



TECHNISCHE UNIVERSITÄT
CHEMNITZ

A Neuro-Cognitive Perspective of Program Comprehension

Dissertation
zur Erlangung des akademischen Grades

Dr. rer. nat.

Herr Norman Peitek, M.Sc.
geboren am 14.11.1989 in Magdeburg

Fakultät für Informatik
an der Technischen Universität Chemnitz

Gutachter:

Prof. Dr.-Ing. Janet Siegmund
Prof. Dr.-Ing. Sven Apel
Prof. Westley Weimer, Ph.D.

Tag der Verteidigung:

27. April 2022

<https://nbn-resolving.org/urn:nbn:de:bsz:ch1-qucosa2-790021>

Abstract

Background Software is an integral part of today's world with an outlook of ever-increasing importance. Maintaining all of these software artifacts is a major challenge for software engineering. A future with robust software primarily relies on programmers' ability to understand existing source code, because they spend most of their time on it.

Program comprehension is the cognitive process of understanding source code. Since program comprehension is an internal cognitive process, it is inherently difficult to observe and measure reliably. Decades of research have developed fundamental models of program comprehension, but there still are substantial knowledge gaps in our understanding of program comprehension.

Novel psycho-physiological and neuroimaging measures provide an additional perspective on program comprehension which promise new insights to program comprehension. Recently, these measures have been permeating software engineering research. The measures include eye tracking and physiological sensors, but also neuroimaging measures, such as functional magnetic resonance imaging (fMRI), which allow researchers to more objectively observe cognitive processes.

Aims This dissertation aims to advance software engineering by better understanding program comprehension. We apply and refine the use of psycho-physiological and neuroimaging measures. The goals are twofold:

First, we develop a framework for studying program comprehension with neuroimaging, psycho-physiological, eye tracking, and behavioral methods. For neuroimaging, we focus on functional magnetic resonance imaging (fMRI), as it allows researchers to unravel cognitive processes in high detail. Our framework offers a detailed, multi-modal view on program comprehension that allows us to examine even small effects.

Second, we shed light on the underlying cognitive process of program comprehension by applying our experiment framework. One major focus is to understand experienced programmers' efficient top-down comprehension. We also link programmers' cognition to common code complexity metrics.

Method and Results To fulfill our goals, we conduct a series of empirical studies on program comprehension. In these studies, we use and combine fMRI, psycho-physiological, and eye-tracking measures. Throughout the experiments, we develop and refine a multi-modal experiment framework to shed light onto program comprehension with a neuro-cognitive perspective. We

demonstrate that the framework provides a reliable approach to quantify and to investigate programmers' cognitive processes.

We explore the neuro-cognitive perspective of program comprehension to validate and extend established program-comprehension models. We show that programmers using top-down comprehension require less cognitive effort, but use the same network of brain areas.

We also demonstrate how our developed experiment framework and fMRI as a measure can be used in software engineering to provide objective data in long-standing debates. For example, we show that commonly used, but criticized code complexity metrics indeed only have a limited predictive power on the required cognitive effort to understand source code.

Conclusion In our interdisciplinary research, we show how neuroimaging methods, such as fMRI, in combination with psycho-physiological, eye tracking and behavioral measures, is beneficial to software-engineering research. This dissertation provides a foundation to further investigate the neuro-cognitive perspective to programmers' brains, which is a critical contribution to the future of software engineering.

Zusammenfassung

Hintergrund Software ist ein fester Bestandteil der heutigen Welt mit einer immer wichtiger werdenden Bedeutung. Das moderne Leben ist infolgedessen zunehmend von funktionierender und möglichst fehlerfreier Software abhängig. Deshalb ist die Pflege aller Software-Artefakte eine wichtige und große Herausforderung für das Software-Engineering. Eine Zukunft mit robuster Software hängt in erster Linie von der Fähigkeit ab, den vorhandenen Quellcode zu verstehen, da damit die meiste Zeit verbracht wird.

Programmverständnis ist der kognitive Prozess des Verstehens von Quellcode. Da dieser kognitive Prozess intern abläuft, ist ein zuverlässiges Beobachten und ein genaues Messen mit erheblichen Schwierigkeiten verbunden. Jahrzehntelange Forschung hat zwar grundlegende Modelle des Programmverständnisses entwickelt, aber das Bild von Programmverständnis weist noch immer erhebliche Wissenslücken auf.

Neuartige psychophysiologische und nicht-invasive human-bildgebende Verfahren bieten zusätzliche Perspektiven auf das Programmverständnis, die neue Erkenntnisse versprechen. In den letzten Jahren haben diese Erfassungsmöglichkeiten die Software-Engineering-Forschung durchdrungen. Zu den Messverfahren gehören Eyetracking und physiologische Sensoren, aber auch nicht-invasive Human-Bildgebung, wie die funktionelle Magnetresonanztomographie (fMRT). Diese innovativen Messverfahren ermöglichen es Forschenden, kognitive Prozesse objektiver und genauer zu verfolgen und auszuwerten.

Ziele Diese Dissertation zielt darauf ab, Software-Engineering durch ein besseres Erfassen des Programmverständnisses voranzubringen. Dafür werden psychophysiologische und nicht-invasive human-bildgebende Verfahren angewendet und verfeinert. Es werden zwei Ziele verfolgt:

Zum einen wird ein Framework für Experimente zum Programmverständnis, die mit Human-Bildgebung, Psychophysiologie, Eyetracking und Verhaltensmethoden durchgeführt werden, entwickelt. Bei der Human-Bildgebung erfolgt die Konzentration auf die funktionelle Magnetresonanztomographie (fMRT), da sie kognitive Prozesse mit hoher Detailschärfe entschlüsseln kann. Das entwickelte Framework bietet eine detaillierte, multimodale Sicht auf das Programmverständnis, die es ermöglicht, auch kleine Effekte zu untersuchen.

Zum anderen wird der zugrunde liegende kognitive Prozess des Programmverständnisses durch den Einsatz des aufgestellten Frameworks analysiert. Ein Hauptaugenmerk liegt dabei auf dem Erfassen des effizienten Top-Down-Verstehens von Quellcode. Zusätzlich wird die Kognition beim Programmieren mit gängigen Komplexitätsmetriken von Quellcode verknüpft und im Zusammenhang ausgewertet.

Methodik und Ergebnisse Um die Ziele zu erreichen, werden eine Reihe empirischer Studien zum Programmverständnis durchgeführt. In diesen Studien werden fMRT, Psychophysiologie sowie Eyetracking verwendet und miteinander kombiniert. Während der Experimente erfolgt eine Entwicklung und Verfeinerung eines multimodalen Experimentframework, um das Programmverständnis mit einer neurokognitiven Perspektive zu beleuchten. Es wird dokumentiert, dass das entwickelte Framework einen zuverlässigen Ansatz bietet, um kognitive Prozesse beim Programmieren zu quantifizieren und zu untersuchen.

Weiterhin wird die neurokognitive Perspektive des Programmverständnisses erforscht, um etablierte Programmverständnismodelle zu validieren und zu erweitern. Im Kontext dessen wird belegt, dass das Top-Down-Verständnis das gleiche Netzwerk von Gehirnbereichen aktiviert, aber zu geringerer kognitiver Last führt.

Es wird demonstriert, wie das entwickelte Experimentframework und fMRT als Messverfahren im Software-Engineering verwendet werden können, um in langjährigen Debatten objektive Daten zu bieten. Dabei wird insbesondere gezeigt, weshalb gängige, aber in Frage gestellte Komplexitätsmetriken von Quellcode tatsächlich nur eine begrenzte Vorhersagekraft auf die erforderliche kognitive Last beim Verstehen von Quellcode haben.

Schlussfolgerung In interdisziplinärer Forschung wird nachgewiesen, dass nicht-invasive human-bildgebende Verfahren wie die fMRT, kombiniert mit Psychophysiologie, Eyetracking sowie Verhaltensmethoden für die Software-Engineering-Forschung von erheblichem Vorteil sind. Diese Dissertation bietet eine belastbare Grundlage für die weitere Untersuchung der neurokognitiven Perspektive auf das Gehirn von Programmierern. Damit wird ein entscheidender Beitrag für ein erfolgreiches Software-Engineering geleistet.

Theses

- Thesis 1** A neuro-cognitive perspective of program comprehension is a promising approach to finally close the gap between research and programmers' minds.
- Thesis 2** Our multi-modal experiment framework offers a novel, data-rich approach to examine programmers' cognition in great detail.
- Thesis 3** Efficient top-down comprehension of experienced programmers relies on the same network of brain areas, but shows higher neural efficiency.
- Thesis 4** Code complexity metrics are linked to programmers' cognition only to a limited degree and are often misused in practice.

Acknowledgments

I decided to pursue a PhD to challenge myself. Initially, I did not realize how lucky I was to be accompanied by people who set me up for success. The work presented in this dissertation was as much their accomplishment as it was mine.

First and foremost, my deepest gratitude goes to my family. Without them, none of this would have been possible. Jesse, my beloved wife and future mother of my children, you are the rock upon which my life is built. My parents, Andrea and Steffen, you have always provided me with unwavering support.

A heartfelt thank you goes to Janet Siegmund, who not only immediately trusted me to succeed in this research project, but also steadily guided me through the challenges of academic research. I thank André Brechmann for always supporting me in all of my endeavors and then gently reminding me to focus on the important questions. I thank Sven Apel for sharing invaluable bits of wisdom every time we spoke. I am thankful to Chris Parnin for his inspirational insights throughout the years. Many PhD students fail due to a lack of support from their supervisor—I was fortunate to have a diverse group of brilliant advisors steadily investing time in me and my research.

My sincere thanks go to Westley Weimer, not only for reviewing my dissertation, but also for passionately driving forward the neuro-scientific research in software engineering.

I thank all my co-authors for teaching me something new, all in their own way: Janet Siegmund, Sven Apel, André Brechmann, Chris Parnin, Johannes C. Hofmeister, Christian Kästner, Andrew Begel, Gunter Saake, Thomas Leich, Michael Hanke, Sebastiaan Mathôt, Adina Svenja Wagner, Eduard Ort, Norbert Siegmund, Thorsten Dickhaus, Karsten Tabelow, Jennifer Bauer, and Jörg Stadler.

I appreciate Andreas Fügner, Anke Michalsky, and Jörg Stadler for their technical support during fMRI data acquisition. I am grateful to Anja Bethmann, Susann Deike, and Nicole Angenstein for their patient guidance on properly analyzing fMRI data.

I am grateful for the funding of the German Research Foundation (DFG), which allowed me to work on a fascinating research project.

Finally, I thank all participants who took the time and effort to take part in my fMRI, eye-tracking, and online studies.

Contents

List of Figures	xvii
List of Tables	xxi
List of Listings	xxiii
List of Abbreviations	xxv
1 Introduction	1
1.1 Program Comprehension	2
1.2 Contributions	3
1.3 Outline	4
2 Program Comprehension	7
2.1 Strategies of Program Comprehension	7
2.1.1 Bottom-Up Comprehension	8
2.1.2 Top-Down Comprehension	8
2.1.3 Extensions of Program-Comprehension Strategies	9
2.1.4 Summary	10
2.2 Classical Measures of Program Comprehension	10
2.2.1 Think-Aloud Protocols	11
2.2.2 Interviews	12
2.2.3 Subjective Rating	12
2.2.4 Task Performance	13
2.2.5 Summary	14
2.3 Psycho-Physiological Measures of Program Comprehension	14
2.3.1 Eye Tracking	14
2.3.1.1 Eye-Gaze Measures: Fixations and Saccades	16
2.3.1.2 Advanced Eye-Tracking Measures	17
2.3.1.3 Summary	18
2.3.2 Physiological Measures	18
2.3.2.1 Electrodermal Activity	19
2.3.2.2 Heart Rate and Heart Rate Variability	19
2.3.2.3 Respiration	19
2.3.2.4 Combination of Physiological Measures	19
2.3.2.5 Summary	20
2.4 Neuroimaging Measures	20
2.4.1 Electroencephalography (EEG)	21

2.4.2	Functional Near-Infrared Spectroscopy (fNIRS)	23
2.4.3	Functional Magnetic Resonance Imaging (fMRI)	24
2.4.3.1	Experiment Design	25
2.4.3.2	Control Condition	26
2.4.3.3	Risks and Participant Requirements	27
2.4.3.4	Analysis of fMRI Data	27
2.4.3.5	fMRI Studies in Software Engineering	31
2.4.4	Comparison of Neuroimaging Measures	32
2.5	Chapter Summary	34
3	A Framework for fMRI Studies of Program Comprehension	35
3.1	First fMRI Experiment from Siegmund et al.	36
3.1.1	Experiment Design	36
3.1.2	Limitations of Experiment Design	37
3.2	Simultaneous Measurement with fMRI and Eye Tracking	39
3.2.1	Motivation	39
3.2.2	Research Objectives	41
3.2.3	Experiment Design	43
3.2.3.1	Experimental Conditions and Task Design	44
3.2.3.2	Study Participants	45
3.2.3.3	Experiment Execution	45
3.2.4	Results	46
3.2.4.1	Data Recording	46
3.2.4.2	Data Quality	47
3.2.4.3	Calibration Validation	48
3.2.4.4	AOI Analysis: Overview	48
3.2.4.5	AOI Analysis: Snippets	50
3.2.4.6	AOI Analysis: Participants	51
3.2.4.7	Optimal Line Height	51
3.2.5	Discussion	53
3.2.5.1	Eye-Tracking Optimizations	53
3.2.5.2	Challenges and Benefits	54
3.2.5.3	fMRI Experiment Framework	56
3.2.6	Threats to Validity	56
3.2.7	Related Work	57
3.2.8	Conclusion	57
3.3	Toward Multi-Modal Data Analysis of Program Comprehension	58
3.3.1	Advanced Eye-Tracking Measures during an fMRI Study	58
3.3.1.1	Objectives	58
3.3.1.2	Pupil Dilation	59
3.3.1.3	Blink Rates and Durations	63
3.3.1.4	Conclusion	63
3.3.2	Conjoint Analysis of Multi-Modal Data	64
3.3.2.1	Strategies for Data Analysis	65
3.3.2.2	Envisioned Tool Support	66

3.3.2.3	Conclusion	68
3.4	Tool Support for Multi-Modal Data Exploration	68
3.4.1	Prototype Implementation of <i>CODERSMUSE</i>	70
3.4.1.1	Integrated Modalities	70
3.4.1.2	Features and Challenges	71
3.4.1.3	Implementation	72
3.4.2	Future Work	73
3.4.2.1	Additional Modalities	73
3.4.2.2	Data Annotation and High-Level Patterns	73
3.4.2.3	Data Aggregation	74
3.4.3	Use Cases for <i>CODERSMUSE</i>	74
3.4.4	Related Work	74
3.4.5	Conclusion	75
3.5	Chapter Summary and Future Work	75
3.5.1	Integration of Psycho-Physiological Measures	75
3.5.2	Effect of Task Design	77
3.5.3	Increasing External Validity with Complex Snippets	77
3.5.4	Reduce Task Reflection during Rest Condition	78
3.5.5	Evaluation of Control Conditions	78
4	Neuro-Cognitive Perspective of Program Comprehension	81
4.1	Neural Efficiency of Top-Down Comprehension	81
4.1.1	Experiment Design	83
4.1.1.1	Experimental Conditions	84
4.1.1.2	Designing and Selecting Code Snippets	86
4.1.1.3	Participants of the fMRI Study	87
4.1.1.4	Task Design	88
4.1.1.5	Experiment Execution	88
4.1.1.6	fMRI Data Analysis	89
4.1.2	Results and Discussion	90
4.1.3	Perspectives	94
4.1.3.1	Relation to Theories of Comprehension	94
4.1.3.2	Implications	95
4.1.4	Threats to Validity	96
4.1.4.1	Construct Validity	96
4.1.4.2	Internal Validity	96
4.1.4.3	External Validity	96
4.1.5	Related Work	97
4.1.6	Conclusion and Future Work	97
4.2	Reading Order of Novices and Experienced Programmers	98
4.2.1	Original Study and Replication	98
4.2.1.1	Original Study (Busjahn et al.)	98
4.2.1.2	Replication Study (Peachock et al.)	101
4.2.2	Experiment Design	102
4.2.2.1	Source Code Linearity Metric <i>i</i>	103

4.2.2.2	Material	104
4.2.2.3	Independent Variables	105
4.2.2.4	Dependent Variables	106
4.2.2.5	Task	108
4.2.2.6	Participants	108
4.2.2.7	Experiment Procedure	109
4.2.3	Data Analysis	109
4.2.3.1	Behavioral Data	109
4.2.3.2	Eye-Tracking Data: Preprocessing	110
4.2.3.3	Eye-Tracking Data: Analysis Procedure	110
4.2.4	Results	111
4.2.4.1	Behavioral Data	111
4.2.4.2	Eye-Tracking Data	111
4.2.5	Discussion	113
4.2.5.1	Behavioral Data	113
4.2.5.2	Eye-Tracking Data	114
4.2.6	Threats to Validity	117
4.2.6.1	Construct Validity	117
4.2.6.2	Internal Validity	117
4.2.6.3	External Validity	118
4.2.7	Related Work	118
4.2.8	Conclusion	119
4.3	Brain-Activation Patterns of Novices and Experienced Programmers	119
4.3.1	Literature Review	119
4.3.1.1	Insights from Cognitive Psychology	120
4.3.1.2	Insights from Neuroscience	120
4.3.1.3	Insights from Software Engineering	121
4.3.2	Data Exploration	122
4.3.2.1	Method	123
4.3.2.2	Results	123
4.3.2.3	Discussion	124
4.3.3	Summary	125
4.4	Chapter Summary and Future Work	126
4.4.1	fMRI Study on Programming Experience Levels	126
4.4.2	Experiment Execution	127
4.4.3	Neural Representations of Programming Constructs	128
4.4.4	Neural Representations of Data Types	128
5	Applications of fMRI Research in Software Engineering	131
5.1	Code Complexity Metrics and Program Comprehension	131
5.1.1	Data Exploration	132
5.1.1.1	Method	132
5.1.1.2	Results	133
5.1.2	fMRI Study	134

