smell study was recorded with the Tobii Pro TX300 and the Tobii Pro Studio software in the eye tracking lab at the University of Applied Sciences and Arts, Dortmund. For the recording of the experts, the equipment was moved to the location of the company (Lünen, Germany). All eye tracking data was recorded as raw data, thereby allowing more options for filtering and analyzing the data stream later. The data was exported as tab-separated value files (.tsv) for later use.

Videos of the Comprehension Process

In addition to capturing the raw eye movement data, videos from the comprehension process were recorded in the Tobii Pro Studio software, thereby enabling eye movements to be analyzed qualitatively if necessary and the comprehension answers, entered into text fields embedded in the presentation slides, to be recorded. Moreover, participants' faces were recorded with a webcam to obtain information about situations when they were not looking at the screen for various reasons.

Metadata with Demographic Data and RTA

The captured metadata for all participants contains, among other things, participants' age, gender, mother tongue, English level, overall programming expertise, and expertise in Java. This demographic data was recorded with a survey at the end of the study for every participant. Overall, while conducting the study, eye movement data; answers to the comprehension questions; answers to the survey, including demographic data and the self-assessment; and the RTA as a voice recording were recorded. The RTA was conducted after

the short survey because a second recording session was started in the Tobii Pro Studio software. Thus, the RTA was recorded verbally not only with a recording device, but also with the microphone of the computer in the RTA session as a form of backup. Apart from this, the Tobii software allows one to start an RTA session where the participants can control the visualization of their eye movements, in the form of a gaze plot, on the source code example. The RTA questions and answers were later transcribed from the audio recording to text. Most of the data was combined to a large metadata file, which helped in analyzing connections between the data (i.e., source code example, answers).

6.1.6 PARTICIPANTS, MATERIAL, AND PROCEDURE

$n = 44$ (32 Students and 12 Experts)

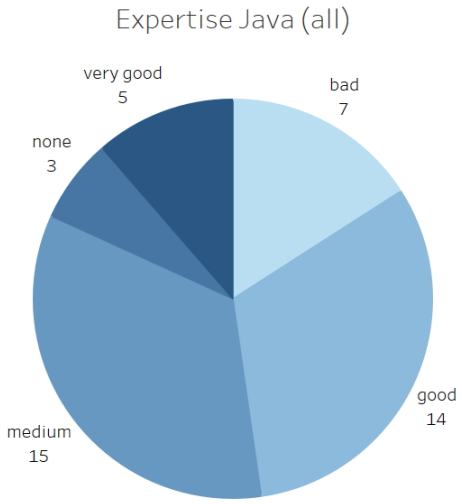
For the code smell study, 42 students from the campus of the University of Applied Sciences and Arts and the TU University, both in Dortmund, were recorded, along with 13 experts from a software development company in Lünen, Germany. Due to data quality restrictions (e.g., eye tracking data with insufficient data points), 10 student recordings and one expert recording were excluded. This quality assessment took place after each recording, so a fast decision was possible regarding whether the data quality was sufficient. This procedure was important to create balanced study conditions. The data basis for the study thus consists of $n = 44$ (32 students + 12 experts) participants for the quantitative and qualitative analyses, with eight students and three experts per condition. Of the 44 individuals in this group, 7 were female and 37 male, with a mean age of 26.43 ($SD = 6.8$).

Java and Programming Expertise

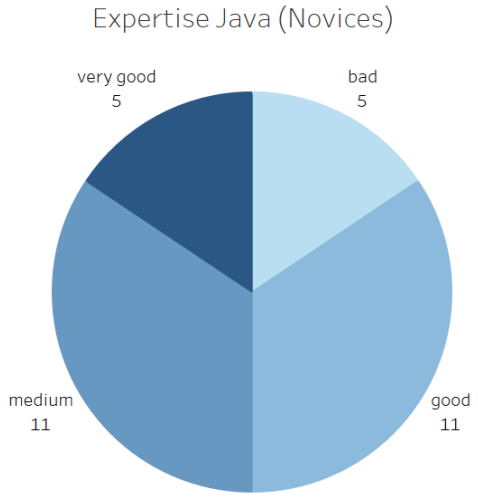
All participants filled in a questionnaire with questions related to, for example, their expertise in the experiment language (Java), which is an important one. The five options for this particular question were *none*, *bad*, *medium*, *good*, and *very good*. The answers, based on all 44 participants in the dataset, were distributed as follows: 3 (none), 7 (bad), 15 (medium), 14 (good), and 5 (very good). For the novices, the distributions were 0 (none), 5 (bad), 11 (medium), 11 (good), and 5 (very good), and for the experts, they were 3 (none), 2 (bad), 4 (medium), 3 (good), and 0 (very good). Figure 6.6 illustrates this distribution first for all participants and then divided for the novice and expert groups.

Ambiguous Expert Skill-Level

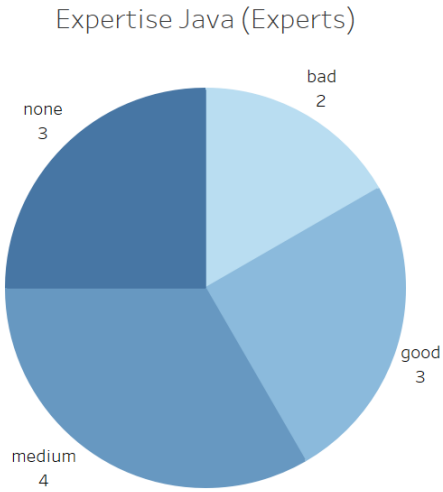
The distribution of answers to the expertise question indicates that the skill level of the experts was not as high as expected for the Java programming language. Three out of the 12 professional participants selected the "none" option and two the "bad" one. This is a significant number of participants in the expert group with a lack of Java knowledge. Nevertheless, these participants were not excluded from the data analysis because a) no other professionals were available who could be recorded for the study, and b) the subsequent analysis revealed that the professionals managed to read Java and solved the comprehension questions sufficiently or that their influence was not as high as expected. As forward references, Table 6.2 (see page 133) indicates that the participants with no Java knowledge, who are only experts, are in Conditions 2 and 4. Both of them are in the question-before



(a) The distribution of answers to the question on expertise in Java (all).



(b) The distribution of answers to the question on expertise in Java (novices).



(c) The distribution of answers to the question on expertise in Java (experts).

Figure 6.6: Distributions for the question about the expertise in Java (all, novices, and experts).

group. Furthermore, Table 6.20 (see page 155) indicates that the results are very good for this particular condition and group; they could have been even higher without the three experts with no Java knowledge. Apart from this, removing at least the three professionals who selected the "none" option for the Java expertise answer would lead to a small number of professional participants and hence an unbalanced study.

Rectangle and Vehicle Source Code Examples

Throughout the study, various source code examples as stimuli were used, separated into two tasks: 1) Rectangle and 2) Vehicle. The Rectangle task contained a Java class with a Rectangle and a method to calculate the perimeter. The Vehicle task contained a Java class with a Vehicle and a method to accelerate the current speed, based on a top speed setting. Each source code example was available with and without a source code smell for the different conditions.

Code Examples Similar to EMIP

The non-smelly source code was based on the stimuli used for collecting the data for the EMIP dataset. Although the EMIP dataset contains stimuli for the three different programming languages – Java, Scala, and Python – the code smell study is focuses on Java, which is a widely used introductory programming language both at the University of Applied Sciences and Arts in Dortmund and at the TU University Dortmund. This improved the likelihood of recruiting participants for a study using Java. The EMIP working group based their Python source code for the Rectangle task on a Python program[14] by Michael Hansen, who is responsible for the eyeCode experiments[15], which are designed to investigate the effects of the comprehensibility of code. The source code for the Vehicle task was shortened and slightly adapted by the EMIP working group from Java teaching material, which is provided by Falko Ripsas, a Computer Science high school teacher from Berlin. The Java versions of the Rectangle and Vehicle source codes from the EMIP dataset were used in the code smell study, and they were modified to include a code smell based on idiomatic programming.

Code Smell and Idiomatic Programming

A code smell, in the sense of idiomatic programming, is a smelly region in the source code that does not stick to common rules ordinarily set by the maintainer of the programming language or the community. The rules are specific for a programming language. For a non-idiomatic Java method, the opening curly brace would start on a separate new line, which is otherwise idiomatic for C#. Vice versa, a method with the opening curly brace on the same line as the method signature is non-idiomatic for C#, but idiomatic for Java. Idiomatic programming was chosen deliberately for the code smells, in favor of design or architectural patterns. Other research has found that larger anti-patterns, such as *spaghetti code* and the *blob*, has an impact on source code comprehension (Politowski et al., 2020). Idiomatic code smells have the advantage that they can be implemented within a few lines of code in a single class. Modifying a design or architectural pattern needs many more classes and hence much more code, which is generally not the best scenario for a source code comprehension study without a specialized recording tool.

Two Code Smells

Listings 6.1 and 6.2 depict the Rectangle source code without and with a source code smell, which is based on an identifier smell, because the method name `per` and the variables in this method are named to be unreadable and difficult to remember (see the lines 11–17 in Listing 6.2). Listings 6.3 and 6.4 present the Vehicle source code without and with a source code smell. This time the smell is based on a structural smell because the `accelerate` method contains an `if-else` structure, where the `true-branch` is empty (see the lines 13–14 in Listing 6.4). This is syntactically and semantically fine, but challenging to read and follow regarding the source code execution and data flow. Thus, the naming smell is aggravating the traceability and recall of source code. The structural smell is complicating the semantics and the logic of the source code.

[14] `http://eyecode.synesthesiam.com/stories/programs.html` – Last accessed on December 14, 2020.
[15] `http://eyecode.synesthesiam.com/` – Last accessed on December 14, 2020.

The Rectangle Code Example Without Smell

```
1  public class Rectangle {
2    private int x1 , y1 , x2 , y2 ;
3
4    public Rectangle ( int x1 , int y1 , int x2 , int y2 ) {
5      this.x1 = x1 ;
6      this.y1 = y1 ;
7      this.x2 = x2 ;
8      this.y2 = y2 ;
9    }
10
11   public int perimeter ( ) {
12     int width = this.x2 - this.x1 ;
13     int height = this.y2 - this.y1 ;
14
15     int perimeter = width + width + height + height ;
16
17     return perimeter ;
18   }
19
20   public static void main ( String [ ] args ) {
21     Rectangle rect1 = new Rectangle ( 0 , 0 , 10 , 10 ) ;
22     int result = rect1.perimeter ( ) ;
23   }
24 }
```

Listing 6.1: The Rectangle source code example without a smell for the code smell study.

The Rectangle Code Example With Smell

```
1  public class Rectangle {
2    private int x1 , y1 , x2 , y2 ;
3
4    public Rectangle ( int x1 , int y1 , int x2 , int y2 ) {
5      this.x1 = x1 ;
6      this.y1 = y1 ;
7      this.x2 = x2 ;
8      this.y2 = y2 ;
9    }
10
11   public int per ( ) {
12     int intVal1 = this.x2 - this.x1 ;
13     int intVal2 = this.y2 - this.y1 ;
14
15     int intVal3 = intVal1 + intVal1 + intVal2 + intVal2 ;
16
17     return intVal3 ;
18   }
19
20   public static void main ( String [ ] args ) {
```

```
21      Rectangle rect1 = new Rectangle ( 0 , 0 , 10 , 10 ) ;
22      int result = rect1.per ( ) ;
23    }
24  }
```

Listing 6.2: The Rectangle source code example with a smell for the code smell study.

Rectangle Questions

The questions for both Rectangle source code variants are presented in the following enumeration (translated to English). The first question is for the non-smell, the second for the smell variant of the code example. The difference is localized in the naming of the method `perimeter` (non-smell) respectively `per` (smell).

1. "Which value is returned when the method perimeter() is called, considering the object state?"
2. "Which value is returned when the method per() is called, considering the object state?"

The Vehicle Code Example Without Smell

```
1  public class Vehicle {
2    String producer , type ;
3    int topSpeed , currentSpeed ;
4
5    public Vehicle ( String p , String t , int tp ) {
6      this.producer = p ;
7      this.type = t ;
8      this.topSpeed = tp ;
9      this.currentSpeed = 0 ;
10   }
11
12   public int accelerate ( int kmh ) {
13     if ( ( this.currentSpeed + kmh ) < this.topSpeed ) {
14       this.currentSpeed=this.currentSpeed + kmh ;
15     }
16
17     return this.currentSpeed ;
18   }
19
20   public static void main ( String args [ ] ) {
21     Vehicle v=new Vehicle ( "Audi" , "A6" , 200 ) ;
22     int result=v.accelerate ( 10 ) ;
23   }
24 }
```

Listing 6.3: The Vehicle source code example without a smell for the code smell study.

The Vehicle Code Example With Smell

```
1  public class Vehicle {
2    String producer , type ;
3    int topSpeed , currentSpeed ;
4
5    public Vehicle ( String p , String t , int tp ) {
6      this.producer = p ;
7      this.type = t ;
8      this.topSpeed = tp ;
9      this.currentSpeed = 0 ;
10   }
11
12   public int accelerate ( int kmh ) {
13     if ( ( this.currentSpeed + kmh ) > this.topSpeed ) {
14     } else {
15       this.currentSpeed=this.currentSpeed + kmh ;
16     }
17
18     return this.currentSpeed ;
19   }
20
21   public static void main ( String args [ ] ) {
22     Vehicle v=new Vehicle ( "Audi" , "A6" , 200 ) ;
23     int result=v.accelerate ( 10 ) ;
24   }
25 }
```

Listing 6.4: The Vehicle source code example with a smell for the code smell study.

Vehicle Question

The question for both Vehicle source code variants is visible in the following enumeration (translated to English). Because the smell for the Vehicle source code is affecting the structure of the `accelerate` method, and not the name, the question stays the same for both source code smells.

1. "Which value is returned when the method accelerate() is called, considering the object state?"

Study Procedure

The procedure for recording the participants was relatively simple. After greeting and shortly introducing the participants to eye tracking as a general technique and technology, every participant had to sign a data protection form. This form provided information about how the data in the study was used and which data was collected. Afterward, the eye tracking equipment was calibrated to start the source code comprehension process and the eye tracking recording. The participants subsequently completed the survey with questions about their known languages, prior knowledge and experiences with programming languages, age, gender, and other basic demographic data. As a last step, to conduct the RTA, in which questions about reading behavior were asked, the participants saw their eye

movements superimposed on the source code examples. At the end, every participant was debriefed and had to sign a form for receiving the 10 € compensation for their effort. In general, the overall process took 25–30 min, depending mainly on how well the eye tracking calibration went and how long the participants needed to fill out the survey. Figure 6.7 visualizes the general study procedure per participant. This procedure was the same for the sessions in the eye tracking lab and in the company.

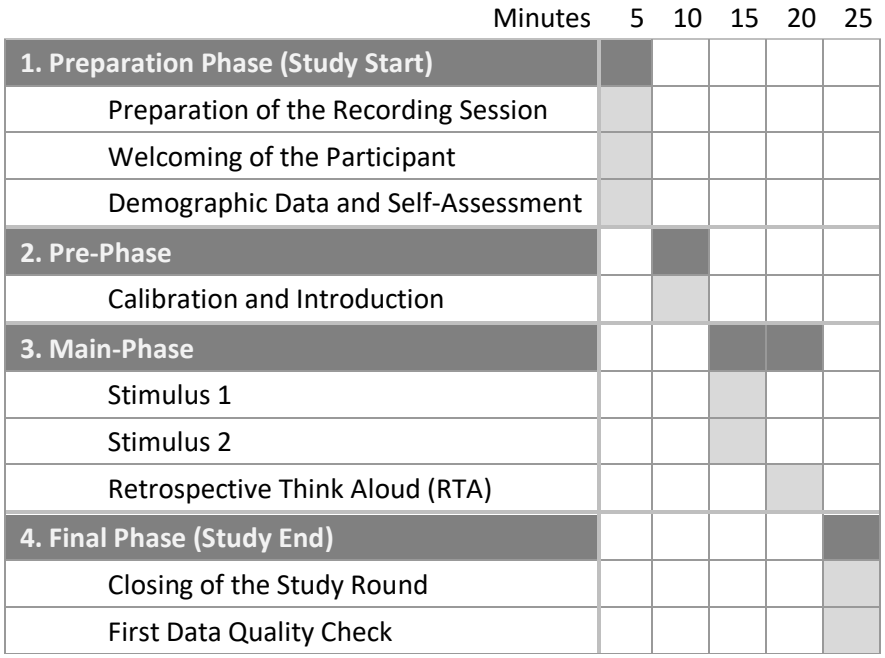


Figure 6.7: The study procedure for the code smell study.

6.1.7 ANALYSIS DESIGN

Encoding of Participant Answers

During the course of the study, every participant encountered one task with the Rectangle and one with the Vehicle source code example. The raw eye movement data was recorded in the Tobii Pro Studio software and exported thereafter. The quantitative part of the analyses was based on the answer quality and eye movement metrics of every participant. For answer quality, the exact answers that a participant entered in the text fields were recorded and coded to have a range from 0 to 100. A correct answer was coded as 100, while a completely incorrect answer, for example leaving the answer intentionally blank or answering with information completely unrelated to the question, was equal to 0. In between, each answer was manually judged to acknowledge a partially correct answer. This coding was done by two researchers individually and matched afterwards.

Common Eye Movement Metrics

In addition to the answers, common eye movement metrics were used, such as the overall duration and fixation count per stimulus, the ratio between smelly and non-smelly AOIs based on the duration and fixation count, the duration and fixation count per smelly AOI, and the duration and fixation count, until a smelly AOI was seen first by a participant. These

eye movement metrics are common in the source code comprehension community, as explained in Section 2.4.2. These analyses used the exported eye tracking recordings, containing the eye movement fixations, exported from Tobii Pro Studio. Before exporting the data, the Tobii Pro Studio integrated I-VT filter was used with a maximum radius of 60 px, a minimum fixation duration of 60 ms, and a maximum of 55 missing gaze samples to count as a fixation. These are the same settings as in the analysis of the EMIP dataset.

AOI Line and Region Model

To match the fixations to the source code, AOIs around every source code line were created. This follows the AOI line model, with a margin between AOIs. In addition, region AOIs were created by logically combining relevant line AOIs to one region (no bounding rectangle). Furthermore, the F# scripts from the EMIP analysis were adapted for an automatic analysis of the eye movement metrics. Every AOI was marked with a letter between A and Z, and the comparison between various groups, to find significant results or differences, was done with ANOVA tests. For the qualitative part of the analysis, AOI-DNAs and AOI-STG were used (Deitelhoff et al., 2019b); they are important to gain insights into participants' reading behaviors and reading patterns. The following four figures 6.8, 6.9, 6.10, and 6.11 display the AOI labeling for the various source code examples. These definitions are the baseline for the eye movement pattern definitions in this study, where the AOI labels are used in the AOI-DNA visualizations, and for the pattern matching.

```
A public class Rectangle {
    B private int x1 , y1 , x2 , y2 ;

    C public Rectangle ( int x1 , int y1 , int x2 , int y2 ) {
        D this.x1 = x1 ;
        E this.y1 = y1 ;
        F this.x2 = x2 ;
        G this.y2 = y2 ;
    H }

    I public int perimeter ( ) {
        J int width = this.x2 - this.x1 ;
        K int height = this.y2 - this.y1 ;

        L int perimeter = width + width + height + height ;

        M return perimeter ;
    N }

    O public static void main ( String [ ] args ) {
        P Rectangle rect1 = new Rectangle ( 0 , 0 , 10 , 10 ) ;
        Q int result = rect1.perimeter ( ) ;
    R }
S }
```

Figure 6.8: The AOI labeling for the Rectangle non-smelly source code example.

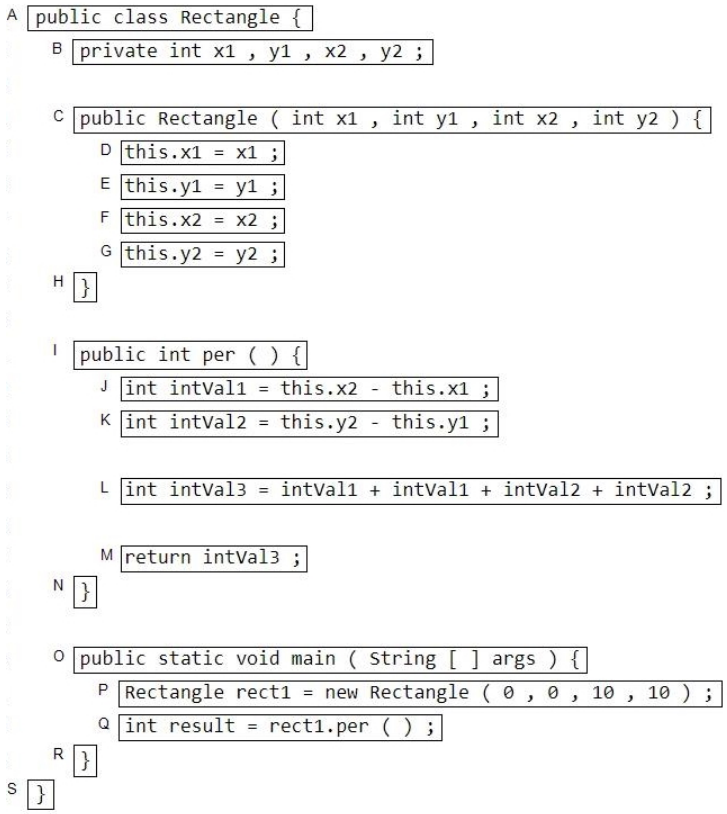


Figure 6.9: The AOI labeling for the Rectangle smelly source code example.

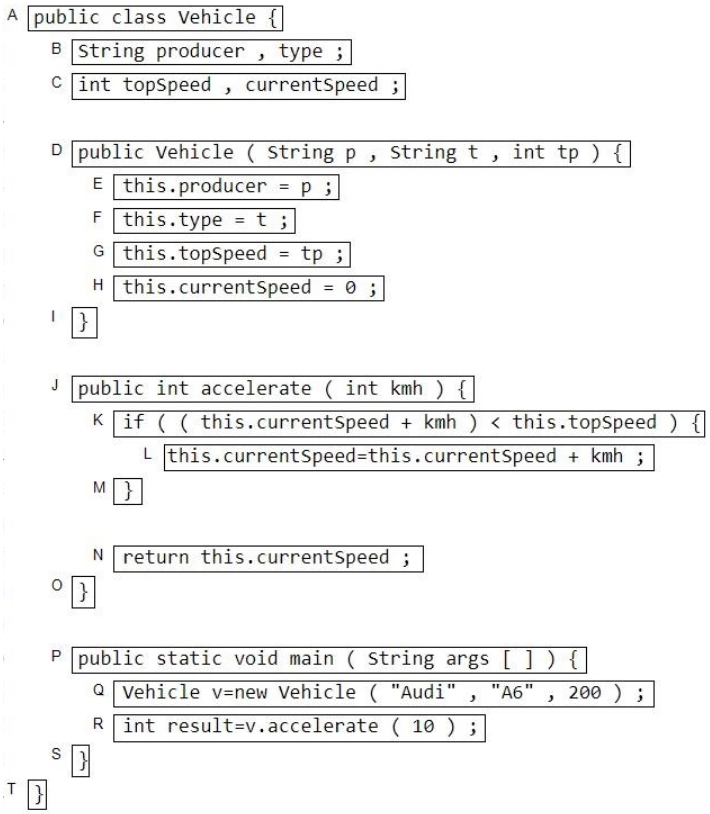


Figure 6.10: The AOI labeling for the Vehicle non-smelly source code example.

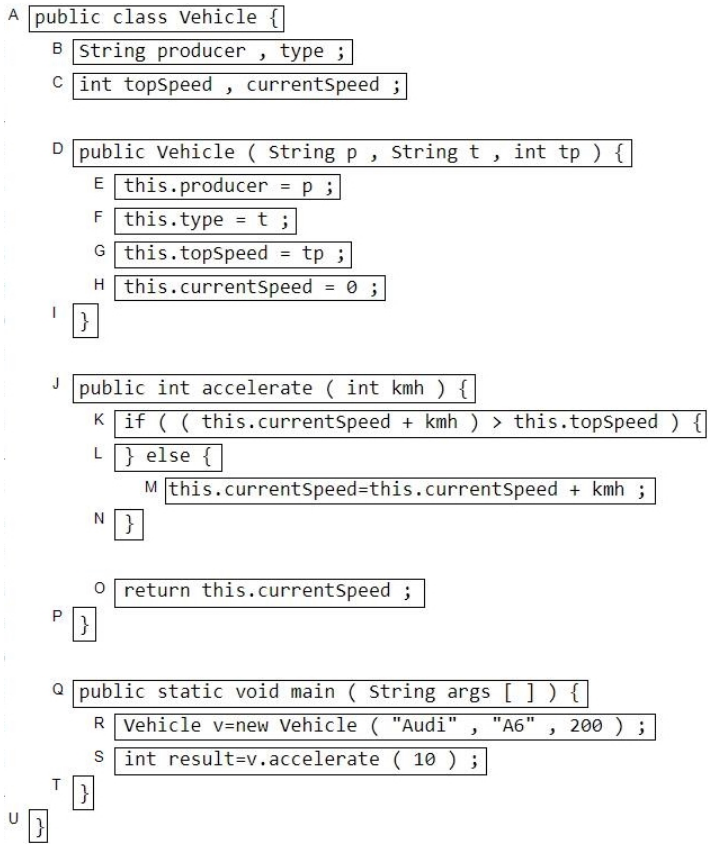


Figure 6.11: The AOI labeling for the Vehicle smelly source code example.

Eye Tracker, I-VT Filter, and Sampling Bias

According to the methodical eye tracking challenges reported in Section 4.1 the following aspects were addressed to minimize the methodical challenges and resulting problems. The involved eye tracker was the *Tobii Pro TX 300* with a sampling rate of 300Hz, which is slightly better than the SMI RED205 and overall sufficient to record fixations and saccades, both of which are important in source code comprehension. As the filter, the default I-VT filter proposed by Tobii was used with the default values reported above. For the AOIs, the maximum margin AOIs were utilized to surround every source code line and important regions with a maximum margin between the AOIs and small padding within the AOIs. The imbalance between the male and female participants with a factor of approximately 5:1 is slightly higher than for the EMIP dataset. However, it still seems to follow the typical distribution of student male and female gender in computer science and engineering faculties, where the data was collected. The increased imbalance was caused by the software company employees, comprising the professionals who were recorded, because no female software developer was present in this group. Another sampling bias based on aggravating recording conditions, such as makeup and glasses, did not require special monitoring, because no participant was excluded based on both conditions. Every time a participant could not be calibrated correctly, the calibration was repeated without glasses as a test to note differences that were not present. For the analysis in this study, participants were excluded due to a low eye tracking data quality for diverse reasons, for example medical conditions

with the eye and different lighting throughout the day. furthermore, a bias towards programming environments was avoided because the source code examples were presented as images in a PowerPoint slide deck without a surrounding or supporting environment. Finally, the comprehension questions targeted the dynamic run time behavior of source code, which is an improvement to the EMIP dataset recordings.

6.2 STUDY RESULTS

The following sections describe the results related to each of the hypotheses (refer to Section 6.1.1).

6.2.1 ANSWER QUALITY ($H_{Answer-Quality}$)

Answer Quality and Code Smells

The first hypothesis deals with the question of whether answer quality, measured by the ratio of correct and incorrect source code comprehension answers, is affected by the source code smells introduced in the source code examples. If that is the case, then the data should indicate more incorrect answers for smelly source code examples. This context was analyzed from different perspectives (e.g., the Rectangle source code analyzed against the Vehicle source code example or each source code stimuli compared against their smell/non-smell variants).

Overview of the Answer Quality

As an introduction to this hypothesis and the analysis, Tables 6.4 (page 135), 6.5 (page 135), 6.6 (page 136), and 6.7 (page 136) list the participant IDs, group affiliations, participant answers, and the coded answers for the source code examples per study condition (refer to Section 6.1.3). A coded answer "0" means an incorrect answer, while an answer coded "100" is completely correct. The rating was done qualitatively by two researchers separately, with a matching of both results afterward. Due to their size, as well as text- and reading-flow-destroying characters, these four tables are placed at the end of this hypothesis, starting from page 135.

Vehicle Stimulus with best Results

The four tables reveal that, overall, the source code comprehension question was best answered for the Vehicle source code example. The assumption was that the difference in answer quality between Conditions 1 and 2 (Tables 6.4 and 6.5) is related to a carryover effect between the Rectangle and Vehicle source code examples. The order was fixed, as explained in Section 6.1.3, and the answer quality differs significantly in the first condition, where the comprehension question was visible afterward. Therefore, the participants may have made an assumption about what they had to do in the second (Vehicle) source code. This difference is much smaller for the second condition, where the comprehension question was visible before the source code was shown, as well as while the question was presented. The results for this condition indicate that many participants answered the question for the

Rectangle source code example correctly, which is an indication that knowing the question beforehand is an advantage.

Carryover Effect

The previous explanation with a carryover effect does not hold as well for Conditions 3 and 4 (Tables 6.6 and 6.7). The difference in the answer quality between the Rectangle and Vehicle source codes is smaller in Condition 3 and much larger in Condition 4, where the comprehension question was asked before the source code was visible. The assumption is that the question time in these conditions is the overall explanation for this effect. In the fourth condition, where the comprehension question was visible before the source code and while the answer was entered by the participants, the answer quality was overall the best between Conditions 3 and 4, and much higher for the Vehicle source code. This is again explainable, colloquially, with the statement "the participants knew what they have to do" (carryover or training effect). This effect can be the reason for the differences in Condition 3. However, the answer quality is much lower; thus, the participants may have had problems answering the comprehension questions.

Cross-Referencing the Skill-Level

A cross-reference with the skill level, collected in the survey as a self-assessment, shows no conclusive result. The options for the answer regarding Java experience was translated to skill points with the following scale: 1 = very good, 2 = good, 3 = medium, 4 = bad, and 5 = none. The number of answers was then multiplied, added, and divided by the number of participants in each group, which is 11 everywhere. Missing answers were left out in the calculation (set to 0). Table 6.2 lists the distributions for the four conditions – lower numbers are better and emphasized.

Table 6.2: The distributions of the skill levels (self-assessment) per condition – lower values are better and emphasized.

# Conditions	Answer Distributions	Results
Condition 1	1 (very good), 4 (good), 6 (medium)	**2.45**
Condition 2	1 (very good), 3 (good), 3 (medium), 3 (bad), 1 (none)	3.00
Condition 3	2 (very good), 3 (good), 4 (medium), 2 (bad)	**2.55**
Condition 4	1 (very good), 4 (good), 4 (medium), 2 (bad), 2 (none)	3.55

Skill-Level Distribution

The results indicate that the skill-level distribution is better for Conditions 1 and 3. In Condition Groups 1 and 2, which are divided by the question time, these results can explain the much better answers for Condition 1. However, for Condition Groups 3 and 4, also divided by the question time, these results do not explain the answer quality differences, because Condition 4 has more correct answers. Furthermore, the three participants (experts) who selected *none* for the Java skill level are in Conditions 2 and 4, both in the question-before condition. Table 6.20 (see page 155) presents the answer distribution for the question-before and -after groups across different dimensions. The experts in the question-before group performed well, which is an indicator that either the three experts with no Java knowledge

managed to solve the comprehension questions or their influence was not as high as expected.

Answers and ANOVA

With these general contemplations in mind, more specific analyses are needed to explain particular situations within the conditions. Of interest are the groups and comparisons shown in Table 6.3, which also lists the rated answers and the calculated values for the ANOVA tests per comparison. A detailed analysis of novices and experts, as well as the time of the comprehensions questions, follows for the hypotheses $H_{Performance-Experts-Novices}$ (see Section 6.2.4) and $H_{Comprehension-Question-Time}$ (see Section 6.2.5).

Table 6.3: Rated answers for various groups and the calculated ANOVA results.

	Groups	Rated Answers		F	p	n^2
a)	non-smell$_{all}$ vs. smell$_{all}$	58.86	**64.55**	F(1, 86) = 0.346	0.558	0.0040
b)	non-smell$_{Rec}$ vs. smell$_{Rec}$	47.73	**48.64**	F(1, 42) = 0.004	0.004	0.0001
c)	non-smell$_{Veh}$ vs. smell$_{Veh}$	70.00	**80.45**	F(1, 42) = 0.678	0.415	0.0159
d)	Rectangle$_{all}$ vs. Vehicle$_{all}$	48.18	**75.23**	F(1, 86) = 8.598	0.004**	0.0910

Significantly Different for Rectangle and Vehicle Groups

The data suggests that the assumption that source code examples with a smell are responsible for more incorrect answers is not correct. The analysis of Comparisons a), b), and c) indicates that the source codes with a smell have more correct answers, which is a direct contradiction of the assumption for this hypothesis. Row b) visualizes the fact that the Rectangle source code example seems to be more difficult overall, because neither the non-smell nor the smell version of the stimulus had more than 50 % correct answers. These effects are explainable with the carryover effect, which plays a crucial role in this experimental design. The Rectangle source code was always presented as the first stimulus, followed by the Vehicle source code. The non-smell and smell, as well as the question-after and -before conditions, were mixed, but the order of the primary source code categories never changed. Thus, as explained earlier for the tables with the answers per condition, a participant could have anticipated that the second comprehension question for the Vehicle code may target the same context as the question for the Rectangle source code. This can be a strong effect and is substantiated with the data of Row d). Putting the smell and non-smell differentiation aside, the comparisons of the Rectangle vs. Vehicle group revealed the only significant result of all ANOVA tests in this hypothesis: There is a significant difference between the answers for the Rectangle and those for the Vehicle source code, which is supported by the results for the rated answers in the table.

Code Smell has no Influence on Answer Quality

This analysis answers the assumption stated in the hypothesis: A code smell, as implemented in this study as an idiomatic code smell, does *not* influence answer quality regarding correct and incorrect answers. The effects observed and the significance calculated are considerably better explainable with the carryover or training effect of participants between

the Rectangle and Vehicle source code examples. Thus, the results do *not* confirm the initial hypothesis.

Table 6.4: Participants' answers for the first condition (Rectangle non-smell, Vehicle smell, question after).

Answers for the **first condition**
(Rectangle non-smell, Vehicle smell, question after)

			Rectangle		Vehicle	
#	PID	Group	Participant Answer	Coded Answer	Participant Answer	Coded Answer
1	1	Student	100	0	10	100
2	9	Student	20	70	210	70
3	13	Student	40	100	10	100
4	17	Student	20	70	10	100
5	21	Student	?	0	10	100
6	25	Student	00,10,10,10	0	10	100
7	29	Student	40	100	10	100
8	33	Student	100	0	10	100
9	45	Professional	N/a	0	10	100
10	49	Professional	width+width+ height+height	30	10	100
11	53	Professional	N/a	0	210	0
		AVG (SD) for coded answers:		**33.63 (47.36)**		**88.18 (34.25)**

Table 6.5: Participants' answers for the second condition (Rectangle non-smell, Vehicle smell, question before).

Answers for the **second condition**
(Rectangle non-smell, Vehicle smell, question before)

			Rectangle		Vehicle	
#	PID	Group	Participant Answer	Coded Answer	Participant Answer	Coded Answer
1	2	Student	perimeter	20	10	100
2	10	Student	0	0	30	0
3	18	Student	40	100	10	100
4	22	Student	40	100	10	100
5	26	Student	40	100	10	100
6	30	Student	40	100	10	100
7	34	Student	height: hint x2 - hint x1	0	This.current Speed = 0;	0
8	36	Student	Umfang des Rechtecks	20	Geschwindigkeit in km/h	0
9	44	Professional	20	70	10	100
10	48	Professional	20	70	10	100
11	54	Professional	40	100	10	100
		AVG (SD) for coded answers:		**61.82 (44.29)**		**72.73 (39.34)**

Table 6.6: Participants' answers for the third condition (Rectangle smell, Vehicle non-smell, question after).

Answers for the **third condition**
(Rectangle smell, Vehicle non-smell, question after)

			Rectangle		Vehicle	
#	PID	Group	Participant Answer	Coded Answer	Participant Answer	Coded Answer
1	3	Student	20	70	210	70
2	11	Student	0	0	this. + km	0
3	19	Student	40	100	10	100
4	23	Student	-40	80	10	100
5	27	Student	-	0	10	100
6	35	Student	40	100	200	0
7	37	Student	intvalue3	10	currentspeed	0
8	42	Student	-	0	aktuelle Geschwindig-keit	0
9	46	Professional	Differenz der x und y Werte multiplziert mit 2 und summiert	0	10	100
10	50	Professional	N/a	0	currentSpeed	0
11	55	Professional	40	100	10	100
		AVG (SD) for coded answers:		**41.82 (46.37)**		**51.82 (47.56)**

Table 6.7: Participants' answers for the fourth condition (Rectangle smell, Vehicle non-smell, question before).