5.1.3	Experiment Design	135
5.1.3.1	Research Goals	136
5.1.3.2	Pilot Studies	136
5.1.3.3	Experiment Design & Execution	138
5.1.4	Results	140
5.1.4.1	Complexity Metrics and Behavioral data	142
5.1.4.2	Complexity Metrics and fMRI Data	142
5.1.4.3	Subjective Complexity	144
5.1.5	Discussion	145
5.1.5.1	Overarching Research Question	145
5.1.5.2	Hypotheses	148
5.1.5.3	Perspectives	148
5.1.6	Threats to Validity	150
5.1.6.1	Construct Validity	150
5.1.6.2	Internal and External Validity	151
5.1.7	Related Work	151
5.1.8	Conclusion	152
5.2	The Role of Aggregation in Human Studies	152
5.2.1	Literature Analysis	153
5.2.2	Re-Analysis of Behavioral Studies	153
5.2.3	Re-Analysis of fMRI Studies on Program Comprehension	154
5.2.4	Conclusion	155
5.3	fMRI Analysis with Participant-Specific Brain Parcellation	156
5.3.1	Method	156
5.3.2	Example Use Case	156
5.4	Chapter Summary	158
6	Conclusion and Future Work	159
7	Appendix	163
7.1	fMRI Scanner Setting and Analyses Procedures	163
7.2	Programmer Experience Questionnaire of Siegmund et al.	165
7.3	Programmer Experience Questionnaire (Extended for Professionals)	166
7.4	Comprehension Snippets	168
7.5	Informed Consent and Participant Safety Questionnaire	169
	Bibliography	171

List of Figures

1.1	Overview of conducted work, chapters, and selected publications.	3
2.1	Visualization of program comprehension as a connection between source code and a programmer's mental model.	7
2.2	Visualization of various measures to observe program comprehension.	11
2.3	Visualization of a sequence of fixations and saccades of one participant comprehending a code snippet.	15
2.4	Visualization of a heat map, which aggregates the number of fixations on a code snippet.	15
2.5	Visualization of a simplified BOLD response to a stimulus that is presented for 13 seconds.	21
2.6	Photo of 64-channel EEG device in a lab.	22
2.7	Visualization of different frequency bands of EEG	22
2.8	Photo of participant wearing an fNIRS in front of a computer	23
2.9	Participant in our fMRI scanner	24
2.10	Visualization of typical fMRI designs: (a) block design and (b) event-related design.	25
2.11	Visualization of a typical voxel intensity series	28
2.12	Visualization of the brain standardization process with BrainVoyager	29
3.1	Visualization of Siegmund et al.'s block-based fMRI experiment design	36
3.2	BOLD response in BA21 of top-down program comprehension.	38
3.3	Eye-tracking setup with a long-range camera in an fMRI scanner.	41
3.4	Illustration of one (out of five) experiment trials for our study on simultaneous fMRI and eye tracking.	44
3.5	Example code snippet as visible in fMRI scanner	45
3.6	Visualization of Inner , 25-px Extra , and 50-px Extra area-of-interests (AOIs) around the task identifier of Figure 3.5	48
3.7	Fixation count for each AOI and code snippet	49
3.8	Fixation count for each AOI and participant	49
3.9	Distribution of vertical distance from fixations on task identifier	50
3.10	Spatial error of rest condition's fixation cross over time. Standard deviation is shown as shades.	52
3.11	Spatial error on x-axis over time for 8 participants with complete eye-tracking data	53
3.12	Spatial error on y-axis over time for 8 participants with complete eye-tracking data	54
3.13	MRI-compatible video camera of a participant's face	55
3.14	Pupil dilation over time for each condition	59
3.15	Correlation between snippet brightness and measured pupil dilation	60

3.16	Visualization of pupil dilation , gaze position on x-axis and y-axis of participant 1 during the first minute of the experiment.	60
3.17	Visualization of normalized pupil dilation and corrected pupil dilation [Bri+13] of participant 1 during the first minute of the experiment.	61
3.18	Mockup of tool user interface for data exploration	66
3.19	Screen of multi-modal annotation tool ATLAS [MBS12]	67
3.20	Screenshot of <i>CODERSMUSE</i>	70
3.21	Example of psycho-physiological data (i.e., heart data, electrodermal activity, respiration, pupil dilation) during an fMRI study of a single participant. Heart data, respiration, and pupil dilation are in arbitrary units.	76
3.22	BOLD response in BA21 of bottom-up comprehension	78
3.23	Example of a distraction task.	79
3.24	fMRI contrast between program comprehension and d2 attention task.	79
4.1	Overview of designing the snippets and conducting the fMRI study.	83
4.2	Illustration of one (out of twelve) experiment trials for our study on top-down comprehension	89
4.3	Significant brain activation during top-down comprehension	90
4.4	Average BOLD response	90
4.5	Response times in seconds per condition.	92
4.6	Average BOLD response	94
4.7	Source code snippet that elicits top-down comprehension with meaningful identifier names	104
4.8	Source code snippet with obfuscated identifiers that prints a student’s age after a birthday	105
4.9	Visualization of the areas-of-interest (AOIs) of the snippet “Calculation”. The AOIs are wrapping individual code lines and functions, as described by Busjahn et al. [Bus+14].	106
4.10	Scatterplot of programming experience and brain activation strength	123
4.11	Scatterplot of Java knowledge and brain activation strength	124
4.12	fMRI study on simultaneous fMRI and eye tracking (cf. Section 3.2): 2 clusters with stronger activation	129
4.13	fMRI study on top-down comprehension (cf. Section 4.1): No statistically significant differences	130
4.14	fMRI study on code complexity metrics (cf. Section 5.1): 1 cluster with stronger activation for word algorithms	130
5.1	Visualization of significant deactivation during program comprehension in the default mode network.	133
5.2	Scatterplot of code complexity metrics and strength of deactivation of BA 31ant, BA 31post, and BA 32	134
5.3	Illustration of one (out of five) experiment trials for our study on code complexity metrics.	139
5.4	Visualization of the activated brain areas	142

5.5	Visualization of the deactivated brain areas	143
5.6	Relationship of the four complexity metrics with BA44/45	143
5.7	Visualization of fine, intermediate, and coarse aggregation levels	154
5.8	Accuracies of the classifier for the ICSE (left), FSE (center), and ESEM study (right)	155
5.9	Illustration of processing of the fMRI data	157
5.10	Comparative visualization of significantly activated brain areas	157
7.1	Pages 1–4 of the informed consent and participant questionnaire (in German) for our fMRI studies.	170

List of Tables

2.1	Visualization of accuracy and precision in eye tracking.	16
2.2	Comparison of fMRI, fNIRS, and EEG as brain activity measures	33
3.1	Participant demographics for our simultaneous fMRI and eye-tracking study. . .	46
3.2	Summary of all fixation counts and lengths within AOIs around task identifiers. Number in brackets is the overall percentage	48
3.3	Mean perceived brightness, pupil dilation (z-score), blink rate (count/minute), and blink duration (ms) for each experiment condition.	61
3.4	Correlation matrix (in r^2) of observed pupil dilation for each snippet's code complexity and participant behavior.	62
3.5	All data streams supported in <i>CODERSMUSE</i> and their characteristics, typical preprocessing, and measurements.	69
4.1	All code snippets used in this fMRI study on top-down comprehension	84
4.2	Participant demographics for our fMRI study on top-down comprehension. . . .	88
4.3	Overview of gaze-based measures taken from Table 1 from Busjahn et al. [Bus+15]	99
4.4	All snippets used in the study, their metric values, and experimental results . . .	107
4.5	Recruitment universities for our two participant groups with basic programming experience (Novices) and intermediate programming experience (Intermediates).	108
4.6	Demographic data of our participants	108
4.7	Behavioral data separated by programming experience	111
4.8	Overview of the three studies' eye-tracking results	112
5.1	Code snippets used in the study with four selected complexity metric scores and experimental results	137
5.2	Participant demographics for our fMRI study on code complexity metrics.	139
5.3	Kendall's τ and the explained variance (r^2 , in brackets) of the dependent variables	141
5.4	Kendall's τ correlation between brain activation and the unique, differentiating top 20 of the 37 explored complexity metrics	146
5.5	Overview of number of papers that include an empirical study with human partic- ipants, in which tasks were defined in ICSE, ESEC/FSE and EMSE between 2011 and 2018.	153
5.6	Illustration of aggregation per participant, per task, and a combination of both	154
7.1	fMRI scanner configuration for EPI sequence of the three fMRI studies.	164
7.2	Overview of all snippet algorithms used in studies of this dissertation	169

List of Listings

- 3.1 Example code snippet inducing bottom-up comprehension from [Sie+14a] . . . 37
- 4.1 Example code snippet with beacons 82
- 4.2 Example code snippet without beacons 82
- 4.3 Example code snippet without beacons and disrupted layout 82
- 4.4 Part of the obfuscated snippet `Street`, which contains a compiler error . . . 114
- 5.1 Complex code snippet that computes the length of the last word in a string . . 138

List of Abbreviations

BA	Brodmann area
BOLD	Blood oxygenation level dependent
EDA	Electrodermal activity
EEG	Electroencephalography
EPI	Echo planar imaging
ERP	Event-related potential
FDR	False discovery rate
fMRI	Functional magnetic resonance imaging
fNIRS	Functional near-infrared spectroscopy
GLM	General linear model
HR	Heart rate
HRV	Heart rate variability
LOC	Lines of code
MRI	Magnetic resonance imaging
RR	Respiration rate
SD	Standard deviation
T	Tesla

1 Introduction

In today's connected world, software invisibly permeates almost all aspects of our lives. In the future of the Internet of things, billions of devices will be connected to the Internet and run with human-written software. The device spectrum ranges from tiny sensors to enormous facilities, such as nuclear power plants. While software enables the development and increased safety of devices that push humanity forward, it can also threaten human life. For example, a software error caused an overexposure of the radiation therapy system Therac-25, killing at least 5 persons [LT93]. Erroneous software contributed to two crashes of brand-new Boeing 737 MAX airplanes in 2018 and 2019, killing 346 persons [JH19]. These are not outliers, but an industry-wide problem, as analyses of newspapers by Software Fail Watch revealed. They identified 606 recorded software failures, impacting half of the world's population, and causing \$1.7 trillion in damage [Tri17]. Thus, due to the spread and criticality of software, we need software to operate on extremely high standards of correctness, reliability, and quality across countless deployments.

The critical importance of high-quality software was already recognized many decades ago at the NATO SE conference, which was considered to be the founding moment of software engineering [NR68]. Software engineering is the discipline that must push forward toward more correct, reliable, and high-quality software. While software-engineering research has made substantial progress in many technical aspects, we still have failed to unravel countless human aspects of software engineering. In particular, we have an insufficient understanding of the underlying cognitive processes that are part of the software-engineering process [Sie+20]. A major part of the daily work of programmers is *comprehending* software as they spend most of their time with existing source code [Sta84; LVD06; Tia11; MML15]. But this internal cognitive process is inherently difficult to measure, which leads to substantial gaps in knowledge. For example, a widely known phenomenon are "10xer"; programmers who show multiple times the output of their peers despite similar backgrounds. How do their thinking processes differ so that they can achieve such performance? Can we teach such skills? We lack the understanding of programmers' cognitive processes to answer these questions.

In this dissertation, we shed light in the internal cognitive process of program comprehension. We will explore and evaluate a new *neuro-cognitive* perspective of program comprehension. Unlike conventional measurement methods used in the last decades, studying programmers' brains is a promising way to *objectively* measure underlying cognitive processes. In the long-run, this would allow us to systematically tackle fundamental questions regarding program comprehension. In turn, understanding program comprehension leads to a better education and training of programmers. We may also be able to optimize programming languages and implementations to better correspond to how programmers comprehend software.

1.1 Program Comprehension

Program comprehension is the fundamental cognitive process of understanding software code. Understanding how programmers comprehend code is essential, because programmers spend most of their everyday work comprehending existing source code [Tia11] and according to studies, more than tenfold of their time writing new code [MML15]. These are not recent discoveries. Research was aware of the special challenge associated with continuous tasks, such as maintenance, as early as the 1970s [LST78].

Due to its importance, researchers have been studying program comprehension for decades. Past research has resulted in valuable models of program comprehension, such as *top-down* comprehension and *bottom-up* comprehension. However, despite decades of research on program comprehension, we still have an insufficient understanding of the underlying cognitive processes [Sie16]. Many effects and aspects of program comprehension are unclear. In particular, none of the developed program comprehension theories, which we describe in Section 2.1, have been properly validated. Despite the developed and refined theories, there is a notable lack of robust, quantifiable answers to critical practical questions in programming. For example, which implementation approaches (i.e., recursion versus iteration) should be taught first to optimize learning programming concepts?

One challenge of studying program comprehension is that it is a complex cognitive process that is inherently difficult to observe and objectively measure. Conventional measures, such as observing programmer behavior (e.g., response time, response correctness) can identify differences between tasks, participants or participant groups, but it often fails to explain *why* such differences exist. Researchers have to turn to qualitative measures, such as interviews, to hypothesize on explanations for observed phenomena. However, this is also problematic since self-reflecting can be incomplete or inaccurate, especially for complex cognitive tasks [SBS94a; Pik+14].

One promising solution is to explore new ways of measuring program comprehension to shed light on program comprehension from a different perspective. In past years, eye tracking has been at the forefront of studying programmers' visual attention and reading patterns [Sha+20b]. Another approach is to observe programmers' physiological responses to programming tasks, for example, by measuring electrodermal activity as an indicator for their stress level [Bou12]. While eye tracking and physiological responses are objective in their observations, they still fail to explain the cognitive processes of program comprehension.

In this dissertation, we will explore a neuro-cognitive perspective with functional magnetic resonance imaging (fMRI) that allows us to observe programmers' brains. Based on their brain activity, we can identify fundamental cognitive processes of program comprehension and objectively measure them. We will develop a robust framework of observing programmers with neuroimaging, validate decade-old program comprehension models, and apply our research to practical questions of code complexity measures.

1.2 Contributions

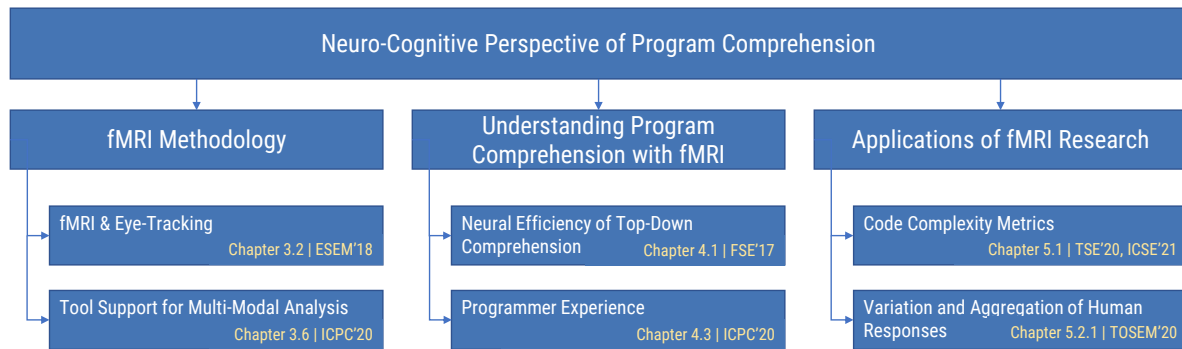


Figure 1.1: Overview of conducted work, chapters, and selected publications.

In Figure 1.1, we provide an overview of our research avenues. In a nutshell, this dissertation makes the following contributions:

Experiment Framework We provide a methodological framework for future, multi-modal fMRI studies of program comprehension and related studies. The framework combines modalities for a thorough view on program comprehension. This comprehensive view on the underlying cognitive processes allows us to study fine-grained effects (e.g., the influence of beacons). Additionally, we outline several future evaluations regarding the importance of task design and suitable contrast conditions. We further share a prototype for multi-modal data exploration that helps to generate new hypotheses.

Validation of Top-Down Comprehension We provide a neuro-cognitive perspective of the model of top-down comprehension. In particular, we find a higher neural efficiency of top-down comprehension in contrast to bottom-up comprehension. We also investigated the impact of code aspects (e.g., beacons, layout).