Answers for the **fourth condition**
(Rectangle smell, Vehicle non-smell, question before)

			Rectangle		Vehicle	
#	PID	Group	Participant Answer	Coded Answer	Participant Answer	Coded Answer
1	8	Student	20	70	10	100
2	12	Student	2	0	210	70
3	16	Student	40	100	10	100
4	20	Student	40	100	10	100
5	24	Student	20	70	10	100
6	28	Student	3	0	10	100
7	32	Student	0	0	10	100
8	40	Student	20	70	10	100
9	43	Professional	40	100	10	100
10	47	Professional	k.A.	0	currentSpeed	0
11	51	Professional	40	100	10	100
		AVG (SD) for coded answers:		**55.45 (46.29)**		**88.18 (34.25)**

6.2.2 RATIO FOR DURATION AND FIXATIONS ($H_{Ratio-Duration-Fixations}$)

Code Smells and Comprehension Process

This hypothesis answers the question of whether a code smell influences the reading and comprehension processes, measured by the *duration* and *fixation count* eye movement metrics. If that is the case, then this effect must be observable in altered fixation counts and longer durations on source code examples with a code smell, compared to the source codes without a smell. The general idea behind this assumptions is that a code smell needs more focus in terms of more eye movement data fixations and longer fixation durations. In addition, the ratio of non-smelly and smelly AOIs was analyzed for both the fixation count and duration eye movement metrics. In most cases, these metrics are more reliable because they take into account the different AOIs for smelly and non-smelly regions of the source code examples, while the overall fixation counts and durations only consider the data for the complete stimulus. The assumption is that a differentiated analysis is only possible with the analysis of the AOIs to determine whether the smelly AOIs are responsible for an increase in duration or fixation count. The following analysis is based on comparing different groups to obtain information from different perspectives.

Duration and Fixation Metrics

Table 6.8 lists the values for the average and standard derivation for the duration and fixation count metrics. The duration data and fixation data have a relation in the sense of a proportionality. The reason is that in many cases, a higher fixation count results in a higher duration overall because every fixation allocates an amount of viewing time that the participant spends on a fixation. On average, more fixations will result in an overall higher duration. Exceptions are participants with fewer fixations and longer durations or many fixations with fewer durations. The first case can occur when a participant focuses on specific parts of the stimulus most of the time. The second case is possible when fast and many saccades are present in the data, resulting from looking all over the stimulus fast. These viewing strategies are on average rare. Furthermore, an unusual combination of duration times and fixation counts can be the result of incorrect data when gaze points are missing. Especially malformed cases will be detected in the qualitative analysis after every recording, when the data quality is assessed. Nevertheless, in the following, both duration data and fixation data are reported to verify this proportionality and to look for cases in which many fixations but a small duration is detected. This can indicate unusual viewing strategies.

Higher Values for Non-Smelly Stimuli

The data for the Rectangle source code example contradicts the assumption that source code with a smell is responsible for overall longer durations and more fixations, because higher values were observable for the non-smell variant. However, it must be noted that the differences between the Rectangle source code with and without a smell are not distinctive. It is conceivable that other effects are accountable for these small differences. Interestingly, the Vehicle source code example shows the opposite behavior: The data indicates that the code variant with a smell had a longer duration and more fixations overall. Moreover the

values for the source code with a smell are more than 30 % larger than the values for the code without a smell.

Higher Values for Vehicle Stimulus

Furthermore, the Vehicle source code has generally higher values for the duration and fixations, which implies that this source code example was more difficult for the participants to process. This result could have led to more incorrect answers for the Vehicle source code, which is not the case, according to the analysis of the previous hypothesis. The carryover or training effect could account for this observable effect. However, the higher values for the smelly Vehicle source code can be attributed to the source code smell because a smell can account for more visual effort, even if the answers are correct in the end.

Table 6.8: The average and standard derivation for the duration and fixation of both source code examples (divided by smell and non-smell variants).

	Duration		Fixation	
Source Code	AVG	SD	AVG	SD
$\text{Rectangle}_{\text{non-smell}}$	**68.91**	31.06	**231.05**	106.36
$\text{Rectangle}_{\text{smell}}$	63.36	41.83	211.55	126.10
$\text{Vehicle}_{\text{non-smell}}$	71.05	41.62	252.73	147.75
$\text{Vehicle}_{\text{smell}}$	**98.36**	47.73	**341.95**	146.75

ANOVA Tests for Smelly and Non-Smelly Code Examples

As a subsequent analysis, the correlations between the non-smelly and smelly source code examples were analyzed based on ANOVA tests. Tables 6.9 and 6.10 display the data for the duration and the fixations respectively. As both tables show, in Row a), no significant correlation exist between the non-smelly and smelly groups over all source code examples. In both cases, this contradicts the hypothesis that a source code smell is responsible for altering the reading behavior of participants regarding the duration and fixations overall. Row d) of the aforementioned tables also indicates that there is a significance for both Rectangle and Vehicle source code examples for the duration and fixation calculations. This supports the previous analysis that the difference is based on the source code examples themselves, and in this case on the Vehicle code, supported with the data in Table 6.8. Moreover, the tables present the comparisons of the non-smell and smell groups for the Rectangle and Vehicle source codes in Rows b) and c) respectively. For duration, there is a significance for the Vehicle source code, whereas for fixation, while the significance is also present, it is smaller. The data supports the result that the effect based on a source code smell, if one is detectable at all, is measurable for the Vehicle source code example. Especially Row c) in both tables consolidates the impression because a significant connection seems to exist between non-smelly and smelly source code for the Vehicle code. These results are interesting because they argue in favor of the increased effort for the Vehicle source code example. However, the results for answer quality do not support this disguise by the carryover effect. The remaining question is whether these findings can be supported by an analysis of the duration

and fixation ratios, which is the goal of the following analysis. The values for the Rectangle source code (non-smelly vs. smelly) are in favor of the non-smell variants for the duration and fixation data. Even if the differences are small, this effect could have occurred because the Rectangle code was always the first code example. It is thus difficult to differentiate what the source for the observable data is.

Table 6.9: The distribution for the duration (overall) per group, as well as the calculated ANOVA results.

	Groups	Duration		F	p	n^2
a)	non-smell$_{all}$ vs. smell$_{all}$	69.98	**80.86**	F(1, 86) = 1.449	0.232	0.0166
b)	non-smell$_{Rec}$ vs. smell$_{Rec}$	**68.91**	63.36	F(1, 42) = 0.249	0.620	0.0059
c)	non-smell$_{Veh}$ vs. smell$_{Veh}$	71.05	**98.36**	F(1, 42) = 4.094	0.049*	0.0888
d)	Rectangle$_{all}$ vs. Vehicle$_{all}$	66.14	**84.70**	F(1, 86) = 4.356	0.040*	0.0482

Table 6.10: The distribution for the fixation (overall) per group, as well as the calculated ANOVA results.

	Groups	Fixation		F	p	n^2
a)	non-smell$_{all}$ vs. smell$_{all}$	241.89	**276.75**	F(1, 86) = 1.373	0.245	0.0157
b)	non-smell$_{Rec}$ vs. smell$_{Rec}$	**231.05**	211.55	F(1, 42) = 0.307	0.582	0.0073
c)	non-smell$_{Veh}$ vs. smell$_{Veh}$	252.73	**341.95**	F(1, 42) = 4.039	0.051.	0.0877
d)	Rectangle$_{all}$ vs. Vehicle$_{all}$	221.30	**297.34**	F(1, 86) = 6.952	0.010**	0.0748

Visual and Comprehension Effort

The assumption for the ratio of the duration and fixation metrics is that these values are a better predictor for visual and comprehension effort because they consider the non-smelly and smelly AOIs and their correlations. The following ANOVA calculations thus visualize different groups based on their duration and fixation ratios. Table 6.11 lists the data for the duration ratio, while Table 6.12 contains the data for the fixation ratio.

Table 6.11: The distribution for the duration ratio per group, as well as the calculated ANOVA results.

	Groups	Duration Ratio		F	p	n^2
a)	non-smell$_{all}$ vs. smell$_{all}$	1.01	**1.20**	F(1, 86) = 0.941	0.335	0.0108
b)	non-smell$_{Rec}$ vs. smell$_{Rec}$	1.14	**1.39**	F(1, 42) = 0.480	0.492	0.0113
c)	non-smell$_{Veh}$ vs. smell$_{Veh}$	0.89	**1.01**	F(1, 42) = 0.813	0.372	0.0190
d)	Rectangle$_{all}$ vs. Vehicle$_{all}$	**1.26**	0.95	F(1, 86) = 2.578	0.112	0.0291

No Significant Differences According to ANOVA

The first result is that there is no significant difference between the various groups, according to the ANOVA tests. This result contradicts the above-mentioned hypothesis. Even when the ratios between non-smelly and smelly AOIs are taken into account for both eye movement metrics, no significant difference is observable in the data. Otherwise, the distribution of the duration and fixation ratios in the tables indicate that the ratio is always higher for the smelly variants of the groups. Whether it be the group for all source codes

Table 6.12: The distribution for the fixation ratio per group, as well as the calculated ANOVA results.

	Groups	Fixation Ratio		F	p	n^2
a)	non-smell$_{all}$ vs. smell$_{all}$	0.99	**1.10**	F(1, 86) = 0.553	0.459	0.0064
b)	non-smell$_{Rec}$ vs. smell$_{Rec}$	1.08	**1.19**	F(1, 42) = 0.174	0.679	0.0041
c)	non-smell$_{Veh}$ vs. smell$_{Veh}$	0.90	**1.02**	F(1, 42) = 0.635	0.430	0.0149
d)	Rectangle$_{all}$ vs. Vehicle$_{all}$	**1.14**	0.96	F(1, 86) = 1.367	0.246	0.0156

– the a) rows – or the Rectangle or Vehicle groups – the b) and c) rows – the duration and fixation ratio is always higher for the smelly groups, which is as originally assumed for this hypothesis (i.e., that a source code smell influences participants' reading behavior, detectable with the duration and fixation eye movement metrics). In summary, there are no significant differences; however, the data suggests that, according to the ratios, the smelly variants increased the visual effort of participants.

More Effort for the Vehicle Stimulus

As a small recourse to the carryover effect of the Rectangle and Vehicle source code examples, the data for the overall duration and fixations in Tables 6.9 and 6.10 suggests more effort for the Vehicle source code in both cases. In contrast, the differences between the Rectangle and Vehicle source code examples in both Tables 6.11 and 6.12 indicate that the Rectangle code accounts for the larger values of duration and fixation ratios. While the overall duration and fixations are prime predictors, the ratios of the non-smelly and smelly AOIs are not well suited to explain carryover effects. The effort for the Vehicle source code variants is higher, yet more correct answers were given by the participants. In summary, the eye movement metrics duration and fixations (ratios) revealed differences between the source code examples and the non-smelly and smelly variants. The data supports the assumption that a source code with a smell accounts for a longer duration and more fixations measured overall and by ratios. However, these effects cannot explain the answer quality results of the previous hypothesis (Vehicle has more correct answers overall). Therefore, these results confirm the initial hypothesis *partially*.

6.2.3 PATTERN READING BEHAVIOR ($H_{Pattern-Reading-Behavior}$)

Common Eye Movement Patterns

To substantiate the assumption that a code smell influences the reading behavior of participants, an analysis of common eye movement patterns within the recorded eye movement data is necessary. The SOR, EOR, and flicking patterns are especially interesting because they indicate common reading patterns, for example obtaining an overview of the source code (SOR), following the execution of the program flow (EOR), and moving back and forth between the declaration and initialization of a variable and its usage (flicking). The last pattern is of particular interest because the code smell in the Rectangle source code example aims to blot out the readiness of the variables. The assumption is that a source code

smell prevents, or at least hinders, participants' ideal-typical reading behavior. The analysis focuses on the differences between smelly and non-smelly source code examples across participants. The developed visualization tool, CodeSight (Deitelhoff et al., 2019b), was used in this analysis to gain insight into eye movement patterns, as well as to have the ability to search for particular patterns, based on regular expressions, in the data. The analyses are based on the AOI line model, because the AOIs in the region model do not allow the analysis of line transitions in the eye movement data.

SOR and EOR Patterns

The SOR pattern is defined as linear reading from the top of the source code examples to the bottom. Furthermore, the EOR pattern is defined for the explicit execution order of the source code Patterns examples, based on the data provided for the participants in the source code comprehension questions. Both the SOR and EOR patterns are thus deterministic.

SOR AND EOR PATTERNS – VISUALIZATIONS

Color Gradients for Visualizations

As the first step, the reading behavior with the SOR and EOR patterns was analyzed. Both patterns occurred frequently in the eye movement data so that they could be spotted easily with an AOI-DNA visualization. To highlight these patterns, different color gradients were used based on the AOI sequences per source code example in Table 6.13. These sequences are directly derived from the AOI definitions shown in the four Figures 6.8, 6.9, 6.10, and 6.11 (pages 129, 130, 130, and 131). The SOR pattern is derived from the top to bottom sequential reading strategy, and the EOR pattern is based on the execution order of the source code with the given parameters in the main method.

Table 6.13: SOR and EOR eye movement patterns encoded as sequences of letters based on the AOIs per source code stimulus.

	SOR	EOR
Rectangle non-smell	ABCDEFGHIJKLMNOPQRS	OPCBDEFGHQIJKLMNQRS
Rectangle smell	ABCDEFGHIJKLMNOPQRS	OPCBDEFGHQIJKLMNQRS
Vehicle non-smell	ABCDEFGHIJKLMNOPQRST	PQDBCEFGHIRJKLMNORST
Vehicle smell	ABCDEFGHIJKLMNOPQRSTU	QRDBCEFGHISJKLMNOPSTU

Similarities and Dissimilarities

This table also emphasizes the similarities between both Rectangle source code examples and the dissimilarities between both Vehicle source code examples: The code smell used for the Rectangle code changes the contents of source code lines, so that the AOI count remains the same, and the code smell used for the Vehicle source code changes the code structure and introduces one additional AOI. First, the patterns were inspected visually with CodeSight, and this led to the results listed in Table 6.14.

Fewer SOR Patterns

As the data indicates, fewer SOR patterns were observed for source code examples with a code smell, which is in line with the hypothesis. The SOR pattern is especially important at the beginning of a source code comprehension task, to obtain an overview of the overall

Table 6.14: Visually identified SOR and EOR patterns for the source-code-varying examples.

	SOR		EOR	
	non-smelly	smelly	non-smelly	smelly
Rectangle	16	11	7	4
Vehicle	14	9	6	5

code. According to Hidetake et al. (2007), the first 30 % of the overall reading time in a source code comprehension task is dedicated to a top-to-bottom reading to obtain a first overview of the source code. This supports the assumption that a smell breaks or hinders this essential reading pattern for understanding the source code.

Qualitative Analysis of Patterns

Deciding whether an eye movement sequence is a SOR or EOR pattern is a delicate process. For this analysis, two researchers individually judged every AOI-DNA on whether it is more likely to be a SOR or EOR pattern or whether the sequence is neither of the two. Figure 6.13 illustrates how individual these sequences and thus the reading processes can be. This AOI-DNA plot, created with TraMineR, uses a grayscale color gradient to visualize the SOR pattern. As the figure shows, the 22 sequences for the Rectangle non-smelly source code examples, based on the line AOI model, are highly individual. The legend for the AOI gray scale color coding is available in Figure 6.12 and the base line for all AOI visualizations with a grayscale.

Figure 6.12: The legend for the mapping of AOIs to the grayscale color coding.

Optimal Matching for Sequence Similarities

The similarities between the sequences were determined using the OM algorithm. Figures 6.14, 6.15, and 6.16 illustrate the three clusters, which are as difficult to compare as those in Figure 6.13. Only the first cluster, with 19 sequences, displays some similarities between some of the eye movement sequences. Overall, this is highly inconclusive and is a contrast to the clusters visualized for the EMIP dataset analysis, starting with Figure 5.13 (see page 101). Apart from the Rectangle non-smelly source code, the Rectangle smelly and the Vehicle non-smelly and smelly source codes, all for the line AOI model, were also tested, and they all yielded the same result: The visualizations portrayed highly individual eye movement sequences.

The assumption was that there is a difference between the non-smelly and smelly source code examples; however, this could not be found. In general, the eye movement sequences differ from each other. One explanation is that this is in fact the result of the smell, combined with the question time condition, but this is highly speculative and cannot be falsified from the data.

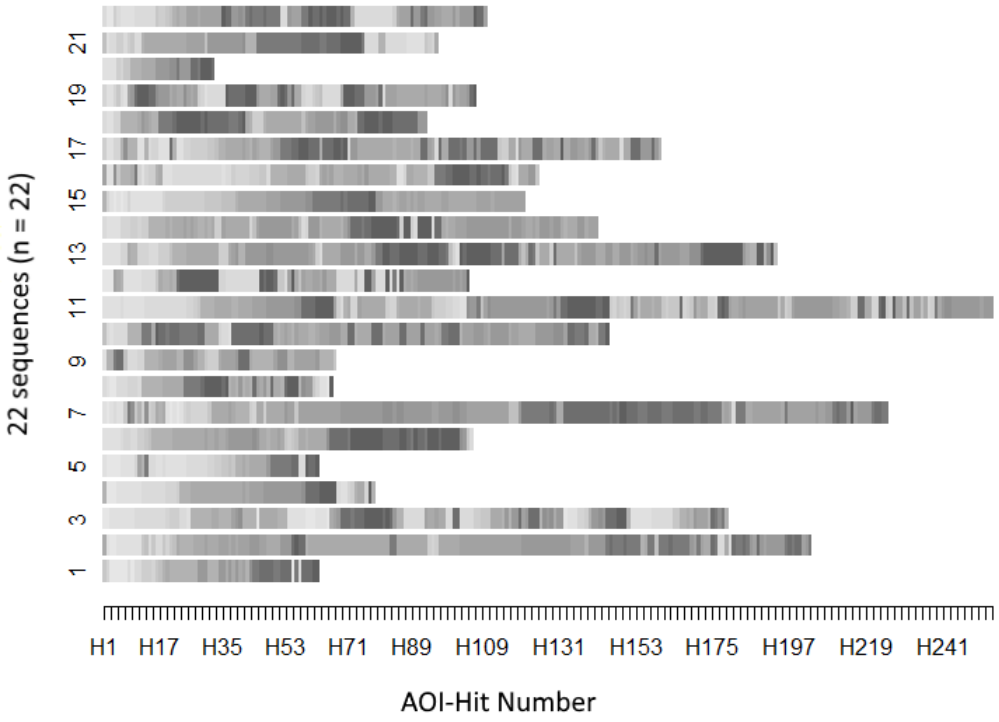


Figure 6.13: Twenty-two clustered sequences for the Rectangle non-smelly source code example.

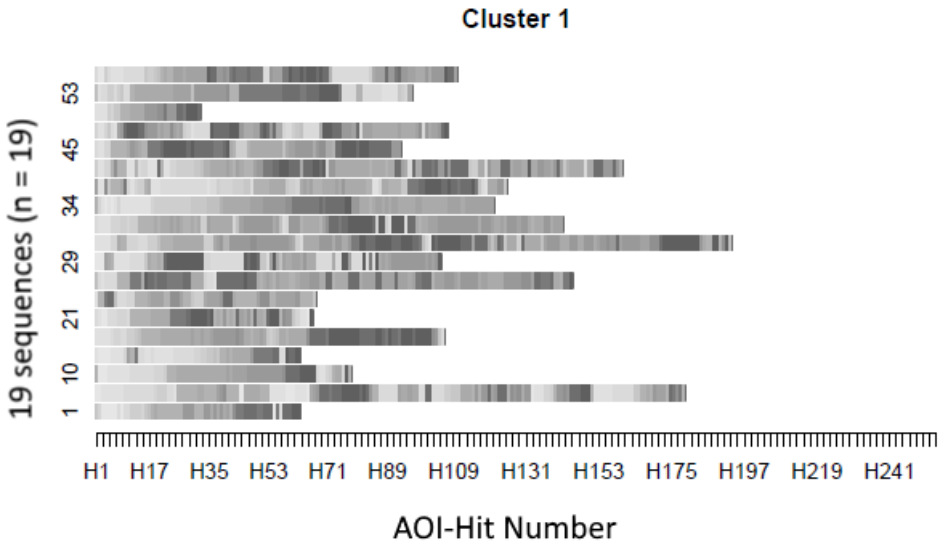


Figure 6.14: The first cluster of the sequences for the Rectangle non-smell source code example.

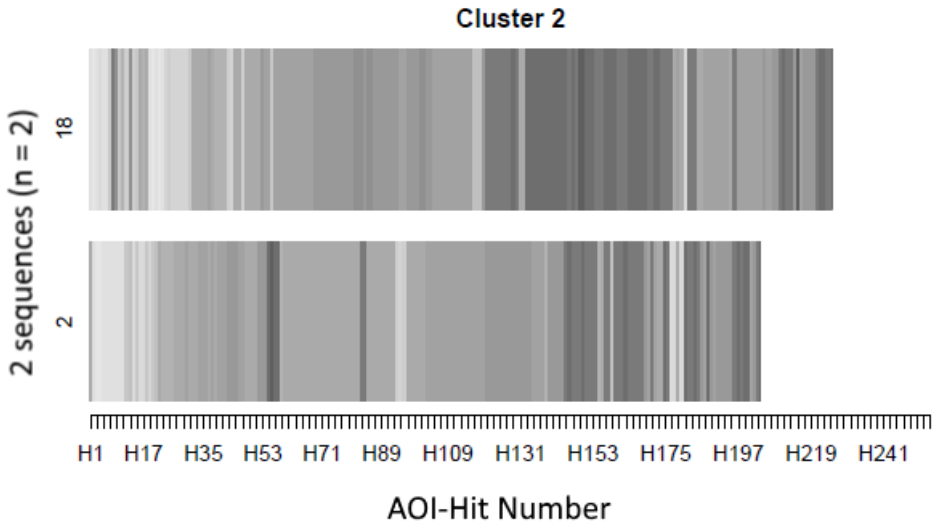


Figure 6.15: The second cluster of the sequences for the Rectangle non-smell source code example.

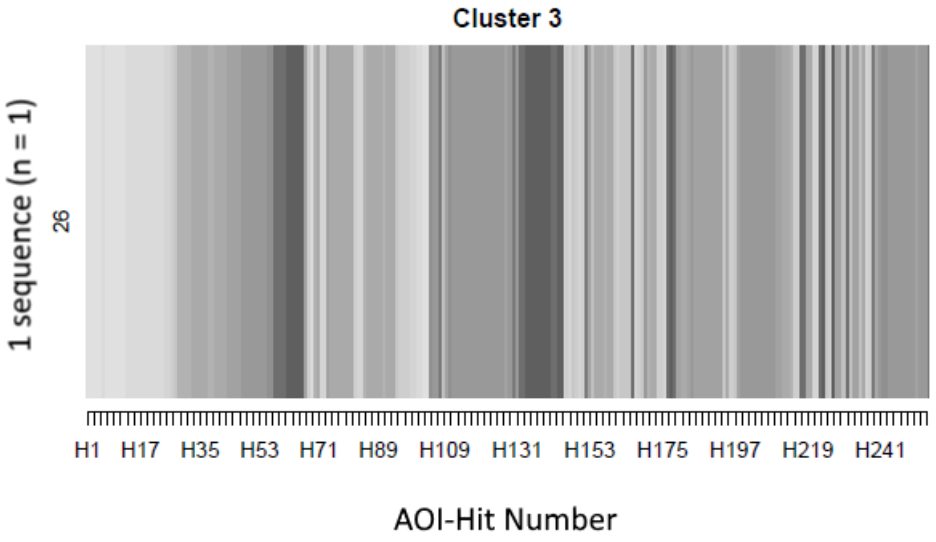


Figure 6.16: The third cluster of the sequences for the Rectangle non-smell source code example.

SOR AND EOR PATTERNS – SEARCHES

Pattern Search with Regular Expressions

For the next part of the analysis, the pattern search in CodeSight was used to search for EOR patterns in the eye movement data. This search was done with a variation of the previously defined EOR patterns, which were shortened to the core of the source code examples, consisting of the main method and the method with the domain logic, which in this case are the calculation methods (Rectangle) and the conditional statements (Vehicle). Table 6.15 lists the important parts of the EOR patterns for both source codes, as well as the non-smelly and smelly variants. The black parts of the EOR string in the table indicate the parts for the pattern matching, and the last column displays the number of matches of each pattern search. Such regular expressions are built like the following example: `Q(?:.{1,5})I(?:.{1,5})J(?:.{1,5})K(?:.{1,5})L` The letters correspond to the AOIs for the shortened EOR pattern (in this example for Rectangle non-smell). The interval expressions `{1,5}` are defining the AOIs between the necessary ones and that there can be between one and five non-relevant AOIs. Finding a pattern with the exact sequence of AOIs without the intervals in between is very unlikely, which makes it necessary to specify the intervals. Thus, the other patterns in this study and the following analyses are defined in the same way.

Table 6.15: Simplified patterns (based on the complete EOR patterns shown in table 6.13 on page 141) and corresponding matches per source code example. The black sub-strings were used for the regular expressions.

	EOR	# Pattern Matches
Rectangle non-smell	OPCBDEFGHQIJKLMNQRS	1
Rectangle smell	OPCBDEFGHQIJKLMNQRS	0
Vehicle non-smell	PQDBCEFGHIRJKLMNORST	0
Vehicle smell	QRDBCEFGHISJKLMNOPSTU	0

Aggregated AOI-DNAs

As an additional adjustment to the prior visual analysis, an aggregation of the AOIs was implemented in CodeSight. The aggregation combines consecutive AOIs with the same letter into one single AOI. From an eye movement data perspective, this means that a participant's dwell time on an AOI, which leads to multiplying this AOI in the resulting sequence of letters, is combined into a single AOI with the combined dwell time of all single AOIs. Figure 6.17 illustrates the aggregation step as an example. The results are shortened AOI-DNAs, which can more easily be analyzed with the pattern search. This aggregation was not applied for the qualitative visual analysis, because an aggregated AOI-DNA can disguise the pattern for the human eye when large parts of the same AOI are aggregated into a single AOI. With this aggregation, missing important pattern matches of the EOR eye movement pattern is more unlikely. In contrast, a pattern search for the SOR pattern is not valuable, because the pattern itself is too large, since every AOI is included within the pattern and thus in

a positive match. Such a pattern is unlikely to be found in real-world eye movement data recordings (e.g., due to noise).

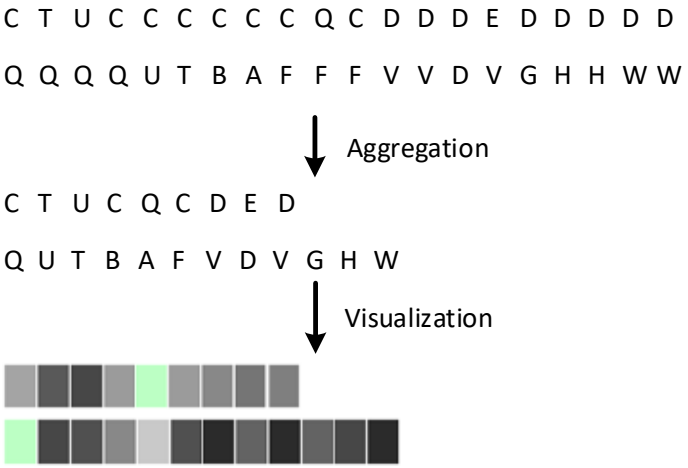


Figure 6.17: The AOI aggregation to combine consecutive AOIs to shorten the sequence.

Searching for Ideal Eye Movement Patterns

The previous results suggest that searching for an ideal eye movement pattern is too shortsighted and cannot hold against the reality of recorded eye movement data. A pattern-matching algorithm is not sufficient to account for all the variance in an eye movement sequence. After finding global patterns, such as SOR and EOR, visually and, in the case of EOR, with a pattern-matching algorithm on aggregated AOI-DNAs, the focus shifts to reading behavior, localized in the area of important source code regions. This optimal or most helpful reading sequence is expressible with a short AOI sequence, which is a better candidate for a pattern search compared to idealistic patterns such as EOR. Table 6.16 presents these patterns for every stimulus and the pattern matches accordingly. For the Rectangle source code, this is the perimeter method, which is called in the main method. For the Vehicle source code, this is the `accelerate` method, which is called in the main method. The basis for these patterns are visualized in Figure 6.18.

Table 6.16: Short reading sequences and the corresponding pattern matches per source code example.

	Pattern	# Pattern Matches
Rectangle non-smell	QI	8 of 22
Rectangle smell	QI	7 of 22
Vehicle non-smell	RJ	5 of 22
Vehicle smell	SJ	4 of 22

The detected pattern matches indicate that some participants use these specific reading patterns. Overall, the found matches are not significant enough to explain better comprehension answers or other effects, such as the comprehension time.

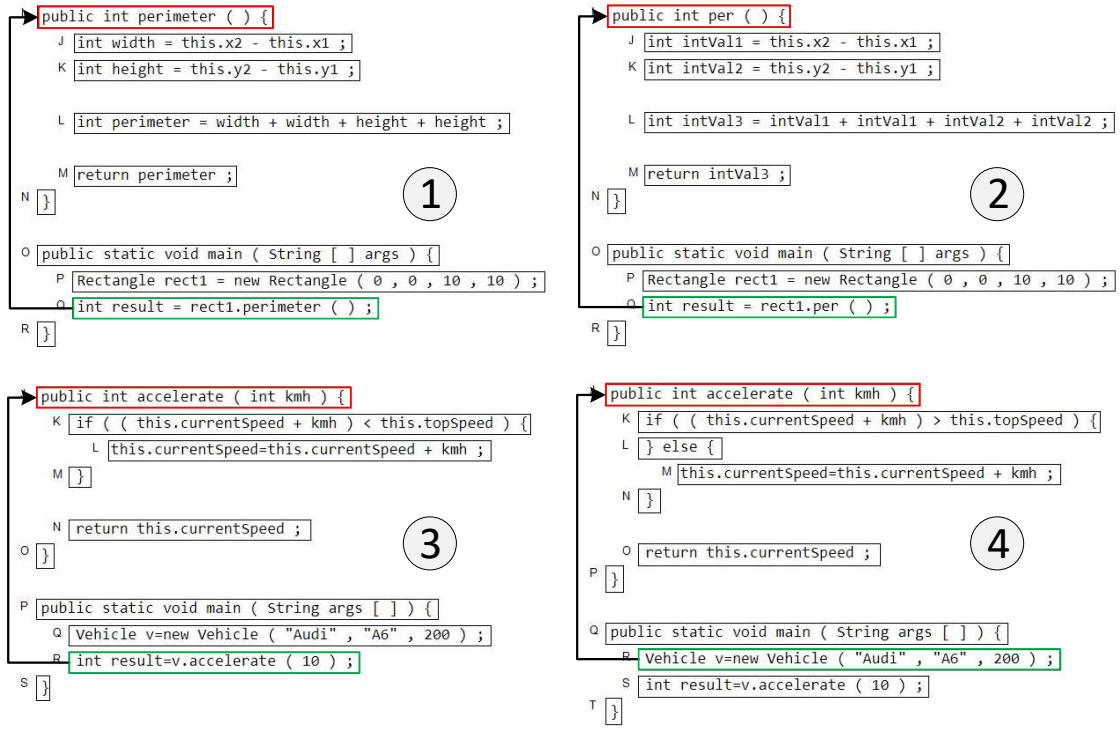


Figure 6.18: The reading sequences used for the patterns of table 6.16 and the resultant regular expressions (1 → Rectangle non-smell, 2 → Rectangle smell, 3 → Vehicle non-smell, and 4 → Vehicle smell).

CODE SMELLS AND PATTERNS

Patterns Depending on Code Smells

As the next part of the analysis, specific patterns connected to the code smells were assessed. The overall assumption was that a code smell changes the reading behavior in an observable way, which can result in discontinuing common reading patterns, such as SOR or EOR, because of the smell. Another influence that a code smell can have on eye movement patterns is that specific reading patterns emerge because of the smell. To test the existence of patterns created when viewing the code smells, the pattern matching in CodeSight was used to find the patterns listed in Table 6.17. The table also contains the associated pattern matches found in the analysis. For the Rectangle source code example, the following pattern describes the calculation in the `perimeter` method, and for the Vehicle source code, the pattern describes the `if-else` structure. The difference is created by the smell. The basis for these patterns are visualized in Figure 6.19.

Ambiguous Results

The results indicate that the defined patterns have no strong relation to a code smell; this is especially true for the Rectangle smelly code example, where no pattern match at all could be found. This analysis is interesting because the pattern is the same for Rectangle non-smell and Rectangle smell. After all, the smell alters the contents of the lines and not the structure. The assumption was that the pattern should be present more often in the smell conditions, which applies to the Vehicle smell stimulus as well. However, only two matches could be

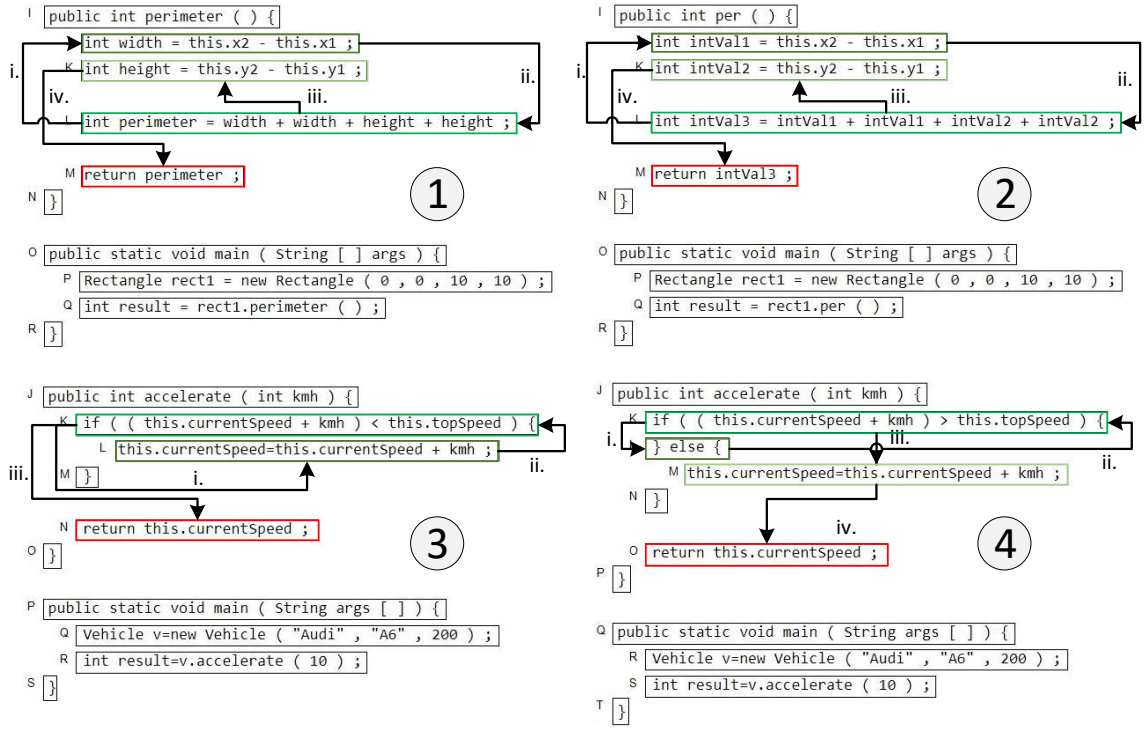


Figure 6.19: The reading sequences used for the patterns of table 6.17 and the resultant regular expressions (1 → Rectangle non-smell, 2 → Rectangle smell, 3 → Vehicle non-smell, and 4 → Vehicle smell).

Table 6.17: Eye movement patterns and corresponding pattern matches based on the source code smells per source code example.

	Pattern	# Pattern Matches
Rectangle non-smell	LJLKM	2 of 22
Rectangle smell	LJLKM	0 of 22
Vehicle non-smell	KLKN	3 of 22
Vehicle smell	KLKMO	2 of 22

found there, and compared to the three the non-smell Vehicle stimuli, this eye movement pattern is not an explanation or distinguishing factor for the existence of a code smell.

Flicking and Retrace Declaration Patterns

Another means of detecting differences in eye movement patterns involves recurring AOI-Hits or transitions. Both phenomena can create specific patterns. Referring to the EMIP working group and the coding scheme proposed by this group, a search for a flicking pattern within the recorded eye movement data was conducted. Flicking occurs when one's gaze moves back and forth between two related items, which could be between a method call and the method definition or the formal and actual parameter list of a method call (EMIP coding scheme). The retrace declaration pattern is a form of flicking where the eye movement jumps back and forth between the variable definition and places where the variable is used. The assumption was that both patterns are also observable for if-conditions, specifically between the definition of the if, the body, the else, and the body of the else block. Table 6.18 presents the specific AOI patterns for flicking for every source code example and

the pattern match occurrences found with CodeSight. The patterns describe the definition and usage of a `variable` in the Rectangle source code example (`perimeter` method) and the `if-else` structure for the Vehicle example. The basis for these patterns are visualized in Figure 6.20.

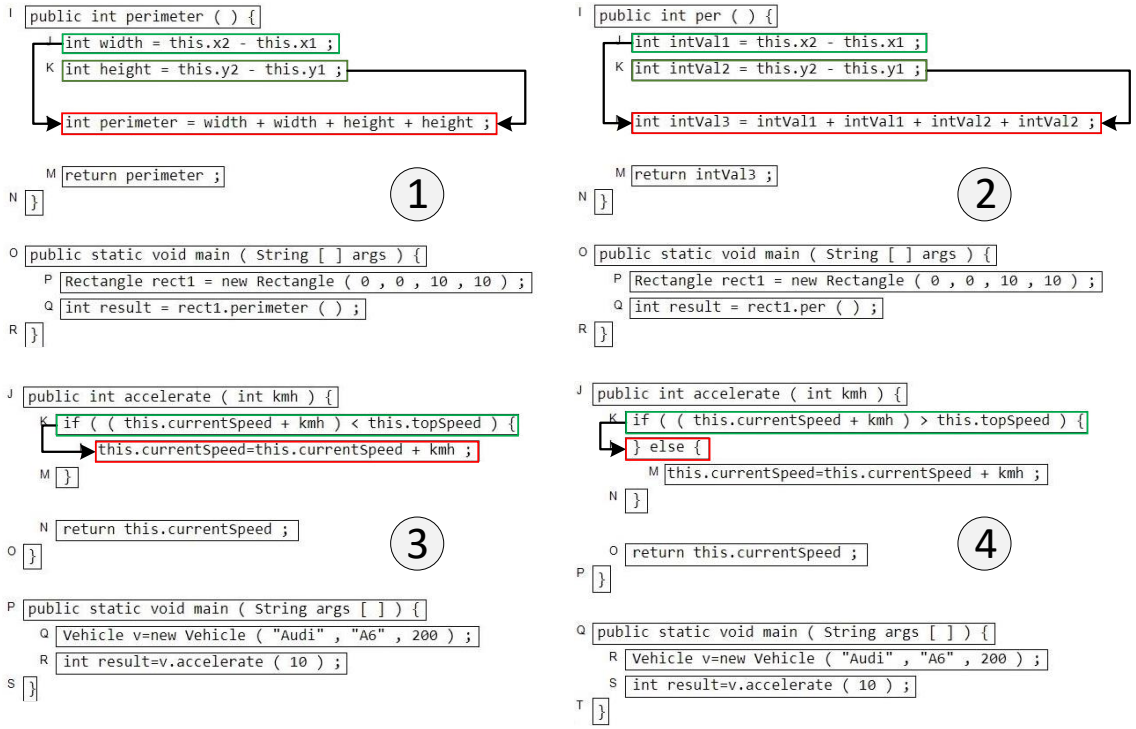


Figure 6.20: The reading sequences used for the patterns of Table 6.18 and the resultant regular expressions (1 → Rectangle non-smell, 2 → Rectangle smell, 3 → Vehicle non-smell, and 4 → Vehicle smell).

Table 6.18: Eye movement pattern definition for the flicking pattern with the corresponding pattern matches per source code example.

	Pattern	# Pattern Matches
Rectangle non-smell	JL	9 of 22
	KL	7 of 22
Rectangle smell	JL	9 of 22
	KL	6 of 22
Vehicle non-smell	KL	9 of 22
Vehicle smell	KL	8 of 22

Non-Important Differences

The table lists two patterns for Rectangle non-smelly and Rectangle smelly source codes because the code smell in the Rectangle stimulus was spread over three lines – two with declaring and initializing variables and one with the calculation of a result with these variables. Therefore, both lines need to be checked with declaring variables, each connected to the calculation. The results in Table 6.18 indicate that a high number of participants made

the flicking eye movement pattern to comprehend the important source code lines. However, in more detail, the difference between the non-smelly and smelly source codes is not as important as initially assumed. Although the data suggests that the analyzed lines of the source codes are indeed important, the smell did not alter the viewing behavior according to the flicking eye movement pattern.

Flicking Pattern not Consistent

Furthermore, the flicking reading pattern of participants was analyzed for both presented source code examples. Every participant encountered two source code stimuli: one with and one without a smell. If the flicking eye movement pattern is of high importance, then the assumption is that a participant displaying the flicking pattern for the first code example utilizes this pattern again for the second source code. The analysis revealed two participants who did this for the Rectangle non-smelly and Vehicle smelly source codes and two participants for the Rectangle smelly and non-smelly Vehicle codes. They make up 20 % of the participants, compared to the overall number of participants making the flicking eye movement pattern. It can thus be assumed that the flicking pattern is not a consistent eye movement strategy in this study.

6.2.4 PERFORMANCE EXPERTS VS. NOVICES ($H_{Performance-Experts-Novices}$)

Novice and Expert Differences

This hypothesis highlights the differences between novices and experts from various perspectives. The fact that 12 professional software developers were recorded for this study makes the classification of both groups implicit because all recorded students are categorized as novices, and the professional software developers are marked as experts. The primary goal was to analyze the overall performance of both groups based on answer quality (correct or incorrect answers). The assumption was that experts provide more correct answers than novices. More generally, the assumptions were that experts demonstrate better performance, not only for answer quality but also for common eye movement metrics.

Three Metrics: Answer Quality, Duration, and Fixation Count

Three metrics are of interest for a more general review of the performance of novices and experts: a) answer quality, b) duration, and c) fixation count. The last two metrics refer to complete source code examples and not to specific AOIs or the ratio of AOIs, as analyzed in a previous hypothesis (see Section 6.2.2). The average values for the correct and incorrect answers for all source code examples and conditions are 61.25 % ($SD = 46.93\,\%$) for the expert participant group and 61.88 % ($SD = 44.79\,\%$) for the novice group. These results are interesting and, at first glance, surprising in themselves. The assumption was that experts issue more correct answers for both source code examples; however, this is not the case, according to the data. An in-depth analysis of the answer quality across the source code examples and conditions follows shortly. A common explanation for such a result between the two skill groups is that the group with more experience cannot apply their knowledge and training on a small set of source code examples used in a study scenario such as this one. Experts, on the one hand, benefit from a larger source code base, with source code

files in the hundreds of lines and many files in a more or less complex project structure. The work of Abid et al. (2019), in which novices and experts read source code to perform a source code summarization, demonstrated that "novices have longer gaze time performing bottom-up compare to experts" (Abid et al., 2019, p. 8). In such scenarios, the experts perform better than novices. Students, on the other hand, are somewhat trained to comprehend small source code examples, find bugs, or make minor modifications, which are common tasks in computer science courses and exams.

Experts are Faster

The data regarding the duration and fixation count is interesting. The assumption was again that experts perform better, which for these metrics means that they need less overall time on the source code examples (duration) and fewer fixations because they can navigate the code with less visual effort and thus faster. The assumption is true for the overall stimuli duration with the results 69.75 sec ($SD = 33.08\,sec$) for the expert group and 77.66 sec ($SD = 45.62\,sec$) for the novice group. These results indicate that the experts are overall faster and, according to the standard derivation, more stable in their performance with fewer differences between the participants. While the overall duration of a source code example is a weak indicator of an expert-level skill set, it is the first indication. A similar picture emerges when using the fixation metric and calculating and comparing the fixation counts. On average, the expert group needed 237.75 fixations ($SD = 105.33 fixations$) and the novice group 267.41 fixations ($SD = 150.95 fixations$). This result again highlights a difference between experts and novices in favor of the first group. As for the overall duration, the overall fixation count on a source code example is a small indicator of an expert-level skill set.

Regarding the answer quality of participants, Table 6.19 lists the various subgroups, which were compared with ANOVA tests, for a detailed view from different perspectives.

Table 6.19: The average answer quality of experts and novices, separated into different subgroups and compared using ANOVA tests.

		Answer Quality				
#	Condition	Experts	Novices	F	p	n^2
a)	All	61.25	**61.88**	F(1, 86) = 0.003	0.954	3.8511
b)	Rectangle	47.50	**48.44**	F(1, 42) = 0.004	0.951	8.9971
c)	Vehicle	75.00	**75.31**	F(1, 42) = 0.000	0.983	1.1259
d)	non-smell	55.83	**60.00**	F(1, 42) = 0.072	0.790	0.0017
e)	smell	**66.67**	63.75	F(1, 42) = 0.036	0.851	0.0008
f)	Question before	**78.33**	66.25	F(1, 42) = 0.703	0.406	0.0165
g)	Question after	44.17	**57.50**	F(1, 42) = 0.703	0.407	0.0165

Table 6.19 reveals that there are no significant effects, according to ANOVA, regarding the tested conditions and subgroups:

Row a) refers to the previously discussed overall differences between experts and novices, with the already explained minimal advantageous difference for novices.

Rows b) and c) present the answer quality differences for the Rectangle and Vehicle source code examples. The data reveals how close the results are between both skill groups for the two source code examples – both times with a slight advantage for the novice earlier analysis (see Section 6.2.1) that the Vehicle source code example in both variants had more correct answers compared to the Rectangle source code in both variants across both skill groups.

Rows d) and e) contain the data for the non-smelly and smelly conditions across both source code examples. As in the comparisons in Rows b) and c), the differences between experts and novices are too minor to explain the effects. For the smell condition, the experts provided more correct answers this time. However, the differences are not large enough to use the skill differences as an explanation for the data.

Rows f) and g) reveal interesting results. For the question-before condition, where the source code comprehension question was visible before the source code example, the experts gave more correct answers than the novices – this time over 10 % more, which can be seen as an important difference. For the question-after condition, the novices provided better answers. This result is fascinating because it seems that experts can use their skill the best if they know what to look for or, to put it colloquially, "if they know what there is coming," as one participant stated in the RTA. This result is an explainable behavior for experts because they are not familiar with such experiments, and in real-world projects, they look at code with a specific reason and process in mind; for example, they want to find a bug, introduce a change, or be sure that the behavior of a particular source code snippet is as they remember it. Novices are usually not accustomed to such an approach. In contrast, explaining the effect that novices perform better for the question-after condition is much more difficult. Possible explanations are that they were better at guessing the reason for the study, or their studies afforded them an advantage in working with such tasks. Finding the real cause is problematic at this point because an in-depth analysis of more subgroups for the question-before and -after conditions results in small case numbers. Conclusions drawn from such tests should therefore be questioned.

No Significant Effects with ANOVA

Surprisingly, no significant effects could be found when calculating and comparing the ANOVA results. The assumption of hypothesis $H_{Performance-Experts-Novices}$ is thus neither entirely correct, nor incorrect. Experts do not have better results regarding answer quality – especially not in general. The data in Table 6.19 suggests that experts seem to

have an advantage for the question-before study condition. Novices predominantly gave more correct answers. In contrast, experts demonstrated better performance regarding the overall duration and fixations on source code examples. Nevertheless, these results do not converge to better answers to the source code comprehension questions. As a summary of these results, three circumstances could be responsible for the reported results:

(a) Novices (students) are accustomed to performing tasks such as those in this study and answering questions for source code examples, which, for the majority of computer science courses, do not contain source code smells. Therefore, students are presumably better prepared to succeed in answering the comprehension questions in this study.

(b) Experts are more accustomed to handling larger source code repositories, with larger files, a folder structure, a more complex software architecture, and many files. Experts will thus presumably excel with an increasing complexity of source code examples, which is in line with other research results (Abid et al., 2019).

(c) Neither novices nor experts have domain knowledge of the source codes used in this study. However, four experts stated in the RTA that they have less experience with Java, which is supported by the analysis of the demographic data (see Section 6.1.6). Since the study was conducted with experts from a company providing web development with JavaScript and TypeScript on the front-end and C# on the back-end, this can be an explanation for the experts' performance.

Qualitative RTA Analysis

According to the qualitative analysis of the RTA, based on the transcribed audio recordings and conducted by two researchers independently from each other, the experts stated that even though they were not prepared for comprehending Java source code examples, that they had managed it. Furthermore, seven experts found the source code smell or could at least point out that something was wrong in the source code examples, while only three students pointed out the smell.

6.2.5 TIME OF THE COMPREHENSION QUESTION ($H_{Comprehension-Question-Time}$)

Different Question Times

The analysis of the comprehension question time is another important and interesting context of the code smell study. Many source code comprehension studies rely on the use of a comprehension question, which is visible *after* the actual source code material is presented to the participants. On the one hand, this minimizes the influence that a question can have on participants, which in turn may significantly affect the comprehension process. On the other hand, this is exactly what a study aims to do: to slightly push a participant in the correct direction; minimize the "wandering around" effect; and minimize incorrect comprehension answers, which are simply based on a participant not knowing what to expect. The

latter, however, is unlikely in real-world scenarios. Moreover, being asked a comprehension question for source code that one cannot look up is, to say at least, a strange use case in the software development world. These questions were taken into account when the code smell study was designed. The assumption was that a source code comprehension question, which is posed before the source code is visible, has a measurable effect on answer quality. As with the previous hypothesis, three metrics are of interest for a general review of the participants' performance for the *question-before* and *question-after* conditions: a) answer quality, b) duration, and c) fixation count.

The Question Time has an Influence

The average values for the correct and incorrect comprehension answers, grouped by the two mentioned conditions, were 53.86 % ($SD = 46.82\,\%$) for the question-after condition and 69.55 % ($SD = 42.42\,\%$) for the question-before condition. These results support the initial hypothesis: The time for the comprehension question has an impact on answer quality. Apart from this result, the data regarding the duration and fixation count shows inconclusive results. The average value for the question-before condition for duration was 77.23 sec ($SD = 42.02\,sec$), while the value for the question after was 73.61 sec ($SD = 43.44\,sec$). Furthermore, the average value for the fixation count was 272.43 sec ($SD = 143.50\,sec$) for the question-before condition and 246.20 sec ($SD = 136.44\,sec$) for the question-after condition. The results reveal higher values for the question-before condition, which is in contrast to the assumption. However, knowing the comprehension question beforehand could have led to a more thorough reading process to be able to answer the question, whereas not knowing the question could have led to participants not knowing how to proceed and move forward in the study. However, any further analysis is not possible with the available data, because detecting such behavior and distinguishing it from noise in the data is problematic. The overall duration indicates no measurable effect because the difference is too small and can be considered as noise over all recorded participants and source code examples. Moreover, the results for the fixation count indicate a larger effect for the question-before condition. Nevertheless, the total fixation count alone is not sufficient to explain both the before and after question conditions. For an in-depth analysis of the data, Table 6.20 lists the various subgroups that were tested with ANOVA regarding answer quality.

Table 6.20 reveals two significant results regarding the tested conditions and subgroups. The results are summarized below:

Row a) shows the already mentioned answer quality for all conditions, divided according to the source code comprehension question time. The data indicates that the question-before condition is responsible for more correct answers overall. This supports the hypothesis that knowing the question before reading the code is beneficial for participants.

Rows b) and c) show no clear indication for the Rectangle source code example and a positive indication for the Vehicle code example that the question-before condition is

Table 6.20: The average answer quality divided according to the question-before and question-after conditions, separated by different subgroups, and compared using ANOVA tests.

		Answer Quality				
#	Condition	Quest. Before	Quest. After	F	p	n^2
a)	All	**69.55**	53.86	F(1, 86) = 2.711	0.103	0.0306
b)	Rectangle	43.46	**44.07**	F(1, 42) = 2.511	0.121	0.0564
c)	Vehicle	**80.45**	70.00	F(1, 42) = 0.678	0.415	0.0159
d)	non-smell	**75.00**	42.73	F(1, 42) = 6.234	0.017*	0.1292
e)	smell	64.09	**65.00**	F(1, 42) = 0.004	0.948	0.0001
f)	Novices	43.98	**45.86**	F(1, 62) = 0.607	0.439	0.0097
g)	Experts	**78.33**	44.17	F(1, 22) = 3.529	0.074.	0.1382

useful for answering the comprehension question. The difference for the Rectangle source code is too small to attribute an effect to the question-after condition. In contrast, the Vehicle source code had a larger effect for the question-before condition, which is as assumed for this hypothesis. With the available data, distinguishing whether this is based on the question-before condition or on the carryover effect is not possible, because the Vehicle source code example overall had more correct answers (see Section 6.2.1).

Rows d) and e) indicate that the non-smell condition for both source code examples had a significant difference between the question time conditions, with a clear preference for the question-before setting, which is considered to be significant, according to the ANOVA test. On the one hand, this supports the hypothesis: The question time influences answer quality. On the other hand, this result is only correct for the non-smelly condition, not for the source code examples with a smell. This result can indicate that a source code smell was distracting for participants so that the time of the comprehension question did not play a crucial role overall.

Rows f) and g) reveal another interesting result: Novices do not seem to benefit from an early source code comprehension question. The results are even higher for the question-after condition, even though the difference is small enough to be considered as noise. In contrast, the data for the experts reveals a more significant difference, according to ANOVA, with a clear preference for the question-before condition: Experts seem to benefit from an earlier question, before they start reading and comprehending the source code example. This finding may be the result of the knowledge and experience advantage. Reading source code with a clear intention in mind is common for professional software developers. Furthermore, this result is interesting because the three experts who selected the “none” option for the Java skill level are in the question-before group. Therefore, the results could

have been even better without these three participants, or the influence is not as high as expected. The data shows that these three participants managed the questions: Only one gave incorrect answers for both source code examples; the others were completely correct or nearly correct (one answer). The average answer was 56.67 % for the first source code example and 66.67 % for the second source code example, which, in both cases, is better than expected for the "none" skill group.

Significant Results with ANOVA

The data in Table 6.20 presents some interesting correlations and significant results. Overall, the question time, specifically the question-before condition, influences the answer quality (correct or incorrect answer) of participants. However, not every difference is large enough to be considered as a realistic effect, such as the ones reported for the Rectangle source code, the smell condition, and the novice skill group. In general, the first row in the table displays a large difference overall. More precisely, larger effects are observable for the Vehicle source code, the smell condition, and the expert skill group. As a summary of these results, three circumstances are responsible for the reported results:

(a) It seems that the non-smelly condition does not interfere with answering the comprehension question in the question-before group, given the greater number of correct answers. This effect is not observable for the source code examples with a code smell.

(b) Experts seem to benefit from knowing the comprehension question and the intention for reading the source code, which is a more realistic scenario in their daily work as software developers.

(c) The larger numbers for the question-before condition on the Vehicle source code are more likely the result of the carryover (training) effect, which is also observable for previous hypotheses. Both comprehension questions are similar; therefore, it is unlikely that the question-before condition is more helpful for the Vehicle source code, compared to the Rectangle source code.

Qualitative RTA Analysis

In the qualitative analysis of the RTA, 14 participants stated that knowing the source code comprehension question is important for reading the source code examples. Not knowing the questions makes the reading unstructured and vague.

6.3 THREATS TO VALIDITY

While analyzing the recorded data, several threats to the internal and external validity were noted.

Internal Validity. The following limitations for causal conclusions apply for the code smell study. a) Both source code examples were based on OOP concepts, which had to be known by the participants. Even though only basic OOP concepts were used, and these are taught early on in universities, participants may have encountered problems solving the source code comprehension questions because of a lack of OOP skills. b) The company from which the professional software developers were recruited offers web and desktop applications, primarily written in JavaScript, TypeScript, and C#. Java is thus not a completely unexpected language; however, the developers had less actual experience with it, compared to the student group. This could have had an effect on the skill groups in several ways (e.g., answer quality). c) All professional software developers were recruited from one company only. It is possible that this company uses a similar recruiting process and internal training program for all its employees. The skill differences could have thus been different, but in a lower variance compared to professional participants recruited from different companies. This could have had an effect on the answer quality and other metrics. d) The expertise is based on a self-assessment. Participants answered questions related to their programming knowledge with Java and other languages. The judgment is likely to be biased because self-assessments are unreliable, which is a common problem in source code comprehension studies, with a suitable solution yet to be known. Therefore, it cannot be guaranteed that measurements based on the skill level are precise enough. e) Although the source code examples were carefully selected, some participants could have worked on similar examples recently in an exam or course, which is advantageous for the novice group. In addition, the source code snippets are not real-word examples and thus far from something the professional group would encounter in actual projects. Therefore, this could have had an impact on answering the source code comprehension questions.

Internal Validity: OOP Code, Skill-Level of Experts, One Company, Self-Assessment, and Known Examples

External Validity. The following limitations for generalizable results were noted. a) The source codes were not representative of large systems or code bases. The code examples had 21 (Rectangle non-smell and smell), 20 (Vehicle non-smell), and 21 (Vehicle smell) code lines (without blanks); these numbers are relatively small but necessary for an eye tracking study without specialized tools. Moreover, especially for the experts, the code is not real-world code that they would encounter in a software project. b) The data collection for the professional software developers was conducted on-site at the company, in an extra room provided for this purpose. The eye tracking setup was the same as in the eye tracking lab of the University of Applied Sciences and Arts in Dortmund; however, because the hardware was brought to the company, it cannot be guaranteed that the data collection was the same as that in the eye tracking lab. The process could have introduced differences and an overall variance in the data collection. c) The introduced time limit of 3 min could be too short. d) The language for the source code examples is Java; therefore, the source code idioms are also

External Validity: Non-Representative Examples, Data Collection at a Company, Time Limit, and Java Programming Language

based on Java. Results regarding the source code smells are valid for Java code, but hardly transferable to another language or to a different set of source code smells.

6.4 SUMMARY AND NEXT STEPS

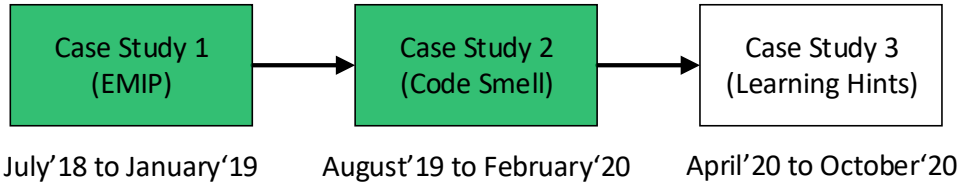


Figure 6.21: A brief overview of the case study timeline (code smell study completed).

Common Eye Movement Patterns

The primary objective of the code smell study was to detect common eye movement patterns related to the presence of a source code smell based on non-idiomatic programming. The second case study is thus complete (see Figure 6.21). As a summary of the code smell study, the initial hypotheses could not be confirmed completely with the gathered data and analyses. In general, the influence of the source code smells is not as high as assumed when designing and conducting the study. A code smell is not a common indicator of an inferior answer quality, higher fixation counts or duration times, or malformed reading patterns. In addition, experts did not perform better overall compared to novices. One explanation, which other researchers have also found, is that experts outperform novices for longer source code examples with more complex code snippets or many different source code files at once.

Question Time Influences Results

Apart from these findings, the time of the comprehension question, before the source code example is visible or afterward, has some effect on the comprehension answers – surprisingly only for the non-smell Rectangle source code example and not for the Vehicle source code with the same surrounding condition. Moreover, the flicking eye movement pattern is dominant for the comprehension question beforehand, but surprisingly, only for the non-smell Rectangle source code.

Code Smells and Answer Quality

In summary of the hypotheses, it must be noted that a code smell does *not* influence answer quality regarding correct and incorrect answers. Visible effects can be better explained with a carryover (training) effect of the participants. Apart from this, the data for the duration and fixation eye movement metrics indicate that a source code smell is generally not responsible for longer durations and more fixations. Such effects were observed only for the Vehicle source code. Nevertheless, the eye movement metrics for the ratio of duration and fixation suggest that source code examples with a smell increase the visual effort of participants, even if this has no effect on answer quality. Regarding the differences between novices and experts, it is noteworthy that experts did *not* provide more correct answers

overall; however, they spent less time (duration) and less fixations on the source code examples. Last but not least, the question-before condition was responsible for more correct answers overall, which is as assumed.

Carryover Effect

Furthermore, the following effect became apparent: Participants gave more correct comprehension answers for the Vehicle source code example, even for the smell condition on this source code. This result can be assigned to the carryover effect as a high training effect between both source code examples. According to the data, the answers for the comprehension questions are significantly better for the smelly Vehicle source code than for the smelly Rectangle code. The assumption is that the Rectangle smell, which is based on an identifier smell, embedded within a calculation, is more difficult to read and comprehend than the Vehicle smell, which is based on a malformed "if" statement (structural smell).

Retrospective-Think-Aloud (RTA)

Analyzing the RTAs revealed two main results: 1) Participants stated that the comprehension question is important for the reading and comprehension process and should be visible before the source code example is shown, not afterward. Participants also argued in favor of showing the question the whole time, side by side with the source code example. 2) Participants had major problems describing the source code they saw and explaining their eye movements, which were superimposed onto the source code examples during the RTA. While explaining their eye movements is understandably challenging because nearly all participants had never been in an eye tracking study for source code comprehension, it was surprising that participants in the novice group lacked the proper vocabulary to explain certain elements of the source code examples and their connections to other parts of the source code. This lack of vocabulary is a key problem in computer science education because explaining source code is a necessity. Therefore, explaining source code should be included in the computer science curriculum.

Comprehension Question Before Stimulus is Visible

In conclusion, the study and subsequent analysis demonstrate that the comprehension question should be asked before the source code examples are visible. This change reduces the necessity of memorizing the complete source code, which is error-prone and a questionable approach to source code comprehension. Moreover, learning hints should be dependent on the training effect of a participant's level of experience.

Larger Effects Assumed

In general, larger effects for the hypotheses were assumed. The code smells did not influence the participants as much as expected when designing the study. Overall, as the next step, the goal is to design and conduct another study for testing learning hints and their influence on source code comprehension processes. The assumption is that help systems, such as syntax or scope highlighting, could change participants' reading behavior, visible in different eye movement patterns. Another assumption is that a help system influences the perception of source code complexity, which can be important for participants when solving source code comprehension questions. These findings are important when designing a system to help students comprehend source code more efficiently.

"Anyone can make an error, Ensign. But that error doesn't become a mistake until you refuse to correct it."

– Grand Admiral Thrawn, Star Wars

7

Case Study 3 – The Learning Hints Study

Learning Hints

While the analysis of the EMIP dataset and the code smell study were both centered around the questions of how to identify eye movement patterns, which methods and tools are necessary to achieve this identification, and how to interpret the results, the so-called learning hints study had different goals: The first goal was to visualize and analyze the ways in which participants perceive varying source code examples, within the newly developed programming environment, based on, for example, answer quality, reading behavior, workspace switches, the rating of code difficulty, and the perception of complexity. The second goal had an exploratory component, wherein participants had to read source code in the new programming environment with gaze-enabled learning hints. In this environment, programming cues such as scope highlighting and tooltips are controllable with eye movements. Participants had to rate the user experience of the interface and the usefulness of the gaze-enabled learning hints, and the environment measured how often the hints were used. These analyses add to the third aim of this thesis (see Section 1.3) because the questions of whether learning hints can help participants and how they should be integrated into a learning environment are crucial aspects of this thesis.

Eye Tracking Data of Varying Learning Hints

This chapter summarizes the quantitative analysis of the eye tracking dataset recorded in the learning hints study. The primary goal is to visualize and analyze the way in which participants perceive source code examples in different study conditions. In addition, the ratings of the difficulty level of the source code examples and the usefulness of the learning hints are presented. The results indicate that the perception of complexity changes when

learning hints such as syntax highlighting are available. These analysis results were matched with participants' answers to comprehension questions, and a determination was made as to whether the different code representations and code interactions have an impact on these answers. Furthermore, without a direct relation to the described hypotheses, sequence analysis with the N-W algorithm and clustering was conducted. The intention was to use the similarity measure between sequences to have the opportunity to project this similarity to other dimensions such as participant groups. This data can be used to qualitatively search for eye movement patterns in these clusters, which supports previous research about syntax highlighting and the N-W algorithm. In summary, the study was conducted to make the following five contributions to the body of work of this thesis:

1. Understanding the influence of the different study conditions, with varying learning hints, on the answer quality of participants;
2. Identifying the differences in eye tracking metrics, such as *Fixation Count (FC)* and *Fixation Time (FT)*, between the various source code examples and learning hints;
3. Determining participants' ratings of the various learning hints, as well as how tooltips are perceived (rated) compared to the other hints;
4. Determining the influence of the learning hints and various source code examples on the perception of difficulty of the source code examples;
5. Understanding the user experience of the gaze-enabled programming environment, whether participants consider such an interface to be useful overall, and the requirements they mentioned to improve this gaze-enabled user interface.

7.1 STUDY DESIGN

Varying Code Examples and Learning Hints

The following sections describe the design of the learning hints study, in which participants were recorded while perceiving different source code examples with numerous learning hints. Furthermore, experimental gaze-enabled learning hints are introduced to test whether such features were used and how they are perceived by participants.

7.1.1 RESEARCH QUESTION AND HYPOTHESES

Plain Text as Default

Source code comprehension studies often use source code examples in the form of plain text. In such study conditions, no learning cues, active help systems, or other means of supporting the comprehension process are active, and this can have various effects on the outcome of these studies. Researchers must ask themselves whether plain text hinders the comprehension process too much and therefore if the study results are too far from real-world examples. In contrast, measuring the actual source code comprehension attainment

of participants can be challenging when learning hints (e.g., syntax or scope highlighting) are active.

Previous Research for Syntax Highlighting

Previous research has demonstrated that, for example, syntax highlighting can have an ambiguous impact on the comprehension outcome. Some studies have found effects for novices or in general (Asenov et al., 2016), and some have not found benefits for participants at all (Hannebauer et al., 2018). One effect is that highlighting is used as a visual cue for programmers to decrease the time required for mental execution (Sarkar, 2015). Another effect is that novices tend to not use syntax highlighting or, in the worst case, misinterpret the meaning entirely (i.e., reading too much into it). As far as is known, no research explicitly compares different learning hints on various source code examples, which is one reason for this study and the chosen design (refer to Section 7.1.3). Apart from this, it is interesting to test the various learning hints with an experimental gaze-enabled interface, where participants can use active learning hints with their gaze rather than using the mouse (e.g., to hover over elements for a tooltip or to highlight the scope of source code regions).

MEASURING SOURCE CODE COMPREHENSION (DEPENDENT VARIABLES)

Dependent Variables: Task Answer, Task Duration, Time Limit, Fixation Count, Interviews, and Visual Focus

As dependent variables, various sources were used to measure the outcome or effort of the source code comprehension tasks. The following specific variables were used to measure the quality of the source code comprehension per participant:

Task answer. Participants' performance of the source code comprehension task was determined by measuring the answers given for each source code example.

Task duration. The behavioral comprehension question cannot be answered without comprehending the source code first; even if a participant recognizes the code, it is necessary to comprehend and build a mental model of it with the given data. Such a question relates to the behavior of a program and asking for the result of an operation (e.g., running a loop three times). The task duration required to answer a question thus reveals, to some extent, the effort needed by a participant.

Time limit. The time limit per source code example ensured that a small amount of pressure was added to the comprehension task. This limit minimizes the "wandering around" effects of participants, which supports other measures such as the task answer and task duration.

Fixation count. Apart from the task duration, the fixation count was measured as an indicator of the effort a participant needed to answer the question.

Semi-structured interview. Answers from a semi-structured interview were used. In this interview, participants were asked what the code examples are implementing. Moreover, questions about the learning hints and the experimental gaze-enabled interface were asked.

Visual focus. Eye tracking data was used to track the process of reading the source code. In combination with AOI models, participants' reading sequences could be obtained while

they comprehended a code example. This process allowed for not only the recording of the order and frequency of source code parts that a participant looked at, but also the measuring of the duration of fixations.

EXPERIMENTAL VARIATION (INDEPENDENT VARIABLES)

Independent Variables: Source Code Examples, and Learning Hints

As the experimental rationale, source code examples with a fixed order and randomly selected learning hints were selected for the participants. The following variables were thus used to create the study conditions:

Source code examples. Three different source code examples, namely *Bubble*, the *Greatest Common Divisor (GCD)*, and *Vehicle*, with varying levels of complexity and different comprehension questions, were used. These code examples remained in the same order to reduce the overall complexity of the study (Siegmund, 2016). The assumption was that a learning hint has a larger effect on the outcome of the comprehension question than the source code example itself.

Learning hints. Comprehending source code can be supported by various hints and help systems. The study used a mix of active and passive learning hints. On the one hand, a dynamic learning hint is active and can highlight scopes in the form of block scopes and variable scopes (usages). On the other hand, a syntax highlighting hint emphasizes identifier names, operations, and keywords, and is passive. The effect that learning hints have on the comprehension process and outcome is interesting, which is the reason participants' were randomly assigned to one of six groups (study conditions). These groups were formed by iterating the available learning hints to achieve a *counterbalanced within-subject design*. If a learning hint influences the outcome of a comprehension task, this can be measured based on the task answer, task duration, or visual focus.

THE VARYING LEARNING HINTS

Syntax Highlighting Most Useful

The original assumption when designing the learning hints study was that a syntax highlighting hint is the most useful one, which would be reflected in high user ratings and optimal results for the source code comprehension questions in all source code examples.

Dynamic Hint Useful but Less Used

The assumption for the dynamic hint was that while this type of hint is useful, it will be less used. The reason for this assumption is that a dynamic hint is an active hint, which participants need to use when comprehending the source code. Moreover, this is a learning hint that requires active recall rather than only recognition of the passive syntax highlighting hint.

Plain Condition Less Useful

Overall, the assumption for the control group without hints (plain) was that this is the least useful learning hint, and it will result in poor ratings and incorrect answers for the source code comprehension questions.

RESEARCH QUESTION

Dynamic Hint Usage

In addition to the three hypotheses, which are described later in this section and were rigorously tested throughout this study, one research question was addressed while analyzing the data:

$RQ_{Dynamic-Hint-Use}$ – The goal of this research question is to determine how learners use a dynamic learning hint to highlight variable and block scopes while comprehending the code examples. This hint is analyzed in more detail because it is a common feature in many programming environments, and using this hint was logged programmatically. The assumption was that the dynamic learning hint is used less than initially expected. The results are described in Section 7.2.1.

HYPOTHESES

Active and Passive Learning Hints

The objective of the study was to investigate the effect of active and passive learning hints on the outcome of source code comprehension processes. In addition, the following were analyzed: How learners use and perceive the learning hints and how the source code examples were rated regarding their complexity. A further interest was in testing the various learning hints with a gaze-enabled interface, where participants could use active learning hints with their gaze rather than using the mouse. The condition with the gaze-enabled interface is an addition to the comprehension tasks (see Section 7.1.3), and it was designed as an exploratory task at the end of the other study conditions. The study was divided into two phases, as described in Section 7.1.3. In summary, the following three hypotheses were analyzed:

$H_{Answer-Quality}$ – For this hypothesis, an examination was conducted to understand how answer quality differs between the various source code examples and conditions. The assumption was that more complex code examples have fewer correct answers, higher fixation counts, and a longer duration overall, with varying results for each condition and hence available learning hint. The results for this hypothesis are described in Section 7.2.2.

$H_{Ratings-Learning-Hints}$ – For this hypothesis, the ratings of all learning hints were analyzed. The ratings, provided by the participants at the end of the study, provided insights into which learning hints are more useful than others from the participants' perspective. The assumption was that the syntax highlighting hint is more useful than the dynamic hint, which is in turn more useful than the plain condition. A poorly rated hint can help to identify positive or negative statements about this particular hint in the semi-structured interviews. Furthermore, the ratings for

the tooltip learning hint, which was additionally presented in the gaze-enabled interface, were also analyzed. Participants had to rate the tooltip hint compared to the other learning hints. The results provide information about the usefulness of tooltips compared to the other hints and compared to the fact that the tooltips were gaze-enabled, which is not a common feature. The results for this analysis can be found in Section 7.2.3.