Code Complexity Metrics and Cognition We collected objective evidence that widely used code complexity metrics exhibit a limited link to programmers' cognitive processes. Our data show that the search for one all-encompassing metric may be misguided. Rather, a combination of simple metrics, in particular measuring data flow, are the most promising. Our experiment framework and this dissertation provide a template for future research.

Analysis of Implementation Details We outline several ongoing projects that further deepen our understanding of program comprehension. Specifically, we present current studies on the neuronal representations of programming constructs (e.g., iteration versus recursion) and data types (e.g., numbers versus words).

Re-Analysis of fMRI Data for Methodological Insights We demonstrate further applications of fMRI research in a broader software-engineering context. First, we showcase that aggregating measures of cognitive processes has a strong influence on the validity and reliability of empirical experiments and must be carefully considered. Second, we illustrate a compelling approach to analyze fMRI data that uses participant-specific anatomies.

Open Science All research presented in this dissertation follows the open science concept. We share replication packages and insights for each study¹ to pave the way for other researchers also to conduct neuroimaging studies of program comprehension. We also created a permanent archive on zenodo to ensure long-term availability.²

1.3 Outline

Background In Chapter 2, we present the state of the art on models of program-comprehension strategies. We introduce which measurement methods researchers are using to observe program comprehension. We also provide an overview of how neuroimaging, eye tracking, and conventional measurements methods provide different perspectives on programmers' cognitive processes. In addition, we provide a close look at eye tracking and fMRI as measurement methods.

Methods and Results This dissertation's work is split into three parts (cf. Figure 1.1). First, Chapter 3 dives into our developed experiment framework, which shows how to conduct multi-modal fMRI experiments. We also present *CODERSMUSE*, a tool for multi-modal data exploration specific to the needs of software-engineering research. Next, Chapter 4 investigates the neuro-cognitive perspective of program comprehension in more detail. We present multiple studies on aspects of top-down comprehension as well as programmer expertise. Last, Chapter 5 shows three practical applications of our framework: An objective view on the relationship between code complexity metrics and programmers' cognition, a re-analysis investigating various aggregation levels of human responses, and an alternative fMRI analysis based on participant-specific anatomies.

¹<https://github.com/brains-on-code/>

²<https://zenodo.org/record/5625142>

Conclusion and Future Work In Chapter 6, we take a step back and discuss our research on a higher abstraction level and provide insightful suggestions on how our work contributes to the overarching questions posted in the introduction. We also conclude this dissertation and outline future work.

2 Program Comprehension

In this chapter, we introduce the necessary background information that this dissertation is built upon. In the following section, we introduce program-comprehension strategies. In Section 2.2, we describe how past research measured program comprehension. In Section 2.3, we take a closer look at more recently used measures, such as eye tracking and functional magnetic resonance imaging.

2.1 Strategies of Program Comprehension

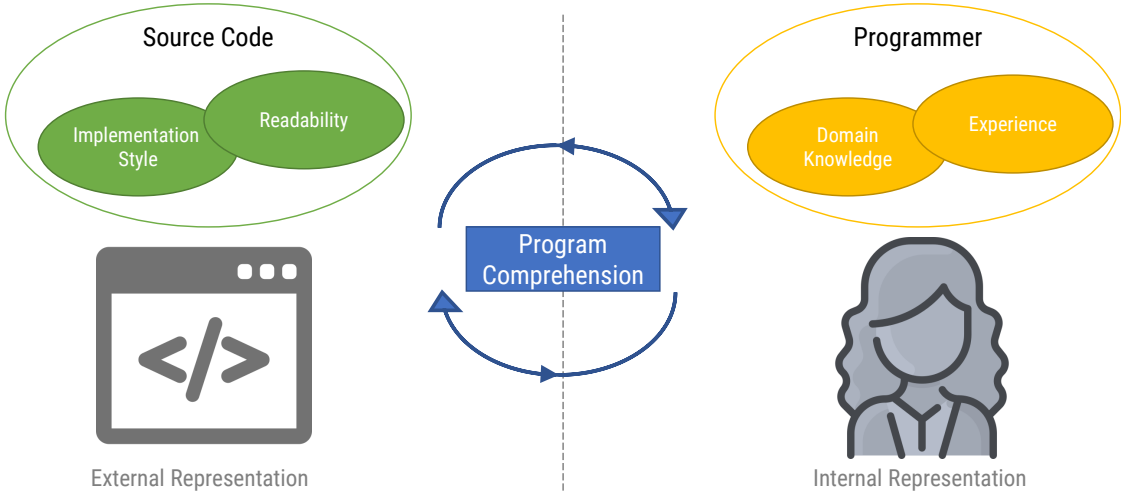


Figure 2.1: Visualization of program comprehension as a connection between source code (external representation) and a programmer’s mental model (internal representation).

Program comprehension is the cognitive process of understanding source code. In essence, it describes the transition in which a programmer understands an existing implementation model formed in source code and constructs an analogous mental model [Ext02], which we visualize in Figure 2.1. Its execution depends on numerous factors in the two realms:

1. Source code. The source code itself heavily influences how programmers understand it. Code readability as well as understandability are key factors.
2. Programmer. Their experience and domain knowledge play essential roles in the cognitive processes of program comprehension.

Both factors influence how a programmer can assimilate source code into a mental model. Researchers have theorized multiple strategies of program comprehension, which we introduce next.

2.1.1 Bottom-Up Comprehension

Bottom-up comprehension describes the most basic strategy of comprehending source code [Pen87]: Programmers read code line by line, understand and extract the meaning of each statement, and then group the semantic information of each individual statement to a larger entity (i.e., *chunking* [SM79]). This recursive process is repeated until programmers achieve a complete understanding of a source code [Pen87]. Since a programmer has to read every single statement of a source code unit, individually comprehend it, and then assimilate all statements until a final, abstract understanding is achieved, this process is slow and tedious. Thus, bottom-up comprehension requires high cognitive effort of programmers [Sie+17].

More recent work further detailed bottom-up comprehension. Hosnieh and Haga investigated an underlying cognitive process of slicing during bottom-up comprehension. They described how programmers decompose source code into their atomic parts, which can be a function, group of statements, individual line or a single variable [HH17]. Similarly, Shargabi et al. described multiple abstraction levels that programmers work up to achieve an overall understanding [Sha+15b].

2.1.2 Top-Down Comprehension

Top-down comprehension, unlike bottom-up comprehension, is a fast and efficient process [Bro78]. Programmers use prior experience and domain knowledge to quickly gain an understanding of a source-code snippet's intent. Programmers look for *beacons* (i.e., "sets of features that typically indicate the occurrence of certain structures or operations in the code" [Bro83]) to build an initial hypothesis of a source-code snippet's intent. Then, programmers validate and refine their hypothesis by quickly jumping between the snippets' points of interest (e.g., input variables, loop structures, return variable). If the implementation confirms their hypothesis, programmers have quickly and efficiently understood a snippets' intent. If an unexpected implementation contradicts their initial hypothesis, programmers have to fall back to bottom-up

comprehension. Top-down comprehension requires lower cognitive effort than bottom-up comprehension, since the process starts with an initial hypothesis and statements do not need to be understood separately [Sie+17]. Thus, programmers use top-down comprehension whenever possible. Due to their lack of experience, novices often cannot employ top-down comprehension and tend to use bottom-up comprehension [SE84].

Research has identified several drivers across the two realms of top-down comprehension. Wiedenbeck showed in several experiments how critical beacons in source code are for the initial phase of program comprehension [Wie86; WS89]. They also showed that inappropriate beacons mislead programmers' comprehension [Wie91]. The code's visual structure, that is the layout with sensible indentation, aids programmers in building hypotheses [Mia+83]. Similarly, *plans* (i.e., "program fragments that represent stereotypic action sequences in programming" [SE84]) aid programmers by identifying programming structures based on their typical natures, such as a sort algorithm typically consists of a loop structure with a swap statement [JS85; SE84]. Recently, Duran, Sorva, and Leite viewed Soloway's plan structure from a cognitive load theory standpoint and developed metrics that reason about a program's cognitive complexity [DSL18]. Pennington described how priming also could induce top-down comprehension [Pen87].

For the realm of the programmer, Shaft and Vessey demonstrated how domain knowledge plays a key role in program comprehension [SV95]. Experienced programmers may read source code like a beginner if they are not familiar with the domain [SV95; KR91]. Belmonte evolved previously described abstraction layers of mental models into three specific layers with increasing finer granularity of the business layer (*why?*), the mapping layer (*what?*), and the implementation layer (*how?*) [BDA14]. Nosal further details the abstraction layers of the situation model and the program model [NP15]. Benomar et al. extended it further by separating program comprehension of source code and comprehension of software evolution over time [BSP15]. All of these further describe when and how programmers are able to employ top-down comprehension.

In a nutshell, programmers preferentially use top-down comprehension. However, several factors can cause programmers to fall back for short moments or generally to bottom-up comprehension. In particular, if the programmer's expectations are violated, for example by missing indentation or disrupted plans, top-down comprehension fails and programmers may fall back to tedious bottom-up comprehension.

2.1.3 Extensions of Program-Comprehension Strategies

In addition to the main classifications of bottom-up and top-down comprehension, numerous refinements extend the two basic strategies.

Letovsky described a *hybrid strategy* that consists of a programmer's knowledge base, the current mental model of the source code, and an assimilation process to map from the source code to the mental model. A programmer applies—depending on the current inquiry and knowledge—both strategies, top-down and bottom-up comprehension, during the assimilation process [Let87].

Littman et al. suggested a further differentiation for individual programmers into an as-needed comprehension versus a systematic comprehension. They observed that programmers who only comprehend as much as necessary (“as-needed strategy”) more often could not correctly modify source code. Programmers who systematically understood source code were more successful [Lit+87]. Roehm et al. observed professional programmers across different companies and tasks. They could not clearly identify an as-needed or systematic comprehension, but found a recurring, structured comprehension strategy. The employed comprehension strategy varied based on task, programmer, and prior knowledge [Roe+12].

Mayrhauser and Vans [MV95] assimilated previous research into one integrated strategy of program comprehension: The program model (source code), plan model (more abstract view, in which beacons indicate the role of an element), and the situation model (most abstract representation of the program, acquired knowledge through bottom-up comprehension of the program model or top-down of the plan model). Mayrhauser and Vans further described that programmers work on multiple abstraction levels and apply a different strategy depending on the current abstraction level [VMV96].

Follow-up research to Mayrhauser’s and Vans’ model extended it by applying learning theories to the comprehension process. For example, Rajlich and Wilde viewed program comprehension from the viewpoint of constructivism theories of learning and separated the comprehension process into assimilation and adaption concepts [RW02].

2.1.4 Summary

From a high-level perspective, research on program comprehension has made substantial steps to an overall theory. Over the decades, research has identified many factors that aid or hinder program comprehension. Research has also established how programmers’ experience and domain knowledge play a key role during the comprehension process.

In the next section, we describe how researchers empirically measured program comprehension in the past and present.

2.2 Classical Measures of Program Comprehension

Measuring program comprehension has been the focus of researchers for almost seven decades. The perspective with regards to program comprehension has changed over time. Harth and Dugerdil classified the research on program comprehension into a classical period (until 1995), optimistic period (1995 to 2015), and pragmatic period (after 2015) [HD17]. Overall, there have been numerous measures of how programmers build up knowledge from source code [HD17; Sto05], which we introduce in the following subsections.

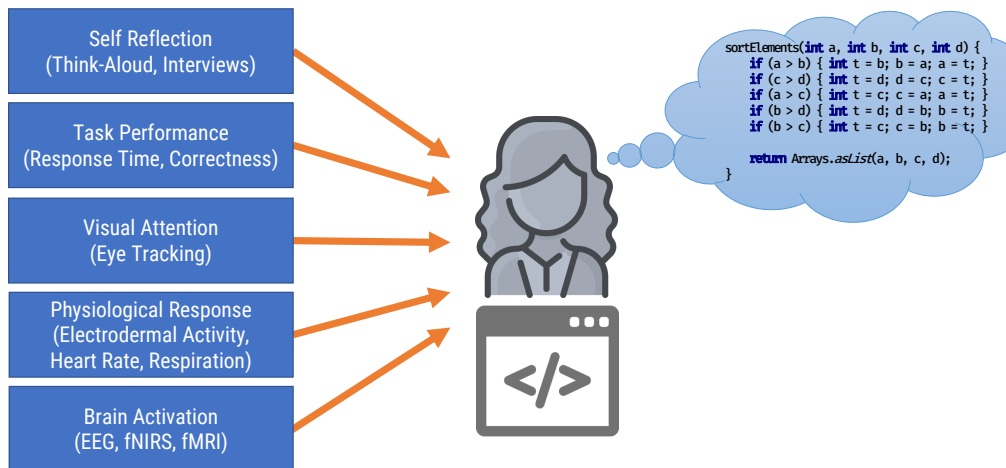


Figure 2.2: Visualization of various measures to observe program comprehension.

Program comprehension is a multi-faceted phenomenon [DR00], so that researchers have used a plethora of operational definitions. For example, typical operationalizations of program comprehension are code comprehension (e.g., [Lee+16; Sie+14a]), debugging (e.g., [Dur+16; Cas+19]), and code review (e.g., [FSW17; Hua+20]). In empirical studies, researchers have many options to measure program comprehension as visualized in Figure 2.2. In this and the following sections, we provide an overview of the conventional methods, before diving into psycho-physiological measures of program comprehension.

2.2.1 Think-Aloud Protocols

In the early days of research on program comprehension, researchers used conventional research methods, such as behavior, think-aloud protocols, and comprehension summaries, to observe programmers. These methods were used to develop the theories on comprehension strategies presented in the previous section.

Think-aloud protocols are a widely used approach in cognitive science to understand participants' thought processes by asking them to self-reflect and verbalize their thoughts aloud during a given task. The collected data is often recorded, transcribed, and qualitatively analyzed, for example, with coding schemes [ES84; SBS94a]. The data provide insights into the thought process and help to shed light on the underlying cognitive processes.

Think-aloud protocols have found their way into program-comprehension research (e.g., Pennington [Pen87], or Shaft and Vessey [SV95]). More recently, LaToza and Myers used in part a think-aloud protocol to observe programmers asking reachability questions, which are common during debugging [LM10]. Gopstein et al. empirically identified small patterns of source

code that consistently lead to programmer confusion and named the patterns *atoms of confusion* [Gop+17]. In a follow-up study, they used a think-aloud protocol to understand how atoms of confusion affect program comprehension [Gop+20].

Think-aloud protocols help to indirectly access the thought processes of programmers. However, think-aloud protocols are time-consuming and can be imprecise or incomplete due to participants filtering their thoughts before verbalization [SBS94a]. While thinking aloud generally does not substantially interfere with a task [ES84], it does increase mental load [Pik+14]. This is especially critical for demanding cognitive processes, such as programming, in which the additional effort for thinking aloud can mentally overload participants, and entirely change their strategy.

2.2.2 Interviews

In the same vein as think-aloud protocols, interviews are a classic tool from psychology. The aim of using interviews is to capture and understand the internal thought processes of participants [FF94]. However, unlike think-aloud protocols, the verbalizations of thoughts happen independently or *after* a given task. Interviews allow follow-up questions to clear up uncertainties.

In program-comprehension research, interviews are commonly used in combination with quantitative approaches. For example, Xia et al. observed the integrated development environment (IDE) interactions of 78 programmers during their workday to understand the role of program comprehension in everyday work. They found that programmers spend a lot of their time with program comprehension, especially for less experienced programmers. Xia et al. conducted follow-up interviews with 10 programmers to confirm their quantitative data, such as extensive use of web browsers to search for the function of unfamiliar code [Xia+17]. Similarly, Roehm observed the actions of 28 programmers with a think-aloud protocol for 45 minutes and then discussed the observations in detail in a subsequent interview. They found that programmers in practice focus on solving the given task rather than a comprehensive understanding [Roe+12].

While research interviews as a method are simple, correctly conducting them is challenging [FF94]. The researcher has to be careful not to introduce their own bias—during the interview and when interpreting the qualitative data [DBC06].

2.2.3 Subjective Rating

Subjective rating asks participants for their perceived comprehension. They are typically measured with questions on a Likert scale (e.g., “How well did you understand the presented code snippet?” with five answering options: “not at all”, “a little”, “about half”, “mostly understood”, “fully understood”, [Lik32]) or a semantic differential scale (e.g., “How difficult is the presented code snippet?” with five answering options: “simple”, “somewhat simple”, “medium”, “somewhat complex”, “complex”) [OST57]). Subjective rating provides quantitative data into a participant’s perception, but is limited to the asked question.

Several program-comprehension studies used subjective ratings as a measure. For example, Miara et al. studied the effect of different indentation depths on a programmer's perception with subjective rating and found that an indentation of 2 to 4 spaces is ideal for comprehending code [Mia+83]. Subjective rating is also used to complement objective measures such as Kosti et al. correlating EEG data of program comprehension with subjectively rated difficulty [Kos+18].

2.2.4 Task Performance

Unlike the subjective measures presented in the previous paragraphs, observing programmers' task performance allows for an objective and quantitative perspective. Typically, researchers use one or both of two measures: response time and correctness [DR00]. Measuring task performance is a coarse proxy of participants' understanding of a topic. Participants who are familiar with a topic will likely be able to effectively and efficiently solve a task in this area. However, participants who do not have the necessary knowledge or skill, will take much longer (response time) or fail to correctly solve the task (correctness). This way, researchers collect an approximate measure of cognitive processes with task-performance measures.

Task performance is one of the most straightforward measures to conduct with large groups. They can be observed with individuals, in a lab with many participants at the same time, or even online. Thus, it is less demanding to conduct an experiment with a large number of participants. This makes it easier to reach minimum participation thresholds to expect significant effects.

Response Time The time a participant takes to complete a task, the *response time*, indicates how long they need to think through a task (sometimes referred to as *efficiency*). In addition to measuring differences between participants, researchers can use this to differentiate tasks. For example, if a participant is presented with two tasks similar in nature, but varying complexity (e.g., compute the faculty of 3 versus compute the faculty of 6). The basic assumption is that the response time is also a proxy of individually perceived task difficulty: easy tasks can be solved quickly, while cognitively challenging problems take longer.

While measuring response time provides an objective indicator of task difficulty, it fails to answer *why* questions. Why does it take participant A longer than participant B? Why is task A more difficult than task B? Response time is useful in identifying differences in behavior between individual participants or groups, but typically requires follow-up studies with different measures to understand a phenomenon fully.

Correctness *Response correctness* measures how often a participant correctly finishes a task. Response correctness allows researchers to measure whether participants were able to correctly solve a given task. Especially for cognitively demanding tasks such as program comprehension, response correctness is useful to identify the limits of participants' understanding of a subject. If a participant (regularly) fails to solve a task correctly, their thought processes are likely flawed.

Researchers often use this effect to objectively find robust differences between participants or between tasks (e.g., comprehending a source code in C# versus source code in Haskell.)

Response correctness is typically independent of response time unless the researchers set a short time limit. Similarly to response time, response correctness does not explain *why* participants fail to solve a given task.

There have been an abundance of program-comprehension studies relying (in part) on task performance as measure. For example, Soloway and Ehrlich presented a fill-in-the-blank code task to novice and expert programmers. Task performance allowed them to (a) find significant differences between their novice and expert group and (b) identify that violated programming plans reduce performance, especially for experts [SE84].

2.2.5 Summary

In a nutshell, all of the presented methods are essential to research on programmers. As visualized in Figure 2.1, all of these methods provide a different perspective to an observed phenomenon. Behavioral and self-reflective methods, which were mostly used in past research, can identify possible strategies of program comprehension. However, since they only provide limited objective insights they cannot capture the entirety of the complex cognitive processes involved in program comprehension. For example, the cognitive effort is subjective and difficult to assess with conventional methods. Thus, in recent years, there has been a push towards using more psycho-physiological and neuroimaging measures, which we introduce in the following sections.

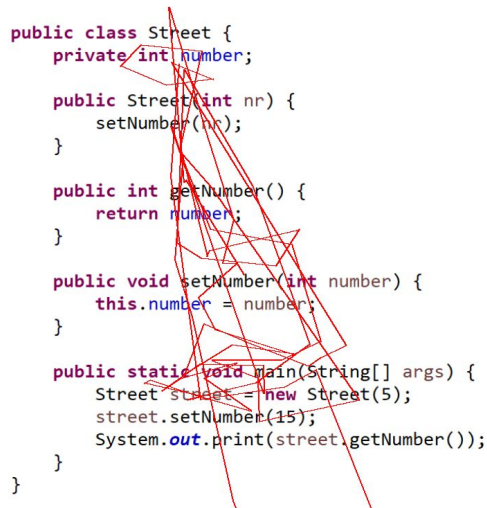
2.3 Psycho-Physiological Measures of Program Comprehension

Psycho-physiological measures objectively capture participants' physiological responses to a given task that induces cognitive processes [Col89]. For example, a programmer thinking through a code snippet containing a complex algorithm may show measurable physiological signals of stress. The human body offers multiple "windows" into its psycho-physiological state, which have been used for research. In the next subsections, we introduce the measures most relevant to program-comprehension research.

2.3.1 Eye Tracking

One of the first measures that achieved broad usage in program-comprehension research was eye tracking. Eye tracking records participants' visual attention by following their eye movements and eye behavior [Ray78; Hol+11; Duc17]. Primitive eye-trackers have been in use since the early 1900s [Sha+20b]. Modern, camera-based eye-trackers provide eye-gaze data with high temporal resolution and accurate spatial resolution. Researchers have used eye tracking to understand

human visual attention and eye-movement behavior in many science areas. For example, reading natural text has been studied with eye tracking for decades providing detailed insights into reading patterns [Ray98].



```

public class Street {
    private int number;

    public Street(int nr) {
        setNumber(nr);
    }

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public static void main(String[] args) {
        Street street = new Street(5);
        street.setNumber(15);
        System.out.print(street.getNumber());
    }
}

```

Figure 2.3: Visualization of a sequence of fixations and saccades of one participant comprehending a code snippet.

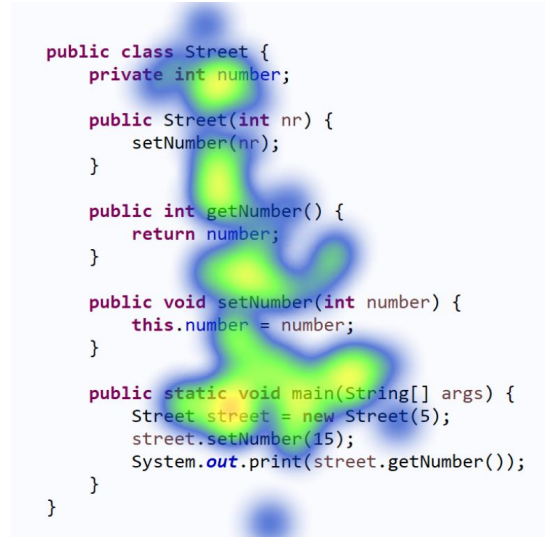


Figure 2.4: Visualization of a heat map, which aggregates the number of fixations on a code snippet.

Technical Background The raw eye-gaze data typically contains a sequence of x,y coordinates of the used screen. Most eye-trackers require a short setup which includes a calibration and validation process.

An event detection algorithm is typically applied to the raw eye-gaze, which separates and assigns the eye-gaze into (typically) two events of *fixation* and *saccade*. A fixation is a stable spatial eye-gaze, which typically lasts for 100-300 ms. When a participant fixates on a specific visual point, they can cognitively process the visual input. A saccade describes a continuous moving eye-gaze. During a saccade, a participant is unable to process visual input in-depth [Hol+11].

The sequence of saccades and fixations can be concatenated to a *scan path*, which we visualize in Figure 2.3. An alternative visualization of fixations is a *heat map*, which we show in Figure 2.4. In addition to qualitative analysis of the visualized data, saccades and fixations can be quantitatively analyzed regarding many metrics, such as the number of fixations, average saccade length, and the number of visits to a specific area of interest (AOI) [Sha+20b]. Finally, there are many algorithms that apply further aggregation. For example, the *K coefficient* classifies a sequence of saccades and fixations into focal and ambient visual attention [Kre+16]. The latter has been shown to be especially important to experts, including programming experts which, in contrast to novices, can use longer saccades because of their larger extrafoveal vision [OB17].

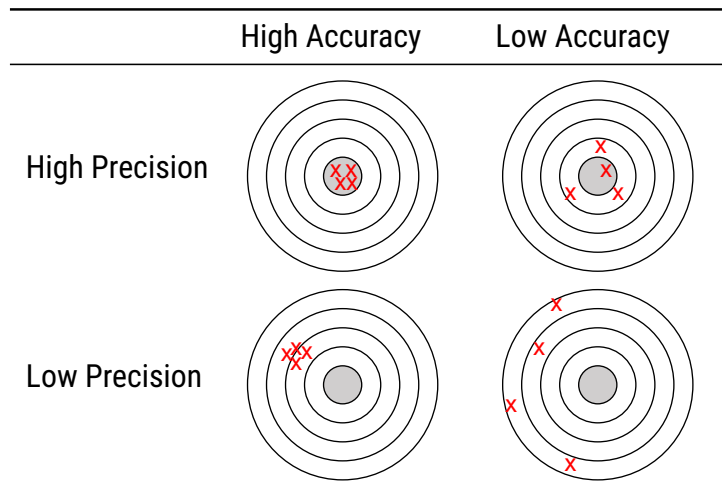


Table 2.1: Visualization of accuracy and precision in eye tracking.

Two concepts are related to data quality: accuracy and precision. *Accuracy* describes the difference between the actual and measured eye-gaze, which threatens results if data is misinterpreted. If it is a systematic error, often named *offset*, it can be corrected by hand [Coh13]. There are also automated corrections in development that would remove researcher bias [PS16]. In addition to low accuracy, there can be low *precision*. Precision describes how consistent the difference between the actual and measured eye-gaze is. We visualize accuracy and precision in Table 2.1. Finally, *drift* describes how accuracy worsens throughout the experiment, for example, as a consequence of changes in eye physiology [Sha+20b].

Use in Software Engineering All eye-tracking measures have massive potential for software-engineering research, as they allow unintrusive, but objective measurement of program comprehension. Sharafi, Soh, and Guéhéneuc’s systematic literature review and Obaidallah, Al Haek, and Cheng’s survey provided an in-depth overview of all eye-tracking studies in software engineering [SSG15; OAH18]. Sharafi et al. provided a practical guide to conducting eye-tracking studies [Sha+20b]. Recent tool development focused on seamlessly integrating eye tracking in more complex and realistic scenarios, such as programmers working in an integrated development environment (IDE) [Gua+18; Sha+15a]. In the next subsection, we introduce a few key eye-tracking studies along with the various eye-tracking measures.

2.3.1.1 Eye-Gaze Measures: Fixations and Saccades

The majority of eye-tracking studies in software engineering focuses on eye-gaze and eye-movement measures (i.e., AOs, fixations, saccades). One of the first to use eye tracking in software engineering were Crosby and Stelovsky, who found differences between the eye-gaze patterns of programmers with different experience levels [CS90]. In 2006, Bednarik and Tukiainen called for using eye tracking in program-comprehension studies. They demonstrated with an

experiment that it is possible to observe programmers with an eye-tracker [BT06]. Sharif and Maletic replicated a behavioral study investigating variable naming styles with eye tracking. Their replication results indicate that variable naming style affects program comprehension in that programmers are able to read under_score style faster than camelCase style [Bin+09b; SM10]. Previous research suggested that the *linearity of the reading order* could be an indicator of how efficiently programmers comprehend source code [Bus+15]. Busjahn et al.'s study described several eye-gaze measures to gauge programmers' linearity of reading order. Their experiment suggests that programmers read source code less linear than natural text. Furthermore, expert programmers read source code less linearly than novice programmers [Bus+15]. Ikehara and Crosby showed that eye movement data could predict task difficulty in a programming context [IC05].

2.3.1.2 Advanced Eye-Tracking Measures

In addition to observing a participant's eye-gaze stream, some advanced eye-tracking measures are established in cognitive psychology, but not widely used in program-comprehension research yet. Nevertheless, they may provide useful for observing programmers and therefore we introduce them next.

Pupil Dilation One commonly measured property is *pupil dilation*, which is a task-evoked pupillary response. Pupil dilation has been shown to directly reflect cognitive load with tasks involving working memory, reasoning, and reading [BLW00]. For example, Beatty and Kahneman's early work showed that an increase in the number of digits to be remembered correlates positively with pupil dilation [BK66]. Similarly, Hess and Polt demonstrated that pupil dilation correlates with the difficulty of mathematical calculations [HP64]. Overall, research has established that pupil dilation can be an accurate measure of mental states [LSG12; HF14].

While pupil dilation is straightforward to observe, it is affected by many external factors, especially light. Thus, extensive preprocessing is necessary in addition to stable environmental conditions throughout the experiment [KSS19].

Pupil dilation has intrigued some researchers as a measure in program-comprehension research. Nolan et al. proposed a study in which novices learned to program. They planned to measure cognitive load with remote eye tracking [NMB15]. In the same vein, Ford et al. proposed to use a variety of eye-tracking metrics (i.e., pupil dilation, saccades, blink rates) to identify mental states during remote interviews of programmers. This would allow interviewers to only interrupt during light thinking phases [FBP15]. Similarly, Behroozi et al. used pupil dilation as a proxy of cognitive load during programming interviews at the whiteboard [Beh+20]. Recently, Ioannou et al. developed an IDE extension that visualizes areas of code, which induced large pupil dilation and thus higher cognitive load in other programmers [Io+].

Blink Rates and Duration Another eye-tracking measure is spontaneous *blink rates*, which correlate with “levels of dopamine in the central nervous system, and can reveal processes underlying learning and goal-directed behavior” [Eck+17]. Blink rates are determined at two levels: The resting baseline and task-evoked blink rate. A higher individual’s blink-rate baseline is related to “better cognitive flexibility but worse maintenance” [Eck+17]. Some research suggests that “higher blink rate at baseline is related with lower distractibility on tasks that place high demands on working memory” [Eck+17].

Using blink rates as a research measure is challenging due to the many factors that can influence it. Blink rates increase with fatigue, which limits experiment length [SBS94b]. Environmental factors, such as air humidity or room temperature, can also affect the observed blink rate [Dou01].

Ford et al. suggested that blink rates are an interesting measure in their proposal for observing remote technical interviews [FBP15]. However, in follow-up work by Behroozi et al., blink rates were not a significant measure to distinguish mental states. Nevertheless, *blink durations* showed significant differences [Beh+18]. While there has been no study directly related to program comprehension, blink duration reliably increases with a participant’s cognitive workload [VG98; Ben+11; Che+11], and were therefore also consider it as a potential measure for our framework.

2.3.1.3 Summary

Eye tracking has been a valuable tool for cognitive psychologists to understand visual attention for decades. It has made its way into software-engineering research. Most researchers use basic eye-gaze measures, such as fixation count in AOs, to understand how programmers are reading code.

The advantage of eye tracking in software engineering is its universal applicability. Researchers can use eye tracking from a controlled lab environment to realistic work scenarios.

While cognitive psychologists have used some advanced measures to better understand cognitive processes using eye tracking, they are not as established in program-comprehension research. This may be due to the complexity of programming as a task. Thus, researchers have been exploring psycho-physiological measures as another alternative to understand participant mental state via its body behavior.

2.3.2 Physiological Measures

Physiological measures comprise measures based on a participant’s physiological responses to a given task, which is designed to induce specific cognitive processes [Col+94]. There are several physiological measures at which we take a closer look in the following paragraphs.

2.3.2.1 Electrodermal Activity

Electrodermal activity (EDA) describes the characteristic of skin to conduct electricity. This characteristic of a participant's skin can change depending on the experienced stress and arousal and can be used to measure participants' cognitive load [Bou12]. The physiological response that results in a measurable change of electrodermal activity takes a few seconds. This delay needs to be taken into account when analyzing electrodermal-activity data.

2.3.2.2 Heart Rate and Heart Rate Variability

There are two commonly used heart-rate measures. *Heart rate* (HR) describes the frequency of heartbeats per minute [And13]. *Heart-rate variability* (HRV) is the variation of the interval between two consecutive heartbeats [And13].

There is plenty of evidence that heart rate and heart-rate variability can be used to estimate cognitive load [VG98; FVT05] and therefore has also been used to observe programmers [MF16; Cou+19b]. For example, Müller and Fritz classified heart rate and heart-rate variability among with other psycho-physiological sensors of ten professional programmers to predict future code quality issues [MF16].

2.3.2.3 Respiration

The most robust respiration measures that have been linked to cognitive load are the *respiratory rate* (RR) and *respiration depth* in which higher respiratory activity indicates higher cognitive load [BRW94; Gra+16]. However, respiration has not been extensively explored in software-engineering research.

2.3.2.4 Combination of Physiological Measures

In software-engineering research, the described physiological measures often were not used in isolation, but in combination. For example, Fritz et al. used a combination of eye tracking, electrodermal activity, and EEG to observe programmers during various programming tasks. They were able to train a classifier based on the three measures that predicted the experienced subjective difficulty [Fri+14]. In a similar vein, Couceiro et al. built a framework to observe programmers using eye tracking, HRV and electrodermal activity while they worked on tasks [Cou+19b]. In a follow-up study, they combined HRV and pupil dilation to annotate potential problematic lines of code [Cou+19a].

However, Züger et al. reported that simple computer interaction data (i.e., keyboard and mouse interaction, active window) may be more accurate than physiological sensors, but also found that a combination of both provides the best results [Züg+18].

Besides predicting task difficulty and code quality, physiological measures have been used to investigate programmers' emotional state. Girardi et al. showed that a combination of physiological measures could identify emotional states without individual training [GLN17]. Müller and Fritz used an approach which combined electrodermal activity and EEG to detect programmers' emotional state and the resulting productivity. They found that it is possible to use physiological measures to fairly reliably identify when a programmer feels stuck during their work [MF15].

2.3.2.5 Summary

Physiological measures show promise in real-world applications and lab research. In the real world, they have been useful in providing real-time feedback during programmers' regular workdays. As described above, they may warn programmers before writing faulty code [Fri+14; MF15; Züg+18; Cou+19a]. While the groundbreaking studies successfully leveraged physiological measures, there are also many challenges ahead with noisy data from physiological sensors in addition to privacy concerns during everyday work [FF20].