$H_{Ratings-Code-Difficulty}$ – For this hypothesis, all three source code examples' ratings, which the participants provided at the end of the study, were analyzed. The data allows insight into whether a learning hint, which was available in some study conditions, can change the difficulty perception of the source code. Moreover, an analysis was performed regarding whether the difficulty ratings of the source code examples match the assumed ratings (vehicle = easy, GCD = medium, and bubble = complex) when the study was designed. The assumption was that the syntax highlighting hint has a greater influence on the difficulty perception than the other hints. This would change participants' perception of complexity. If the GCD source code example had higher difficulty ratings, this could have affected the statements in the semi-structured interview about the usefulness of the gaze-enabled hints on this particular source code example. The results for this hypothesis are described in Section 7.2.4.

7.1.2 METHODS

AOI-DNAs and AOI-STGs

The analysis of the code smell study was based on various methods and visualizations to achieve a triangulation of the results and to gain richer and broader insights. As already used in the previous case studies, the visualizations for eye movement patterns were based on AOI-DNAs (Deitelhoff et al., 2019b). These were integrated into the CodeSight application, which allows for visualizing of the AOI-DNAs and searching for eye movement patterns. Visualizing the data is closer to a bottom-up approach, while searching for patterns is a top-down one.

Sequence Alignment

Furthermore, some methods identical to the previous case study were used; however, in this study, they were utilized without any direct relation to the described hypotheses and results. These methods were thus not directly used for hypotheses testing: The sequence alignment algorithm N-W was used in combination with a distance calculation and hierarchical clustering with dendrograms to visualize the results. These methods are crucial to identify and visualize important eye movement patterns and hence differences between study conditions and participant groups. Since the results provided no benefit over the already described results of the previous case studies, the results are omitted in this chapter; however, it is worth mentioning that these methods were used in this study as a common set of analysis methods and tools, continuously developed throughout this thesis.

Difficulty Ratings and UEQ-S

Throughout the study, participants had to rate the difficulty level of the source code examples and the usefulness of the learning hints, thereby making it possible to calculate the ratings across learning hints and source code examples. To obtain data about the user experience of the overall study prototype, the short version of the *user experience questionnaire (UEQ-S)* was conducted at the end of Phase 1 of the study (see Section 7.1.3). Moreover, participants' usage counts and time periods of learning hints were recorded in the gaze-enabled interface. The calculation was done with an event log – created throughout the study – that contains log entries for every learning hint interaction, including the *activity data* (e.g., scope and variable name), the *timestamp*, and the *element* the help was viewed for with the gaze interaction. Moreover, the answers to the semi-structured interview, which was conducted with participants at the end of every study phase, were analyzed. In addition to the questionnaire for the programming skill levels as a self-assessment, a pre-test was employed in this study. Within this test, 10 questions were asked regarding common programming constructs such as loops and constructors, tailored to the Java programming language. The results of the self-assessment and the pre-test can be found in Section 7.1.6.

7.1.3 EXPERIMENTAL DESIGN

Two Study Parts and Gaze-Enabled Interface

The learning hints study was divided into *two consecutive parts* that every participant had to go through. In the first phase, the influence of learning hints on the source code comprehension process of three different source code examples was analyzed. The code examples *Bubble*, the *GCD*, and *Vehicle* were used; they implement the well-known *bubble sort* and *GCD* algorithms, as well as a class that represents a *vehicle*. This primary study phase thus consists of the later described conditions (groups), the recording of the demographic data, the pre-test, and a semi-structured interview. The *second phase*, the secondary study phase, consisted of an experimental part with a gaze-enabled interface. The source code examples and learning hints of the first phase were used in a structured way with specific source code comprehension questions, and the gaze-enabled interface of the second phase was used in an exploratory way without a specific comprehension question. Participants were urged to read the source code and actively use the gaze-enabled learning hints. They could control the learning hints in the header of the application (e.g., to turn the syntax highlighting or code lines on and off). In both phases, a semi-structured interview was conducted to ask specific questions about the source code examples, learning hints, and gaze-enabled features. In addition, the helpfulness rating for the interface was measured with the UEQ-S. The study procedure is described in Section 7.1.6.

Three Source Code Examples and Three Learning Hints

The study is based on three different source code examples and three learning hints, which are described in detail in Section 7.1.6. The source code examples and learning hints form the study conditions depicted in Figure 7.1. The advantages are, that a) the fixed order of code examples reduces the overall complexity of the study (Siegmund, 2016), and b) that there

are always two groups with the same condition, which can be analyzed as one group. The participants were randomly assigned to one of the six groups in which they saw the source codes relevant for the group one by one with the accompanying learning hint active and the source code comprehension question visible all the time. The learning hints are labeled as P = plain, S = syntax, and D = dynamic throughout this chapter. When comprehending the source code, every code example had a time constraint of *4* min. After this time, the code was hidden, but the participant could still enter an answer and move forward.

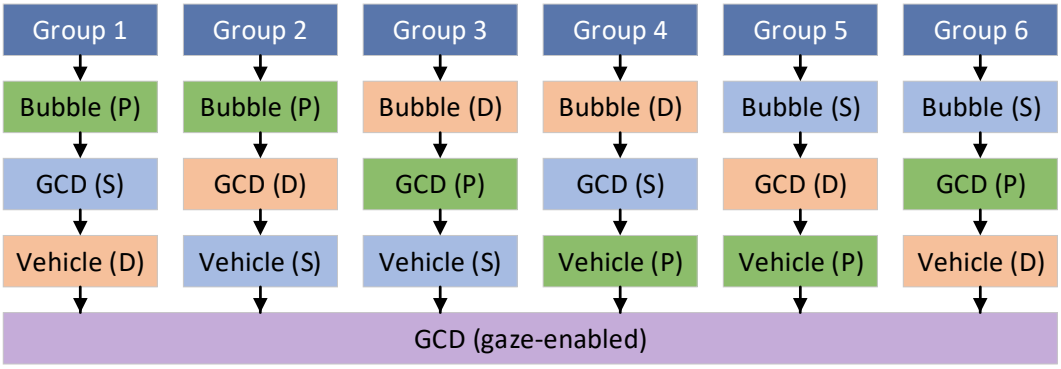


Figure 7.1: The six study conditions (groups) formed by three source code examples and three learning hints – **(S)**yntax, **(D)**ynamic, **(P)**lain. The experimental gaze-enabled interface was used by every participant as the last step.

7.1.4 PREREQUISITES AND ENVIRONMENT

Tobii Pro TX300 Eye Tracker

The prerequisites for the learning hints study affected the eye tracking setup, while the environmental requirements focused on the programming environment prototype. The eye tracking setup remained the same as in the code smell study: All recordings were done with the Tobii Pro TX300 eye tracker. Figure 7.2 illustrates the setup in the eye tracking laboratory at the University of Applied Sciences and Arts, Dortmund. The hardware setup stayed the same, but in contrast to the code smell study, this time a new software environment was necessary for conducting the study.

Custom Study Environment

The custom programming environment, implemented in Node.js and JavaScript, provided a Web interface for all study phases and conditions. Moreover, the environment provided a fully functional back-end with administrative options to start the study (e.g., with options for the condition and status information about the database connection). The actual study UI for the participants contained a short briefing, the pre-test, an onboarding directly within the environment, the three source code examples with or without learning hints, the questions about the user interface after each source code example, the UEQ-S, the rating for the code complexity, and the experimental front-end for the gaze-enabled interface. This environment was thus a functional platform for the learning hints study, manageable via the admin interface in the back-end. The following figures present some example screenshots

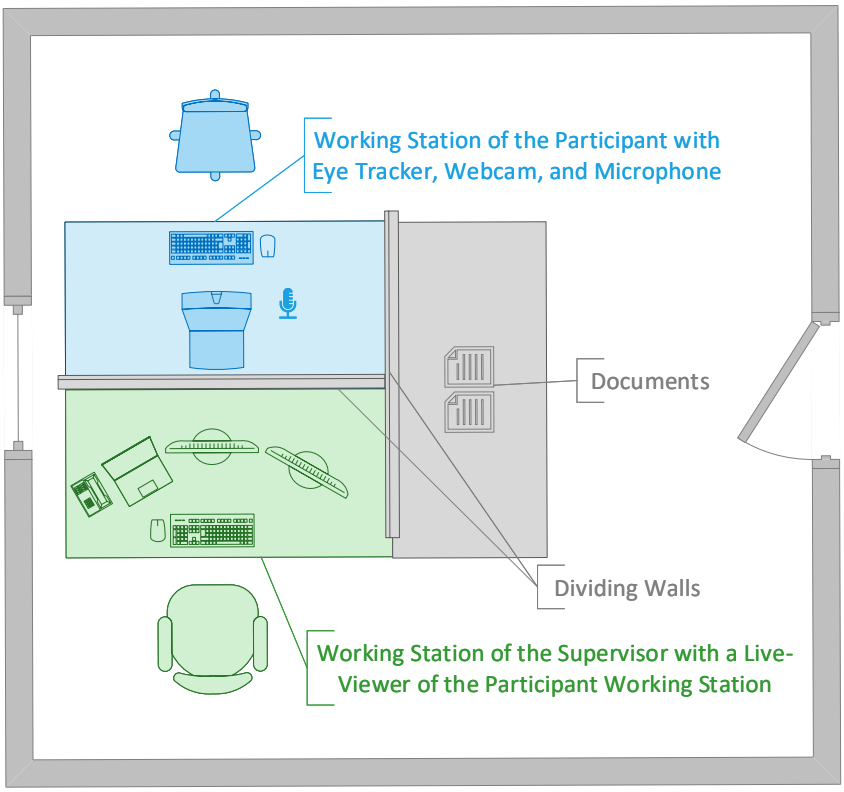


Figure 7.2: The layout of the setup in the eye tracking lab at the University of Applied Sciences and Arts, Dortmund.

from the study environment. Figure 7.3 displays a part of the pre-test, Figure 7.4 presents the vehicle source code example with the syntax highlighting learning hint, Figure 7.5 highlights the UEQ-S, and Figure 7.6 focuses on the gaze-enabled interface from the experimental part of the study (second phase). The screenshots of the study environment remain in German because the study was conducted in German, with German-speaking participants.

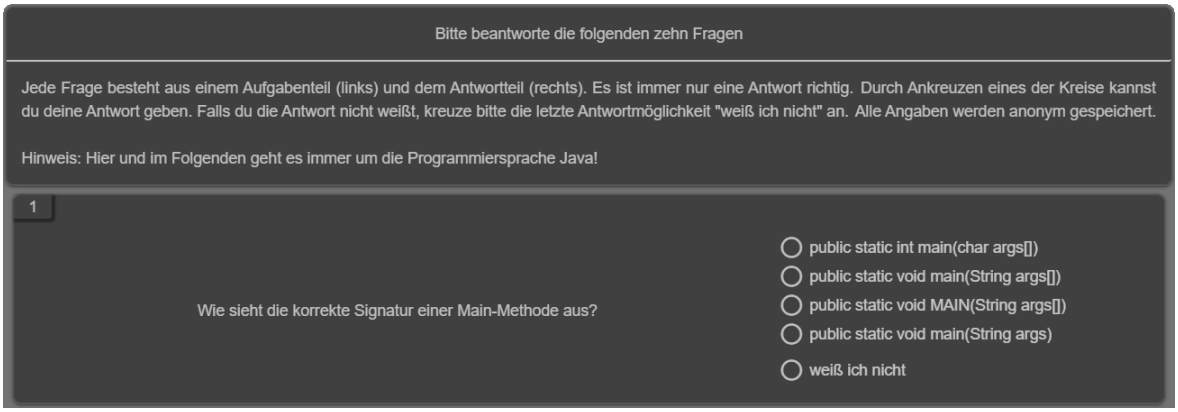


Figure 7.3: The first question of the pre-test as an excerpt from the full pre-test.

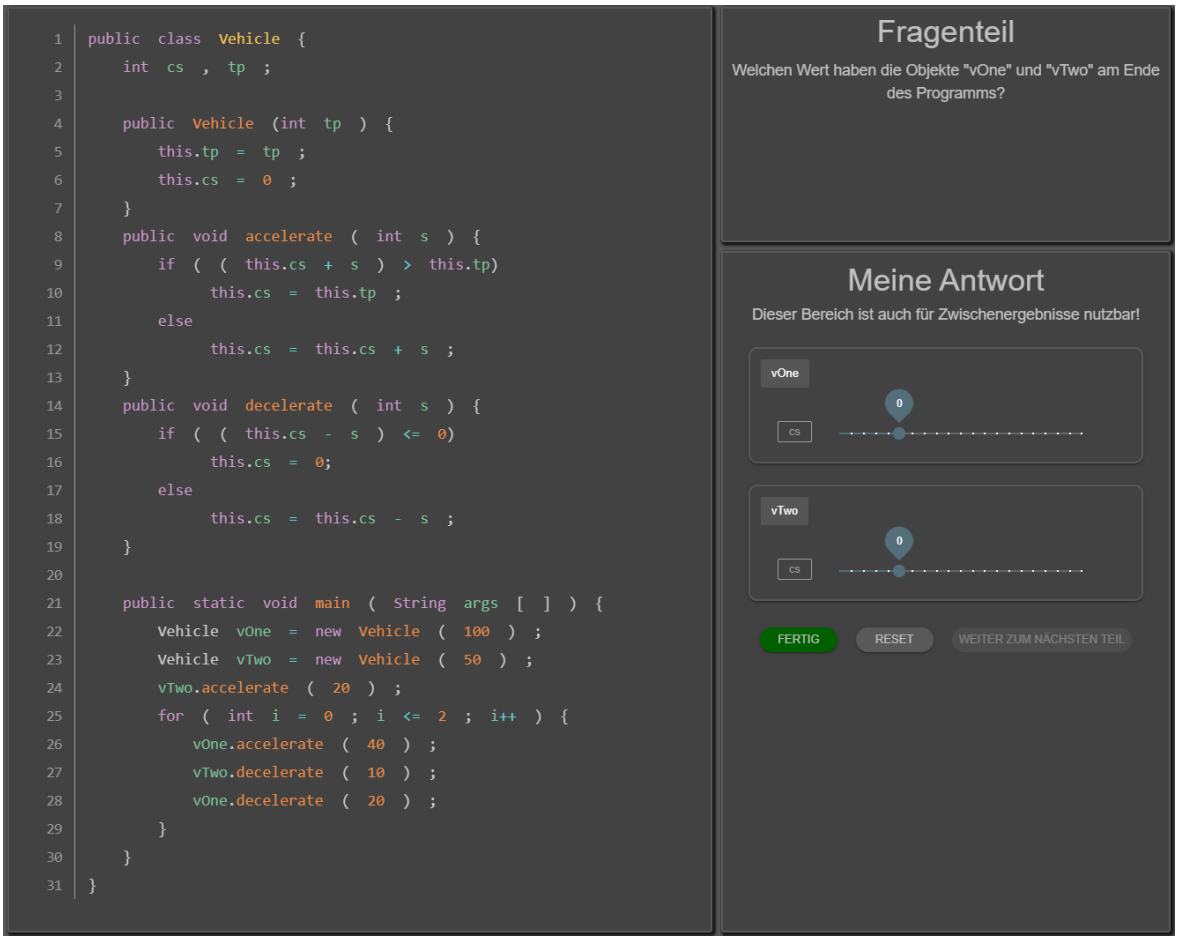


Figure 7.4: The Vehicle source code examples with the syntax highlighting learning hint.

Figure 7.5: The questions for the short UEQ (UEQ-S).

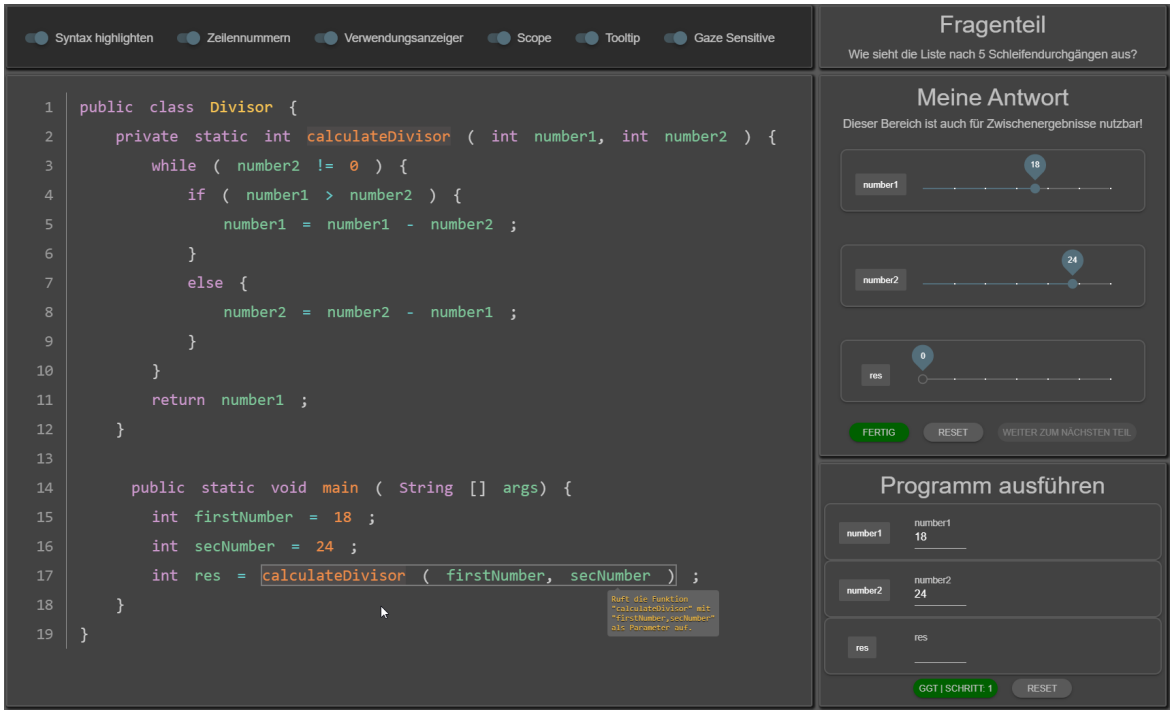


Figure 7.6: The experimental interface with the gaze-enabled learning hints.

Study Prototype as Master Thesis

This environment, implemented in the master thesis of Benedikt Schröder[16] (Schröder, 2019), saved a great deal of effort when conducting the study. Whenever a participant had an appointment at the eye tracking lab of the University of Applied Sciences and Arts, Dortmund, the next condition was chosen in the back-end to set up the study environment properly. In this way, a balanced participant group was possible because a gap in one of the groups could be filled without problems. The evaluation of the UEQ-S results are described in Section 7.2.

7.1.5 CAPTURED DATA

Eye Tracking Raw Data in the Study Prototype

Unlike the code smell study, the eye tracking data for the learning hints study was *not* recorded with the Tobii Pro Studio software. Instead, the study prototype was implemented to receive the eye tracking data from the Tobii Pro TX300 directly. Data receipt was implemented via the *Eye Tracking Connector*[17], visible in Figure 7.7. This application routes data from an eye tracker to the study prototype. The programming environment was thus aware of the coordinates that a participant was looking at. For the first phase of the study, this was not actually necessary, but a useful way to save the eye tracking data in a MongoDB instance, along with all other incoming data. This proved to be valuable in the later data analyses. While receiving eye tracking data in the first phase of the study was not a necessity, it became essential for the second phase. Furthermore, implementing a gaze-enabled

[16] My gratitude to Benedikt Schröder for his outstanding work!
[17] My gratitude to Christian Schlösser for his outstanding work!

interface with gaze-controllable learning hints demands receipt of the eye tracking coordinates in the application. This implementation allows for gaze points and UI elements to be checked to activate animations and context-related actions. From the viewpoint of the implementation, this made the study prototype more complex and required more effort.

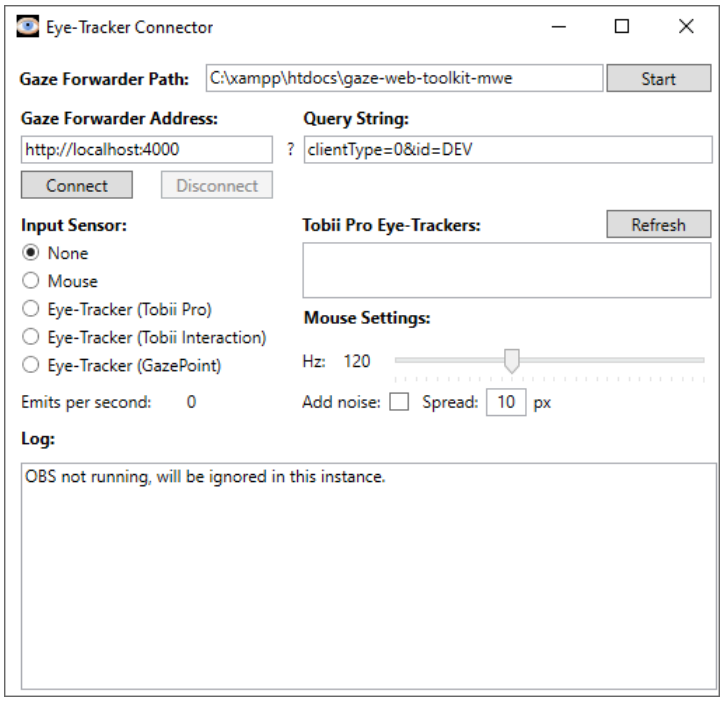


Figure 7.7: Screenshot of the Eye Tracking Connector application to route the eye tracking data from Tobii Pro TX300 to the study prototype.

Demographic Data, UEQ-S, and MongoDB as Database

Apart from the eye tracking data, the captured metadata for all participants contains, among other things, the age, gender, mother tongue, English level, and expertise in Java. This demographic data was recorded with a survey at the end of the study for every participant. Overall, eye movement raw data, answers to the comprehension questions, answers to the surveys, the source code and learning hint ratings, learning hint usage counts, and answers to the UEQ-S were recorded. For the source code difficulty, the answers were on a scale from "*simple (1)*" to "*complex (3).*" For the learning hint usefulness, the answers ranged from "*helpful (1)*" to "*not helpful (3).*" All data was captured within the study prototype in a single MongoDB, with one document study step per participant; this resulted in 13 documents per participant, each with distinct data for one study step. This design with a flexible scheme simplified the subsequent analyses because the data was accessible programmatically, document by document, whereas every document was designed specifically for one study step. Figure 7.8 (see page 173) presents the MongoDB scheme as a table. The entries in the `probandID` field illustrate that a participant has multiple rows, one for each study step.

The semi-structured interviews were conducted manually. The answers were written down and simultaneously recorded as an audio file for backup purposes in case of missing handwritten data.

Hauptstudie

	_id ObjectId	probandID Int64	probandZeit String	schritt Mixed
1	5cf7a6c4b2d2ba3528822b3b	1559733076997	"5.6.2019, 13:11:16"	"1 pre"
2	5cf7a741b2d2ba3528822b3c	1559733076997	"5.6.2019, 13:11:16"	"2 Intro"
3	5cf7a7edb2d2ba3528822b3d	1559733076997	"5.6.2019, 13:11:16"	"3 bubble"
4	5cf7a81db2d2ba3528822b3e	1559733076997	"5.6.2019, 13:11:16"	4
5	5cf7a823b2d2ba3528822b3f	1559733076997	"5.6.2019, 13:11:16"	"5 Intro"
6	5cf7a943b2d2ba3528822b40	1559733076997	"5.6.2019, 13:11:16"	"6 GGT"
7	5cf7a95eb2d2ba3528822b41	1559733076997	"5.6.2019, 13:11:16"	7
8	5cf7a966b2d2ba3528822b42	1559733076997	"5.6.2019, 13:11:16"	"8 Intro"
9	5cf7aa5fb2d2ba3528822b43	1559733076997	"5.6.2019, 13:11:16"	"9 vehicle"
10	5cf7aa70b2d2ba3528822b44	1559733076997	"5.6.2019, 13:11:16"	10
11	5cf7aaa1b2d2ba3528822b45	1559733076997	"5.6.2019, 13:11:16"	11
12	5cf7aad6b2d2ba3528822b46	1559733076997	"5.6.2019, 13:11:16"	12
13	5cf7af2fb2d2ba3528822b47	1559733076997	"5.6.2019, 13:11:16"	"14 GGTEye"
14	5cf8e66a3fda2f07ecc5c5d7	1559815393340	"6.6.2019, 12:03:13"	"1 pre"
15	5cf8e6c03fda2f07ecc5c5d8	1559815393340	"6.6.2019, 12:03:13"	"2 Intro"

Figure 7.8: Excerpt of the data from the MongoDB scheme for the learning hints study.

7.1.6 PARTICIPANTS, MATERIAL, AND PROCEDURE

$n = 24$

Altogether, 35 participants from the nearby University campus were recorded. Six participants were used in the master thesis as part of a pre-study to test the newly developed study prototype. These six participants are not included in the data reported in this chapter, which results in 29 participants overall for the main study described here. Due to data quality restrictions regarding the eye tracking data (i.e., too few gaze samples), five participants had to be excluded from subsequent analyses. The technical eye tracking data quality was verified after each recording to ensure an optimal analysis basis, thereby allowing for the creation of a balanced study design. The eye tracking data quality of the remaining 24 participants was very good to good. The data basis for the learning hints study was thus $n = 24$, seven of whom are female and 17 male, with a mean age of 26.29 ($SD = 4.28$). The participants were all students between semesters 1 and 10, with various backgrounds in computer science.

Programming Knowledge

All participants completed a questionnaire with questions regarding, for example, their knowledge in English and Java, measured on a scale from "*very good*" to "*very bad.*" No participant was in the "*very bad*" category; therefore, no one had to be excluded due to a lack of English reading skills. The exact distribution was "very good" (6), "good" (11), "medium" (5), and "bad" (2). Furthermore, answers between "very good" and "no knowledge" measured participants' knowledge of the Java programming language. No participant selected the "no knowledge" answer; therefore, no one had to be excluded due to a lack of Java knowledge. The exact distribution was "very good" (5), "good" (9), "medium" (8), and "bad" (2), as illustrated in Figure 7.9.

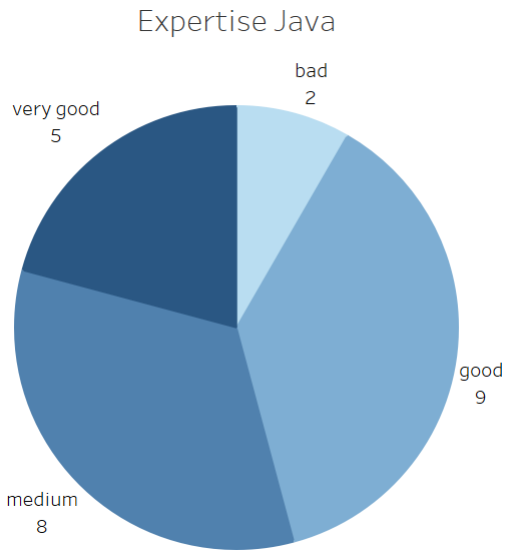


Figure 7.9: The distribution of answers to the question on expertise in Java in the learning hints study (all participants).

Self-Assessment and Pre-Test

In addition to the self-assessment of Java knowledge, a pre-test was conducted at the beginning of the study. Ten questions were asked per participant regarding common Java and programming elements. The ratings were high among the participants. Eight participants got 100 % of the questions right, seven answered 90 % correctly, five got 80 % right, two 70 %, one 60 %, and another one 50 %. The average was 86.67 % correct answers ($SD = 13.726$ %) over all participants, which is a positive result. Overall, the pre-test questions seem to have been quite easy and could be more difficult for future versions of the test.

Code Examples Bubble, GCD, and Vehicle

Throughout the study, the three code examples, namely *Bubble*, the *GCD*, and *Vehicle*, were used as stimuli. These code snippets implement the algorithms *Bubble Sort*, *Greatest Common Divisor*, and a class that represents a *Vehicle* with methods such as accelerate and decelerate, and properties such as top speed and current speed. These code examples serve as exemplary implementations for well-known algorithms and as an implementation for an object-oriented task. All code examples are often used at various levels and for different tasks throughout computer science education courses. The complexity of these code examples varies between *complex* (Bubble), *medium* (GCD), and *easy* (Vehicle), assessed with the help of researchers involved in education, learning analytics, and teaching. The complexity assessment is also based on experience with tasks in courses such as CS1. A metric introduced by Ward Cunningham in the workshop "Software Archeology: Understanding Large Systems" (OOPSLA, 2001[18]) demonstrates a different categorization. In this metric, opening and closing brackets, as well as lines with code statement, are counted and visualized with the characters "*{*", "*}*", and "*;*". According to this metric, the *Bubble* and *GCD* code examples are nearly equally complex, and the *Vehicle* example is the most complex one. Listings 7.1, 7.2, and 7.3 show the Java source code for the three code examples.

[18] `http://c2.com/doc/SignatureSurvey/` – Last accessed on December 14, 2020.

The Bubble Code Example

```
1  public class SortAlgorithm {
2    public static int[] sorting ( int[] listVar ) {
3      int temp;
4      for ( int i = 1 ; i < listVar.length ; i++ ) {
5        for ( int j = 0 ; j < listVar.length-i ; j++ ) {
6          if ( listVar[j] > listVar[j+1] ) {
7            temp = listVar[j] ;
8            listVar[j] = listVar[j+1] ;
9            listVar[j+1] = temp ;
10         }
11       }
12     }
13     return listVar ;
14   }
15
16   public static void main ( String[] args ) {
17     int[] unsortedList = { 1, 5, 8, 2, 7, 4 } ;
18     int[] sortedList = sorting ( unsortedList ) ;
19   }
20 }
```

Listing 7.1: The Bubble source code example for the learning hints study.

The GCD Code Example

```
1  public class Divisor {
2    private static int calculateDivisor ( int number1, int number2 ) {
3      while ( number2 != 0 ) {
4        if ( number1 > number2 ) {
5          number1 = number1 - number2 ;
6        }
7        else {
8          number2 = number2 - number1 ;
9        }
10     }
11     return number1 ;
12   }
13
14   public static void main ( String[] args ) {
15     int firstNumber = 18 ;
16     int secNumber = 24 ;
17     int res = calculateDivisor ( firstNumber, secNumber ) ;
18   }
19 }
```

Listing 7.2: The GCD source code example for the learning hints study.

The Vehicle Code Example

```
public class Vehicle {
  String p , t ;
  int cs = 0 , tp ;

  public Vehicle ( String p, String t, int tp ) {
    this.p = p ;
    this.t = t ;
    this.tp = tp ;
  }

  public void changeVelocity1 ( int s ) {
    if ( ( this.cs + s ) > this.tp ) { this.cs = this.tp ; }
    else { this.cs = this.cs + s ; }
  }

  public void changeVelocity2 ( int s ) {
    if ( ( this.cs - s ) > 0 ) { this.cs = this.cs - s ; }
    else { this.cs = 0 ; }
  }

  public static void main ( String args[] ) {
    Vehicle v1 = new Vehicle ( "Audi" , "A8" , 200 ) ;
    Vehicle v2 = new Vehicle ( "Claas" , "T1" , 50 ) ;
    v2.changeVelocity1 ( 20 ) ;
    for ( int i = 0 ; i <= 2 ; i++ ) {
      v1.changeVelocity1 ( 40 ) ;
      v2.changeVelocity2 ( 10 ) ;
      v1.changeVelocity2 ( 20 ) ;
    }
  }
}
```

Listing 7.3: The Vehicle source code example for the learning hints study.

To measure how successfully participants can comprehend the source codes, the following comprehension questions were asked for the different code examples:

For Bubble: "What does the list look like after two runs of the outer loop?"

For the GCD: "To which values are the variables 'number1' and 'number2' set after three runs of the loop?"

For Vehicle: "To which values are the objects 'vOne' and 'vTwo' set at the end of the program?"

Learning Hints Syntax Highlighting, Dynamic, and Plain

In addition to the three source code examples, three different learning hints were used; they are classified as passive and active learning hints. A learning hint in the first category is always available and visible in the code editor, while a hint in the latter category needs to be used actively by the participant. The *syntax highlighting* hint is passive and emphasizes identifier names, operations, and keywords. This learning hint helps one to navigate the code and focus on parts such as variable assignment and logic. The *dynamic* learning hint is active and can highlight scopes in the form of block scopes and variable scopes (usages). This hint is active because a participant uses the mouse to highlight curly braces or variables. Both hints are standard features of many IDEs and help one to focus on how code is structured. The third hint, called *plain*, and serves as the control group. In this group, no additional learning hint was available. Furthermore, the *code lines* hint was always available in every study condition because no specific comprehension questions were asked referring to source code lines. Therefore, source code lines are not counted as learning hint in the learning hints study.

Four Phases: Preparation, Pre, Main, and Post

The procedure for recording the participants was more complex than in the code smell study, and it was divided into the following five stages: *preparation phase*, *pre phase*, *main phase*, *post phase*, and *end phase*. After greeting and shortly introducing the participants to eye tracking as a study technique and technology, every participant had to sign a data protection form. This form provided information about how the data in the study was used and which data was collected. Then, the survey with the demographic data had to be filled out, followed by the first calibration and answering the pre-test questions for the assessment. The main study phase consisted of one onboarding part per source code example, with explanations regarding the available hint and a short survey about both the complexity of the past source code (ignoring the presentation of the code). In addition, participants rated how helpful the presentation was. Thus, one onboarding process was followed by one stimulus and then the short survey. This was repeated three times for every condition per participant. Thereafter, the post-phase began with the UEQ-S, a survey regarding the complexity of the source codes overall (ranking) and the usefulness of the learning hints overall (ranking). This was followed by the first semi-structured interview about the source code examples, including the presentation of source code and the learning hints. Afterward, another eye tracking calibration was started to ensure high eye tracking data quality and precise fixations. Finally, the participants had time to use the gaze-enabled interface with the gaze-enabled learning hints. After some exploratory time, the second semi-structured interview was conducted, asking questions about the usefulness and perception of the gaze feature and the new tooltip learning hint, and to conclude the study, the participants were debriefed. This whole process took approximately 55–60 min per participant, mainly dependent on how long both calibration phases and the survey process took. Figure 7.10 presents the general study procedure per participant.

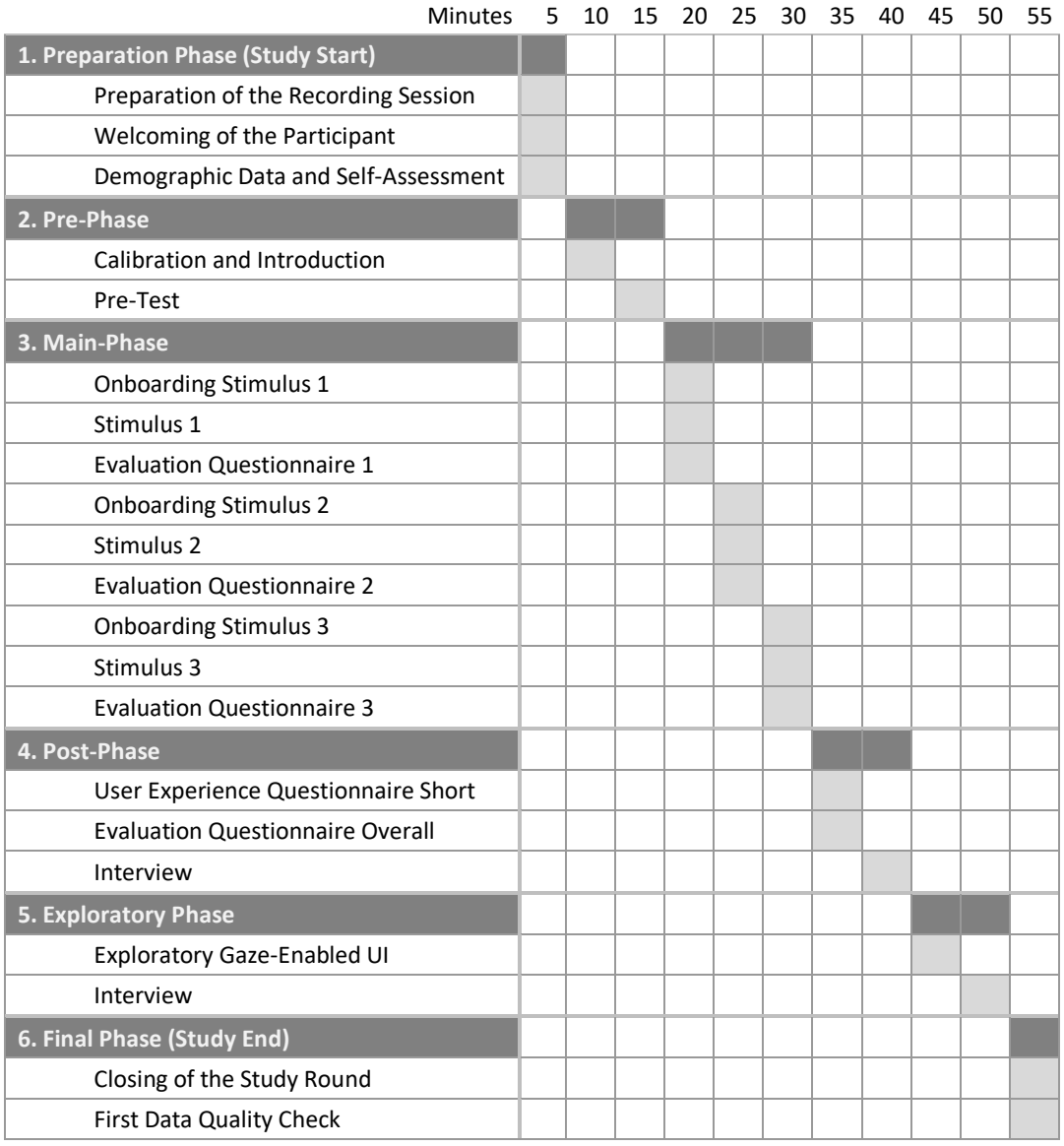


Figure 7.10: The study procedure for the learning hints study.