In lab research, physiological measures enable researchers to measure an objective proxy of a programmer's cognitive state and provide more insight than basic behavioral measures. Physiology is closer to measuring programmers' cognitive processes, but they are still insufficient in identifying and explaining program comprehension's underlying cognitive processes. To better understand these, we need neuroimaging measures, which we explain in the next section.

2.4 Neuroimaging Measures

Neuroimaging allows researchers to measure the brain, such as determining a participant's individual anatomical brain structure. *Functional* neuroimaging measures the brain's neural activity, typically to a presented task. Functional neuroimaging relies on measuring the brain's electrical signals or underlying biological processes, such as the *blood oxygenation level dependent* (BOLD) response.

The BOLD effect is based on the following principle: Oxygenated and deoxygenated blood have different physical characteristics. On the one hand, oxygenated and deoxygenated blood reflects light differently back to the light source. On the other hand, oxygenated and deoxygenated blood also behaves differently when exposed to magnetic fields. Neuroimaging measures use these characteristics to distinguish different levels of brain activation. The underlying BOLD effect is based on the haemodynamic response to neural activity, which means the acquired brain-activation data follow exposure to the stimuli onset by several seconds [Cha+93]. We visualize a typical BOLD response in Figure 2.5. A statistical analysis of the data needs to account for such activation delay.



Figure 2.5: Visualization of a simplified BOLD response to a stimulus that is presented for 13 seconds.

There is substantial variability among BOLD responses. In particular, the response characteristics can be different between participants. Within one participant, the BOLD response is reasonably consistent even across multiple sessions [AZD98].

Several neuroimaging measures have explored for measuring program comprehension. We describe the three major measures next.

2.4.1 Electroencephalography (EEG)

Electroencephalography (EEG) is a neuroimaging measure to detect brain activity via electrical sensors typically placed on the participant's scalp. EEG was discovered almost 100 years ago in 1924 [Ber29]. EEG receives the weak electrical potential (5–100 μ V) of the simultaneous activity of large groups of neurons in the brain [Phi+97]. We show an EEG device in Figure 2.6.

Background EEG can be used to capture brain activity related to an event (e.g., presentation of a stimulus such as source code) or spontaneous brain activity that occurs without a presented stimulus. *Event-related potentials* (ERP) describe the change in activity that occurs after presenting a stimuli [Phi+97].

EEG data can be divided into different frequency bands: Delta (δ , 1–4 Hz), theta (θ , 4–8 Hz), alpha (α , 8–13 Hz), beta (β , 13–30 Hz), and gamma (γ , 30–70 Hz) [CKS15]. Each band provides a time series that varies in frequency, shape, and amplitude, of which we show an example in Figure 2.7.

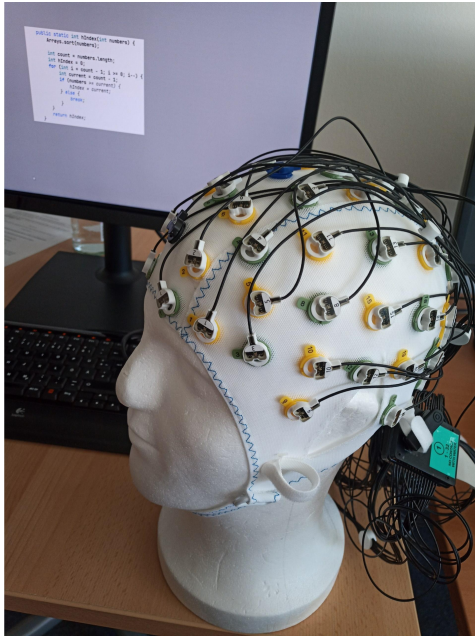


Figure 2.6: Photo of 64-channel EEG device in a lab.

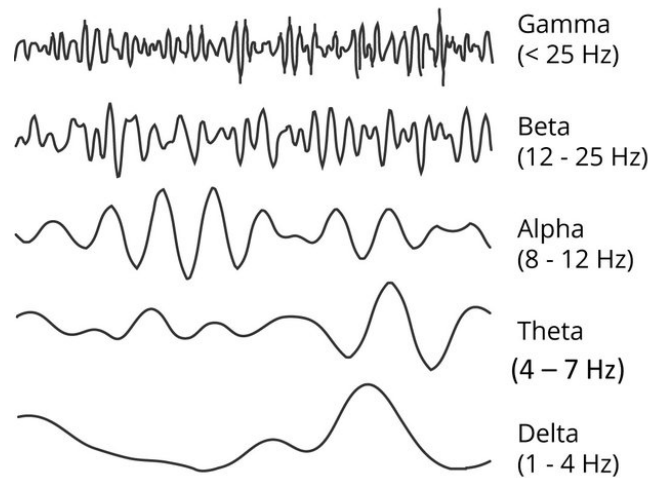


Figure 2.7: Visualization of different frequency bands of EEG (taken from Figure 1.5 of [Sak17])

Alpha and theta frequency bands both have been shown to correlate to mental load and working memory. They act in opposite ways so that when alpha power increases, theta power decreases [Kli99].

EEG Studies in Program Comprehension EEG has been widely adopted in neuroscience, for example, for studying working memory load [Ber+07]. EEG now also has found its way into several studies focusing on software engineering. Crk et al. used EEG in software engineering to study programmer expertise. They found that a programmer's experience level induces higher levels of alpha waves during program comprehension tasks [CKS15].

Yeh et al. studied atoms of confusions (small patterns of source code that lead to programmer confusion, [Gop+17]) in C code with EEG. They found significant increases in alpha and theta frequency bands for confusing code [Yeh+17].

Kosti et al. replicated the first fMRI study by Siegmund et al. [Sie+14a] with EEG [Kos+18]. They confirmed that the comprehension tasks designed by Siegmund et al. are more demanding than the control tasks of finding syntax errors. In addition, they were able to train a classifier based on the participants' brain wave patterns to predict the subjective difficulty of the presented tasks [Kos+18].

Lee et al. first compared novice and expert programmers with EEG during comprehension tasks and found significant activation in the left frontal lobe [Lee+16]. In a follow-up study, Lee et al. extended their EEG study with eye tracking and successfully predicted task difficulty and programmer expertise [Lee+17].

Recently, Ishida and Uwano used EEG and eye tracking and found a significant increase in the alpha frequency band for programmers who successfully finished a task [IU19]. Madeiros et al. used EEG to investigate the accuracy of widely used software complexity metrics and found little predictive power for all frequency bands [Med+19]. In a follow-up study, they investigated several EEG measures to distinguish different experience levels [Med+21]. Fucci et al. used a combination of EEG, EDA, and heart sensors to replicate an fMRI study of Floyd et al. [FSW17] on program comprehension. Interestingly, the heart sensor alone could determine which condition was presented to participants [Fuc+19]. Busechian et al. demonstrated on studying the cognitive processes of pair programming with EEG [Bus+18]. In a similar study, Ikramov et al. found an increased need for attention when fulfilling the navigator role in pair programming [Ikr+19].

Overall, research has shown that EEG can be used to study source code patterns, program comprehension, and differences between programmers' experience levels. Due to its low cost, non-invasive nature, and few requirements, EEG has been suggested to be a constant, real-time monitoring device during software development. Radevski et al. proposed to use EEG monitoring, for example, to automatically alert programmers to take breaks when they become stressed or overwhelmed with a task [RHM15].

2.4.2 Functional Near-Infrared Spectroscopy (fNIRS)

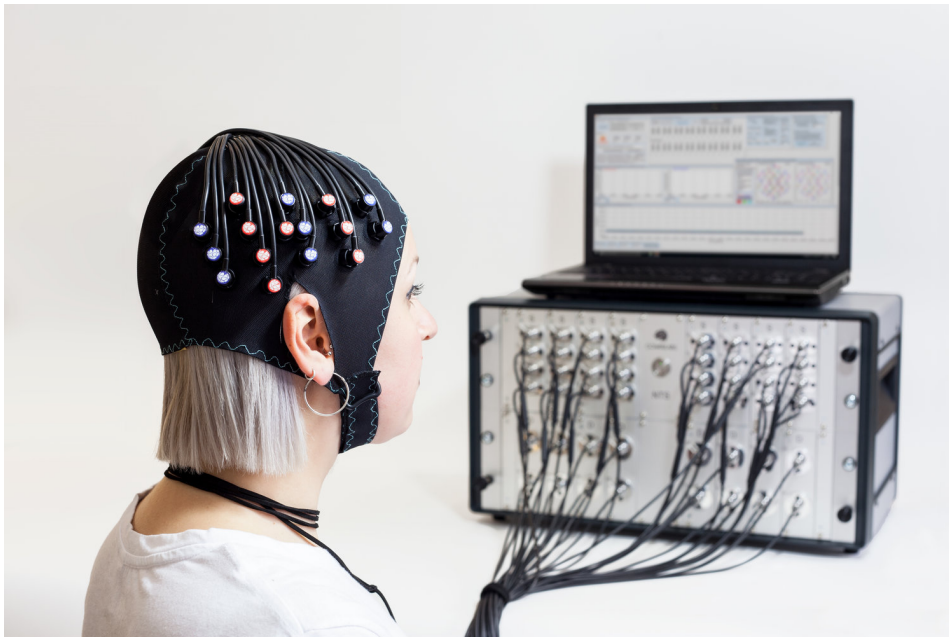


Figure 2.8: Photo of participant wearing an fNIRS in front of a computer (taken from [Elisenicolegray, CC BY-SA 4.0](#), via Wikimedia Commons)

²blah blah blah

Functional near-infrared spectroscopy (fNIRS) uses near-infrared light emitters to detect brain activity. This idea of shining light into the skull to measure blood flow existed for a century but finally came to fruition in 1993. Since then, many commercial versions of fNIRS have been developed with wide adoption in neuroscience [Sch+14].

fNIRS Studies in Program Comprehension So far, a few studies have used fNIRS in a software-engineering context. First, Nakagawa et al. showed the feasibility of using fNIRS while completing programming tasks. They presented comprehension tasks with code snippets with two difficulty levels and observed higher blood flow during the more challenging code snippets [Nak+14]. Ikutani and Uwano contrasted program and arithmetic tasks with several subtypes. They found that programmers use their frontal pole for variable memorization, but observed no significant differences for arithmetic tasks [IU14]. Fakhoury et al. combined fNIRS and eye tracking and showed that problematic identifier names significantly increase cognitive load [Fak+18; Fak+20]. Endres et al. investigate program comprehension, natural language reading, and mental rotation with fNIRS. They find distinct activation patterns of novices for each task and, unlike most neuroimaging studies, a bilateral activation [End+21].

2.4.3 Functional Magnetic Resonance Imaging (fMRI)



Figure 2.9: Participant in our fMRI scanner. Participants can view stimuli on a small plastic screen via a mirror on the head coil (cf. Figure 3.3). Participants respond to tasks with fMRI-compatible two-button response device.

Functional magnetic resonance imaging (fMRI) is a measurement method that also allows researchers to observe ongoing brain activation [HSM14]. In the same vein as fNIRS, it relies on the

BOLD response and the haemodynamic response (cf. Section 2.4 and Figure 2.5). Observing participants' brain activation during tasks lets researchers infer occurring cognitive processes [GIM13]. We show an image of a participant in our fMRI scanner in Figure 2.9.

Collected fMRI data is structured in a sizable three-dimensional array of numerical values representing *voxel* (i.e., 3D pixel) intensities. Depending on the protocol, voxels have a typical edge length of 2–3 mm. fMRI does not measure the entire brain at once, but rather in typically around 35 slices, one slice at a time. Each slice takes around 50 ms to record, which means that an entire brain can be scanned in two seconds. More recently, *multi-band protocols* concurrently measure multiple slices, allowing researchers to increase spatial or temporal resolution. All values can be adjusted depending on the particular needs of an experiment. For example, one may reduce spatial accuracy to gain a higher temporal resolution.

Besides *functional* magnetic resonance imaging to observe brain activation, MRI scanners can use different protocols for different foci. One commonly used protocol is *structural* MRI to measure a participant's anatomy, which we describe in detail in Section 2.4.3.4.

2.4.3.1 Experiment Design

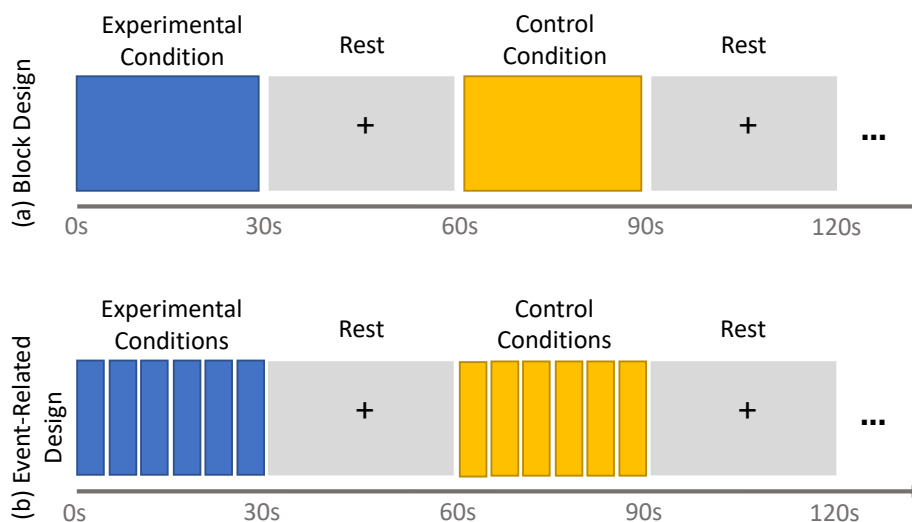


Figure 2.10: Visualization of typical fMRI designs: (a) block design and (b) event-related design.

An experiment in an fMRI scanner is often different from conventional experiments. By necessity, fMRI requires a specific experiment design.³ In particular, it typically requires at least two tasks: One task for the cognitive process in question and a task that acts as the control condition. Further, because we observe a change in the BOLD response to a presented task, we need to

³This section only provides a high-level overview without all variations and details, which can be found, for example, in [Agu11].

allow the change in brain activation to return to the baseline. To this end, standard experiment designs have prolonged rest conditions between tasks in which participants are asked to rest (e.g., by looking on a fixation cross on a screen).

In addition to the experiment task, control condition, and rest in between, the researcher needs to configure the order and length of each task. We present two prototypical experiment designs next.⁴

Block Design The most commonly used experiment design in fMRI research is the *block design*. A block design uses long-lasting tasks of typically 15–30 seconds with intermittent rest conditions. We visualize an experiment with block design in Figure 2.10. A block design can be used if the presented tasks require prolonged thinking from participants. For example, understanding a code snippet with 20 lines of code induces program comprehension for a significant time (i.e., more than 5 seconds). A block design assumes brain activation and thus a BOLD response for the entire block. The rest condition that follows the stimuli allows the brain activation to return to the baseline.

Typically, a control condition will also be presented. The pattern of task-rest-control-rest sequences is repeated several times until the experiment is over. The overall time in the fMRI scanner does typically not exceed 45 minutes.

Event-Related Design A different approach is an *event-related design*, which we also visualize in Figure 2.10. An event-related design is typically used when a rapid cognitive process is under observation. For example, recognizing a single face takes less than a second and thus is inadequate to induce sufficient brain activation to be detectable by fMRI. Therefore, an event-related design uses multiple similar repetitions of a task, for example, rapidly recognizing 15 faces in a row. This combination of multiple instances of the cognitive process effectively induces a sufficient BOLD response. Like the block design, a rest condition follows to allow a return to the brain activation baseline.

2.4.3.2 Control Condition

When observing participants with fMRI, we collect *all* induced brain activation. In the context of complex cognitive tasks, this can be problematic as there are plenty of basic cognitive activities that are also triggered. For example, program comprehension induces a strong activation in the visual cortex due to programmers physically moving their eyes and recognizing characters on the screen. However, this is entirely irrelevant to understanding the essence of program comprehension. Thus, we need to filter out such irrelevant brain activation with a suitable control condition.

⁴There are various other options, such as mixed designs, but are outside of this dissertation's scope.

A control condition is designed to be very similar to the experimental task. It only differs in a way that the key cognitive tasks are not involved. For example, a control condition for reading and comprehending a word is presenting scrambled letters with the same amount of characters as the original word. Both tasks require a participant to visually process the presented text, but only the original task induces semantic processing. In this case, researchers can understand which parts of the brain are involved in semantic processing during reading, which without the control task, would not have been as clear-cut.

A suitable control condition for program-comprehension studies is not as obvious. The first fMRI study by Siegmund et al. used the task of locating syntax errors as the control condition [Sie+14a]. The idea is that locating basic syntax errors, such as a missing semicolon, does not require program comprehension, but induces similar basic cognitive processes (e.g., visual processing, attention). More recently, Ivanova et al. instead contrasted program comprehension with the same content in prose form [Iva+20]. These divergent control conditions result in different activation patterns and show how critical choosing the control condition is. We outline current and future work on this topic in Section 3.5.5.

2.4.3.3 Risks and Participant Requirements

Empirical experiments with human participants need to evaluate their risks. Despite exposure to strong magnetic fields in the fMRI scanner, there is no known direct biological risk [Har+09b].

However, there are direct physical risks due to the strong magnetic field, which limit the participant pool. In particular, participants with metallic implants or pacemakers can be at risk. Thus, we screen all participants interested in our fMRI studies for such exclusion criteria, which also includes claustrophobia, pregnancy, and tattoos (low-quality ink may contain iron interacting with the MRI scanner). The informed consent and questionnaire used for the studies presented in this dissertation are in the Appendix (Section 7.5).

2.4.3.4 Analysis of fMRI Data

Preprocessing of Functional Data Acquired fMRI data contains substantial amounts of noise from various sources. Neuroscience has developed several preprocessing steps that are nowadays standard to reduce artifacts' influence on the analysis results and interpretation.

Typical preprocessing steps, which we also applied to all studies presented in this dissertation, include the following:

- **Slice-scan-time correction:** An fMRI measurement observes the brain in separate slices (cf. Section 2.4.3), so the first slice can be observed up to two seconds earlier than the last slice. To correct this temporal offset, we apply a slice-scan-time correction that corrects this offset by interpolating and smoothing the intensity series for each voxel.

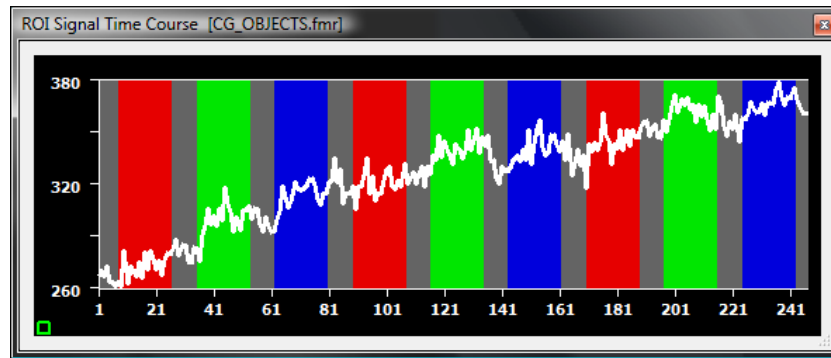


Figure 2.11: Visualization of a typical voxel intensity series with a strong linear drift. The x-axis is experiment time in terms of fMRI scans. The y-axis is the voxel intensity (i.e., brain activation strength). The image is from the BrainVoyager documentation (online version, accessed 25th Nov 2020): <https://www.brainvoyager.com/bv/doc/UsersGuide/Preprocessing/TemporalHighPassFiltering.html>

- **3D-motion correction:** While foam paddings are used to minimize head movements, there are still small head movements creating artifacts. These can significantly affect data quality, especially if they are rhythmic to the presented stimulus. To ensure robust results, a head-motion-detection algorithm can help researchers exclude data that has too many artifacts (e.g., more than 3 mm of movement within one session). If a data set can be used, a 3D-motion correction attempts to correct the data for minor and slow movements by realigning the image.
- **High-based filtering:** Similar to head motion, low-frequency drifts throughout an experiment can substantially affect the statistical analysis if not taken into account. We visualize an example of a typical linear drift of a voxel intensity series in Figure 2.11. To correct such drifts, we preprocess the data with a high-pass filter (GLM-Fourier) of typically three cycles per scan.
- **Spatial smoothing:** Finally, as part of the statistical analysis, functional data is often spatially smoothed. The parameters for the spatial smoothing differ between research groups. We use 4 mm as configuration for the full-width-at-half-maximum (FWHM) parameter for spatial smoothing with a Gaussian filter. It is slightly wider than our voxel size (3 mm) and provides a good balance between retaining spatial accuracy and ensuring data robustness.

For our research presented in this dissertation, all steps above were conducted with BrainVoyager™ QX 2.8.4.⁵. The specific configurations can be found in the description of the respective study.

⁵Brain Innovation BV, Maastricht, The Netherlands, <https://brainvoyager.com>

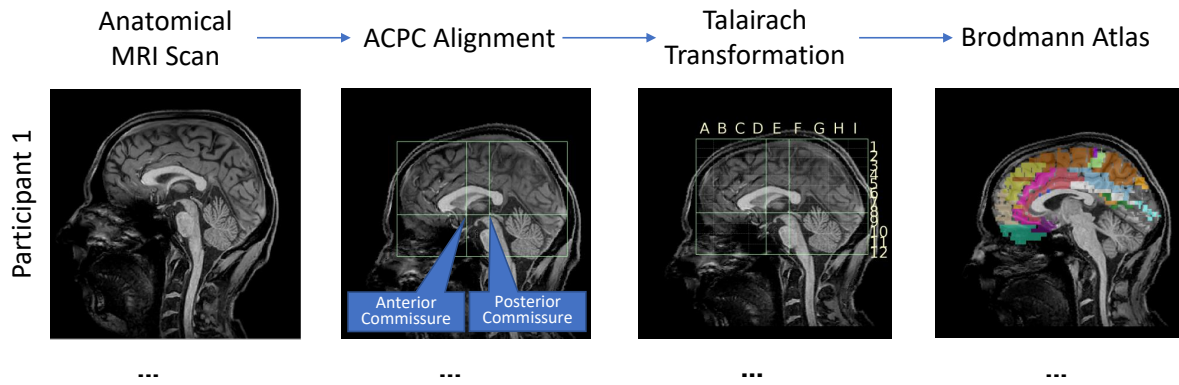


Figure 2.12: Visualization of the brain standardization process with BrainVoyager. One individual raw anatomical scan is transformed into a Talairach-space brain with an overlaid Brodmann atlas. For a full study, this process is repeated for every participant.

Standardization of Anatomical Data We collect anatomical data to understand each participant’s individual brain anatomy. Since we average functional data across participants, we need to correct for slight differences in brain anatomies. To this end, we use a *standardization* process, which serves multiple purposes. First, most neuroimaging studies analyze and report their results on an aggregated group level. Standardization of each participant’s brain to a common, corresponding space enables group-level analysis and interpretation. Second, anatomical standardization allows comparing and synthesizing activation clusters across different fMRI studies, but only if they are using the same 3D-coordinate space.

There is a multitude of 3D-coordinate spaces and atlases of the human brain [Eva+12]. A commonly used template is the Talairach space [TT88]. The Talairach space normalizes participants’ brains by warping them along their anatomical structure. Specifically, brains are *ACPC-aligned*, which means the cerebrum is rotated into an imaginary plane connecting two prominent anatomical landmarks (i.e., anterior commissure and posterior commissure). Then, the brain’s six outside edges are defined (i.e., the points that the most to the front, back, left, right, top, and bottom of the brain). Based on these overall 8 landmarks, we can warp an individual brain onto the Talairach template. We visualize the entire process in Figure 2.12.

A closely related space to the Talairach space is the Montreal Neurological Institute (MNI) space [Eva+93]. It is newer than Talairach, but not as widely used yet. Additionally, there are several versions with slightly different atlases. While closely related to the Talairach space, the MNI space cannot directly be translated into Talairach space without errors [Bre+01].

The Talairach space and MNI space can be mapped to an atlas. A common atlas is the Brodmann areas (BA), which serve as an anatomical classification system. The entire brain is split into several areas based on cytoarchitectonic differences suggested to serve different functional brain processes [Bro09; Bro06].

Functional and anatomical data are acquired with different MRI protocols, which leads to differences in the structure of the data. For example, the field of view may be larger during the

anatomical scan capturing the entire head in contrast to a focus on only the cortex for the functional data. To correct for these differences, functional and anatomical data is typically *co-registered*, which refers to a process that aligns the separately acquired functional and anatomical data based on anatomical landmarks [Kle+09].

For the research in this dissertation, we follow standard procedures established in neuroscience. We use the Talairach space with the Brodmann atlas. All group analyses are conducted and all results reported on a study-averaged Talairach brain.

Statistical Analysis with GLMs After the anatomical and functional preprocessing, researchers must conduct a statistical analysis of the acquired brain-activation data. Notably, conventional parametric and nonparametric statistical tests cannot be used to analyze fMRI data, because of the experiment-design complexity with many conditions, the amount of data, and its time-course nature. Instead, a general linear model (GLM) is computed with a model of a haemodynamic response function [WF95].

In a nutshell, the statistical analysis with a GLM entails two steps. First, on a participant-specific level, each voxel's time series is independently analyzed with a (univariate) statistic. Our data sets contain around 90 000 voxels for each brain scan. Thus, there are 90 000 statistical results based on the computed GLM. These results are then transformed into a three-dimensional map.

Second, we further process the voxel-based dimensional maps of each participant on a group level. There are two main approaches to this: A fixed-effects analysis or a random-effects analysis. In essence, a fixed-effects analysis concatenates the data from all participants together, essentially creating one large data set. A fixed-effects analysis has higher statistical power, but does not allow to draw conclusions on the population level. A random-effects analysis treats each participant as a separate subject and considers between-subject variance. If the participants were a representative sample of the population, random-effects analysis allows conclusions to be drawn on a population level. Two further options are mixed-effects analysis and balanced designs which are outside this dissertation's scope.

The resulting group-level statistical map is then tested for significance. One crucial aspect that needs to be considered is the sheer amount of data. In our case, we test 90 000 voxels, which inevitably, due to random and systematic noise, will lead to false positives. Thus, it is imperative to correct for multiple comparisons to avoid making false conclusions (see famous dead fish example [BMW09]). We use *false discovery rate* (FDR) correction [BH95], because it provides a good balance between correctly identifying voxels as significantly activated (i.e., rejecting null hypothesis) and avoiding falsely identifying voxels as activated (i.e., incorrectly rejecting the null hypothesis). Bonferroni correction [BA95] is an alternative, which, however, can be overly conservative [Per98]. Additionally, activated voxels need to be part of a cluster of a certain size (e.g., 27 mm^3).

Classifiers with Machine Learning In contrast to merely identifying activated brain areas, some researchers aim to make a reverse inference, that is making predictions based on observed brain activations. To this end, they use machine-learning classifiers to check whether the brain-activation data can be distinguished between different tasks or participants. They generate the same statistical maps as described in the previous section. However, instead of selecting activation clusters exceeding a significance threshold, they feed the maps into classifiers.

A few fMRI studies in software engineering use machine learning or a similar approach (e.g., [FSW17; Hua+19; Sie+21; Iku+21]), which we describe in further detail in the following section.

2.4.3.5 fMRI Studies in Software Engineering

Siegmund et al. were the first to use fMRI in 2014 when they developed an experiment for studying program comprehension with fMRI and applied it to provide a neuro-cognitive perspective to bottom-up comprehension [Sie+12; Sie+14a]. Since then, around one dozen studies have used fMRI to answer software-engineering research questions. We present them grouped in the targeted cognitive process.

Program Comprehension Most studies used some form of a program-comprehension task to induce brain activation. Specifically, the first study observed bottom-up comprehension and identified a network of five areas in the left hemisphere of the brain [Sie+14a]. Our follow-up study largely confirmed these results and found that programmers use the same network of brain areas during top-down comprehension, just with higher neural efficiency [Sie+17].

Directly inspired by the first study, Floyd, Santander, and Weimer conducted an fMRI study with three tasks: program comprehension, program review, and prose reading. They found similar activation patterns and identified that program comprehension becomes neurally closer to reading with more extensive programming experience [FSW17].

Duraes et al. published the second fMRI study on programmers debugging code snippets. Their focus was on debugging tasks that required participants to locate possible defects. In general, the observed activated areas are associated with language processing and mathematics. In particular, they found brain activation in the right anterior insula when a bug was spotted and confirmed [Dur+16]. In a follow-up study, Castelhana et al. used a similar setting with expert programmers to confirm the crucial role of the anterior insula [Cas+19].

More recently, there were studies using program comprehension tasks to understand different phenomena. We conducted another bottom-up comprehension study, but with snippets varying along various code complexity metrics revealing that most metrics show little predictive power (cf Section 5.1, [Pei+21]).

Ivanova et al. used a novel approach and contrasted short Python snippets with content-matching natural-language text. They found no strong left-lateralized bias of the brain activation with a

direct contrast between these two conditions, but a bilateral response in the domain-general network [Iva+20]. Liu et al. invited expert programmers for program comprehension and memory tasks, including various localizer tasks. They identified a left-lateralized fronto-parietal network that activates during program comprehension. It overlaps with brain areas related formal logic, language, and math [Liu+20].

Various Programming Activities In addition to program comprehension and debugging, there have been several one-off studies with various research questions. Huang et al. contrasted the underlying cognitive processes between mental rotation tasks and data structure manipulation and found overlapping brain activation, but distinct activation patterns. Notably, they used both, fNIRS and fMRI, to also compare each measure's power in the context of software engineering [Hua+19]. In a follow-up paper, they also included a comparison of eye tracking that was recorded during the fMRI sessions [Sha+20a].

Huang et al. studied code review with fMRI, investigating how gender and humanity change prospective reviewers' perspectives. They identified that there are differences in code review behavior between genders and identify several cognitive biases during code review (e.g., bias against automatically generated code) [Hua+20].

Ikutani et al. used a program categorization task to understand the neural differences between programmers with different expertise levels. Their classifier could predict programmers' expertise based on the observed brain activation [Iku+21].

Finally, unlike all previous studies that required mostly passive interaction with the stimuli, Krueger et al. studied writing source code. In a commendable effort, they created a custom-designed keyboard that works in the fMRI environment and found a stronger activation in the right hemisphere while writing source code [Kru+20]. This complements evidence of a mostly left-hemisphere lateralization of program comprehension. In a follow-up analysis, Karas et al. investigated the *functional connectivity*, that is how similarly different areas of the brain change in activation, of the same data set. They find a significant link between Broca's area and the number form area strengthening the combination of natural language processing and mathematics in programming [Kar+21].

2.4.4 Comparison of Neuroimaging Measures

The presented neuroimaging techniques, due to their different approaches to observe brain activity, have advantages and limitations. We provide a generalized overview of fMRI, fNIRS, and EEG as measurement measures in Table 2.2.

EEG is a comparatively low-cost sensor for measuring brain activity. It provides a high temporal resolution with typical measurements in 100 Hz. There is minimal delay between the neurons' electrical discharge and the received EEG signal allowing to precisely pinpoint cause and effect between stimuli and event-related potential.

	fMRI	fNIRS	EEG
Measures	BOLD	BOLD	Electrical activity
Delay	Several seconds	Several seconds	Milliseconds
Temporal Resolution	~1-2 seconds	~1-2 seconds	Milliseconds
Spatial Resolution	++	+	—
Participant Restrictions	—	+	++
Environmental Limitations	—	++	+
Portable	No	Yes*	Yes*
Financial Costs	—	+	+

Table 2.2: Comparison of fMRI, fNIRS, and EEG as brain activity measures. Note that only a general overview of an abstract, generalized perspective is provided. * it depends on the device and may not be universally true. ++ very positive, + positive, and - a (comparatively) negative characteristic.

However, EEG data have a low signal-to-noise ratio, because of the sensors' sensitivity to body and head movements as well as environmental influences. Thus, EEG experiments often require averaging across many tasks, which dilutes the high temporal resolution. Further, as it measures large groups of neurons, there is little spatial resolution, since the observed activity cannot be linked to a specific source in the brain. A higher number of sensors of the EEG device allows a more accurate spatial localization, but is also more tedious. Because EEG is quite sensitive to the environment, a robust analysis requires extensive data preprocessing [Sha+08].

Similarly to EEG, fNIRS allows lab experiments that relatively closely resemble real-world scenarios. For example, researchers could conduct experiments in which programmers sit in front of a computer interacting with an IDE. It also would be possible to observe writing code with fNIRS.

Like EEG, fNIRS is limited in spatial accuracy and how deep into the skull it can measure. Specifically, it is limited to 2-3 cm into the skull and therefore cannot measure brain activation deep in the skull.

Studies with fMRI provide the most detailed data on cognitive processes. A direct comparison of mental rotation tasks in software engineering showed that fNIRS has a weaker signal in addition to less skull-penetration depth than fMRI [Hua+19]. But, the setup comes with the inherent restrictions with participants being in an fMRI scanner. Specifically, interaction with stimuli is severely limited. Most experiments are designed with little interaction (i.e., only a button press at the end of a task rather than continuous interaction with an IDE).

In essence, fMRI, fNIRS, and EEG all have advantages and disadvantages and their use depends on the research question. EEG and fNIRS allow for more realistic scenarios and thus enable higher external validity, for example, by conducting studies with programmers in front of a regular computer and IDE. On the other hand, fMRI provides the most comprehensive measure of brain activity.

Synthesis of Neuroimaging Measures In the past decades, research on program comprehension has been using more novel psycho-physiological measures to explore programmers' cognition. Research with eye tracking started in the early 1990s and has made substantial progress in methodology and gained significant insights into programmers' visual attention patterns [Sha+20b]. A yearly workshop⁶ discusses current topics on eye tracking in programming.

Research with neuroimaging measures, such as EEG, fNIRS or fMRI, is not as established yet. It is already clear that neuroimaging measures have their advantages and disadvantages. Overall, the community is still in the early, exploratory stages in which common standards for methodology have to be defined. For example, the basic question on which control condition is suitable to distill the essence of program comprehension is still unclear (cf. Section 3.5.5). This dissertation is part of the effort to establish a standard framework for observing programmers with fMRI in combination with further modalities.