7.1.7 ANALYSIS DESIGN

MongoDB-Exporter

During the study, the eye movement data of every participant was recorded within the MongoDB programming environment prototype and saved in a MongoDB. Raw eye movement data was thus available for every step of the study, even for the surveys, the UEQ-S, and the onboarding. As a data preparation step, a tool named *MongoDB-Exporter* was implemented to export the data out of the various documents in the MongoDB instance and into a usable format for later analysis. The tool exported data for the pre-test, the source code examples (i.e., calculated answers), the mini evaluations between each stimuli, the overall UEQ-S, the difficulty ratings for the source code examples, the helpfulness ratings for the

learning hints, and the event log data for the dynamic learning hint. These steps also aggregated the data by participants.

AOI Line and Block Model

The bases of every analysis that involves eye tracking data are fixation hits on specific regions of the source code examples or surrounding elements. In the source code examples, AOIs were placed around every code line (*AOI line model*) or every important workspace area (*AOI block model*) with a maximum margin between the AOIs and a small padding within the AOIs. These were marked a) with letters in the line model from top to bottom starting with A; b) with additional AOIs for the question and answer areas; and c) with $A = answer$, $C = code$, and $Q = question$ workspace area model. In addition, non-hits were labeled with "_" to identify gaps in, for example, transitions. These labels are used in all further analyses to refer to the specific regions in the source code. The recorded raw eye tracking data was used to calculate fixations, utilizing a *I-VT* filter with a maximum radius of *60 px*, a minimum fixation duration of *60 ms*, and a maximum of *55* missing gaze samples in order to count as a fixation.

Combined Metadata

The metadata for all participants, including the demographic data; the answers to the source code comprehension questions; and the calculated eye movement metrics were saved in one Excel file for an overview and for simpler analysis. The data of the UEQ-S was analyzed with the official *Data Analysis Tool*[19]. As in the previous case studies, the primary analysis was done by the *F#* scripts, which were slightly modified for this study.

Sampling Bias

According to the methodical eye tracking challenges reported in Section 4.1, the following aspects were addressed to minimize the methodical challenges and resulting problems. The involved eye tracker was the same as the one employed in the code smell study: the *Tobii Pro TX 300* with a sampling rate of 300Hz, which is sufficient to record fixations and saccades (both are important in source code comprehension research). As the filter, the default I-VT filter proposed by Tobii was used with the default values reported above. For the AOIs, the maximum margin AOIs were utilized to surround every source code line and important regions with a maximum margin between the AOIs and small padding within the AOIs. The imbalance between male and female participants with a factor of approximately 2.5:1 is the best in all three studies. It slightly breaks the typical distribution of student male and female genders in computer science and engineering faculties, where the data was collected. The improved gender balance was achieved by posting recruiting calls for participants not only within the computer science faculty but on the campus of the University of Applied Sciences and Arts, Dortmund. The call addressed everyone with knowledge in programming and Java, and this could have had an effect on the participant group. Another sampling bias based on aggravating recording conditions, such as makeup and glasses, did not require special monitoring, because no participant was excluded based on both conditions. Every time a participant could not be calibrated correctly, the calibration was repeated without glasses

[19] `https://www.ueq-online.org/` – Last accessed on December 14, 2020.

as a test to note differences, which were not present. For the analysis in this study, participants were excluded due to a low eye tracking data quality with diverse reasons such as medical conditions with the eye and different lighting throughout the day. A bias towards programming environments was avoided because the source code examples were presented in a self-development programming environment with specific requirements for this study. Furthermore, the user experience was tested within the study to ensure average user experience and avoid significant problems in using the environment. The comprehension questions targeted the dynamic run time behavior of source code, which is an improvement to the EMIP dataset recordings.

7.2 STUDY RESULTS

The following section describes the results for the research question and for every hypothesis (refer to Section 7.1.1). Before the results are presented, the evaluation of the UEQ-S is reported as the premise for the usability of the study prototype.

UEQ-S AND GAZE-ENABLED INTERFACE

UEQ-S for the Prototype User Interface

To gain an impression of the user experience of the study prototype, a *user experience questionnaire (UEQ)*[20] was included in the study. After the first three source code comprehension tasks of Phase I, the participants had to rate the interface with the short version of the UEQ (UEQ-S) (Schrepp, Hinderks, & Thomaschewski, 2017a, 2017b). This questionnaire was conducted to obtain a fresh interpretation of the source code visualizations and learning hints. The assessment of the gaze-enabled interface was done with an interview at the end of Phase II. The reason for the simple UEQ was that the participants already performed three tasks that required intense focus because of the eye tracking recording. In addition, the gaze-enabled interface was presented and used thereafter, and not too much time should be occupied for the UEQ. For this pre-study of the gaze-enabled interface, included in the overall learning hints study, the small version of the UEQ leads to sufficient insights. For the UEQ-S, all 29 participants were used, instead of the 24 with correct eye tracking data, because the eye tracking data does not affect the perception of the interface in terms of the UEQ-S.

Slightly Above Positive Evaluation

With the UEQ-S, the dimensions *pragmatic quality* (goal-directed) and *hedonic quality* (not goal-directed) of Phase I can be reported. The overall UEQ-S scale indicates that the pragmatic quality has a *positive evaluation* (1.009), and the hedonic quality has a *neutral evaluation* (0.595). Thus, the overall scale has a *positive evaluation* (0.802), with the note that the value is slightly above the positive evaluation of 0.8. Figure 7.11 depicts the values per item of the UEQ-S.

[20] UEQ(-S) Website: `https://www.ueq-online.org/` – Last accessed on December 14, 2020.

Item	Mean	Variance	Std. Dev.	No.	Negative	Positive	Scale
1	1,0	0,8	0,9	29	obstructive	supportive	Pragmatic Quality
2	1,1	1,7	1,3	29	complicated	easy	Pragmatic Quality
3	1,2	1,0	1,0	29	inefficient	efficient	Pragmatic Quality
4	0,8	2,5	1,6	29	confusing	clear	Pragmatic Quality
5	0,5	1,9	1,4	29	boring	exciting	Hedonic Quality
6	1,0	1,3	1,1	29	not interesting	interesting	Hedonic Quality
7	0,6	2,9	1,7	29	conventional	inventive	Hedonic Quality
8	0,3	2,6	1,6	29	usual	leading edge	Hedonic Quality

Figure 7.11: Mean, variance, and standard deviation per item of the UEQ-S.

Satisfying User Experience

To summarize the data and results of the UEQ-S, the overall user experience of the study prototype is satisfying, based on the pragmatic quality, which is the more critical scale in the current study scenario because it describes the usefulness and usability of the system. In detail, the values for the pragmatic and hedonic quality, as well as the overall evaluation, are *below average*, and Participants therefore presumably had no usability problems using the prototype throughout the study.

Gaze-Enabled Learning Hints Usage

Apart from the question of how the participant group rated the gaze-enabled interface, the usage count and time of the gaze-enabled learning hints were also analyzed. Every time a participant activated a hint, an entry was written into the event log, including the type of learning hint that was activated, the timestamp, and the element in the source code with the hint. For this analysis, all 29 participants who performed the exploratory task at the end of the study were considered. The eye tracking data quality for Phase II was good for all participants after the second calibration; therefore, no participant had to be excluded. The lowest use count for the gaze-enabled learning hints was 99 (Participant 3), and the highest was 798 (Participant 8). The average use count for all participants was 406.86 ($SD = 188.95$). If it is considered that the activation of the learning hints is done via a fixation, then a use count below the threshold of 200 seems to be minor. The reasons could be an insufficient re-calibration of the participant or that they did not use the learning hints often. Only five participants were below this threshold (Participants 3, 29, 27, 15, and 24) – correlated with the answers in the semi-structured interview, they are not the participants who gave lower ratings for the gaze-enabled interface overall or for the tooltips in particular. Moreover, they all mentioned requirements for improving the system, but they are not in the group who disliked the system as a whole. Therefore, a non-optimal re-calibration phase of the eye tracker seems to be the reason for the values, which influenced the precision and thus the gaze-enabled learning hints usage. Furthermore, for all participants, an investigation was undertaken to determine whether a correlation exists between a high use count of the gaze-enabled learning hints and positive answers in the semi-structured interview, but no connection could be found, neither for a high use count and positive answers nor for a low use count and negative answers. All participants seem to have used the learning hints more or less frequently and made suggestions for improvement.

Usage Distribution and Time

The use count of learning hints indicates how often the gaze-enabled features were used, but not for how long they were used. A connection exists between use count and use time because utilizing learning hints is presumably often correlated with using them overall over a longer period, but this is not always the case. The event log and the learning hints, especially the tooltips, were designed in such a way that using a hint for, for example, 5 sec; activating the tooltip; and reading the line to which the tooltip is attached will generate only one entry in the event log. Therefore, it is possible to have a higher usage time. The lowest use time among all participants was $76.40\,sec$ (Participant 15), and the highest was $605.31\,sec$ (Participant 28). These values are not correlated with the lowest or highest use counts of the previous analysis. Furthermore, the average use time was $354.38\,sec$ ($SD = 142.94\,sec$). Again, there is no correlation between participants using the learning hints for a longer period and more positive answers in the semi-structured interview or between the lowest period and more negative answers. All participants seem to have used the learning hints for more or less time, and they made suggestions for improvement. Apart from the use count for the three different gaze-enabled learning hints, *scope*, *variable*, and *tooltip* were also analyzed separately. Scope means highlighting source code blocks, while variable means highlighting the use of a variable, and tooltips are those with short descriptions about the elements to which they are attached. For the scope hint, the average use count for all participants was 29.38 ($SD = 16.38$); for the variable hint, the average use count was 335.10 ($SD = 156.33$); and for the tooltip hint, it was 42.38 ($SD = 25.49$). The data suggests that the variable highlighting was used the most. The reason for this could be that there are more variables in the source code than code blocks for the scope learning hint and tooltips. Interestingly, the participants used the tooltip more often than the scope highlighting, which was not expected, because highlighting a scope is more convenient than tooltips, especially when it comes to activating such hints with the gaze. Overall, the use count of the three different learning hints is satisfactory, which makes the participants' suggestions in the semi-structured interview to improve the learning hints even more important.

High Usage Count and Time on Average

To summarize the findings, on average, a high usage count and usage time for all participants can be noted. These results are encouraging because no participant gave up early. The participants were told that they could stop the exploratory use of the gaze-enabled interface whenever they wanted to, but the results indicate that they were interested in using the new interface. The usage count of the three different learning hints was also high. Overall, the participants used the gaze-enabled hints frequently; therefore, the answers and suggestions in the semi-structured interview can be deemed valuable.

7.2.1 DYNAMIC HINT USE ($RQ_{Dynamic-Hint-Use}$)

Activity Stream

For this analysis, the study prototype collected data about the usage of the dynamic learning hint. Every time a participant highlighted the scope of a variable, method, or a source code

block, an entry was created in the event log with the *activity data* (e.g., scope or method), the *timestamp*, and the *element* for which the dynamic help was viewed. Therefore, when and how the participants used the dynamic help are analyzable because the dynamic hint is the only active learning hint.

Reasonable Threshold

For a systematic analysis of the usage of the dynamic learning hint, a reasonable threshold was necessary to split the participants into two groups to operationalize the usage scheme: one with a relatable usage count (D_{use}) and one without a relatable usage count. The first analysis of the overall use of the dynamic hint per participant revealed significant differences across the participant group. On average, $D_{use} = 34,71$, but with a high standard deviation ($SD = 50,90$). For 14 participants the value was $D_{use} \leq 20$, and the other 10 participants had values between $27 \leq D_{use} \leq 244$. For one participant (ID = 17), the data shows that $D_{use} = 35$, which is close to the average usage count. This participant had the dynamic help condition for the vehicle code example and used the hint over a total period of *3* min. The average dynamic hint use time for all participants was $98.02\,s$ ($SD = 76.08$).

Lower Than Expected Utilization

These first results indicate that the dynamic hint was used throughout the participant group, but less often than initially expected. To better understand this, the distribution of D_{use} across the various code examples was analyzed. To create two groups, the participants were separated at the threshold of 20. Table 7.1 lists the distribution of D_{use} across the source code examples, additionally with the answer quality and the average time used.

Table 7.1: Distribution of correct/incorrect answers across code examples for the two D_{use} groups with the average time used.

D_{use}	# Answers	Bubble	GCD	Vehicle	
$D_{use} <= 20$	6 correct	1	5	0	47.95 s
	8 incorrect	3	1	4	**51.46 s**
		40.71 s	46.00 s	**65.15 s**	Ø s
$D_{use} > 20$	4 correct	2	1	1	137.58 s
	6 incorrect	2	1	3	**183.79 s**
		153.76 s	153.21 s	**182.90 s**	Ø s

Average Usage Data

Table 7.1 presents data regarding the average time the dynamic learning hint was used; this data was analyzed for the correct/incorrect groups and the three different learning hints. According to the findings, the average usage time was higher for both incorrect groups and higher for the vehicle code example across all groups. The first result implies that participants with problems during the source code comprehension task did not necessarily use the dynamic hint more often regarding the usage count D_{use}, but they used it overall for a more extended period of time compared to the correct groups. It can be assumed that they tried to use the hint as a helpful resource but that this was not successful, as is evident in

the comprehension answers. The second result implies that participants with an active dynamic learning hint for the vehicle code example used this type of help more often than for other source code examples. However, again, the data suggests that this was not successful in terms of more correct answers.

Cross-Referencing Java Knowledge

Furthermore, an analysis was performed to determine whether participants with excellent results regarding their Java knowledge in the pre-test and the self-assessment used the dynamic learning hint more often, which would indicate that more experienced participants knew that this learning hint could help them and tried to make use of it. For the pre-test, a rating scheme was used that divided the participants into two groups: Those with 70 % or less correct answers fell into the low group, and more than 70 % of the participants belonged to the high group. In addition, the participants had to answer programming experience-related questions for various programming languages in the questionnaire. The answers for the Java programming language were used in this analysis because the source code examples were implemented in Java. The experience rating could be scored between *very good (1)* and *no knowledge (5)*, but no participant selected the latter option, so the answers 1/2 and 3/4 were grouped into two groups. The questionnaire and the pre-test data are available in Section 7.1.6. Table 7.2 lists the usage time of the dynamic learning hint, divided into groups based on the pre-test and additionally on the self-assessment (experience with Java).

Table 7.2: The usage time of participants, divided into groups based on the pre-test or self-assessment (Java experience) results.

Data Source	Thresholds	# Participants	Usage (Ø s)
Pre-Test	≤ 70	4	59.14 s
	> 70	20	105.78 s
Self-assessment	groups 1+2	14	69.06 s
	groups 3+4	10	138.56 s

Increased Usage Time for Experienced Participants

These results indicate that the average usage time of the dynamic learning hint increased for more experienced participants. While this analysis is based on the pre-test and the self-assessment, and the data may be inaccurate, the analysis nevertheless highlights a trend that seems to be realistic.

Used more by Participants with Incorrect Answers

The data shows that the hint was used more often by participants who ended up given incorrect answers. The learning hint thus did not seem to help in answering the comprehension questions. Based on the source code examples, the hint was used overall more often for the vehicle stimulus, which is compliant with answer quality. In addition to these results, the usage time was higher for more experienced participants. On the one hand, this is not surprising because these participants already knew how the hints work. On the other hand, according to the quality of the answers, the learning hint usage made no difference. The fact

that less experienced participants did not use the hint is essential for programming education. It is necessary to explain these features to less experienced learners and to show them how dynamic learning hints can be used.

7.2.2 ANSWER QUALITY ($H_{Answer-Quality}$)

Eye Movement Metrics

Answer quality is one basic dimension in a source code comprehension study to analyze the comprehension quality of participants. The results are described below. In addition, the duration and fixation count eye movement metrics are presented to analyze differences for the source code examples and study conditions.

Vehicle with Less Correct Answers

The overall correct answers for the Bubble source code example, for the GCD, and for the Vehicle are 12, 12, and 5 respectively. Therefore, the Vehicle code example seems to be the most complex one of the three, based on the answer quality. This result is contrary to the assumption that the Bubble is the most complex code example (nested loops). It seems that many participants had problems with the object-oriented task. If the time limit of every code example is additionally considered, then the impression can be confirmed that many participants had problems with the Vehicle task. Four participants exceeded the time limit for the Bubble source code, two for the GCD, and nine for the Vehicle code example. The overall poor results for the Vehicle source code may be related to the Java skill distribution among the participants. As reported in Section 7.1.6 eight participants selected "medium" and two "bad" for expertise in Java. Thus, this can be an influencing factor.

Learning Hint Availability

Furthermore, it is interesting to analyze which learning hint was available for which source code example for the participants. Table 7.3 lists the number of correct and incorrect answers to the learning hints per source code example.

Table 7.3: Correct and incorrect answers for all three code examples and for all three learning hints (S → Syntax, D → Dynamic, and P → Plain).

Code	# Answers	S	D	P
Bubble	12 correct	4	3	5
	12 incorrect	4	5	3
GCD	12 correct	4	6	2
	12 incorrect	4	2	6
Vehicle	5 correct	3	1	1
	19 incorrect	5	7	7

Syntax Highlighting Balanced, Dynamic and Plain Indecisive

The data shows that the syntax learning hint was balanced for the correct and incorrect answers. Syntax highlighting does not seem to be an essential factor related to answering a comprehension question for the code examples. While the navigational aspect of the underlying source code comprehension may be different, the core of the source code examples

is the same: A complex code is still complicated. Furthermore, the differences between the dynamic learning hint and the plain code examples are indecisive for the Bubble and GCD code examples. For the first code, plain had a more significant effect on correct answers than the dynamic learning hint (five to three participants). This result was reversed for the GCD code examples, with a more substantial effect for the dynamic help on correct answers (two to six participants). This is positive for the dynamic learning hint of the GCD code example and interesting for the Bubble code example. Overall, a more in-depth analysis is required to determine how often the dynamic learning hint was used across code examples. For the Vehicle code, both conditions with the dynamic help and the plain text seem to have had no positive effect on the comprehension result. Again, while this result must be analyzed further regarding how often the dynamic learning hint was used, it is overall in line with the results of the answer quality analysis. In general, the Vehicle code example seems to have been more difficult compared to the other two.

Table 7.4 lists the overall fixation count and fixation time for every stimulus, additionally separated per learning hint and therefore study condition. The highest numbers are emphasized per line per learning hint and the overall highest numbers per stimuli.

Table 7.4: Overall fixation counts and fixation times per stimuli, additionally separated by learning hint (S → Syntax, D → Dynamic, and P → Plain). The highest values are emphasized.

Code	S	D	P	Overall	
Bubble	4,330	4,472	**4,525**	13,323	fix.
	1,313.12	**1,317.69**	1,314.57	3,945.37	sec.
GCD	**4,272**	3,891	3089	11,231	fix.
	1,189.53	1,037.66	892.33	3,119.51	sec.
Vehicle	4,112	4,956	**5,638**	**14,690**	fix.
	1,158.85	1,435.62	**1,648.88**	**4,243.36**	sec.

Fixation Count and Time Metrics

These metrics already reveal some interesting results. The source code examples were originally designed with varying complexity levels. According to the fixation count and fixation time distributions in Table 7.4, the data does not seem to match the assumptions. Both the overall fixation counts and fixation times were the highest for the Vehicle code example. Various studies on source code and UML models have found that a higher fixation count indicates more visual effort to perform bug fixing tasks (Sharif & Maletic, 2010), find defects (Sharif et al., 2012), perform the task at all, or recall the name of identifiers (Sharafi, Soh, Guéhéneuc, & Antoniol, 2012). Therefore, high values in the data presumably indicate more visual effort, at least for the fixation count. In addition, if participants overall need more time on a stimulus, then this could be linked to visual effort, which can be related to source code complexity. The higher values for the plain learning hint for the Vehicle stimulus

are as expected, since plain source code should be more difficult to read and comprehend without syntax highlighting than with it. What is surprising is that for the plain condition, only the fixation count for the Bubble stimulus had higher values. The plain condition should be the most difficult to read for every stimulus; however, this was not the case. Moreover, the values for the fixation counts and fixation times for the syntax highlighting condition were the highest for the GCD code example. On the one hand, this is the code that is considered to be the medium complex one, but the syntax highlighting should have been useful in that regard. On the other hand, high values were assumed for the GCD stimulus, but not in the syntax highlighting condition.

Plain Condition Influencing

To summarize the data in Table 7.4, the plain text condition seems to be an influencing factor for the Vehicle code example. The other two learning hints have no evident influence on the fixation count and fixation time metrics. As surprising as these results are, the differences in the values are not significant enough to assume that the learning hints have no other effect. Furthermore, participants who gave up trying to comprehend the source code after one or two minutes had a significant impact on the values of these two eye movement metrics.

Vehicle Source Code Most Difficult

As a summary to this hypothesis, the Vehicle source code seems to have been the most difficult one for the participant group. One reason could be the semantic understand of a class called "Vehicle." This is a semantic level other than the GCD or Bubble source code, and it uses well-known algorithmic concepts. Another explanation for the answer quality of the Vehicle source code is that the object-oriented structuring is problematic for a considerable number of participants. Possible reasons are the skill level and skill distribution of the participant group. Two participants answered the Java expertise question with "bad," and eight with "medium," which is generally not particularly promising. However, the answers from participants with "bad" Java expertise were very good, and for all three source code examples, their answers were close to correct or completely correct. In addition, the results for the eight participants with "medium" Java expertise are mixed. For example, the answer quality for the third source code example (Vehicle) was exactly 50 % on average. Moreover, the ratings on the other two stimuli were close to this result, which is generally not promising. The analysis of the interviews provide no further explanation for the answer quality result regarding the Vehicle source code, because that source code was not mentioned exceptionally often in terms of the difficulty or complexity level, but it is at a similar level to the number of mentions of the other two code examples.

Plain Influences Duration Time and Fixation Count

According to the data, the syntax highlighting hint is balanced, and both the dynamic and plain hints are indecisive. The data about the eye movement metrics revealed that the plain condition seemed to influence the duration time and fixation count for the Vehicle code example. The overall time was also highest for the vehicle source code; this adds to the assumption that the source code was the most complex one for the participant group.

7.2.3 RATINGS FOR LEARNING HINTS ($H_{Ratings-Learning-Hints}$)

Learning Hints Ratings

The participants' ratings of the learning hints can offer insight into the hints' perceived usefulness. At the end of the three source code comprehension tasks of Phase I, the participants were asked to order the learning hints on a scale from *useful (1)* to *not useful (3)*, thereby allowing for the overall usefulness of the learning hints to be calculated. (The tooltip hint was not part of the rating because participants did not know this learning hint existed at this point in the study. The corresponding data for the tooltip ratings can be found later in this section.) The overall ratings of all participants were $Syntax = 1.208$, $Dynamic = 2.08$, and $Plain = 2.70$. According to these results, the syntax highlighting learning hint was rated best and much better than the dynamic hint and the plain condition. These results are in line with the initial expectations because the assumption was that a passive hint, such as syntax highlighting, is better known than an active hint, such as dynamic highlighting, and therefore receives a better rating. However, it should be noted at this point that the dynamic hint was used less often than originally assumed. The rating for this hint is thus not completely wrong or neglectable, but of limited expressiveness, because there is no way to determine whether the rating is poor because the learning hint was used less often or whether the hint was not used as often as expected because the dynamic hint was perceived as not useful.

Tooltips as Additional Hints

The second phase of the study had tooltips for the source code as an additional learning hint. In the semi-structured interview in this second phase, participants were asked what their impression of the tooltips are, how they would rate the tooltips compared to the other learning hints, and what can be improved to make the tooltips even more helpful.

Tooltips Ratings

For the tooltip rating, participants had to order the usefulness of the learning hints between 1 and 4, where 1 is the most useful hint. The results are ambiguous. Seven participants rated the learning hint with 1, 13 with the rating 2, six with the rating 3, and three with the rating 4. The average rating for the tooltip was 2.17 ($SD = 0.881$). The data basis comprised the 29 participants because the data was collected in the semi-structured interview, which was independent of the eye tracking data quality.

Good but not Overly Useful Tooltips as Result

These results reflect the statements that the participants issued as an explanation for their ratings. For the best rating, participants mentioned that the tooltips are helpful overall but that they expected more information about the element associated with the tooltip. For the second rating, the participants mentioned that the idea is promising, but that the tooltips are not as helpful as they would have expected. The reasons for these statements are that the information within the tooltips are too generic and not specific enough for the actual elements. The overall sentiment was that the tooltips in their current form are not as helpful as they could be. In the third rating category, participants mentioned that the content is not helpful, that tooltips need to be visible for longer, and that they could be annoying after a while when one's gaze activates them unintentionally. For the last category, two

participants stated that comments would be a better way of providing help and that the gaze-activation is problematic because sometimes a tooltip should be visible as an additional source of information, while one is looking at something else after a short period.

Learning Hints of Varying Utility

The answer related to this hypothesis is twofold. First, the data collection for the three learning hints, namely *syntax highlighting*, *dynamic*, and *plain*, found strong evidence that the learning hints are perceived differently regarding their usefulness. Second, participants rated the syntax highlighting much better than the rest. These results influence the perception of the complexity of the three source code examples, as the results of the next hypothesis demonstrate. Furthermore, to summarize the findings for the tooltip ratings, the participants thought the tooltips were "good" or "okay" regarding the first two rating categories. Apart from this, the results highlight the potential of such a gaze-enabled system, but they also demonstrate that the tooltips need to be adjusted and improved, primarily in terms of duration and content.

7.2.4 RATINGS FOR CODE DIFFICULTY ($H_{Ratings-Code-Difficulty}$)

Source Code Examples' Complexity Ratings

In addition to the ratings for the learning hints, data for the complexity ratings of the source code examples were collected. Every participant had to answer a question about the difficulty level of each of the three source code examples. They had to order the codes on a scale from *simple (1)* to *difficult (3)*, which allowed for the calculation of the overall complexity perception of source code examples. In general, the data for the Bubble and GCD source code examples are more comparable to each other than to the Vehicle source code. The reasons are the similar algorithmic concepts and programming paradigms used in both stimuli, whereas the Vehicle source code utilizes object-oriented code and is thus on a different semantic level. Table 7.5 lists the ratings of the source code difficulty for all participants, divided by learning hint, and the lowest values are emphasized. According to the participants, the learning hints change the complexity perception completely. As the table indicates, for the syntax highlighting condition, every value is lower (and therefore better) than the other ratings, and the differences are relatively high for the Bubble code example and very high for the GCD code example. Moreover, the syntax highlighting for the GCD code could not be rated better, and the results for the dynamic and plain conditions are inconclusive – they are matching for the Bubble and Vehicle codes, and they are even lower for the dynamic learning hint on the GCD code example. This result is in line with the answer quality described in Section 7.2.2, where the distribution of correct answers is also inconclusive between the dynamic and plain conditions.

Syntax Highlighting is Rated Best

The data in Table 7.5 matches the overall rating of participants regarding the use of learning hints mentioned in the previous hypothesis ($Syntax = 1.208$, $Dynamic = 2.08$, and $Plain = 2.70$). The ratings also match the fixation counts and duration used by the participants in the various conditions (see Table 7.4). These findings are fascinating because

Table 7.5: The difficulty rating for all three source code examples and for all three learning hints (S → Syntax, D → Dynamic, and P → Plain). The lowest values are emphasized.

Code	S	D	P
Bubble	**1.75**	2.25	2.25
GCD	**1.00**	1.75	1.375
Vehicle	**2.25**	2.5	2.5
Ø	**1.6**	2.16	2.04

most of the participants had the impression that the syntax highlighting is much more useful than the other learning hints. Twenty participants gave the syntax highlighting a rating of 1, three gave a rating of 2, and one gave a rating of 3. However, this only changed their perception of the complexity of the source code examples. The answer quality (see Table 7.3 related to the first hypothesis) reveals that the highlighting did not help them with the source code comprehension, at least not in the sense that the results are better compared to the other conditions. This result is consistent with other studies, which have also found no effect for syntax highlighting in novice groups (Hannebauer et al., 2018). Nevertheless, the subjective perception of source code complexity is supported by the eye tracking data regarding the fixation count and duration eye tracking metrics. This is indeed an interesting result because these metrics are more fine-grained than the answer quality. Overall, the subjective ratings of the source code difficulty are reflected in (correspond to) the source code reading behavior.

Learning Hints Were Recognized

One explanation for this effect is that the different source code stimuli blurred the effect of the learning hints. For the Vehicle example in particular, perhaps the code itself or the object-oriented concepts were already so difficult to understand that the availability of a learning hint played no significant role in the comprehension process. Furthermore, the semi-structured interview attempted to shed some light on the differences between the usefulness and the comprehension results. The participants were asked if they knew some of the source code visualizations used throughout the study and which advantages these visualizations could give them. Overall, 20 of the 24 participants could remember seeing the syntax highlighting somewhere else (e.g., in development environments such as Eclipse or Visual Studio Code). A more detailed analysis of the interview data revealed the following (not mutually exclusive): nine participants mentioned "better overview of semantics;" 11 mentioned "a good overview of types;" for 16 participants, the "overview of keywords" was essential; and seven mentioned an "overview of variables" as an advantage. In addition, 10 participants mentioned the "focus on important parts of the code" or "navigational aspects" as their main advantage. Most of the participants thus recognized syntax highlighting, and many of them could at least provide one advantage of the highlighting for them.

Complexity Perception Differences Found

Regarding this hypothesis, it is clear that differences were found in the participants' complexity perception. The difficulty and usefulness ratings revealed that a learning hint seems to change the way participants see a source code stimulus. However, it appears that only the perception was changed. The overall answers indicate that, especially for the vehicle source code, a hint is not always helpful; in some cases, it might be distracting or annoying. Regarding the source code comprehension community and the studies conducted in this field, the study provides the first sign of evidence that a learning hint has no effect at all, respectively an effect, which is not as high as originally thought. These results are notable for syntax highlighting. Nevertheless, because of the inconclusive specific results for syntax highlighting, a recommendation is to provide at least basic syntax highlighting to help individuals read the source code and for navigational tasks.

7.3 THREATS TO VALIDITY

While analyzing the recorded data, several threats to the internal and external validity were noted.

Internal Validity: OOP Code, Known Code Examples, Self-Assessment, and Pre-Test

Internal validity. The following limitations for causal conclusions apply to the learning hints study. a) The vehicle code example used OOP. Although OOP concepts are taught in initial courses on campuses, not all participants knew these concepts, and this was not controlled for in the study. b) The code examples were carefully selected to match code snippets that participants may already have encountered during their studies. Nevertheless, some participants may have worked on similar examples recently in an exam or course, and they would have therefore had an advantage in answering behavioral source code comprehension questions. c) In addition to a self-assessment, a pre-test was added to obtain additional data about the skill level of participants, especially regarding foreknowledge in computer science and programming. This worked well but is only an approximation of the real knowledge. Therefore, measurements based on the skill level cannot be guaranteed to be precise enough. d) The code example in Phase II of the study was the GCD example, which was already used in Phase I. This may have caused irritations when the source code was presented a second time, which could then have led to participants not using the learning hints properly. e) The goal for the participants in Phase II was to trial the gaze-enabled interface. While the results presented in the research questions indicate that this went satisfactorily, a real comprehension question and a time limit for answering it could have had a better effect on using learning hints due to external motivation.

External Validity: Non-Representative Examples, Eye Tracking Calibration, Self-Developed Study Prototype, and Java Programming Language

External validity. The following limitations for generalizable results were noted. a) Since the code examples had to be relatively small for an eye tracking study without specialized tools, the code snippets were not representative of large systems or code bases. In addition, the source codes were formatted to have the option for tracking single tokens, which means adding spaces between keywords and semicolons to obtain better fixation matches on single elements. This visualization may have interfered with the proper use of the learning hints in Phase II of the study. b) Even if the eye tracking calibration was repeated before starting the exploratory phase, the gaze-enabled learning hints are highly sensitive to a well-calibrated participant, and an eye tracker that was not well calibrated could have interfered with the proper use of the gaze-enabled learning hints, which is even more critical here than in Phase I because the usability and interactivity are based on accurate eye tracking data. c) The study prototype was explicitly designed and implemented for this study. Even though usability was tested in a pre-study, and the results are above average, it cannot be guaranteed that the prototype interfered with the results, because participants compared the software with full-sized development environments. d) The analyzed eye movement data was based on the Java source code used in the study, and source code different from Java will likely produce other eye movement patterns and results.

7.4 SUMMARY

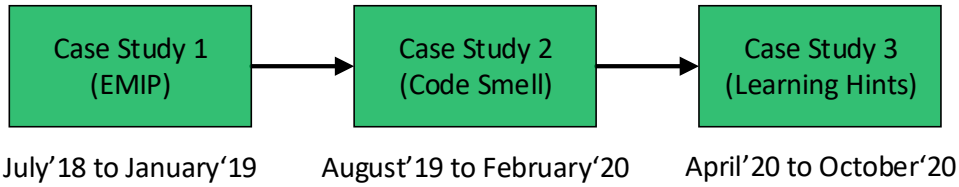


Figure 7.12: A brief overview of the case study timeline (learning hints study completed).

Answer Quality and Learning Hints Influence

The main objectives of this study and the subsequent analysis were a) to reveal differences in the answer quality of participants, including eye movement metrics; b) to analyze how learning hints influence source code comprehension regarding answer quality (correct/incorrect); c) to use the dynamic learning hint; d) to understand how participants perceive the learning hints; e) to determine how they influence the perception of complexity regarding source code examples; and f) to understand how a gaze-enabled user interface is rated and used by the participants, all with the different code examples and learning hints in mind. The third case study is thus complete (see Figure 7.12).

Satisfying UEQ-S Results

The results of the UEQ-S and the ratings for the tooltips are satisfactory. The data suggests that the interface is one step in the right direction. It is positive that many participants made suggestions and requirements for future versions of the gaze-enabled interface in the interviews. Apart from this, the study analyzed how gaze-enabled learning hints are used,

how useful the gaze-enabled interface was, and how the tooltips were rated and used compared to the other learning hints.

Dynamic Learning Hint Less Used

The dynamic learning hint was used less than expected. The assumption was that the target group, with knowledge in programming and therefore development environments, would use this hint more frequently. Overall, the hint seems be useful for the GCD example but ambiguous for the other two source code examples. On the one hand, this finding requires more in-depth analysis and a specific study of whether the dynamic learning hint is a candidate for the dynamic learner support system. On the other hand, it is possible that the hint is too distracting to obtain useful eye tracking movements and is therefore not an ideal choice for an automatic support system.

Vehicle Source Code most Difficulty

Based on participants' answers, the Vehicle code example was surprisingly the most difficult one. The initial assumption was that the Bubble code is the most complex one regarding the complexity of the program structure (nested loops). However, the study revealed that many participants had problems with the object-oriented code, no matter which learning hint was available. Possible reasons for this outcome are the skill level and the skill distribution of the participant group and the differences in the semantic level of the Vehicle source code because it uses object-oriented rather than arithmetically oriented code as well as procedurally comparable concepts such as Bubble and GCD. The real reasons cannot be provided with the data available for this study, because the skill-level answers, the pre-test, and the interviews have limited expressiveness regarding this aspect.