2.5 Chapter Summary

In this chapter, we provided an overview of the state of program-comprehension research. Program comprehension is a complex cognitive process that often requires a combination of top-down and bottom-up processes, which are part of most strategy models. The measurement of program comprehension is challenging and requires different approaches to fully understand it. In particular, a lack of comprehensive measures of program comprehension prevented validation and refinement of the program-comprehension models. Research with eye tracking and neuroimaging offers a new perspective on program comprehension, but comes with its own drawbacks.

Having laid the groundwork, we apply fMRI to open research questions in software engineering in the next chapters. We demonstrate how neuroimaging can open new doors for software-engineering research.

⁶Eye Movements on Programming (EMIP), <http://emipws.org>

3 A Framework for fMRI Studies of Program Comprehension

This chapter shares material with several prior publications [PSB17; Pei+18a; Pei+18b; Pei+18c; Pei+19; Han+20].

Past program-comprehension research mainly used conventional measures such as response time or response correctness. Since they only look at the result of a comprehension task, it is challenging to provide insights into the underlying cognitive processes. Thus, researchers also relied on self-reflection measures, such as think-aloud protocols or interviews, to gather data on cognitive processes that are part of program comprehension. But, these measures tend to be somewhat unreliable when it comes to cognitively demanding tasks such as program comprehension. In essence, program comprehension consists of internal cognitive processes that are inherently difficult to observe [Sie16].

Researchers have begun to explore new psycho-physiological and neuroimaging methods to *objectively* measure program comprehension and its underlying cognitive processes. For example, eye tracking allows researchers to observe eye movements while programmers comprehend source code [Bus+15]. Functional near-infrared spectroscopy (fNIRS) [Fak+18] and electroencephalogram (EEG) [Fri+14] provide insights into programmers' cognitive load during tasks. Siegmund et al. have been at the forefront of adopting functional magnetic resonance imaging (fMRI) as one method to understand developers. Siegmund et al. developed the first framework to study programmers with fMRI [Sie+14a]. This seminal work has inspired other researchers to conduct fMRI studies on their software engineering sub-areas such as debugging [Dur+16; Cas+19], data structures [Hua+19], and code comprehension [FSW17; Kru+20; Iku+21].

In summary, in addition to the conventional methods developed decades ago, software-engineering researchers are adding novel psycho-physiological measures to their toolset. While neuroimaging provides insights into ongoing cognitive processes in great detail, programming consists of extremely complex cognitive processes that pose additional challenges. A programmer's behavior inside the fMRI scanner appears as a black box to researchers, as we cannot capture the specific strategy to solve a presented task. While we can ensure the completion after a task of 30 to 60 seconds, we miss details throughout the task. Also, due to the temporal resolution of fMRI (i.e., depending on the protocol, 1 to 2 seconds) [HSM14], we may miss some rapid cognitive processes when observing a programmer with fMRI only.

This dissertation contributes to the foundation of using fMRI as a measure in software engineering by defining an experiment framework. We developed and tested several improvements, most

importantly conjoint eye tracking, to allow researchers to understand even fine-grained effects (Section 3.2). Further, we sketch a vision of a multi-modal future such as an eye-tracking-informed fMRI analysis (Sections 3.3.2, 3.5.1 and 3.5.4). As a step into this future, we developed *CODER-SMUSE*, a multi-modal data exploration tool specific for software engineering (Section 3.4). Finally, we outline several steps that can further establish a robust framework for fMRI research in a programming context (Section 3.5).

3.1 First fMRI Experiment from Siegmund et al.

3.1.1 Experiment Design

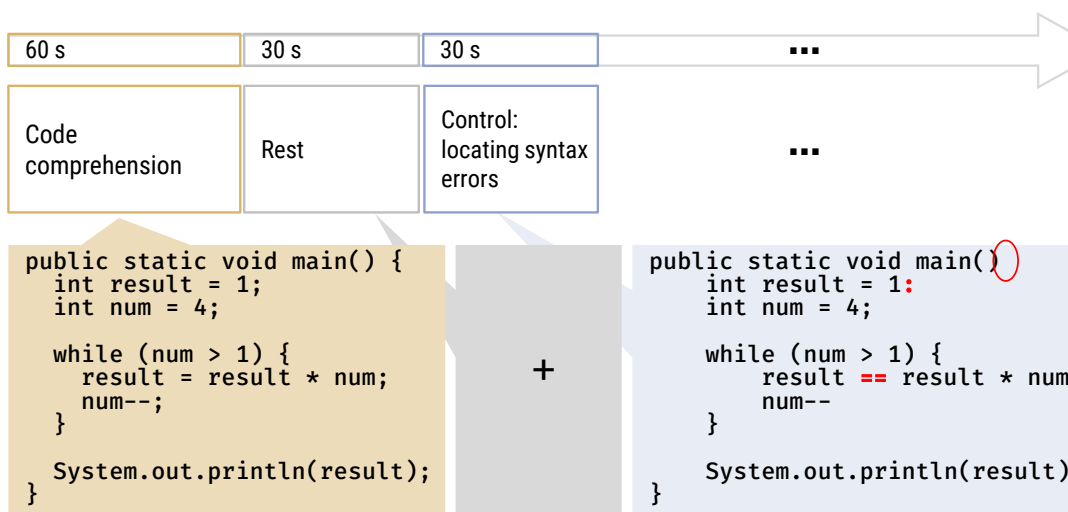


Figure 3.1: Visualization of Siegmund et al.’s block-based fMRI experiment design. In this case, a program-comprehension task with control condition (i.e., locate syntax errors, indicated in red) and intermittent rest periods. The visualized sequence was repeated with 12 different snippets.

In 2012, Siegmund et al. were the first to propose studying program comprehension with fMRI and outlined an experiment design that contrasted bottom-up program comprehension with locating syntax errors [Sie+12]. Siegmund et al. chose to use a fixed-length block design, switching between a 1-minute program-comprehension task and 30 seconds of finding syntax errors, and an intermittent 30-second rest condition (cf. Figure 3.1). This was repeated twelve times for a total experiment duration of around 30 minutes. We show an example snippet in Listing 3.1. After positive feedback, the experiment design was then applied to study bottom-up comprehension, which was published at the top conference of software engineering [Sie+14a]. Besides the methodological contributions, the study successfully identified a network of five activated brain areas involved during program comprehension.

```
1 public float arrayAverage(int[] array) {  
2     int counter = 0;  
3     int sum = 0;  
4  
5     while (counter < array.length) {  
6         sum = sum + array[counter];  
7         counter = counter + 1;  
8     }  
9  
10    float average = sum / (float) counter;  
11    return average;  
12 }
```

Listing 3.1: Example code snippet in Java inducing bottom-up comprehension from [Sie+14a] that computes the length of the last word in a string. The snippet uses non-meaningful identifiers to induce bottom-up comprehension. Participants needed to figure out the output of this snippet “5”.

3.1.2 Limitations of Experiment Design

A closer look at Siegmund’s experiment design reveals the structure visualized in Figure 3.1. This design was effective for identifying the activated brain areas during bottom-up comprehension. In addition to brain activation, Siegmund et al. recorded behavioral data (i.e., response correctness and response time), but did not further use them beyond a basic analysis.

Initially, the research presented in later chapters of this dissertation was planned to apply Siegmund’s experiment design without modification to further research questions (e.g., top-down comprehension). However, during our study on top-down comprehension (cf. Section 4.1), we encountered limitations in the experiment design. Specifically, our research question on understanding the effect of meaningful identifiers on top-down comprehension could not be answered holistically. While participant comments and fMRI data hinted toward an effect of meaningful identifiers, we did not find compelling evidence.

Limitations of fMRI With Siegmund’s experiment design, the sequence of cognitive (sub)processes for each individual task and participant during an fMRI session cannot be easily inferred. Since program comprehension is a complex task with many layers (e.g., attention, working memory, problem-solving) and a sequence of different phases (e.g., [KR91; MW01]), this limits our gained insights. Further, because the neuro-cognitive perspective of program comprehension is still rather new, we lack the knowledge about brain activation during program comprehension to accurately identify such program-comprehension phases. In addition, the temporal resolution of fMRI is low (i.e., depending on the protocol, 1 to 2 seconds) and is delayed by about 5 seconds [HSM14; Cha+93]. Thus, we are unable to observe rapid cognitive subprocesses (e.g., identifying a meaningful identifier). We may miss some cognitive subprocesses of program comprehension assuming a uniform fMRI activation across the entire period of understanding a presented source code.

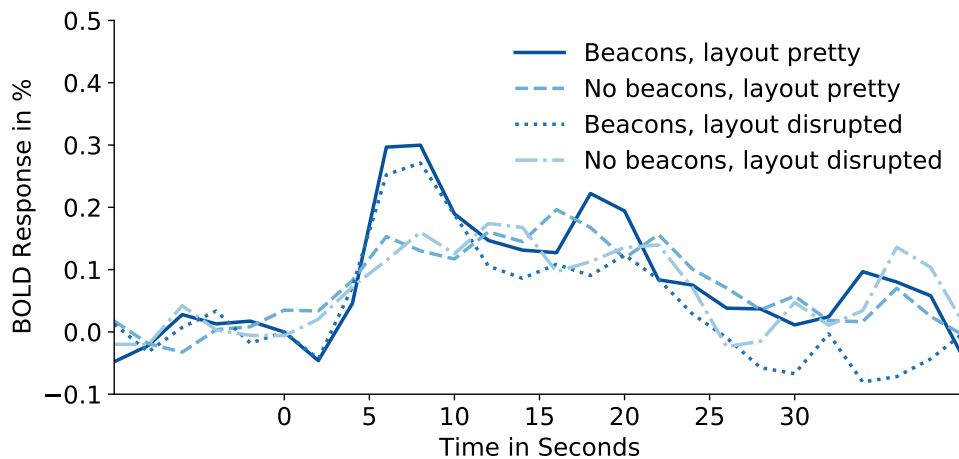


Figure 3.2: BOLD response in BA21 of top-down program comprehension.

For example, Figure 3.2 shows the BOLD response for BA21 (i.e., one of the activated areas associated with language comprehension) of four conditions from our fMRI study on top-down comprehension [Sie+17]. The BOLD response shows a higher activation strength between 5 to 10 seconds for both conditions, which include a meaningful identifier. A possible explanation for the increased activation strength is that, if participants recognize a meaningful identifier, they recall an appropriate programming plan [Bro83; Wie91]. However, because fMRI alone provides insufficient insight into a participant's comprehension strategy during a task, this is untested hypothesis.

In summary, this observed effect and possible future fine-grained effects cannot be pinned down to a cause highlighting a fundamental limitation of an experiment design relying only on fMRI. Specifically, fMRI alone suffers from the following limitations:

1. It provides limited insight into participant behavior within a long-lasting, complex task. The cognitive (sub)phases during a long task cannot easily be inferred.
2. The low temporal resolution of fMRI limits our insight in rapid cognitive processes.
3. A participant's strategy to solve a program comprehension task cannot easily be understood with fMRI alone.

To tackle these limitations, we initially outlined a possible solution: fMRI with simultaneous eye tracking would provide insight into participants' visual attention during long-lasting tasks such as program comprehension [PSB17]. We then implemented and evaluated this vision of observing programmers with fMRI and eye tracking simultaneously, which we describe in Section 3.2.

In this context, we applied further refinements to Siegmund's original experiment design and also evaluate them with studies in this chapter.

Further Modalities While the combination of fMRI and eye tracking increases our insights during program-comprehension experiments, there are further possible improvements. In another fMRI study, we added a third modality to observing program comprehension besides fMRI and eye tracking: psycho-physiological measures, which allow us to objectively measure a participant's physiological state (e.g., stress level). We delve further into this modality in Section 3.5.1.

fMRI Contrasts and Control Condition Another limitation is that Siegmund et al.'s experiment design may not provide clean contrasts between rest and program comprehension (cf. Section 2.4.3.1). A closer look at the BOLD responses of the first experiment showed that there is substantial brain activation during the rest condition. Participants indicated that they may reflect on the previously presented task if they were unable to solve it in time. In a follow-up fMRI experiment, we improved fMRI data contrasts by reducing reflective thinking during rest and investigated a possible candidate for a control condition. We describe this in detail in Section 3.5.4.

Similarly, the initially used control condition (cf. Section 2.4.3.2) of locating syntax errors may trigger some level of program comprehension (as indicated by participants). We therefore tested using a simple search task in the fMRI experiment described in Section 5.1. We describe this possible refinement in Section 3.5.5.

Tool Support Finally, a conventional analysis of fMRI experiment does not maximize insights by using the full potential of all modalities. For future multi-modal experiments, we need suitable analysis protocols and proper tool support. We outline such analysis in Section 3.3 and present a prototype implementation in Section 3.4.

Together, these improvements provide an experiment framework to conduct (multi-modal) fMRI studies of program comprehension.

3.2 Simultaneous Measurement with fMRI and Eye Tracking

3.2.1 Motivation

In this section, we present and evaluate our vision of an fMRI study with simultaneous eye tracking. By simultaneously recording eye movements, we aim to identify when a participant is dealing with which part of the code [BT06], thereby connecting program-comprehension phases to the resulting brain activation [PSB17]. Specifically for the example shown in Figure 3.2, we may find participants fixating on a meaningful identifier shortly before an increased BOLD response. In this case, eye tracking may provide evidence for the suspected fixation on a meaningful identifier, which would support the theory of semantic recall of programming plans during top-down comprehension [Sie+17; SE84]. In general, by observing and understanding programmer

behavior with eye tracking, the brain-activation data become more valuable, because we may be able to *explain* it.

With eye tracking, we can observe visuo-spatial attention by collecting eye-movement data [Hol+11], from which we can infer what the programmer is focusing on. Thus, eye tracking allows us to better understand the behavior of a programmer during a program-comprehension task. Furthermore, due to the high temporal resolution of eye tracking (i.e., depending on the device, 50 – 2000 Hz), we can capture even rapid eye movements. Observing visuo-spatial attention allows us to infer details of ongoing cognitive processes during program comprehension. For example, Duchowski et al. have shown that high saccadic amplitudes patterns after fixations indicate that the participants are scanning for a feature in a presented stimulus [Duc17]. However, longer fixations with shorter saccades indicate a pursuit search, thus suggesting that a participant is trying to verify a task-related hypothesis rather than overviews a stimulus [Duc17].

Eye tracking as a lone perspective is becoming a commonly used measure to observe visual attention during program comprehension (cf. Section 2.3.1). However, unlike neuro-cognitive measures, such as fMRI, eye tracking is limited regarding insights into higher-level cognitive processes (e.g., language comprehension, decision making, working memory). Thus, researchers have begun to combine eye tracking with neuro-cognitive measures, for example, simultaneous recording of electroencephalography (EEG) [Fri+14; Lee+17] or functional near-infrared spectroscopy (fNIRS) [Fak+18]. While an fMRI experiment is more restrictive than EEG or fNIRS, fMRI offers a higher spatial resolution, which motivated us to integrate fMRI and eye tracking as simultaneous measures.

We integrate simultaneous eye tracking to Siegmund’s fMRI experiment design to understand programmers’ behavior and identify rapid cognitive (sub)processes. Eye tracking has been used in previous fMRI studies in neuroscience, but mostly as an indicator of whether participants are fulfilling the task (and not sleeping) [Hol+11]. At the time of this study, only Duraes et al. also attempted to collect simultaneous eye tracking data during an fMRI session, but disregarded it during analysis. They also did not report how successful and precise the recorded eye tracking was in their experiment [Dur+16]. Hanke et al. observed the exact eye gazes during an fMRI session in which participants watched a movie, but also did not provide further analysis [Han+16]. At the time of this study, no prior study has combined fMRI and eye tracking in software-engineering research.

Combining fMRI and eye tracking is promising for program-comprehension research, because the two measures complement each other’s strengths. The high temporal resolution of eye tracking, in combination with information about which part of the code a participant is focusing on, allows us to identify the origin of brain activations more precisely in time. By combining the two measures, we can reason about causal relationships: What part of a program gives rise to the activation of a specific brain area or triggers a certain cognitive process? With a simultaneous observation with fMRI and eye tracking, we intend to benefit from the strengths of both measures.

We describe a new experiment framework for observing programmers with simultaneous fMRI and eye tracking in the following subsections. We include eye tracking into our experiment framework for more fine-grained fMRI analysis. Further, we present a study in which we test the

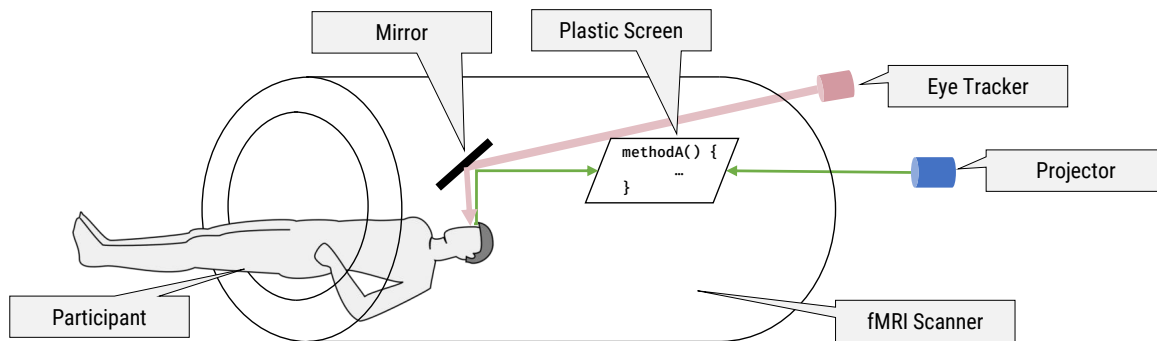


Figure 3.3: Eye-tracking setup with a long-range camera in an fMRI scanner.

new experiment design for its feasibility and reliability. Then, we use the acquired eye-tracking and fMRI data for more fine-grained analysis.