Learning Hints Perception and Ratings

Another goal of the study was to investigate how participants perceive learning hints, how they influence the perception of complexity regarding source code examples, and how a gaze-enabled user interface is rated and used by the participants. Learning hints are indeed changing the viewpoint of source code examples and their complexity. Participants rated the syntax highlighting learning hint better than the other hints, and they rated the source code examples with the syntax highlighting hint active as the easiest ones. Therefore, it can be assumed that learning hints influence the perception of code and thus the source code comprehension process. Syntax highlighting could change the way participants perceive source code; however, the core, similar to the complexity, remains the same.

Common Eye Movement Patterns

Regarding the reading patterns on both AOI models, a common analysis in the previous two case studies, common patterns across all participants, code examples, and learning hints could be found. The bases for this analysis, without a direct relation to the hypotheses described in this chapter, are the participants, who were used as grouping levels. The analysis demonstrates that these patterns form groups through calculating the edit distances (N-W) and with hierarchical clustering. Furthermore, a first analysis revealed that these patterns are not the distinguishing factors for the answer quality of participants. The reasons for this result need to be analyzed further to exclude factors of the study conditions and the participant group, since a group with more knowledge in OOP, for example, could change

the outcome of the analysis considerably. The results are in line with the findings of the EMIP analysis and the code smell study: Common eye movement patterns can be found, and the AOI-DNA, AOI-STG, N-W, and TraMineR methods are useful, but there is no evidence of dominant patterns throughout participant groups, which can specifically explain a positive or negative outcome of a source code comprehension task, at least based on the used methods.

Gaze-Enabled Interface Possible

One overall result is that building a gaze-enabled user interface or including gaze-enabled learning hints in a programming environment is possible. However, this has to be done carefully. Another overall result is that participants expressed doubts regarding the calibration phase, the precision of eye tracking itself, and the usefulness of gaze-enabled learning hints for non-novices; these doubts should be addressed in future versions of the interface. Moreover, some participants mentioned the acceptance of eye tracking as a crucial factor for implementing such a system into an IDE used for daily tasks. This finding should also be addressed, for example with smaller eye tracking devices that are less apparent, much cheaper, and usable in more scenarios.

"Honest disagreement is often a good sign of progress."

– Mahatma Gandhi

8 Contributions and Discussion

Three Aims as Foundation

The introduction of this thesis (see Chapter 1) outlined the problem statement, the overall vision, and the three specific aims, which led to the research agenda for this thesis. These aims, tailored to detecting source code comprehension strategies, methods and tools, as well as describing the toolbox, are revisited in this chapter to summarize the individual contributions. In addition, a discussion is presented regarding how these achievements can help in developing the idea of a dynamic learner support system and which steps are reasonable for future development.

8.1 First Aim – Detect Source Code Comprehension Strategies

Reliable Detection and Categorization Necessary

The reliable detection and categorization of source code comprehension strategies (see Section 2.3.3 for comprehension strategies) is a necessity in source code comprehension research. The contributions in this thesis focus on comprehension strategies, which can be defined and detected by eye movement patterns. Furthermore, the test of learning hints regarding their usefulness (see Chapter 7 for the learning hints study), their effect on source code comprehension, and their influence on eye movement patterns is part of the first aim. The data collection as well as data analysis to support this aim were conducted in three case studies. The contributions of this thesis are marked as C_n in the following text.

FINDING STRATEGIES

Top-Down and Bottom-Up Approaches

The approaches used for finding eye movement strategies and patterns were first tested on the EMIP dataset, primarily with top-down and bottom-up approaches based on AOIs (see Chapter 5 for the EMIP analysis). The design of steps to detect comprehension strategies is important in source code comprehension research. One contribution is that the definition of AOIs is simultaneously important and yet a non-standardized process in the source code comprehension community, often done manually or with little tool support[21], which leads to AOIs that vary in size and location in relation to the source code elements. These non-standardized approaches are problematic for the comparability and reproducibility of results across researchers.

Contribution C_1

Defining AOIs needs to be a standardized process in the source code comprehension community to allow for comparisons and reproductions of the results across researchers.

AOIs with Padding and Margins

The investigations conducted to analyze the AOI sizes, published in EMIP'19 (Deitelhoff et al., 2019a), came to the conclusion that AOIs must be defined with a small padding around the source code elements (see Section 4.2), such as tokens and lines. Moreover, with respect to the aforementioned padding, the margin between the AOIs needs to be as maximal as possible for lines and regions. This reduces noise when matching fixations and AOIs.

Contribution C_2

The padding in an AOI needs to be defined, respecting common eye tracker parameters (degrees of visual angle), and the margin between AOIs must be defined as large as possible.

Such decisions and the overall process should be made public for every analysis and in every publication. Without knowing AOI parameters, the chances of reproducibility are slim to none.

Contribution C_3

To foster reproducibility, AOI parameters must be made public for every analysis and publication.

[21] eyeCode: `http://synesthesiam.github.io/eyecode/` – Last accessed on December 14, 2020.

First Categorization of AOI Models

Another contribution is a first categorization for AOI models, how they can be defined, and which characteristics they model. Three commonly used examples in this thesis are the *line*, *region*, and *workspace area* models. The region model is modeled as an abstraction of the line model in this thesis, whereas the workspace area model is designed to capture AOI-Hits for characteristic areas of a workspace. It was therefore surprising that a comparison between a line and a workspace area model found similarities, which were visualized with dendrograms. Thus, AOI models provide a different view on the same eye movement data, which can be useful in the analysis process.

Contribution C_4

AOI models provide a different view on the data. A standardized definition of AOI models could help the source code comprehension community.

SOR and EOR

Furthermore, common eye movement patterns such as SOR and EOR depend on the mentioned AOI definitions (see Section 2.4.4 for patterns of novices, and Section 2.6, for methods to analyze eye movement patterns). Moreover, they depend on the source code example used in a study and the way in which the patterns are defined. Some are more explicit, for example a SOR pattern starting from the top and moving to the bottom of the source code, while others offer a certain scope for interpretation (e.g., the EOR pattern). This pattern can be defined as a perfect path following the execution of a code snippet. Another definition would consider minor deviations that a reader would apply when reading the code. The fundamental question is whether an EOR pattern should be defined based on the perfect technical flow of code execution or with logical deviations based on source code comprehension. This becomes even more complicated when source code comprehensions with different arguments for functions or methods are asked, because the execution order of the source code is highly dependent on this data. Moreover, the paradigm(s) provided by the programming language has a high impact on the reading order of participants. Functional programming is accountable for completely different patterns than procedural or object-oriented source code. These differences must be taken into account when hypotheses about participant behavior are created and when the data is analyzed.

Contribution C_5

A standardized definition of eye movement patterns per programming paradigm is necessary to build a common understanding of these patterns. Overall, a standardized process for defining these patterns is crucial to support source code comprehension research. In addition, common coding schemes for common programming languages are necessary to add to this understanding in the community.

AOIs Depending on the Source Code Structure

A major shortcoming with AOIs and the patterns defined on this basis is the complete dependence on the underlying source code structure and programming paradigm. While the latter is difficult to change because object-oriented source code is not nearly identical to, for example, functional code, a solution to the first problem must be found to move the source code comprehension research forward. Changing the structure of a source code example (e.g., changing the structural organization of code such as curly braces) currently still results in a changed eye movement pattern, which makes it much more challenging to find generalizable results. Abstractions or allowed variances in eye movement patterns are needed, since they can compensate for some source code structure-induced problems and differences.

Contribution C_6

The abstraction of eye movement patterns from AOIs is necessary to gather generalizable results because AOIs are inherently dependent on the stimulus material.

Comprehension Question Target and Time

Finding comprehension strategies and eye movement patterns depends on precise comprehension questions at the correct time. Questions should not target the specification of source code if not absolutely necessary. How something is implemented is easier to memorize than how things turn out if the source code needs to be evaluated and executed with actual data in the mental model because this mental model is created by reading and comprehending the source code. Apart from this, the questions should be asked before the source code is visible or simultaneously with the source code. This removes the effects from simply remembering parts of the source code stimulus, which is an odd way of measuring comprehension.

These thesis results are contrary to many source code comprehension studies that have been conducted in the source code comprehension community. The recommendation is to use behavioral comprehension questions with a question visible before and after the source code stimulus or simultaneously with the stimulus.

Contribution C_7

Source code comprehension questions should target the execution of source code (behavioral questions), and the questions should be asked before and after the source code is visible or simultaneously while showing the code stimulus.

NOVICES AND EXPERTS

More Accurate Self-Assessment or Pre-Test

Comparing novices and experts is important to find problems in the source code comprehension process of learners. Using a heterogeneous group of novices as the baseline to find experts in it is a delicate process because in most studies, the evaluation of skill depends on a self-assessment component in a survey. This is error-prone as it does not reflect the real-world appropriately. Nevertheless, the data about participants' knowledge is important because it can be used to evaluate other aspects of the recorded data. This was done in the learning hints study (see Chapter 7). The subjective ratings of the source code difficulty corresponded to the source code reading behavior, which is an interesting result (see Tables 7.5 and 7.4 respectively). Matching objective eye movement data with the knowledge assessment data is an important step in source code comprehension studies.

Contribution C_8

The self-assessment for measuring knowledge or experience is error-prone and needs to be replaced by a more reliable and objective approach. A pre-test can be an ideal replacement if a balance can be found in the degree of difficulty of the questions, which is not a simple task either.

Select Experts with Caution

In contrast, using software developers as the expert group promises better results. However, these experts should be selected with the same caution and diligence as novices, as the second case study demonstrated (see Chapter 6). The knowledge gaps in an expert group can be as large as in the novice group, and too much trust in the expertise of experts, simply because they are experts, is not recommended.

Contribution C_9

The same caution is required when selecting experts as when selecting novices. Furthermore, if possible, experts should be recruited from different companies to distribute the knowledge and training background.

Code Reading and Explaining are Important

The use of an RTA revealed different results. In this task, experts are much more consistent because they can explain what they read and have a sense of why they read it at a specific time. In addition, they can use appropriate vocabulary to explain their reading process, which is not the case for novices. Therefore, code reading and code explaining should be taught explicitly. Both abilities should not be seen as side products of code writing.

Contribution C_{10}

The quality of explanations in an RTA varies significantly, partly because of the missing vocabulary of participants. Code reading and code explaining should thus be taught explicitly when learning programming.

LEARNING HINTS

Learning Hints and their Influence

Another approach for detecting source code comprehension strategies is to use some form of learning hint in the source code examples (see Chapter 7). The goal was twofold: a) test learning hints and their usefulness, and b) analyze the data to detect varying comprehension strategies and eye movement patterns when the hints are active. Hints have an influence on the complexity perception of source code examples. When hints, such as syntax highlighting, were active, the source code was rated as less complex and complicated compared to a source code without the highlighting, even if the answers to the comprehension questions do not reflect this. This result is important for further studies because it demonstrates that a learning hint can be misleading when source codes are rated. They have an effect on the eye movements, but not consistently. Moreover, passive learning hints were "used" more often compared to active hints implemented as on-demand hints, which needed to be used with the mouse. This effect of recall, rather than only recognition, must be considered in future studies.

Contribution C_{11}

Learning hints can change both the complexity perception of source code and eye movement patterns. However, the hints did not change the outcome of the source code comprehension questions significantly. This result must be taken into account when designing a source code comprehension study.

General Availability of Syntax Highlighting

Further studies should answer the question of whether syntax highlighting as a passive learning hint should be available in source code comprehension studies. It changes the perception of the code, and the unavailability of the highlighting could hinder an interpretable study result.

Furthermore, the gaze-enabled interface was easy to implement, rated high by participants, and used with curiosity. However, when such an interface is considered for implementation in a real-world application, the experiments in this thesis indicate that thorough planning is essential. Such an interface is too uncommon and needs much training. Moreover, eye tracking as a technology must be more reliable, be more transparent, and work

without a full calibration process. These were requirements mentioned by participants in the semi-structured interviews.

Contribution C_{12}

A gaze-enabled interface needs thorough planning and training. Furthermore, it depends on an even more reliable eye tracking technology.

The learning hints overall, as implemented and used in this thesis, were helpful in terms of comprehension processes, but not as equally helpful for detecting eye movement patterns.

8.2 SECOND AIM – IDENTIFY, CREATE, AND TEST METHODS AND TOOLS

Specialized Tools and Methods

To detect and analyze source code comprehension strategies and eye movement patterns, specialized tools and methods are necessary. Some are already successfully used in the source code comprehension community, while others have specifically been developed over the course of this thesis. The following contributions and discussions focus on these results.

AOI-DNAs

AOI-DNA as Representation...

The matching of eye tracking fixations and AOIs, which are distinctly labeled, are called AOI-Hits. This encoding allows it to represent eye movement data as a sequence, called AOI-DNAs. This representation is used throughout this thesis and is a solid foundation for analyzing and visualizing eye movement data.

Contribution C_{13}

The primary representation used in this thesis is comprised of AOI-DNAs, which are easy to create and useful for various visualizations and calculations.

...and Color-Coding...

AOI-DNAs were the basis for a diverse set of visualizations. When the same labeled AOI-Hits in an AOI-DNA are marked with the same color, eye movement patterns can be visualized with a small amount of effort. Specific color codes for different eye movement patterns can visualize their existence or absence, even for a large number of AOI-DNAs. This was demonstrated for the SOR, EOR, flicking, and retrace declaration patterns with color gradients and with color codes, where only a few AOI-Hits were highlighted while the rest were left intentionally blank.

Contribution C_{14}

AOI-DNAs can visualize eye movement patterns with different color codes and color gradients.

...and Sequence Alignment...

Furthermore, AOI-DNAs are the bases for different sequence alignment methods. The representation as a continuous string of characters is ideal for using methods such as N-W and OM. Therefore, AOI-DNAs are viable not only for visualizing eye movement patterns, but also for comparing them with alignment algorithms based on a similarity score. The N-W algorithm is often used in the source code comprehension community to calculate the similarity score for different sequences with the intention of projecting these results to different levels of the data. Grouping by participant, answers, and other aspects are thus possible. Other methods and algorithms, for example those used in social science, should be explored because the results in this thesis, based on OM and TraMineR, are promising.

Contribution C_{15}

AOI-DNAs are useful as the bases for sequence alignment methods such as N-W and OM.

...as well as Pattern Matching

Moreover, the representation as AOI-DNAs has proved to be useful for a pattern matching based on regular expressions. For this purpose, the AOI labels were used in a regular expression with varying arbitrary AOIs in between. An eye movement pattern can thus be represented as a regular expression, which is then used on every AOI-DNA to return all matches. Tests revealed that this can work for smaller eye movement patterns (e.g., flicking and retrace declaration). However, finding larger and thus more complex patterns such as EOR is more difficult because with every new AOI in a regular expression, the variance of possible AOIs in between the wanted AOIs increases. This fact makes the pattern matching in the current form only usable for smaller patterns.

Contribution C_{16}

The representation as AOI-DNAs is useful for a pattern matching based on regular expressions. The results have shown that this approach, in the current stage of development, is only useful for shorter eye movement patterns.

AOI-STGS

AOI-STGs as Model

The representation as AOI-DNAs is not useful solely for visualizing or searching of eye movement patterns. These sequences of continuous strings of characters can be seen as the bases for some sort of state transition graphs. A variant of these graphs, called AOI-STG, was developed in this thesis. This is not another representation, but a model for visualizing states (AOIs) and the transitions between them, which are implicitly encoded in the AOI-DNAs. Every edge is marked with the index of the AOI-DNA transition, which allows for the sequence within the graph to be seen. Due to the dot visualization and the used layout algorithm, strongly connected clusters are also visible, along with high dwell times on AOIs, which are called fountains.

Contribution C_{17}

AOI-STGs are valuable models for spotting strong transition characteristics of AOI-DNAs, such as strongly or loosely connected clusters and high dwell times (fountains).

8.3 THIRD AIM – THE TOOLBOX

CodeSight as Analysis-Tool

The third aim of this thesis is to propose a toolbox, which describes methods and tools for detecting source code comprehension strategies and eye movement patterns. As an artefact of this toolbox, *CodeSight* was developed to combine the possibilities of AOI-DNAs (see Section 4.6 for information about CodeSight). With CodeSight, the eye movement data of the three studies can a) be visualized with different color schemes and b) be searched with the pattern matching. This tool allows for easy analysis of the data, and it was valuable over the course of this thesis. At this point, this application is not ready for public use; however, it can be a base for further development.

Contribution C_{18}

CodeSight proved to be a valuable application for visualizing and analyzing eye movement patterns with the potential for a comprehensive analysis platform.

Table 8.1 fulfills the role of a toolbox by describing the contributions of this thesis. In fact, many of the case studies, methods, processes, representations, visualizations, and tools of this thesis can be seen as a toolbox. The combination of content- and artefact-related results forms the set of tools for source code comprehension strategy and eye movement pattern research. Furthermore, Figure 8.1 presents the 18 contributions distributed over eight main categories. The case studies mainly influenced the development and usage of contributions like the AOI-DNAs, AOI-STGs, and CodeSight.

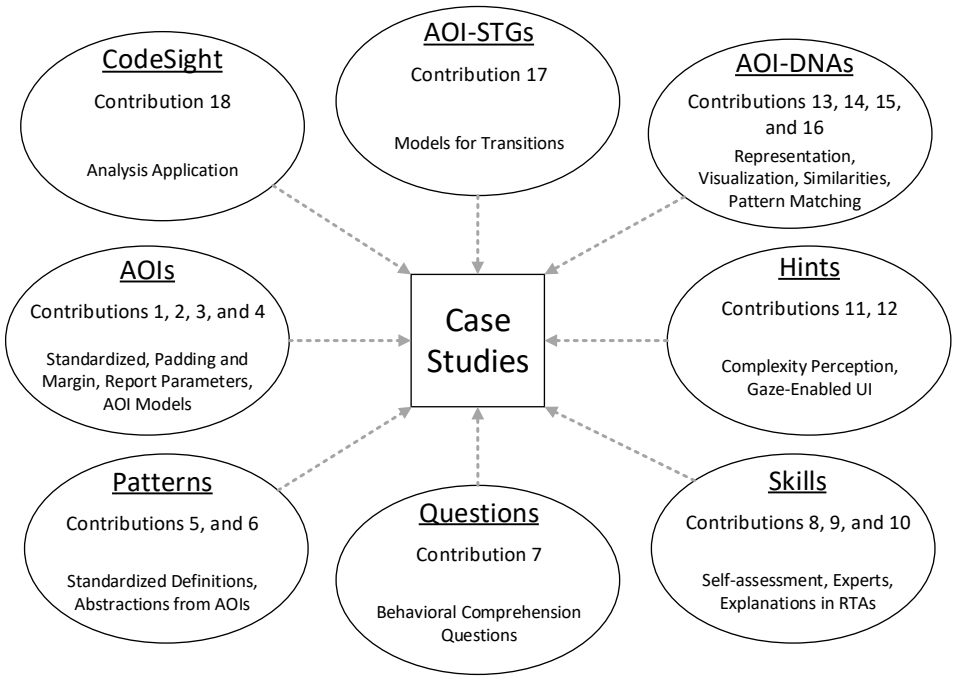


Figure 8.1: Overview of the 18 contributions of this thesis, distributed among eight main categories.

Table 8.1: Overview of the summarized contributions of this thesis.

#	Summarized Contributions	Type
1.	The thesis offers a standardized definition of AOIs, including margins, paddings, models, and the reporting of the configuration for the source code comprehension community.	Process
2.	The thesis offers a standardized definition of eye movement patterns with an abstraction from AOIs and thus the underlying source code.	Process
3.	The thesis recommends making behavioral comprehension questions visible before and after or simultaneously with the source code stimulus.	Process
4.	The thesis contributes self-assessment and pre-tests for well-selected participant groups and valuable data in the RTAs and interviews.	Process
5.	Learning hints can change the complexity perception, especially for syntax highlighting, which leads to the recommendation to activate syntax highlighting in source code comprehension studies.	Process
6.	AOI-DNAs are useful for representing viewing sequences, which are the bases for sequence alignment and pattern matching.	Representation
7.	AOI-DNAs are valuable for the visualization of eye movement sequences via color codes and color gradients to highlight reading patterns.	Visualization
8.	AOI-STGs are valuable models for analyzing strong transition characteristics.	Model
9.	CodeSight is a tool for integrating many of the above-mentioned representations and visualizations.	Tool

8.4 RETROSPECTIVE OF SOURCE CODE COMPLEXITY, SIMILARITY MEASURES AND CLUSTERING

This thesis demonstrates that, especially in the learning hints study (see Chapter 7), the perspective on source code complexity can change when learning hints are available, which was the case for syntax highlighting in particular.

Different Perception of Algorithmic Results

Overall, the results for different source code stimuli are interesting and insightful. The learning hints study used three different source code examples, namely Bubble, the GCD, and Vehicle. The first two utilized well-known algorithmic concepts such as the bubble sort algorithm and the calculation of the greatest common divisor (GCD), and both examples and their complexity are thus comparable. The Vehicle source code used object-oriented concepts to model a vehicle with a maximum speed and methods for accelerating and decelerating.

Different Programming Paradigms

These different technical and semantic concepts will influence answer quality and thus create challenges in analyzing and comparing the data. This became clear when conducting and especially when analyzing the learning hints study: the Vehicle source code received consistently worse answers. The initial expectations were that the Bubble source code example is the most complex one because of the program structure with two nested loops, followed by the GCD and the Vehicle code stimuli. However, the participant ratings revealed that the Vehicle source code was considered the most difficult one. The primary reason seems to be the different semantics due to OOP. A different viewpoint is the algorithmic complexity and not the aforementioned complexity based on the program structure (nested loops). Both can be and are in fact often highly coupled, but differences are possible. The GCD, for example, is not as structurally complex as the Bubble source code, but it can be algorithmically more challenging. Different programming paradigms, such as functional or logical programming, can further change these characteristics.

These various complexity viewpoints, induced by different programming paradigms, as well as structural, arithmetical, and semantical concepts have a great impact on the outcome of a source code comprehension study and must be taken into account when planning and conducting a study, as well as when analyzing the data.

Similarity Measures

Calculating similarity measures is a frequently used method in the source code comprehension community. The majority of the known studies focus on algorithms such as the N-W, Smith-Waterman, or OM algorithms.

The similarities can be used as a grouping mechanism to project the measures to different levels of the data, such as participant groups, skill level, answer quality, or eye tracking characteristics (e.g., duration and fixation counts). This projection is an important part of analyzing viewing patterns through clustering.

Tight Coupling to AOIs

The major obstacle and future challenge for this type of analysis is decoupling these similarity measures from the dependence of AOIs, and thus sequences such as the AOI-DNAs, because these are again dependent on the structure of source code. While this is inevitably

the current state of source code comprehension research, it must be solved for generalizable approaches in the future.

8.5 RETROSPECTIVE OF THE METHODICAL EYE TRACKING CHALLENGES

In Section 4.1 different dimensions for eye tracking challenges were discussed, ranging from technology-, method-, and participant-related problems.

Sampling Rate Considerations

Various measures were taken in the course of this thesis to minimize these challenges and the resulting problems. Throughout this thesis, the *Tobii Pro TX 300* with a sampling rate of 300Hz was used in self-conducted studies. Furthermore, for the EMIP dataset, the *SMI RED250* mobile eye tracker with a 250Hz rate was used. In both cases, the sampling rate is sufficient to record the fixation and saccade eye movement characteristics, both of which are critical for source code comprehension studies. A lower sampling rate can cause problems when recording fixations and saccades, but a higher one does not improve these measures significantly (refer to the sampling rate explained in Section 4.1).

Fixation Filter

For the fixation filter, which is responsible for calculating fixations from the raw eye tracking data, the *I-VT* proposed by Tobii was employed in every study with the standard parameters that are common in other published research: a maximum radius of *60 px*, a minimum fixation duration of *60 ms*, and a maximum of *55* missing gaze samples in order to count as a fixation.

Sampling Bias Considerations

The often-reported imbalance between male and female participants is unfortunately common for studies in which participants are recruited in computer science or engineering faculties. The factor ranged from approximately 2.5:1 to 5:1 in the reported studies, which seems to be common for the mentioned faculties. Extending the call for participants to include not only technically oriented degree programs can help to find more female participants if the requirements to participate in the study are mentioned in the call. However, it must be noted that searching explicitly for gender is inappropriate from a gender equality perspective. From experience, the sampling bias based on aggravating recording conditions, such as makeup and glasses, did not require special monitoring. All participants should have the opportunity to participate. By the time of the calibration, it becomes clear whether the participant can be recorded or not. In the vast majority of cases, a poor calibration result does not depend on wearing makeup or glasses, but rather than on other circumstances such as eye-related medical conditions or the lighting (e.g., the time of the day).

Programming Environment Biases

A potential bias towards programming environments can be avoided by using simple images with code or, if the additional effort is necessary, with specialized study prototypes. Both options were used throughout this thesis: The EMIP dataset was recorded while showing images; for the code smell study, the source code examples were embedded in a PowerPoint presentation; and for the learning hints study, a functional prototype was built to include different learning hints and an experimental gaze-enabled interface.

Comprehension Question Time

The data collected regarding the comprehension question time were insightful. Asking the question before versus after showing the source code stimulus influenced the answer quality. Moreover, asking behavioral questions instead of questions targeting the specification of the source code is more appropriate to measure the real comprehension of the source code. One reason is that a behavioral question needs actual data, which is used to execute the code in the mental model built while reading and comprehending the source code. Asking questions regarding the structure of the source code or posing the questions after the code example is, in both cases, beneficial for participants who can memorize and recall information better; however, this is an odd way of defining comprehension.

Margin and Padding Models

The analysis of AOI margins and paddings revealed that a maximum margin and a padding model, respecting the viewing angles, form a well-thought approach in defining AOIs. This approach is more realistic than defining the smallest possible AOIs around source code elements.

In summary of the measures in this thesis to address the methodical eye tracking challenges, balancing the participant group in terms of gender and skill level is one of the most difficult challenges. Finding participants can take time, which must be considered when planning a study. Choosing the correct eye tracking hardware and setting up working parameters is less complicated because fixation and saccades are the important eye movement data points for source code comprehension studies. Both can be collected satisfactorily with an eye tracker upwards of the 250Hz scale.

8.6 FUTURE WORK

Improvements for…

This section covers open research directions and issues for future projects. Not every topic could be addressed by this thesis (e.g., enhancements for the newly developed methods and tools, ideas for detecting eye movement patterns, and additional research directions).

…AOI-DNA Improvements…

The proposed **AOI-DNAs** should be enhanced with the fixation time per AOI-Hit. This would allow for improved visualizations (e.g., with the fixation time as an indicator for larger or smaller AOI-Hit rectangles). Furthermore, this information could be used in the pattern search as well. A more dynamic version of AOI-DNAs is described in the context of enhancements for CodeSight.

…AOI-STG Improvements…

The developed **AOI-STGs** can also benefit from the fixation time per AOI-Hit to visualize the dwell time per AOI. Apart from this, different layout algorithms should be tested to emphasize various characteristics of the data, such as clusters and regressions. A more dynamic version of AOI-STGs is described in the context of enhancements for CodeSight. In general, fully dynamic STGs would be beneficial for analyzing data to highlight transitions and to include a search feature. In addition, splitting AOI-STGs into time periods (*time-sliced STGs*) would allow for an analysis of the transitions in pre-defined time slots with more detail, which could then highlight the changes in the viewing strategies.

...Pattern Matching...

Pattern matching is the third method that can benefit from the fixation time per AOI-Hit. The version used in this thesis did not take the fixation time into account when matching AOIs. This improvement would allow for the creation of pattern matches searching for eye movement patterns with a sequence of AOIs explicitly below or above a fixation time threshold. In addition, the fixation time could be used to search for patterns in which a certain amount of fixation time can be matched between two important AOIs. Overall, using fixation time in pattern matching offers many new opportunities to search for more complex eye movement patterns on the one hand and more specialized patterns on the other.

...and CodeSight

CodeSight, as an application mainly for AOI-DNAs, can be improved in a comprehensive way. AOI-DNAs can be completely dynamic, with the ability to select parts thereof for further information, such as: 1) The AOI-Hits count, 2) the fixation time for the selected parts, 3) an automatic analysis if the selection holds one or more patterns, 4) the feature for zooming in and out of an AOI-DNA to change the abstraction levels between AOI models, 5) automatic calculation of other AOI models, 6) using different fixation filters to see the changes in the sequences, 7) using sequence alignment within the application, to emphasize similarities and dissimilarities, 8) using the demographic data to show data from participants, 9) adding other methods such as AOI-STGs, and many features more. The application can be seen as the starting point for a complete analysis platform for eye movement data and eye movement patterns.

AOI Model Analyses

The analysis targeting the **AOI models** and **AOI gaps** revealed significant differences for two extreme gap models (full and no gaps). These results could be analyzed further and automatically for multiple datasets to detect thresholds when the differences in detected fixations are rising significantly, which is a sign of more noise being covered by the gap model. This analysis can be achieved with an automatic tool and algorithm.

Learning Hints

Furthermore, the learning hints study suggests that the **hints** are used and rated differently. This raises the question of whether common hints in development environments are used as initially expected and intended, as well as whether they are used at all. Studying the use of such hints would be valuable for analyzing how developers work and which help is used to comprehend source code faster and more reliably.

Sequence Mining and Machine Learning

Regarding the analysis of eye movement patterns, **sequence mining** and methods from the **machine learning community** are worth exploring. These ideas include storing sequence data (AOI-DNAs) in a sequence database and searching for patterns with sequence mining algorithms. Furthermore, 1D-CNNs (Castner et al., 2020) and hidden Markov models for pattern classification are promising and have been successfully used in other research areas.

Domain Specific Languages

The handling of eye movement data can be improved as well; this includes regular expressions for pattern searches. It is worth a try, if both areas could be more effective with specialized **query languages**, implemented as *domain specific languages* (DSLs). Regular ex-

pressions can be replaced by a language, which makes it easy to express sequences of AOIs with a specific length, gaps, as well as minimum and maximum fixation times. Expressing eye movement patterns is thus usable for scientists without a strong background in regular expressions, which will become increasingly complex when the mentioned features are included. Furthermore, a language for querying eye tracking and eye movement data could be a valuable approach. If fixations and AOIs are considered as lists or tables, to use the vocabulary of relational databases, then a language comparable to SQL could be used to simplify eye movement data analysis. Many eye movement metrics are expressible with a simple SQL statement (e.g., looking for the sum over fixation times or the first hit of an AOI). The query language, implemented as a DSL, can be specialized to make querying for eye movement data and pattern-related information easy, convenient, and fast.

Code Reading and Explaining in Education

Computer science education could benefit from specifically taught code reading and code explaining skills. Both tasks are often performed by professional software developers and used in source code comprehension studies in interviews or RTAs. The studies in this thesis revealed that especially novices have problems reading source code in general and explaining what they have read due to their lack of vocabulary. This must be solved in the educational process by viewing code reading and code explaining as first-level skills and not as skills learned on the side while writing source code.

Community-Wide Analyses Tool

Overall, a specialized tool for analyzing eye tracking data, and eye movement data in particular, could help with the challenging task. This tool could target numerous of analyses and visualizations, such as CodeSight, but with a much broader spectrum. This new focus should also include interviews and RTAs because a specialized tool for conducting RTAs with the option to highlight regions, AOIs, or parts of a participant's scan path is valuable in analyzing such data.

Myriad of Ideas

There are a myriad of ideas and areas to explore for future projects. These include Wizard of Oz studies to test learning hints in the context of source code comprehension, more difficult code examples, machine learning for eye movement data, and a more extensive analysis tool such as CodeSight, to name a few examples. Especially the areas of analyzing eye movement data and testing learning hints are important for moving the vision of a dynamic learner support system forward.

Bibliography

A., V., M.S., H., Hayashi, M., Renema, F., Elkins, S., McCandless, J. W., & McCann, R. S. (2005). Characterizing Scan Patterns in a Spacecraft Cockpit Simulator: Expert Vs. Novice Performance. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, *49*(1), 83-87. Retrieved from https://doi.org/10.1177/154193120504900119 doi: 10.1177/154193120504900119

Abid, N. J., Maletic, J. I., & Sharif, B. (2019). Using Developer Eye Movements to Externalize the Mental Model Used in Code Summarization Tasks. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications (ETRA'19)* (pp. 1–9). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/3314111.3319834 doi: 10.1145/3314111.3319834

Adelson, B., & Soloway, E. (1985). The Role of Domain Experience in Software Design. *IEEE Transactions on Software Engineering*, *SE-11*(11), 1351–1360. Retrieved from https://doi.org/10.1109/TSE.1985.231883 doi: 10.1109/TSE.1985.231883

Alaoutinen, S., & Smolander, K. (2010). Student Self-Assessment in a Programming Course Using Bloom's Revised Taxonomy. In *Proceedings of the 2010 ACM SIGCSE Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'10)* (pp. 155–159). Retrieved from https://doi.org/10.1145/1822090.1822135 doi: 10.1145/1822090.1822135

Anderson. (1985). *Cognitive Psychology and Its Implications, 2nd ed.* New York, NY, US: W H Freeman/Times Books/ Henry Holt & Co.

Anderson, Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences*, *4*(2), 167–207. Retrieved from https://doi.org/10.1207/s15327809jls0402_2 doi: 10.1207/s15327809jls0402_2

Anderson, Krathwohl, D. R., & Bloom, B. S. (2001). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives.* Retrieved from http://books.google.com/books?id=JPkXAQAAMAAJ&pgis=1

Andrienko, G., Andrienko, N., Burch, M., & Weiskopf, D. (2012). Visual Analytics Methodology for Eye Movement Studies. *IEEE Transactions on Visualization and Computer Graphics*, *18*(12), 2889–2898. Retrieved from https://doi.org/10.1109/TVCG.2012.276 doi: 10.1109/TVCG.2012.276

Aschwanden, C., & Crosby, M. (2006). Code Scanning Patterns in Program Comprehension. In *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS'06).* Retrieved from http://pdf.aminer.org/000/641/141/is_there_any_difference_in_novice_comprehension_of_a_small.pdf

Asenov, D., Hilliges, O., & Müller, P. (2016). The Effect of Richer Visualizations on Code Comprehension. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI'16)* (pp. 5040–5045). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2858036.2858372 doi: 10.1145/2858036.2858372

Astrachan, O., Berry, G., Cox, L., & Mitchener, G. (1998). Design Patterns: An Essential Component of CS Curricula. In *Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education (ITiCSE'98)* (pp. 153–160). Retrieved from https://doi.org/10.1145/273133.273182 doi: 10.1145/273133.273182

Awasthi, P., & Hsaio, I. H. (2015). INSIGHT: A Semantic Visual Analytics for Programming Discussion Forums. In *Proceedings of the First International Workshop on Visual Aspects of Learning Analytics (VISLA 2015)* (Vol. 1518, pp. 24–31).

Bauer, J., Siegmund, J., Peitek, N., Hofmeister, J. C., & Apel, S. (2019). Indentation: Simply a Matter of Style or Support for Program Comprehension? In *Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC'19)* (pp. 154–164). Piscataway, NJ, USA: IEEE Press. Retrieved from https://doi.org/10.1109/ICPC.2019.00033 doi: 10.1109/ICPC.2019.00033

Bayman, P., & Mayer, R. E. (1988). Using Conceptual Models to Teach BASIC Computer Programming. *Journal of Educational Psychology*, *80*(3), 291–298. Retrieved from https://doi.org/10.1037/0022-0663.80.3.291 doi: 10.1037/0022-0663.80.3.291

Beaubouef, T., & Mason, J. (2005). Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. *ACM SIGCSE Bulletin*, *37*(2), 103–106. Retrieved from https://dl.acm.org/doi/10.1145/1083431.1083474 doi: 10.1145/1083431.1083474

Becker, B. A., Glanville, G., Iwashima, R., McDonnell, C., Goslin, K., & Mooney, C. (2016). Effective Compiler Error Message Enhancement for Novice Programming Students. *Computer Science Education*, *26*(2-3), 148–175. Retrieved from https://doi.org/10.1080/08993408.2016.1225464 doi: 10.1080/08993408.2016.1225464

Becker, B. A., Murray, C., Tao, T., Song, C., McCartney, R., & Sanders, K. (2018). Fix the First, Ignore the Rest: Dealing With Multiple Compiler Error Messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE'18)* (pp. 634–639). Retrieved from https://dl.acm.org/doi/10.1145/3159450.3159453 doi: 10.1145/3159450.3159453

Bednarik, R. (2012). Expertise-Dependent Visual Attention Strategies Develop Over Time During Debugging With Multiple Code Representations. *International Journal of Human-Computer Studies*, *70*(2), 143–155. Retrieved from http://dx.doi.org/10.1016/j.ijhcs.2011.09.003 doi: 10.1016/j.ijhcs.2011.09.003

Bednarik, R., Busjahn, T., & Schulte, C. (2013). *Eye Movements in Programming Education Analyzing the Expert's Gaze* (Tech. Rep.). Finland: University of Eastern Finland, Joensuu. Retrieved from https://epublications.uef.fi/pub/urn_isbn_978-952-61-1539-9/urn_isbn_978-952-61-1539-9.pdf

Bednarik, R., Schulte, C., Budde, L., Heinemann, B., & Vrzakova, H. (2018). Eye-Movement Modeling Examples in Source Code Comprehension: A Classroom Study. In *Proceedings of*

the 18th Koli Calling International Conference on Computing Education Research (Koli Calling '18) (pp. 2:1–2:8). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/3279720.3279722 doi: 10.1145/3279720.3279722

Bednarik, R., & Shipilov, A. (2012). Usability of Gaze-Transfer in Collaborative Programming: How and When Could it Work, and Some Implications for Research Agenda. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work Companion (CSCW'12)* (pp. 1–7). Retrieved from https://api.semanticscholar.org/CorpusID:61333062

Belenky, D., Ringenberg, M., & Olsen, J. (2014). Using Dual Eye-Tracking to Evaluate Students' Collaboration with an Intelligent Tutoring System for Elementary-Level Fractions. *Proceedings of 36th Annual Meeting of the Cognitive Science Society (CogSci 2014)*, *36*, 176–181. Retrieved from https://escholarship.org/uc/item/8m3543hd

Belmonte, J., Dugerdil, P., & Agrawal, A. (2014). A Three-Layer Model of Source Code Comprehension. In *Proceedings of the 7th India Software Engineering Conference (ISEC'14)*. New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2590748.2590758 doi: 10.1145/2590748.2590758

Bennedsen, J., & Caspersen, M. E. (2007). Failure Rates in Introductory Programming. *ACM SIGCSE Bulletin*, *39*(2), 32. Retrieved from https://doi.org/10.1145/1272848.1272879 doi: 10.1145/1272848.1272879

Bergin, S., & Reilly, R. (2005). The Influence of Motivation and Comfort-Level on Learning to Program. In *Proceedings of the 17th Workshop of the Psychology of Programming Interest Group (PPIG'17)* (pp. 293–304). Retrieved from https://www.researchgate.net/publication/228969033

Biggs, John, Tang, & Catherine. (2011). Teaching For Quality Learning At University. *Berkshire, UK: McGraw-Hill Education*, *2011*. Retrieved from http://books.google.se/books/about/Teaching_for_Quality_Learning_at_Univers.html?id=XhjRBrDAESkC&pgis=1

Blascheck, T., Kurzhals, K., Raschke, M., Burch, M., Weiskopf, D., & Ertl, T. (2014). State-of-the-Art of Visualization for Eye Tracking Data. In R. Borgo, R. Maciejewski, & I. Viola (Eds.), *Proceedings of the Eurographics Conference on Visualization (EuroVis'14)* (pp. 1–20). The Eurographics Association. Retrieved from https://dx.doi.org/10.2312/eurovisstar.20141173 doi: 10.2312/eurovisstar.20141173

Blascheck, T., Kurzhals, K., Raschke, M., Burch, M., Weiskopf, D., & Ertl, T. (2017). Visualization of Eye Tracking Data: A Taxonomy and Survey. *Computer Graphics Forum*, *36*(8), 260–284. Retrieved from https://doi.org/10.1111/cgf.13079 doi: 10.1111/cgf.13079

Blignaut, P. J., Beelders, T. R., & So, C. Y. (2008). The Visual Span of Chess Players. In *Proceedings of the 2008 Symposium on Eye Tracking Research & Applications (ETRA'08)* (pp. 165–171). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1344471.1344514 doi: 10.1145/1344471.1344514

Bloom, B. S., & others. (1956). *Taxonomy of Educational Objectives. Vol. 1: Cognitive Domain.* New York: Longman.

Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., & Selby, R. (1995). Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *Annals of Software Engineering*, *1*(1), 57–94. Retrieved from https://doi.org/10.1007/BF02249046 doi: 10.1007/BF02249046

Boysen, J. P., & Keller, R. F. (1980). Measuring Computer Program Comprehension. *ACM SIGCSE Bulletin*, *12*(1), 92–102. Retrieved from https://dl.acm.org/doi/10.1145/953032.804619 doi: 10.1145/953032.804619

Bransford, J., & Stein, B. (1993). *The Ideal Problem Solver*. Retrieved from https://digitalcommons.georgiasouthern.edu/ct2-library/46/

Brooks, R. (1983, 6). Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, *18*(6), 543–554. Retrieved from https://doi.org/10.1016/S0020-7373(83)80031-5 doi: 10.1016/S0020-7373(83)80031-5

Brown, N., & Altadmri, A. (2014). Investigating Novice Programming Mistakes: Educator Beliefs vs. Student Data. In *Proceedings of the 10th Annual International Conference on International Computing Education Research (ICER'14)* (pp. 43–50). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2632320.2632343 doi: 10.1145/2632320.2632343

Brown, N., & Altadmri, A. (2017, 5). Novice Java Programming Mistakes. *ACM Transactions on Computing Education*, *17*(2), 1–21. Retrieved from http://dl.acm.org/citation.cfm?doid=3090098.2994154 doi: 10.1145/2994154

Brown, N., & Wilson, G. (2018). Ten Quick Tips for Teaching Programming. *PLoS Computational Biology*, *14*(4), 1–8. Retrieved from https://doi.org/10.1371/journal.pcbi.1006023 doi: 10.1371/journal.pcbi.1006023

Brusilovsky, P. (1992). Intelligent Tutor, Environment and Manual for Introductory Programming. *Educational and Training Technology International*, *29*(1), 26–34. Retrieved from https://doi.org/10.1080/0954730920290104 doi: 10.1080/0954730920290104

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., ... Tamm, S. (2015). Eye Movements in Code Reading: Relaxing the Linear Order. In *Proceedings of the IEEE 23rd International Conference on Program Comprehension (ICPC'15)* (pp. 255–265). Retrieved from https://doi.org/10.1109/ICPC.2015.36 doi: 10.1109/ICPC.2015.36

Busjahn, T., Bednarik, R., & Schulte, C. (2014). What Influences Dwell Time During Source Code Reading? Analysis of Element Type and Frequency as Factors. In *Proceedings of the Symposium on Eye Tracking Research and Applications (ETRA'14)* (pp. 335–338). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2578153.2578211 doi: 10.1145/2578153.2578211

Busjahn, T., & Schulte, C. (2013). The Use of Code Reading in Teaching Programming. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli Calling '13)* (pp. 3–11). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2526968.2526969 doi: 10.1145/2526968.2526969

Busjahn, T., Schulte, C., & Busjahn, A. (2011). Analysis of Code Reading to Gain More Insight in Program Comprehension. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli Calling'11)* (pp. 1–9). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2094131.2094133 doi: 10.1145/2094131.2094133

Busjahn, T., Schulte, C., & Kropp, E. (2014). Developing Coding Schemes for Program Comprehension Using Eye Movements. In *Proceedings 25th Annual Workshop of the Psychology of Programming Interest Group (PPIG'14)* (pp. 111–122). Retrieved from https://ris.uni-paderborn.de/record/15662

Busjahn, T., Schulte, C., Sharif, B., Simon, Begel, A., Hansen, M., … Antropova, M. (2014). Eye Tracking in Computing Education. In *Proceedings of the 10th Annual International Conference on International Computing Education Research (ICER'14)* (pp. 3–10). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/2632320.2632344 doi: 10.1145/2632320.2632344

Busjahn, T., Schulte, C., & Tamm, S. (2015). *Eye Movements in Programming Education II : Analyzing the Novice' s Gaze* (Tech. Rep. No. March). Berlin: Technical Report TR-B-15-01, Freie Universität Berlin, Department of Mathematics and Computer Science, Berlin, Germany. Retrieved from https://api.semanticscholar.org/CorpusID:1998500

Campbell, W., & Bolker, E. (2002, 11). Teaching Programming by Immersion Reading and Writing. In *Proceedings of the 32nd Annual Frontiers in Education* (Vol. 1, pp. 23–28). Boston, MA, USA: IEEE. Retrieved from https://doi.org/10.1109/fie.2002.1158015 doi: 10.1109/fie.2002.1158015

Carter, J., & Jenkins, T. (1999). Gender and Programming: What's Going on? In *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'99)* (pp. 1–4). Retrieved from https://doi.org/10.1145/305786.305824

Castner, N., Kasneci, E., Kübler, T., Scheiter, K., Richter, J., Eder, T., … Keutel, C. (2018). Scanpath Comparison in Medical Image Reading Skills of Dental Students: Distinguishing Stages of Expertise Development. In *Proceedings of the 2018 ACM Symposium on Eye Tracking Research and Applications (ETRA'18)* (pp. 39:1–39:9). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/3204493.3204550 doi: 10.1145/3204493.3204550

Castner, N., Küebler, T. C., Scheiter, K., Richter, J., Eder, T., Hüettig, F., … Kasneci, E. (2020). Deep Semantic Gaze Embedding and Scanpath Comparison for Expertise Classification During OPT Viewing. In *Proceedings of the 2020 ACM Symposium on Eye Tracking Research and Applications (ETRA'20)* (pp. 1–10). ACM. Retrieved from https://doi.org/10.1145/3379155.3391320 doi: 10.1145/3379155.3391320

Chen, Z., & Sun, W. (2018). Scanpath Prediction for Visual Attention Using IOR-ROI LSTM. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI'18)* (pp. 642–648). Retrieved from https://doi.org/10.24963/ijcai.2018/89 doi: 10.24963/ijcai.2018/89

Chuk, T., Chan, A. B., & Hsiao, J. H. (2014). Understanding Eye Movements in Face Recognition Using Hidden Markov Models. *Journal of Vision*, *14*(11), 8. Retrieved from https://doi.org/10.1167/14.11.8 doi: 10.1167/14.11.8

Clear, T., Whalley, J., Robbins, P., Philpott, A., Eckerdal, A., Laakso, M.-J., & Lister, R. (2011). Report on the Final BRACElet Workshop: Auckland University of Technology, September 2010. *Journal of Applied Computing and Information Technology*, *15*(1). Retrieved from https://api.semanticscholar.org/CorpusID:60708568

Clifton, C., Ferreira, F., Henderson, J. M., Inhoff, A. W., Liversedge, S. P., Reichle, E. D., & Schotter, E. R. (2016). Eye Movements in Reading and Information Processing: Keith Rayner's 40 Year Legacy. *Journal of Memory and Language*, *86*, 1–19. Retrieved from https://doi.org/10.1016/j.jml.2015.07.004 doi: 10.1016/j.jml.2015.07.004

Corbett, A. (2001). Cognitive Computer Tutors: Solving the Two-Sigma Problem. In *User Modeling 2001* (Vol. 2109, pp. 137–147). Berlin, Heidelberg: Springer-Verlag. Retrieved from https://doi.org/10.1007/3-540-44566-8_14 doi: 10.1007/3-540-44566-8_14

Cornia, M., Baraldi, L., Serra, G., & Cucchiara, R. (2018). Predicting Human Eye Fixations Via an LSTM-Based Saliency Attentive Model. *IEEE Transactions on Image Processing*, *27*(10), 5142–5154. Retrieved from https://doi.org/10.1109/TIP.2018.2851672 doi: 10.1109/TIP.2018.2851672

Costa-Gomes, M., Crawford, V. P., & Broseta, B. (2001). Cognition and Behavior in Normal-Form Games: An Experimental Study. *Econometrica*, *69*(5), 1193–1235. Retrieved from http://dx.doi.org/10.2139/ssrn.145793 doi: 10.2139/ssrn.145793

Cristino, F., Mathôt, S., Theeuwes, J., & Gilchrist, I. D. (2010, 8). ScanMatch: A Novel Method for Comparing Fixation Sequences. *Behavior Research Methods*, *42*(3), 692–700. Retrieved from https://doi.org/10.3758/BRM.42.3.692 doi: 10.3758/BRM.42.3.692

Crosby, M. E., Scholtz, J., & Wiedenbeck, S. (2002). The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group (PPIG'02)* (pp. 58–73). Retrieved from https://api.semanticscholar.org/CorpusID:16012754

Crosby, M. E., & Stelovsky, J. (1990). How Do We Read Algorithms?: A Case Study. *Computer*, *23*(1), 25–35. Retrieved from https://doi.org/10.1109/2.48797 doi: 10.1109/2.48797

Crow, T., Luxton-Reilly, A., & Wuensche, B. (2018). Intelligent Tutoring Systems for Programming Education: A Systematic Review. In *Proceedings of the 20th Australasian Computing Education Conference (ACE'18)* (pp. 53–62). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3160489.3160492 doi: 10.1145/3160489.3160492

Dalbey, J., & Linn, M. C. (1985). The Demands and Requirements of Computer Programming: A Literature Review. *Journal of Educational Computing Research*, *1*(3), 253–274. Retrieved from https://doi.org/10.2190/bc76-8479-ym0x-7fua doi: 10.2190/bc76-8479-ymox-7fua

D'Angelo, S., & Begel, A. (2017). Improving Communication Between Pair Programmers Using Shared Gaze Awareness. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI'17)* (pp. 6245–6255). Retrieved from http://doi.org/10.1145/3025453.3025573 doi: 10.1145/3025453.3025573

D'Angelo, S., & Gergle, D. (2018). An Eye for Design: Gaze Visualizations for Remote Collaborative Work. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI'18)* (Vol. 2018-April). {ACM} Press. Retrieved from https://doi.org/10.1145%2F3173574.3173923 doi: 10.1145/3173574.3173923

Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012). Mediated Transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)* (pp. 141–146). Retrieved from https://doi.org/10.1145/2157136.2157180 doi: 10.1145/2157136.2157180

Deitelhoff, F., & Harrer, A. (2018). Towards a Dynamic Help System: Support of Learners During Programming Tasks Based Upon Historical Eye-Tracking Data. In *Proceedings of the IEEE 18th International Conference on Advanced Learning Technologies (ICALT'18)* (pp. 77–78). New York, NY, USA: ACM. doi: 10.1109/ICALT.2018.00116

Deitelhoff, F., Harrer, A., & Kienle, A. (2019a). The Influence of Different AOI Models in Source Code Comprehension Analysis. In *Proceedings of the IEEE/ACM 6th International Workshop on Eye Movements in Programming (EMIP'19)* (pp. 10–17). Piscataway, NJ, USA: IEEE Press. Retrieved from https://doi.org/10.1109/EMIP.2019.00010 doi: 10.1109/EMIP.2019.00010

Deitelhoff, F., Harrer, A., & Kienle, A. (2019b). An Intuitive Visualization for Rapid Data Analysis: Using the DNA Metaphor for Eye Movement Patterns. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research and Applications (ETRA'19)* (pp. 78:1–78:5). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/3317958.3318227 doi: 10.1145/3317958.3318227

Denny, P., Becker, B. A., Craig, M., Wilson, G., & Banaszkiewicz, P. (2019). Research This! Questions That Computing Educators Most Want Computing Education Researchers to Answer. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER'19)* (pp. 259–267). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/3291279.3339402 doi: 10.1145/3291279.3339402

Denny, P., Luxton-Reilly, A., Tempero, E., & Hendrickx, J. (2011). Understanding the Syntax Barrier for Novices. In *Proceedings of the 16th Annual Conference on Innovation and Technology in Computer Science (ITiCSE'11)* (pp. 208–212). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/1999747.1999807 doi: 10.1145/1999747.1999807

Dessus, P., Cosnefroy, O., & Luengo, V. (2016). "Keep Your Eyes on 'em all!": A Mobile Eye-Tracking Analysis of Teachers' Sensitivity to Students. In *Lecture Notes in Computer Science* (Vol. 9891, pp. 72–84). Retrieved from https://doi.org/10.1007/978-3-319-45153-4_6 doi: 10.1007/978-3-319-45153-4_6

Détienne, F. (1990). Expert Programming Knowledge: A Schema-Based Approach. In J.-M. Hoc and T.R.G. Green and R. Samurçay and D.J. Gilmore (Ed.), *Psychology of programming*

(pp. 205–222). London: Academic Press. Retrieved from https://doi.org/10.1016/B978-0-12-350772-3.50018-5 doi: 10.1016/B978-0-12-350772-3.50018-5

Détienne, F., & Soloway, E. (1990). An Empirically-Derived Control Structure for the Process of Program Understanding. *International Journal of Man-Machine Studies*, *33*(3), 323–342. Retrieved from https://doi.org/10.1016/S0020-7373(05)80122-1 doi: 10.1016/S0020-7373(05)80122-1

D'Mello, S. K., Olney, A., Williams, C., & Hays, P. (2012). Gaze Tutor: A Gaze-Reactive Intelligent Tutoring System. *International Journal of Human-Computer Studies*, *70*(5), 377–398. Retrieved from https://doi.org/10.1016/j.ijhcs.2012.01.004 doi: 10.1016/j.ijhcs.2012.01.004

Doberstein, D., Hecking, T., & Hoppe, H. U. (2017). Sequence Patterns in Small Group Work Within a Large Online Course. In *Lecture Notes in Computer Science* (Vol. 10391, pp. 104–117). Retrieved from https://doi.org/10.1007/978-3-319-63874-4_9 doi: 10.1007/978-3-319-63874-4_9

Dreyfus, H. L., Drey-fus, S. E., & Zadeh, L. A. (2008). *Mind over Machine: The Power of Human Intuition and Expertise in the Era of the Computer* (Vol. 2) (No. 2). USA: The Free Press. Retrieved from https://doi.org/10.1109/mex.1987.4307079 doi: 10.1109/mex.1987.4307079

Du Boulay, B. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, *2*(1), 57–73. Retrieved from https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9 doi: 10.2190/3lfx-9rrf-67t8-uvk9

Du Boulay, B., O'Shea, T., & Monk, J. (1999). Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. *International Journal of Human-Computer Studies*, *51*(2), 265–277. Retrieved from https://doi.org/10.1006/ijhc.1981.0309 doi: 10.1006/ijhc.1981.0309

Duchowski, A. T. (2017). *Eye Tracking Methodology: Theory and Practice: Third Edition*. Retrieved from https://doi.org/10.1007/978-3-319-57883-5 doi: 10.1007/978-3-319-57883-5

Duchowski, A. T. (2018). Gaze-Based Interaction: A 30 Year Retrospective. *Computers & Graphics*, *73*, 59–69. Retrieved from https://doi.org/10.1016/j.cag.2018.04.002 doi: 10.1016/j.cag.2018.04.002

Duchowski, A. T., Driver, J., Jolaoso, S., Tan, W., Ramey, B. N., & Robbins, A. (2010). Scanpath Comparison Revisited. In *Proceedings of the 2010 Symposium on Eye-Tracking Research and Applications (ETRA'10)* (pp. 219–226). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/1743666.1743719 doi: 10.1145/1743666.1743719

Ebrahimi, A. (1994, 10). Novice Programmer Errors: Language Constructs and Plan Composition. *International Journal of Human-Computer Studies*, *41*(4), 457–480. Retrieved from https://doi.org/10.1006/ijhc.1994.1069 doi: 10.1006/ijhc.1994.1069

Efopoulos, V., Dagdilelis, V., Evangelidis, G., & Satratzemi, M. (2005). WIPE: A Programming Environment for Novices. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'05)* (pp. 113–117). Retrieved from https://doi.org/10.1145/1151954.1067479 doi: 10.1145/1067445.1067479

Eraslan, S., Yesilada, Y., & Harper, S. (2016). Eye Tracking Scanpath Analysis on Web Pages: How Many Users? In *Proceedings of the Ninth Biennial ACM Symposium on Eye Tracking Research and Applications (ETRA'16)* (Vol. 14, pp. 103–110). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/2857491.2857519 doi: 10.1145/2857491.2857519

Ettles, A., Luxton-Reilly, A., & Denny, P. (2018, 1). Common Logic Errors Made By Novice Programmers. In *Proceedings of the 20th Australasian Computing Education Conference (ACE'18)* (pp. 83–89). New York, NY, USA: Association for Computing Machinery (ACM). Retrieved from https://doi.org/10.1145/3160489.3160493 doi: 10.1145/3160489.3160493

Exton, C. (2002). Constructivism and Program Comprehension Strategies. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)* (pp. 281–284). Retrieved from https://doi.org/10.1109/WPC.2002.1021349 doi: 10.1109/WPC.2002.1021349

Ferguson, R. (2012). Learning Analytics: Drivers, Developments and Challenges. *International Journal of Technology Enhanced Learning (IJTEL)*, *4*(5/6), 304–317. Retrieved from https://doi.org/10.1504/IJTEL.2012.051816 doi: 10.1504/IJTEL.2012.051816

Feurzeig, W., Papert, S., Bloom, M., Grant, R., & Solomon, C. (1970). Programming-Languages as a Conceptual Framework for Teaching Mathematics. *ACM SIGCUE Outlook*, *4*(2), 13–17. Retrieved from https://doi.org/10.1145/965754.965757 doi: 10.1145/965754.965757

Fix, V., Wiedenbeck, S., & Scholtz, J. (1993). Mental Representations of Programs by Novices and Experts. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI'93)* (pp. 74–79). Association for Computing Machinery. Retrieved from https://doi.org/10.1145/169059.169088 doi: 10.1145/169059.169088

Foster, J. (1993). *Cost Factors in Software Maintenance* (Doctoral dissertation, Durham University). Retrieved from http://etheses.dur.ac.uk/1561/

Frey, B. J., & Dueck, D. (2007). Clustering by Passing Messages Between Data Points. *Science*, *315*(5814), 972–976. Retrieved from http://doi.org/10.1126/science.1136800 doi: 10.1126/science.1136800

Friend, J. (1975). *Programs Students Write* (Tech. Rep. No. 257). California: Citeseer.

Giannakos, M. N., Sharma, K., Pappas, I. O., Kostakos, V., & Velloso, E. (2019). Multimodal Data as a Means to Understand the Learning Experience. *International Journal of Information Management*, *48*, 108–119. Retrieved from https://doi.org/10.1016/j.ijinfomgt.2019.02.003 doi: 10.1016/j.ijinfomgt.2019.02.003

Goldberg, J. H., & Helfman, J. I. (2010). Visual Scanpath Representation. In *Proceedings of the 2010 Symposium on Eye-Tracking Research and Applications (ETRA'10)* (pp. 203–210). New York, NY, USA: ACM. Retrieved from http://doi.org/10.1145/1743666.1743717 doi: 10.1145/1743666.1743717

Good, J., & Brna, P. (2003). Toward Authentic Measures of Program Comprehension. In *Proceedings 15th Annual Workshop of the Psychology of Programming Interest Group (PPIG'03)* (pp. 29–50). Retrieved from https://ppig.org/papers/2003-ppig-15th-good/

Good, J., & Brna, P. (2004). Program Comprehension and Authentic Measurement: A Scheme for Analysing Descriptions of Programs. *International Journal of Human-Computer Studies*, *61*(2 SPEC. ISS.), 169–185. Retrieved from https://doi.org/10.1016/j.ijhcs.2003.12.010 doi: 10.1016/j.ijhcs.2003.12.010

Greiff, S., Scheiter, K., Scherer, R., Borgonovi, F., Britt, A., Graesser, A., ... Rouet, J.-F. (2017). Adaptive Problem Solving. *OECD*, *156*(156), 1–57. Retrieved from https://doi.org/10.1787/90fde2f4-en doi: 10.1787/90fde2f4-en

Greller, W., & Hoppe, U. (2017). *Learning Analytics: Implications for Higher Education* (Vol. 1). BoD–Books on Demand.

Guzdial, M. (2008). Education: Paving the Way for Computational Thinking. *Communications of the ACM*, *51*(8), 25–27. Retrieved from https://doi.org/10.1145/1378704.1378713 doi: 10.1145/1378704.1378713

Guzdial, M., & du Boulay, B. (2019). The History of Computing Education Research. In S. A. Fincher & A. V. Robins (Eds.), *The Cambridge Handbook of Computing Education Research* (p. 11–39). Cambridge University Press. Retrieved from https://doi.org/10.1017/9781108654555.002 doi: 10.1017/9781108654555.002

Hanenberg, S. (2010). An Experiment about Static and Dynamic Type Systems: Doubts about the Positive Impact of Static Type Systems on Development Time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)* (p. 22–35). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1869459.1869462 doi: 10.1145/1869459.1869462

Hanks, B., McDowell, C., Draper, D., & Krnjajic, M. (2004). Program Quality with Pair Programming in CS1. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'04)* (p. 176–180). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1007996.1008043 doi: 10.1145/1007996.1008043

Hannebauer, C., Hesenius, M., & Gruhn, V. (2018). Does Syntax Highlighting Help Programming Novices? In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)* (p. 704). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3180155.3182554 doi: 10.1145/3180155.3182554

Hansen, M., Goldstone, R. L., & Lumsdaine, A. (2014). Towards Automated Coding of Program Comprehension Gaze Data. *Eye Movements in Programming Education*, 9.

Harth, E., & Dugerdil, P. (2017). Program Understanding Models: An Historical Overview and a Classification. In *Proceedings of the 12th International Conference on Software Technologies (ICSOFT'17)* (pp. 402–413). Retrieved from https://doi.org/10.5220/0006465504020413 doi: 10.5220/0006465504020413

Harvey, B., & Mönig, J. (2010). Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists? *Constructionism*, 1–10. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.182.658&rep=rep1&type=pdf

Hein, G. (1991). *Constructivist Learning Theory.* Jerusalem Israel: International Committee of Museum Educators Conference. Retrieved from https://www.exploratorium.edu/education/ifi/constructivist-learning

Hembrooke, H., Feusner, M., & Gay, G. (2006). Averaging Scan Patterns and What They Can Tell Us. In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications (ETRA'06)* (p. 41). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/1117309.1117325 doi: 10.1145/1117309.1117325

Heminghous, J., & Duchowski, A. T. (2006). iComp: A Tool for Scanpath Visualization and Comparison. In *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization (APGV'06)* (p. 152). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/1140491.1140529 doi: 10.1145/1140491.1140529

Hertz, M. (2010). What do "CS1" and "CS2" Mean? Investigating Differences in the Early Courses. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)* (pp. 199–203). Retrieved from https://doi.org/10.1145/1734263.1734335 doi: 10.1145/1734263.1734335

Hidetake, U., Masahide, N., Akito, M., & Ken-Ichi, M. (2007). Exploiting Eye Movements for Evaluating Reviewer's Performance in Software Review. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, *E90-A*(10), 2290–2300. Retrieved from https://doi.org/10.1093/ietfec/e90-a.10.2290 doi: 10.1093/ietfec/e90-a.10.2290

Höfer, A. (2011). Exploratory Comparison of Expert and Novice Pair Programmers. In *Lecture Notes in Computer Science* (Vol. 4980, pp. 218–231). Retrieved from https://doi.org/10.1007/978-3-642-22386-0_17 doi: 10.1007/978-3-642-22386-0_17

Holmqvist, K. (2011). *Eye Tracking: A Comprehensive Guide to Methods and Measures.* Oxford University Press Oxford, England.

Hoppe, S., & Bulling, A. (2016). End-to-End Eye Movement Detection Using Convolutional Neural Networks. *CoRR*, *abs/1609.02452*. Retrieved from http://arxiv.org/abs/1609.02452

Hou, T. Y., Lin, Y. T., Lin, Y. C., Chang, C. H., & Yen, M. H. (2013). Exploring the Gender Effect on Cognitive Processes in Program Debugging Based on Eye-Movement Analysis. In *Proceedings of the 5th International Conference on Computer Supported Education (CSEDU'13)* (pp. 469–473). Retrieved from https://doi.org/10.5220/0004415104690473 doi: 10.5220/0004415104690473

Husic, F. T., Linn, M. C., & Sloane, K. D. (1989). Adapting Instruction to the Cognitive Demands of Learning to Program. *Journal of Educational Psychology*, *81*(4), 570–583. Retrieved from https://doi.org/10.1037/0022-0663.81.4.570 doi: 10.1037/0022-0663.81.4.570

Hutt, S., Mills, C., White, S., Donnelly, P. J., & D'Mello, S. K. (2016). The Eyes Have it: Gaze-Based Detection of Mind Wandering During Learning With an Intelligent Tutoring System. In *Proceedings of the 9th International Conference on Educational Data Mining (EDM 2016)* (pp. 86–93). ERIC. Retrieved from https://eric.ed.gov/?id=ED592729

Ihantola, P., Vihavainen, A., Ahadi, A., Butler, M., Börstler, J., Edwards, S. H., ... Toll, D. (2015, 7). Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports (ITiCSE-WGP'15)* (pp. 41–63). Association for Computing Machinery, Inc. Retrieved from https://doi.org/10.1145/2858796.2858798 doi: 10.1145/2858796.2858798

Izu, C., Weerasinghe, A., & Pope, C. (2016). A Study of Code Design Skills in Novice Programmers Using the SOLO Taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER'16)* (pp. 251–259). Retrieved from https://doi.org/10.1145/2960310.2960324 doi: 10.1145/2960310.2960324

Jackson, G. T., & McNamara, D. S. (2013). Motivation and Performance in a Game-Based Intelligent Tutoring System. *Journal of Educational Psychology*, *105*(4), 1036–1049. Retrieved from https://doi.org/10.1037/a0032580 doi: 10.1037/a0032580

Jarodzka, H., Holmqvist, K., & Gruber, H. (2017). Eye Tracking in Educational Science: Theoretical Frameworks and Research Agendas. *Journal of Eye Movement Research*, *10*(1). Retrieved from https://doi.org/10.16910/jemr.10.1.3 doi: 10.16910/jemr.10.1.3

Jarodzka, H., Van Gog, T., Dorr, M., Scheiter, K., & Gerjets, P. (2013). Learning to See: Guiding Students' Attention via a Model's Eye Movements Fosters Learning. *Learning and Instruction*, *25*, 62–70. Retrieved from https://doi.org/10.1016/j.learninstruc.2012.11.004 doi: 10.1016/j.learninstruc.2012.11.004

Jenkins, T. (2002). On the Difficulty of Learning to Program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences* (Vol. 4, pp. 53–58). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.596.9994

Johnson, A. M., McCarthy, K. S., Kopp, K. J., Perret, C. A., & McNamara, D. S. (2017). Adaptive Reading and Writing Instruction in iSTART and W-Pal. In *Proceedings of the 30th International Florida Artificial Intelligence Research Society Conference (FLAIRS'17)* (pp. 561–566).

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying Student Misconceptions of Programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)* (pp. 107–111). Retrieved from https://doi.org/10.1145/1734263.1734299 doi: 10.1145/1734263.1734299

Kaijanaho, A.-j. (2015). Evidence-Based Programming Language Design: A Philosophical and Methodological Exploration. *Jyväskylä studies in computing*(222), 259.

Kalyuga, S. (2010). Schema Acquisition and Sources of Cognitive Load. In J. L. Plass, R. Moreno, & R. Brünken (Eds.), *Cognitive Load Theory* (p. 48–64). Cambridge University Press. Retrieved from https://doi.org/10.1017/CBO9780511844744.005 doi: 10.1017/CBO9780511844744.005

Kelleher, C., & Pausch, R. (2005). Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Computing Surveys*, *37*(2), 83–137. Retrieved from https://doi.org/10.1145/1089733.1089734 doi: 10.1145/1089733.1089734

Khairuddin, N. N., & Hashim, K. (2008). Application of Bloom's Taxonomy in Software Engineering Assessments. In *Proceedings of the 8th Conference on Applied Computer Scince (ACS'08)* (p. 66–69). Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS).

Kintsch, W. (1998). *Comprehension: A Paradigm for Cognition.* Cambridge University Press.

Kluthe, T. (2014). *A Measurement of Programming Language Comprehension Using p-BCI: An Empirical Study on Phasic Changes in Alpha and Theta Brain Waves* (Doctoral dissertation, Southern Illinois University Edwardsville). doi: 10.1017/CBO9781107415324.004

Knuth, D. E. (1972, 8). George Forsythe and the Development of Computer Science. *Communications of the ACM*, *15*(8), 721–726. Retrieved from https://doi.org/10.1145/361532.361538 doi: 10.1145/361532.361538

Kölling, M. (2010, November). The Greenfoot Programming Environment. *ACM Trans. Comput. Educ.*, *10*(4). Retrieved from https://doi.org/10.1145/1868358.1868361 doi: 10.1145/1868358.1868361

Koulouri, T., Lauria, S., & Macredie, R. D. (2014). Teaching Introductory Programming: A Quantitative Evaluation of Different Approaches. *ACM Transactions on Computing Education*, *14*(4). Retrieved from https://doi.org/10.1145/2662412 doi: 10.1145/2662412

Krajcik, J., & Merritt, J. (2012). Engaging Students in Scientific Practices: What Does Constructing and Revising Models Look Like in the Science Classroom? *The Science Teacher*, *79*(3), 38–41. Retrieved from https://search.proquest.com/openview/4f29da8460ffb46a119972b904b8a7f6/

Kübler, T. C., Rothe, C., Schiefer, U., Rosenstiel, W., & Kasneci, E. (2017). SubsMatch 2.0: Scanpath Comparison and Classification Based on Subsequence Frequencies. *Behavior Research Methods*, *49*(3), 1048–1064. Retrieved from https://doi.org/10.3758/s13428-016-0765-6 doi: 10.3758/s13428-016-0765-6

Kuittinen, M., & Sajaniemi, J. (2004). Teaching Roles of Variables in Elementary Programming Courses. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'04)* (p. 57–61). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1007996.1008014 doi: 10.1145/1007996.1008014

Kulik, J. A., & Fletcher, J. D. (2016). Effectiveness of Intelligent Tutoring Systems: A Meta-Analytic Review. *Review of Educational Research*, *86*(1), 42–78. Retrieved from https://doi.org/10.3102/0034654315581420 doi: 10.3102/0034654315581420

Lemaignan, S., Garcia, F., Jacq, A., & Dillenbourg, P. (2016). From Real-Time Attention Assessment to "With-Me-Ness" in Human-Robot Interaction. In *Proceedings of the 11th ACM/IEEE International Conference on Human-Robot Interaction (HRI'16)* (Vol. 2016-April, pp. 157–164). Retrieved from https://doi.org/10.1109/HRI.2016.7451747 doi: 10.1109/HRI.2016.7451747

Le Meur, O., & Baccino, T. (2013). Methods for Comparing Scanpaths and Saliency Maps: Strengths and Weaknesses. *Behavior Research Methods*, *45*(1), 251–266. Retrieved from https://doi.org/10.3758/s13428-012-0226-9 doi: 10.3758/s13428-012-0226-9

Leutenegger, S., & Edgington, J. (2007). A Games First Approach to Teaching Introductory Programming. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'07)* (pp. 115–118). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1227310.1227352 doi: 10.1145/1227310.1227352

Lieberman, H., Paternò, F., Klann, M., & Wulf, V. (2006). End-User Development: An Emerging Paradigm. In H. Lieberman, F. Paternò, & V. Wulf (Eds.), *End user development* (pp. 1–8). Dordrecht: Springer Netherlands. Retrieved from https://doi.org/10.1007/1-4020-5386-X_1 doi: 10.1007/1-4020-5386-x_1

Linn, M. C. (1985). The Cognitive Consequences of Programming Instruction in Classrooms. *Educational Researcher*, *14*(5), 14–29. Retrieved from https://doi.org/10.3102/0013189X014005014 doi: 10.3102/0013189X014005014

Linn, M. C., & Dalbey, J. (1985). Cognitive Consequences of Programming Instruction: Instruction, Access, and Ability. *Educational Psychologist*, *20*(4), 191–206. Retrieved from https://doi.org/10.1207/s15326985ep2004_4 doi: 10.1207/s15326985ep2004_4

Lister, R., Fidge, C., & Teague, D. (2009). Further Evidence of a Relationship Between Explaining, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the Conference on Integrating Technology into Computer Science Education (ITiCSE'09)* (pp. 161–165). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/1595496.1562930 doi: 10.1145/1562877.1562930

Lister, R., & Leaney, J. (2003). Introductory Programming, Criterion-Referencing, and Bloom. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03)* (p. 143–147). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/611892.611954 doi: 10.1145/611892.611954

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE'06)* (p. 118–122). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1140124.1140157 doi: 10.1145/1140124.1140157

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the ACM Workshop on International Computing Education Research (ICER'08)* (pp. 101–112). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/1404520.1404531 doi: 10.1145/1404520.1404531

Ludi, S., Natarajan, S., & Reichlmayr, T. (2005). An Introductory Software Engineering Course that Facilitates Active Learning. *Proceedings of the Thirty-Sixth SIGCSE Technical Symposium on Computer Science Education (SIGCSE'05)*, *37*(1), 302–306. Retrieved from https://doi.org/10.1145/1047124.1047449 doi: 10.1145/1047124.1047449

Madison, S., & Gifford, J. (1997). Parameter Passing: The Conceptions Novices Construct. *Proceedings of the Annual Meeting of the American Educational Research Association*, 2–29. Retrieved from https://api.semanticscholar.org/CorpusID:59968841

Mavroudi, A., Giannakos, M., & Krogstie, J. (2018). Supporting Adaptive Learning Pathways Through the use of Learning Analytics: Developments, Challenges and Future Opportunities. *Interactive Learning Environments*, *26*(2), 206–220. Retrieved from https://doi.org/10.1080/10494820.2017.1292531 doi: 10.1080/10494820.2017.1292531

Mayer, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys (CSUR)*, *13*(1), 121–141. Retrieved from https://doi.org/10.1145/356835.356841 doi: 10.1145/356835.356841

Mayrhauser, A. V., & Vans, A. M. (1995). Program Comprehension During Software Maintenance and Evolution. *Computer*, *28*(8), 44–55. Retrieved from https://doi.org/10.1109/2.402076 doi: 10.1109/2.402076

McCall, D., & Kölling, M. (2014). Meaningful Categorisation of Novice Programmer Errors. In *Proceedings of the 2014 IEEE Frontiers in Education Conference (FIE'14)* (pp. 1–8). Retrieved from https://doi.org/10.1109/FIE.2014.7044420 doi: 10.1109/FIE.2014.7044420

McDowell, C., Werner, L., Bullock, H. E., & Fernald, J. (2003). The Impact of Pair Programming on Student Performance, Perception and Persistence. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)* (pp. 602–607). Washington, DC, USA: IEEE Computer Society. Retrieved from https://doi.org/10.1109/icse.2003.1201243 doi: 10.1109/icse.2003.1201243

McGill, T. J., & Volet, S. E. (1997). A Conceptual Framework for Analyzing Students' Knowledge of Programming. *Journal of Research on Computing in Education*, *29*(3), 276–297. Retrieved from https://doi.org/10.1080/08886504.1997.10782199 doi: 10.1080/08886504.1997.10782199

Mendelsohn, P., Green, T., & Brna, P. (1990). Programming Languages in Education: The Search for an Easy Start. In *Psychology of Programming* (pp. 175–200). Elsevier. Retrieved from https://doi.org/10.1016/B978-0-12-350772-3.50016-1 doi: 10.1016/b978-0-12-350772-3.50016-1

Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information. *Psychological Review*, *63*(2), 81–97. Retrieved from https://doi.org/10.1037/h0043158 doi: 10.1037/h0043158

Moreno-Leon, J., & Robles, G. (2016). Code to Learn With Scratch? A Systematic Literature Review. In *Procedings of the IEEE Global Engineering Education Conference (EDUCON'16)* (Vol. 10-13-Apri, pp. 150–156). IEEE. Retrieved from https://doi.org/10.1109/EDUCON.2016.7474546 doi: 10.1109/EDUCON.2016.7474546

Moskal, B., Lurie, D., & Cooper, S. (2004). Evaluating the Effectiveness of a New Instructional Approach. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'04)* (p. 75–79). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/971300.971328 doi: 10.1145/971300.971328

Murphy, L., McCauley, R., & Fitzgerald, S. (2012). 'Explain in Plain English' Questions: Implications for Teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)* (pp. 385–390). Retrieved from https://doi.org/10.1145/2157136.2157249 doi: 10.1145/2157136.2157249

Murray, T. (1999). Authoring Intelligent Tutoring Systems: An Analysis of the State of the Art. *International Journal of Artificial Intelligence in Education (IJAIED)*, *10*, 98–129. Retrieved from https://api.semanticscholar.org/CorpusID:6259313

Murtagh, F., & Legendre, P. (2014). Ward's Hierarchical Agglomerative Clustering Method: Which Algorithms Implement Ward's Criterion? *Journal of Classification*, *31*(3), 274–295. Retrieved from https://doi.org/10.1007/s00357-014-9161-z doi: 10.1007/s00357-014-9161-z

Myers, B. A., Burnett, M. M., Ko, A. J., Rosson, M. B., Scaffidi, C., & Wiedenbeck, S. (2010). End User Software Engineering: CHI 2010 Special Interest Group Meeting. In *CHI '10 Extended Abstracts on Human Factors in Computing Systems* (p. 3189–3192). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1753846.1753953 doi: 10.1145/1753846.1753953

Najar, A. S., Mitrovic, A., & Neshatian, K. (2014). Utilizing Eye Tracking to Improve Learning From Examples. In C. Stephanidis & M. Antona (Eds.), *Lecture Notes in Computer Science* (Vol. 8514, pp. 410–418). Cham: Springer International Publishing. Retrieved from https://doi.org/10.1007/978-3-319-07440-5_38 doi: 10.1007/978-3-319-07440-5_38

Newell, A., Perlis, A. J., & Simon, H. A. (1967). Computer Science. *Science*, *157*(3795), 1373–1374. Retrieved from https://doi.org/10.1126/science.157.3795.1373-b doi: 10.1126/science.157.3795.1373-b

Njeru, A. M., & Paracha, S. (2017). Learning Analytics: Supporting At-Risk Student Through Eye-Tracking and a Robust Intelligent Tutoring System. In *Proceedings of the IEEE International Conference on Applied System Innovation (ICASI'17)* (pp. 1002–1005). Sapporo, Japan: IEEE Computer Society. Retrieved from https://doi.org/10.1109/ICASI.2017.7988616 doi: 10.1109/ICASI.2017.7988616

Obaidellah, U., & Haek, M. A. (2018). Evaluating Gender Difference on Algorithmic Problems Using Eye-Tracker. In *Proceedings of the 2018 ACM Symposium on Eye Tracking Research and Applications (ETRA'18).* New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3204493.3204537 doi: 10.1145/3204493.3204537

O'brien, M. P., Shaft, T. M., & Buckley, J. (2001). An Open-Source Analysis Schema for Identifying Software Comprehension Processes. In *Proceedings of the 13th Psychology of Programming Interest Group Annual Conference (PPIG'01)* (p. 11). Retrieved from www.ppig.org

Orquin, J. L., Ashby, N. J., & Clarke, A. D. (2016). Areas of Interest as a Signal Detection Problem in Behavioral Eye-Tracking Research. *Journal of Behavioral Decision Making*, *29*(2-3), 103–115. doi: 10.1002/bdm.1867

Papert, S. (1980). *Mindstorms: Children, Computers and Powerful Ideas*. New York N.Y.: Basic Books.

Papert, S. (1996). An Exploration in the Space of Mathematics Educations. *International Journal of Computers for Mathematical Learning*, *1*(1), 95–123. Retrieved from https://doi.org/10.1007/BF00191473 doi: 10.1007/BF00191473

Parihar, S., Dadachanji, Z., Singh, P. K., Das, R., Karkare, A., & Bhattacharya, A. (2017). Automatic Grading and Feedback Using Program Repair for Introductory Programming Courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'17)* (p. 92–97). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3059009.3059026 doi: 10.1145/3059009.3059026

Pea, R. D., & Kurland, D. M. (1984). On the Cognitive Effects of Learning Computer Programming. *New Ideas in Psychology*, *2*(2), 137–168. Retrieved from https://doi.org/10.1016/0732-118X(84)90018-7 doi: 10.1016/0732-118X(84)90018-7

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., ... Paterson, J. (2007). A Survey of Literature on the Teaching of Introductory Programming. In *Proceedings of the Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR'07)* (p. 204–223). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1345443.1345441 doi: 10.1145/1345443.1345441

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research*, *2*(1), 37–55. Retrieved from https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL doi: 10.2190/gujt-jcbj-q6qu-q9pl

Perkins, D. N., & Martin, F. (1986). Fragile Knowledge and Neglected Strategies in Novice Programmers. In *Proceedings of the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers* (p. 213–229). USA: Ablex Publishing Corp.

Perkins, D. N., Schwartz, S., & Simmons, R. (1988). Instructional Strategies for the Problems of Novice Programmers. In *Teaching and Learning Computer Programming: Multiple Research Perspectives* (p. 26). Lawrence Erlbaum Associates, Inc. Retrieved from https://doi.org/10.4324/9781315044347 doi: 10.4324/9781315044347

Piattini, M., Polo, M., & Ruiz, F. (Eds.). (2003). *Advances in Software Maintenance Management*. IGI Global. Retrieved from https://doi.org/10.4018/978-1-59140-047-9 doi: 10.4018/978-1-59140-047-9

Politowski, C., Khomh, F., Romano, S., Scanniello, G., Petrillo, F., Guéhéneuc, Y. G., & Maiga, A. (2020). A Large Scale Empirical Study of the Impact of Spaghetti Code and Blob Anti-Patterns on Program Comprehension. *Information and Software Technology*, *122*, 181–190. Retrieved from https://doi.org/10.1016/j.infsof.2020.106278 doi: 10.1016/j.infsof.2020.106278

Porter, L., Guzdial, M., McDowell, C., & Simon, B. (2013, 8). Success in Introductory Programming: What Works? *Communications of the ACM*, *56*(8), 34–36. Retrieved from https://doi.org/10.1145/2492007.2492020 doi: 10.1145/2492007.2492020

Price, T. W., Dong, Y., & Lipovac, D. (2017). ISnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE'17)* (p. 483–488). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3017680.3017762 doi: 10.1145/3017680.3017762

Prieto, L. P., Sharma, K., Wen, Y., Dillenbourg, P., & Caballero, D. (2014). Studying Teacher Cognitive Load in Multi-Tabletop Classrooms Using Mobile Eye-Tracking. In *Proceedings of the 2014 ACM International Conference on Interactive Tabletops and Surfaces (ITS'14)* (pp. 339–344). Retrieved from https://doi.org/10.1145/2669485.2669543 doi: 10.1145/2669485.2669543

Raschke, M., Herr, D., Blascheck, T., Ertl, T., Burch, M., Willmann, S., & Schrauf, M. (2014). A Visual Approach for Scan Path Comparison. In *Proceedings of the Symposium on Eye Tracking Research and Applications (ETRA'14)* (p. 135–142). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2578153.2578173 doi: 10.1145/2578153.2578173

Rayner, K. (1998). Eye Movements in Reading and Information Processing: 20 Years of Research. *Psychological Bulletin*, *124*(3), 372–422. Retrieved from https://doi.org/10.1037/0033-2909.124.3.372 doi: 10.1037/0033-2909.124.3.372

Rayner, K. (2009). Eye Movements and Attention in Reading, Scene Perception, and Visual Search. *Quarterly Journal of Experimental Psychology*, *62*(8), 1457–1506. Retrieved from https://doi.org/10.1080/17470210902816461 doi: 10.1080/17470210902816461

Reani, M., Peek, N., & Jay, C. (2018). An Investigation of the Effects of n-gram Length in Scanpath Analysis for Eye-Tracking Research. In *Proceedings of the 2018 ACM Symposium on Eye Tracking Research and Applications (ETRA'18)* (pp. 1–8). Retrieved from https://doi.org/10.1145/3204493.3204527 doi: 10.1145/3204493.3204527

Reingold, E. M., & Sheridan, H. (2012). Eye Movements and Visual Expertise in Chess and Medicine. In *The Oxford Handbook of Eye Movements* (p. 528). Oxford University Press Oxford, England. Retrieved from https://doi.org/10.1093/oxfordhb/9780199539789.013.0029 doi: 10.1093/oxfordhb/9780199539789.013.0029

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., … Kafai, Y. (2009, 11). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60–67. Retrieved from https://doi.org/10.1145/1592761.1592779%0A doi: 10.1145/1592761.1592779

Riedl, T. R., Weitzenfeld, J. S., Freeman, J. T., Klein, G. A., & Musa, J. (1991). What we Have Learned About Software Engineering Expertise. In J. E. Tomayko (Ed.), *Lecture Notes in Computer Science* (Vol. 536, pp. 261–270). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from https://doi.org/10.1007/bfb0024298 doi: 10.1007/bfb0024298

Rist, R. S. (1989). Schema Creation in Programming. *Cognitive Science*, *13*(3), 389–414. Retrieved from https://doi.org/10.1207/s15516709cog1303_3 doi: 10.1016/0364-0213(89)90018-9

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *International Journal of Phytoremediation*, *21*(1), 137–172. Retrieved from https://doi.org/10.1076/csed.13.2.137.14200 doi: 10.1076/csed.13.2.137.14200

Rogalski, J., & Samurçay, R. (1990). Acquisition of Programming Knowledge and Skills. In *Psychology of Programming* (pp. 157–174). Retrieved from https://doi.org/10.1016/B978-0-12-350772-3.50015-X doi: 10.1016/b978-0-12-350772-3.50015-x

Rosenholtz, R., Li, Y., Mansfield, J., & Jin, Z. (2005). Feature Congestion: A Measure of Display Clutter. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'05)* (p. 761–770). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1054972.1055078 doi: 10.1145/1054972.1055078

Rouse, W. B. (1982). Software Psychology: Human Factors in Computer and Information Systems. *Information Processing & Management*, *18*(3), 164–165. Retrieved from https://doi.org/10.1016/0306-4573(82)90043-7 doi: 10.1016/0306-4573(82)90043-7

Sajaniemi, J., Kuittinen, M., & Tikansalo, T. (2008). A Study of the Development of Students' Visualizations of Program State During an Elementary Object-Oriented Programming Course. *Journal on Educational Resources in Computing*, *7*(4), 1–31. Retrieved from https://doi.org/10.1145/1316450.1316453 doi: 10.1145/1316450.1316453

Saldaña, J. (2009). *The Coding Manual for Qualitative Researchers.* SAGE Publications Ltd. Retrieved from https://uk.sagepub.com/en-gb/eur/the-coding-manual-for-qualitative-researchers/book243616

Samurçay, R. (1989). The Concept of Variable in Programming: Its Meaning and use in Problem-Solving by Novice Programmers. *Studying the Novice Programmer*, *9*, 161 – 178.

Sanders, I., Galpin, V., & Götschi, T. (2006). Mental Models of Recursion Revisited. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE'06)* (p. 138–142). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1140124.1140162 doi: 10.1145/1140124.1140162

Sarkar, A. (2015). The Impact of Syntax Colouring on Program Comprehension. In *Proceedings of the 26th Annual Conference of the Psychology of Programming Interest Group (PPIG'15)* (pp. 49–58). Retrieved from https://ppig.org/papers/2015-ppig-26th-sarkar1/

Scaffidi, C., Shaw, M., & Myers, B. (2005). Estimating the Numbers of End Users and End User Programmers. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)* (Vol. 2005, pp. 207–214). IEEE. Retrieved from https://doi.org/10.1109/VLHCC.2005.34 doi: 10.1109/VLHCC.2005.34

Schankin, A., Berger, A., Holt, D. V., Hofmeister, J. C., Riedel, T., & Beigl, M. (2018). Descriptive Compound Identifier Names Improve Source Code Comprehension. In *Proceedings of the 26th Conference on Program Comprehension (ICPC'18)* (pp. 31–40). Retrieved from https://doi.org/10.1145/3196321.3196332 doi: 10.1145/3196321.3196332

Schlösser, C. (2020). *Towards Concise Gaze Sharing* (Doctoral dissertation, Hagen). doi: 10.18445/20191219-085351-0

Schneider, B., Sharma, K., Cuendet, S., Zufferey, G., Dillenbourg, P., & Pea, R. (2018). Leveraging Mobile Eye-Trackers to Capture Joint Visual Attention in Co-Located Collaborative Learning Groups. *International Journal of Computer-Supported Collaborative Learning*, *13*(3), 241–261. Retrieved from https://doi.org/10.1007/s11412-018-9281-2 doi: 10.1007/s11412-018-9281-2

Schrepp, M., Hinderks, A., & Thomaschewski, J. (2017a). Construction of a Benchmark for the User Experience Questionnaire (UEQ). *International Journal of Interactive Multimedia and Artificial Intelligence*, *4*(4), 40. Retrieved from https://doi.org/10.9781/ijimai.2017.445 doi: 10.9781/ijimai.2017.445

Schrepp, M., Hinderks, A., & Thomaschewski, J. (2017b). Design and Evaluation of a Short Version of the User Experience Questionnaire (UEQ-S). *International Journal of Interactive Multimedia and Artificial Intelligence*, *4*(6), 103. Retrieved from https://doi.org/10.9781/ijimai.2017.09.001 doi: 10.9781/ijimai.2017.09.001

Schröder, B. (2019). *Sublime Coding* (Unpublished doctoral dissertation). University of Applied Sciences and Arts, Dortmund, Dortmund.

Schulte, C. (2008). Block Model: An Educational Model of Program Comprehension As a Tool for a Scholarly Approach to Teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICRE'08)* (pp. 149–160). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/1404520.1404535 doi: 10.1145/1404520.1404535

Schulte-Mecklenbeck, M., Kühberger, A., & Johnson, J. G. (2011). *A Handbook of Process Tracing Methods for Decision Research.* Psychology Press. Retrieved from https://doi.org/10.4324/9780203875292 doi: 10.4324/9780203875292

Sevella, P. K., & Lee, Y. (2013). Determining the Barriers Faced by Novice Programmers. *International Journal of Software Engineering (IJSE), Vol.4*(1), 10–22.

Shaft, T. M., & Vessey, I. (1998). The Relevance of Application Domain Knowledge: Characterizing the Computer Program Comprehension Process. *Journal of Management Information Systems*, *15*(1), 51–78. Retrieved from https://doi.org/10.1080/07421222.1998.11518196 doi: 10.1080/07421222.1998.11518196

Sharafi, Z., Shaffer, T., Sharif, B., & Gueheneuc, Y. G. (2016). Eye-Tracking Metrics in Software Engineering. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC'16)* (Vol. 2016-May, pp. 96–103). Retrieved from https://doi.org/10.1109/APSEC.2015.53 doi: 10.1109/APSEC.2015.53

Sharafi, Z., Soh, Z., Guéhéneuc, Y. G., & Antoniol, G. (2012). Women and Men - Different but Equal: On the Impact of Identifier Style on Source Code Reading. In *Proceedings of the 2012 20th IEEE International Conference on Program Comprehension (ICPC'12)* (pp. 27–36). Retrieved from https://doi.org/10.1109/icpc.2012.6240505 doi: 10.1109/icpc.2012.6240505

Sharif, B., Falcone, M., & Maletic, J. I. (2012). An Eye-Tracking Study on the Role of Scan Time in Finding Source Code Defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications (ETRA'12)* (pp. 381–384). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/2168556.2168642 doi: 10.1145/2168556.2168642

Sharif, B., & Maletic, J. I. (2010). An Eye Tracking Study on the Effects of Layout in Understanding the Role of Design Patterns. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM'06)* (pp. 1–10). Retrieved from https://doi.org/10.1109/ICSM.2010.5609582 doi: 10.1109/ICSM.2010.5609582

Sharma, K., Alavi, H. S., Jermann, P., & Dillenbourg, P. (2016). A Gaze-Based Learning Analytics Model: In-Video Visual Feedback to Improve Learner's Attention in MOOCs. In *Proceedings of the Sixth International Conference on Learning Analytics and Knowledge (LAK'16)* (p. 417–421). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2883851.2883902 doi: 10.1145/2883851.2883902

Sharma, K., Chavez-Demoulin, V., & Dillenbourg, P. (2017). An Application of Extreme Value Theory to Learning Analytics: Predicting Collaboration Outcome from Eye-Tracking Data. *Journal of Learning Analytics*, *4*(3), 140–164. Retrieved from https://doi.org/10.18608/jla.2017.43.8 doi: 10.18608/jla.2017.43.8

Sharma, K., D'Angelo, S., Gergle, D., & Dillenbourg, P. (2016). Visual Augmentation of Deictic Gestures in MOOC Videos. In *Proceedings of International Conference of the Learning Sciences (ICLS'16)* (pp. 202–209). International Society of the Learning Sciences (ISLS). Retrieved from http://www.scopus.com/inward/record.url?scp=84987836154&partnerID=8YFLogxK

Sharma, K., Jermann, P., & Dillenbourg, P. (2015). Displaying Teacher's Gaze in a MOOC: Effects on Students' Video Navigation Patterns. In *Lecture Notes in Computer Science* (Vol. 9307, pp. 325–338). Retrieved from https://doi.org/10.1007/978-3-319-24258-3_24 doi: 10.1007/978-3-319-24258-3_24

Shneiderman, B. (1975). Cognitive Psychology and Programming Language Design. *ACM SIGPLAN Notices*, *10*(7), 46–47. Retrieved from https://doi.org/10.1145/987305.987314 doi: 10.1145/987305.987314

Shneiderman, B. (1976). Exploratory Experiments in Programmer Behavior. *International Journal of Computer & Information Sciences*, *5*(2), 123–143. Retrieved from https://doi.org/10.1007/BF00975629 doi: 10.1007/BF00975629

Shneiderman, B., & Mayer, R. (1979). Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer & Information Sciences*, *8*(3), 219–238. Retrieved from https://doi.org/10.1007/BF00977789 doi: 10.1007/BF00977789

Shute, V. J. (1991). Who is Likely to Acquire Programming Skills? *Journal of Educational Computing Research*, *7*(1), 1–24. Retrieved from https://doi.org/10.2190/VQJD-T1YD-5WVB-RYPJ doi: 10.2190/vqjd-t1yd-5wvb-rypj

Siegmund, J. (2016). Program Comprehension: Past, Present, and Future. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)* (pp. 13–20). Retrieved from https://doi.org/10.1109/saner.2016.35 doi: 10.1109/saner.2016.35

Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., … Brechmann, A. (2014). Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)* (p. 378–389). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2568225.2568252 doi: 10.1145/2568225.2568252

Sime, M. E., Arblaster, A. T., & Green, T. R. (1977). Structuring the Programmer's Task. *Journal of Occupational Psychology*, *50*(3), 205–216. Retrieved from https://doi.org/10.1111/j.2044-8325.1977.tb00376.x doi: 10.1111/j.2044-8325.1977.tb00376.x

Simon. (2011). Assignment and Sequence: Why Some Students Can't Recognise a Simple Swap. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research (Koli Calling '11)* (p. 10–15). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2094131.2094134 doi: 10.1145/2094131.2094134

Singh, R., Gulwani, S., & Solar-Lezama, A. (2013). Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)* (p. 15). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2491956.2462195 doi: 10.1145/2491956.2462195

Sleeman, D., Putnam, R. T., Baxter, J., & Kuspa, L. (1986). Pascal and High School Students: A Study of Errors. *Journal of Educational Computing Research*, *2*(1), 5–23. Retrieved from https://doi.org/10.2190/2XPP-LTYH-98NQ-BU77 doi: 10.2190/2xpp-ltyh-98nq-bu77

Soloway, E. (1986). Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*, *29*(9), 850–858. Retrieved from https://doi.org/10.1145/6592.6594 doi: 10.1145/6592.6594

Soloway, E., & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, *SE-10*(5), 595–609. Retrieved from http://doi.org/10.1109/TSE.1984.5010283 doi: 10.1109/TSE.1984.5010283

Soloway, E., Lochhead, J., & Clement, J. (1982). Does Computer Programming Enhance Problem Solving Ability? Some Positive Evidence on Algebra Word Problems. In *Computer Literacy* (pp. 171–201). Elsevier. Retrieved from https://doi.org/10.1016/B978-0-12-634960-3.50023-3 doi: 10.1016/B978-0-12-634960-3.50023-3

Spohrer, J. C., & Soloway, E. (1986, 7). Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM*, *29*(7), 624–632. Retrieved from https://doi.org/10.1145/6138.6145 doi: 10.1145/6138.6145

Spohrer, J. C., & Soloway, E. (1989). Simulating Student Programmers. In *Ann arbor 1001* (Vol. 89, pp. 543–549).

Sridharan, S., Bailey, R., McNamara, A., & Grimm, C. (2012). Subtle Gaze Manipulation for Improved Mammography Training. In *Proceedings of the Symposium on Eye Tracking Research and Applications (ETRA'12)* (p. 75–82). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2168556.2168568 doi: 10.1145/2168556.2168568

Stamper, J., Eagle, M., Barnes, T., & Croy, M. (2013). Experimental Evaluation of Automatic Hint Generation for a Logic Tutor. *International Journal of Artificial Intelligence in Education*, *22*(1-2), 3–17. Retrieved from https://doi.org/10.3233/JAI-130029 doi: 10.3233/JAI-130029

Startsev, M., Agtzidis, I., & Dorr, M. (2019). 1D CNN with BLSTM for Automated Classification of Fixations, Saccades, and Smooth Pursuits. *Behavior Research Methods*, *51*(2), 556–572. Retrieved from https://doi.org/10.3758/s13428-018-1144-2 doi: 10.3758/s13428-018-1144-2

Steenbergen-Hu, S., & Cooper, H. (2013). A Meta-Analysis of the Effectiveness of Intelligent Tutoring Systems on K-12 Students' Mathematical Learning. *Journal of Educational Psychology*, *105*(4), 970–987. Retrieved from https://doi.org/10.1037/a0032447 doi: 10.1037/a0032447

Steenbergen-Hu, S., & Cooper, H. (2014). A Meta-Analysis of the Effectiveness of Intelligent Tutoring Systems on College Students' Academic Learning. *Journal of Educational Psychology*, *106*(2), 331–347. Retrieved from https://doi.org/10.1037/a0034752 doi: 10.1037/a0034752

Stefik, A., Sharif, B., Myers, B. A., & Hanenberg, S. (2018). Evidence About Programmers for Programming Language Design (Dagstuhl Seminar 18061). In *Dagstuhl Reports* (Vol. 8, pp. 1–25). Retrieved from https://doi.org/10.4230/DagRep.8.2.1 doi: 10.4230/DagRep.8.2.1

Stefik, A., & Siebert, S. (2013). An Empirical Investigation Into Programming Language Syntax. *ACM Transactions on Computing Education*, *13*(4), 19. Retrieved from https://doi.org/10.1145/2534973 doi: 10.1145/2534973

Sudol-DeLyser, L. A., Stehlik, M., & Carver, S. (2012). Code Comprehension Problems as Learning Events. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'12)* (p. 81–86). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2325296.2325319 doi: 10.1145/2325296.2325319

Sun, M., Tsujikawa, M., Onishi, Y., Ma, X., Nishino, A., & Hashimoto, S. (2018). A Neural-Network-Based Investigation of Eye-Related Movements for Accurate Drowsiness Estimation. In *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBS'18)* (pp. 5207–5210). Retrieved from https://doi.org/10.1109/EMBC.2018.8513491 doi: 10.1109/EMBC.2018.8513491

Sweller, J. (2010). Cognitive Load Theory: Recent Theoretical Advances. In *Cognitive Load Theory* (pp. 29–47). Cambridge University Press. Retrieved from https://doi.org/10.1017/CBO9780511844744.004 doi: 10.1017/CBO9780511844744.004

Tabesh, Y. (2017). Computational Thinking: A 21st Century Skill. *Olympiads in Informatics*, *11*(Special Issue), 65–70. Retrieved from https://doi.org/10.15388/ioi.2017.special.10 doi: 10.15388/ioi.2017.special.10

Teague, D., Corney, M., Ahadi, A., & Lister, R. (2012). Swapping as the "Hello World" of Relational Reasoning: Replications, Reflections, and Extensions. In *Proceedings of Conferences in Research and Practice in Information Technology (CRPIT'12)* (pp. 87–94).

Tew, A. E., & Guzdial, M. (2011). The FCS1: A Language Independent Assessment of CS1 Knowledge. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11)* (pp. 111–116). Retrieved from https://doi.org/10.1145/1953163.1953200 doi: 10.1145/1953163.1953200

Trætteberg, H., Mavroudi, A., Sharma, K., & Giannakos, M. (2018). Utilizing Real-Time Descriptive Learning Analytics to Enhance Learning Programming. In M. J. Spector, B. B. Lockee, & M. D. Childress (Eds.), *Learning, Design, and Technology* (pp. 1–22). Cham: Springer International Publishing. Retrieved from https://doi.org/10.1007/978-3-319-17727-4_117-1 doi: 10.1007/978-3-319-17727-4_117-1

Traver, V. J. (2010). On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction*, *2010*. Retrieved from https://doi.org/10.1155/2010/602570 doi: 10.1155/2010/602570

Ulrich Hoppe, H., & Werneburg, S. (2019). Computational Thinking – More Than a Variant of Scientific Inquiry! In *Computational Thinking Education* (pp. 13–30). Springer. Retrieved from https://doi.org/10.1007/978-981-13-6528-7_2 doi: 10.1007/978-981-13-6528-7_2

Utting, I., Cooper, S., Kölling, M., Maloney, J., & Resnick, M. (2010, November). Alice, Greenfoot, and Scratch – A Discussion. *ACM Trans. Comput. Educ.*, *10*(4). Retrieved from https://doi.org/10.1145/1868358.1868364 doi: 10.1145/1868358.1868364

Uwano, H., Nakamura, M., Monden, A., & Matsumoto, K.-i. (2006). Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. In *Proceedings of the 2006 Symposium on Eye Tracking Research and Applications (ETRA'06)* (p. 133–140). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1117309.1117357 doi: 10.1145/1117309.1117357

Vainio, V., & Sajaniemi, J. (2007). Factors in Novice Programmers' Poor Tracing Skills. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'07)* (p. 236–240). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1268784.1268853 doi: 10.1145/1268784.1268853

Van Der Maas, H. L., & Wagenmakers, E. J. (2005). A Psychometric Analysis of Chess Expertise. *American Journal of Psychology*, *118*(1), 29–60. Retrieved from http://www.jstor.org/stable/30039042

van Gog, T., Jarodzka, H., Scheiter, K., Gerjets, P., & Paas, F. (2009). Attention Guidance During Example Study via the Model's Eye Movements. *Computers in Human Behavior*, *25*(3), 785–791. Retrieved from https://doi.org/10.1016/j.chb.2009.02.007 doi: 10.1016/j.chb.2009.02.007

van Marlen, T., van Wermeskerken, M., Jarodzka, H., & van Gog, T. (2016). Showing a Model's Eye Movements in Examples Does Not Improve Learning of Problem-Solving Tasks. *Computers in Human Behavior*, *65*, 448–459. Retrieved from https://doi.org/10.1016/j.chb.2016.08.041 doi: 10.1016/j.chb.2016.08.041

Venables, A., Tan, G., & Lister, R. (2009). A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In *Proceedings of the 2009 ACM Workshop on International Computing Education Research (ICER'09)* (pp. 117–128). Retrieved from https://doi.org/10.1145/1584322.1584336 doi: 10.1145/1584322.1584336

Vine, S. J., Masters, R. S., McGrath, J. S., Bright, E., & Wilson, M. R. (2012). Cheating Experience: Guiding Novices to Adopt the Gaze Strategies of Experts Expedites the Learning of Technical Laparoscopic Skills. *Surgery (United States)*, *152*(1), 32–40. Retrieved from https://doi.org/10.1016/j.surg.2012.02.002 doi: 10.1016/j.surg.2012.02.002

Volet, S. E. (1991). Modelling and Coaching of Relevant Metacognitive Strategies for Enhancing University Students' Learning. *Learning and Instruction*, *1*(4), 319–336. Retrieved from https://doi.org/10.1016/0959-4752(91)90012-W doi: 10.1016/0959-4752(91)90012-W

Von Mayrhauser, A., & Lang, S. (1999). A Coding Scheme to Support Systematic Analysis of Software Comprehension. *IEEE Transactions on Software Engineering*, *25*(4), 526–540. Retrieved from https://doi.org/10.1109/32.799950 doi: 10.1109/32.799950

von Mayrhauser, A., & Vans, A. M. (1994). *Program Unterstanding: A Survey*. Colorado State Univ.

Von Mayrhauser, A., & Vans, A. M. (1995). Program Understanding: Models and Experiments. In M. C. Yovits & M. Zelkowitz (Eds.), *Advances in Computers* (Vol. 40, pp. 1–38). Elsevier. Retrieved from https://doi.org/10.1016/S0065-2458(08)60543-4 doi: 10.1016/S0065-2458(08)60543-4

Watson, C., & Li, F. W. (2014). Failure Rates in Introductory Programming Revisited. In *Proceedings of the 2014 Innovation and Technology in Computer Science Education Conference (ITiCSE'14)* (pp. 39–44). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2591708.2591749 doi: 10.1145/2591708.2591749

Weinberg, G. M. (1985). *The Psychology of Computer Programming*. USA: John Wiley & Sons, Inc.

Weintrop, D., & Wilensky, U. (2018). How Block-Based, Text-Based, and Hybrid Block/Text Modalities Shape Novice Programming Practices. *International Journal of Child-Computer Interaction*, *17*, 83–92. Retrieved from https://doi.org/10.1016/j.ijcci.2018.04.005 doi: 10.1016/j.ijcci.2018.04.005

Whalley, J., Clear, T., Robbins, P., & Thompson, E. (2011). Salient Elements in Novice Solutions to Code Writing Problems. In *Proceedings of the Thirteenth Australasian Computing Education Conference (ACE'11)* (Vol. 114, pp. 37–46). Australian Computer Society.

Wiedenbeck, S. (1985, 8). Novice/Expert Differences in Programming Skills. *International Journal of Man-Machine Studies*, *23*(4), 383–390. Retrieved from https://doi.org/10.1016/S0020-7373(85)80041-9 doi: 10.1016/S0020-7373(85)80041-9

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM*, *49*(3), 33–35. Retrieved from https://doi.org/10.1145/1118178.1118215 doi: 10.1145/1118178.1118215

Wing, J. M. (2008). Computational Thinking and Thinking About Computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *366*(1881), 3717–3725. doi: 10.1098/rsta.2008.0118

Wing, J. M. (2014). Computational Thinking Benefits Society. *Journal of Computing Sciences in Colleges*, *24*(6), 6–7.

Winslow, L. E. (1996, 9). Programming Pedagogy – A Psychological Overview. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)*, *28*(3), 17–22. Retrieved from https://doi.org/10.1145/234867.234872 doi: 10.1145/234867.234872

Wirth, R. (2000). CRISP-DM: Towards a Standard Process Model for Data Mining. In *Proceedings of the Fourth International Conference on the Practical Application of Knowledge Discovery and Data Mining (PADD'00)* (pp. 29–39).

Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., … Ko, A. J. (2019, 7). A Theory of Instruction for Introductory Programming Skills. *Computer Science Education*, *29*(2-3), 205–253. Retrieved from https://doi.org/10.1080/08993408.2019.1565235 doi: 10.1080/08993408.2019.1565235

Yadin, A. (2011). Reducing the Dropout Rate in an Introductory Programming Course. *ACM Inroads*, *2*(4), 71–76. Retrieved from https://doi.org/10.1145/2038876.2038894 doi: 10.1145/2038876.2038894

Ye, N., & Salvendy, G. (2007). Expert-Novice Knowledge of Computer Programming at Different Levels of Abstraction. *Ergonomics*, *39*(3), 461–481. Retrieved from https://doi.org/10.1080/00140139608964475 doi: 10.1080/00140139608964475

Yenigalla, L., Sinha, V., Sharif, B., & Crosby, M. (2016). How Novices Read Source Code in Introductory Courses on Programming: An Eye-Tracking Experiment. In *Lecture Notes in Computer Science* (Vol. 9744, pp. 120–131). Springer Verlag. Retrieved from https://doi.org/10.1007/978-3-319-39952-2_13 doi: 10.1007/978-3-319-39952-2_13

Youngs, E. A. (1974). Human Errors in Programming. *International Journal of Man-Machine Studies*, *6*(3), 361–376. doi: 10.1016/S0020-7373(74)80027-1

Yusuf, S., Kagdi, H., & Maletic, J. I. (2007). Assessing the Comprehension of UML Class Diagrams via Eye Tracking. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)* (pp. 113–122). Retrieved from https://doi.org/10.1109/ICPC.2007.10 doi: 10.1109/ICPC.2007.10