3.2.2 Research Objectives

While the simultaneous measurement and analysis of fMRI and eye-tracking data would open the door to a novel neuro-cognitive perspective of program comprehension, the strict non-magnetic environment around an fMRI scanner poses a challenge for the use of an eye-tracker. A regular camera-based eye-tracker is mounted outside the scanner bore, which is roughly 90 centimeters away from a participant's head. From this position, the camera and infrared light hit the small mirror, on which participants see displayed program code, and then hits a participant's eye through a small opening of the head coil. We visualize the setup in Figure 3.3. Due to its complexity, we first need to evaluate how stable and precise the eye-tracker is throughout an experiment of 30 minutes. We pose the first research question:

RQ 3.1: Can we simultaneously observe program comprehension with fMRI and eye tracking?

As a first step, we evaluate whether we can reliably collect fMRI and eye tracking simultaneously. Both data streams capture a different aspect of program comprehension. We could separately examine the fMRI and eye-tracking data. For example, we could analyze a participant's reading order linearity, which indicates expertise [Bus+15; PSA20], and match it to the resulting brain activation.

For our goal of combining the information of both data streams, an *eye-tracking-informed fMRI analysis*, we do not only require both data streams in parallel, but need to confidently recognize and synchronize specific events of program comprehension with spatial precision (e.g., fixation on a meaningful identifier). Thus, for our purposes, we need, at least, a word-level spatial precision.

To evaluate whether we can expect such precision from our setup, we pose our second research question:

RQ 3.2: Is eye tracking sufficiently precise for fMRI studies of program comprehension?

In our experiment to evaluate this new design, we used the same code snippets and display configuration as in previous studies [Sie+14a; Sie+17]. It is possible that the font sizes and line spacing were too small to reliably detect fixations at the word level and that we may need to increase them in future experiments. This optimization is critical, as the display in the fMRI scanner is restricted in size and resolution. Although possible, scrolling comes with new challenges, as it may induce movement (which further reduces the number of usable fMRI data sets) and continuously changes the visual input for each individual task (which makes the eye-tracking analysis more complicated). An answer to RQ 3.2 helps us understand how to design appropriate code snippets (regarding text size, line spacing) for our experiment framework.

To be certain that the eye-tracking data are consistently precise throughout an experiment, we pose our third research question:

RQ 3.3: Is there a drift throughout the experiment?

After initial calibration, eye-tracking accuracy is expected to *drift* (a spatial imprecision worsening over time) [Hol+11]. Participant movements challenge an eye-tracker to consistently capture a precise result [Hol+11]. In conventional eye-tracking experiments in front of a computer, the calibration can be repeated if the eye-tracking accuracy falls below a threshold [HH02; ERK08]. Such on-demand experiment interruption is infeasible in an fMRI study: The functional measurement has a fixed length and cannot be stopped without the need for a full restart. A possible split in multiple shorter runs (as done by Floyd et al. [FSW17]) with intermittent eye-tracking re-calibrations increases the experiment length and thus participant discomfort. Furthermore, it may decrease fMRI data quality due to participant movements in between the sections.

A common length for fMRI experiments is around 30 minutes, which is above our eye-tracker's recommended time for continuous eye tracking without a repeated calibration and validation procedure. That is, our experiment design requires us to significantly exceed the recommended eye-tracking time threshold, which raises the question of how precise eye tracking will be towards the end of an experiment.

Despite these concerns, two aspects of fMRI experiments suggest that a stable measurement throughout a 30-minute fMRI experiment is possible: First, a participant's head is fixated during the fMRI session. Head movement, which is a common cause for impaired eye-tracking accuracy, is strictly limited. Second, the fMRI protocol includes a rest condition, which displays a fixation cross in the screen center. We instruct participants to fixate the fixation cross during the rest condition. The precision and accuracy of the eye-tracker during the periodic fixation-cross condition is a clear indicator of the eye-tracking stability throughout an experiment. Answering RQ 3.3 will help us to decide whether we need to split the experiment into multiple sections with intermittent eye-tracker (re)calibrations.

3.2.3 Experiment Design

We based our study design on an fMRI study of program comprehension, which we describe in Section 4.1 [Sie+17]. Specifically, we have contrasted tasks of bottom-up comprehension, top-down comprehension, and locating syntax errors. We induced bottom-up comprehension by removing all semantic information from a code snippet. We facilitated top-down comprehension by familiarizing participants with the code snippets in a training before the fMRI session. To understand how meaningful identifiers (i.e., acting as beacons [Bro83]) influence top-down comprehension, we operationalized two versions of top-down comprehension by manipulating how meaningful a snippet's identifiers are (e.g., `arrayAverage` versus a scrambled `beeBtBurebZr`). All top-down and bottom-up comprehension snippets were part of our previous top-down comprehension study (more details in Section 4.1.1). The snippet complexity is similar to the previous studies: 7–14 lines of code (LOC), DepDegree of 10–24 [BF10]. In the fMRI scanner, participants had to compute the output of a specific function call for each snippet.

Experiment Framework For the conducted study, we aimed at developing an experiment framework with an improved experiment design based on the experiences from our previous fMRI studies. In a nutshell, an fMRI analysis's success depends on how suitable the chosen conditions are. fMRI analysis is based on computing contrasts between appropriately different conditions to carefully exclude cognitive processes from the brain-activation data unrelated to a research question. For example, in the previous studies, we use locating syntax errors as a control condition, which we contrast with comprehension tasks to obtain brain activation specific for program comprehension (e.g., working memory, but not the less relevant visual cortex). Thus, it is critical that our control condition maximizes contrasts by eliminating all unrelated brain activation, but also without triggering any comprehension-related brain activation. Furthermore, a rest condition, in which participants think about programming as little as possible, is necessary to provide a brain-activation baseline.

However, in our previous study [Sie+17], the data indicated that participants in fact partially comprehend code when finding syntax errors, and furthermore reflect on the comprehension tasks during the rest condition. Both reduce the statistical power of our fMRI analysis. Thus, inspired by Mallow et al. [Mal+15], we attempted to mitigate these problems by adding a new distractor task. For this purpose, we used a d2 task, which is a psychological test of attention in which participants scan through a row of letters and decide for each letter whether it is a *d* with two marks [BSAL10]. Integrating a distractor task may block snippet-related cognitive processes during the rest condition and also provide a better control task (as it is not related to programming).

To integrate simultaneous eye tracking, we made several changes at the technical level. First, we calibrated and validated the eye-tracker, which was scheduled after the anatomical pre-measurements (cf. Section 2.4.3.4), but before the functional fMRI scan. Second, we adapted the used Presentation® software⁷ scripts to synchronize the eye-tracker and the presented source

⁷Version 19.0, Neurobehavioral Systems, Inc., Berkeley, CA, USA, <https://neurobs.com>

code. We connected the Presentation software and the EyeLink eye-tracker with the *PresLink* plugin. Third, we changed the presented source code from text-based to image-based stimuli. The image-based snippets do not look any different for participants, but make the analysis of eye-movement data more precise. From a participant’s perspective, there is almost no extra effort with integrated eye tracking compared to a basic fMRI experiment. The calibration and validation at the beginning usually take only an extra minute.

We provide more details on the experiment design, including materials and used scripts on our project Web site.⁸

3.2.3.1 Experimental Conditions and Task Design

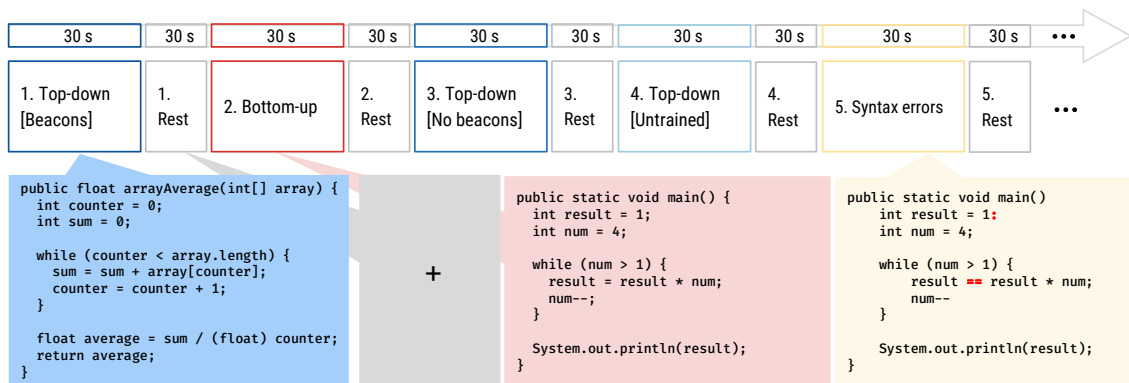


Figure 3.4: Illustration of one (out of five) experiment trials for our study on simultaneous fMRI and eye tracking.

Our experiment design essentially used the same design as our study on top-down comprehension, which we present in more detail in the next chapter (cf. Section 4.1, [Sie+17]). The fMRI study consisted of top-down and bottom-up comprehension tasks as well as a locating syntax-error task. We asked participants to determine the output of a presented Java function call. Figure 3.4 visualizes one out of five trials. Figure 3.5 shows an exemplary snippet as seen in the fMRI scanner. We kept the snippets’ computational complexity low (e.g., square root of 9 or 25), so that participants focus on program comprehension. The participants responded to each task via a two-button response device.

We asked participants to click the response button during the locating syntax-error task whenever they found a syntax error. Each snippet contained three syntax errors, which did not require comprehension of the snippet (e.g., missing semicolon).

```

Was ist das Ergebnis von squareRoots([9,25,16,100])?

public double[] squareRoots(int[] numbers) {
    double[] squareRootArray = new double[numbers.length];

    for (int i = 0; i < numbers.length; i++) {
        if (numbers[i] < 0) {
            squareRootArray[i] = Math.sqrt(-1 * numbers[i]);
        } else {
            squareRootArray[i] = Math.sqrt(numbers[i]);
        }
    }

    return squareRootArray;
}

```

Figure 3.5: Example code snippet as visible in fMRI scanner. Task on the top line is in German and can be translated to “What is the result of”?

3.2.3.2 Study Participants

We recruited 22 students from the Otto von Guericke University Magdeburg via bulletin boards. Requirements for participating in the study were some experience in object-oriented programming and the ability to participate in an fMRI experiment (cf. Section 7.5). Every participant completed a programming experience questionnaire [Sie+14b], which showed that our participants are a relatively homogeneous group in terms of programming experience. Table 3.1 shows the participants’ demographic data and their programming experience.

3.2.3.3 Experiment Execution

We invited interested participants to the study. When they arrived at our location, we explained our study’s goals, the risks of fMRI, and asked for their informed consent. Then, they completed a programming experience questionnaire and a brief training. We then conducted the fMRI session and finished with a short post-session interview.

We describe the fMRI scanner configuration in the Appendix (Section 7.1). We used an MRI-compatible EyeLink 1000⁹ eye-tracker for our study. The EyeLink eye-tracker offers 1000 Hz temporal resolution, <math><0.5^\circ</math> average accuracy, and

We tracked a participant’s left eye on a display with a resolution of 1280 by 1024 pixel. We calibrated the eye-tracker with a randomized 9-dot grid, and we conducted a 9-dot validation to

⁸<https://github.com/brains-on-code/simultaneous-fmri-and-eyetracking>

⁹SR Research Ltd, Ottawa, Ontario, Canada, <https://www.sr-research.com>

Characteristic		N (in %)
Participants		22
Gender	Male	20 (91%)
	Female	2 (9%)
Pursued academic degree	Bachelor	9 (41%)
	Master	13 (59%)
Age in years \pm SD		26.70 \pm 6.16
Programming experience	Years of experience \pm SD	6.14 \pm 4.57
	Experience score [Sie+14b] \pm SD	2.73 \pm 0.75
	Java experience [Sie+14b] \pm SD	1.93 \pm 0.33

Table 3.1: Participant demographics for our simultaneous fMRI and eye-tracking study.

identify possible issues with the calibration. If the error during validation exceeded the EyeLink's recommended thresholds, we repeated the calibration and validation process.

We calibrated and validated the eye-tracker after the (anatomical) pre-measurements but before the functional fMRI scan. Once the eye-tracker was calibrated and validated, we started the functional fMRI scan. Our Presentation script showed the first code snippet with the first scan period and triggered the EyeLink eye-tracker to record eye movements. We logged the start time to both systems (i.e., stimulus and eye-tracking computer). After every stimulus change (e.g., from comprehension to distractor task), we added a log with timestamp and stimulus name to the eye-tracker output. This way, we were able to accurately pinpoint the observed eye movements to the presented stimulus.

We used a combination of vendor-provided and custom scripts to extract and convert the obtained eye-tracking data. We also imported the data into Ogama for further analysis [Voß+08]. We provide the complete workflow, all custom scripts, and raw and processed eye-tracking data on this project's Web site.

3.2.4 Results

In this section, we present the results of this study, following our three research questions.

RQ 3.1: Can we simultaneously observe program comprehension with fMRI and eye tracking?

3.2.4.1 Data Recording

We successfully calibrated, validated, and recorded eye-tracking data for 20 out of 22 invited participants. We did not gather eye-tracking data from 2 participants, where calibration was

impossible (once due to Strabismus [Bil03], once due to a technical failure). Due to the delicate setup of the eye-tracking camera outside the bore, the angle to a participant's eyes is not ideal. When calibration initially could not be completed, we had to ask 8 out of 20 participants to widely open their eyes. This way, we were able to complete the calibration. The validation with widely opened eyes showed good precision. However, when this request was necessary, and participants returned to their eyes' natural state, the eye-tracker was unable to consistently capture the pupil, which resulted in incomplete eye-tracking data (for all cases less than 30% of recorded frames). Overall, for 10 out of 20 participants, the eye-tracker could not consistently capture the pupil.

3.2.4.2 Data Quality

An important aspect of eye-tracking data quality is how many frames the eye-tracker captured. In general, 100% of recorded frames are not realistic because of blinks. This matter is more severe in an fMRI environment due to participants looking into a bright, projector-backlit screen and the ventilation with dry air increasing the number of blinks. For 8 out of 20 (40%) participants, the number of recorded frames was excellent (more than 85%). For 10 out of 20 (50%) participants, the eye-tracker captured, at least, 65% of all frames. For 8 out of 20 (40%), the eye-tracker captured less than 10% of all frames.

RQ 3.1

Our study indicates that it is feasible to simultaneously measure program comprehension with fMRI and eye tracking. However, the comparatively high failure rate of eye tracking due to the fMRI environment has to be considered when designing experiments.

RQ 3.2: Is eye tracking sufficiently precise for our fMRI studies of program comprehension?

While the results of RQ 3.1 show that we can capture simultaneous eye tracking for around half of our participants, it does not reveal how much we can rely on the eye-tracking data and its spatial precision. Thus, we investigate the required degree of spatial precision in RQ 3.2. For our goal of an eye-tracking-informed fMRI analysis, it is critical not only to record eye movements (e.g., to analyze generic metrics such as fixation counts or average saccade lengths), but also to provide eye-tracking data with high spatial precision (e.g., to detect fixation on meaningful identifier). When eye tracking is successful, are the spatial errors small enough to confidently detect fixations on an individual identifier? To obtain an accurate result, we only analyze the eye-tracking data for all suitable participants (i.e., at least 65% of recorded frames, $n = 10$) for RQ 3.2.

To confidently detect fixations on a single identifier, we need appropriately small spatial errors. For our stimuli, one line of code was 40 pixels high and a single character around 20 pixels wide. Thus, we would require a spatial error of smaller than 40 pixels.

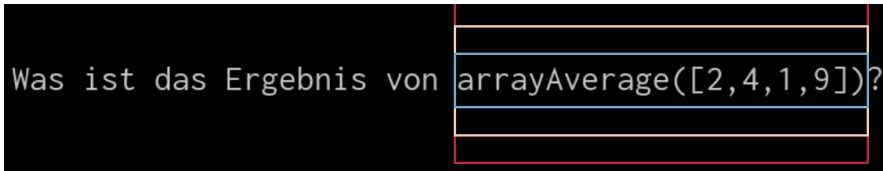


Figure 3.6: Visualization of Inner, 25-px Extra, and 50-px Extra area-of-interests (AOIs) around the task identifier of Figure 3.5 for the AOI analyses of RQ 3.2.

	Inner AOI	25-px Extra AOI	50-px Extra AOI
Fixation Count	1 266 (46%)	2 244 (81%)	2 772 (100%)
Fixation Length (sec)	2 215 (48%)	3 885 (83%)	4 661 (100%)

Table 3.2: Summary of all fixation counts and lengths within AOIs around task identifiers. Number in brackets is the overall percentage

3.2.4.3 Calibration Validation

Initially, we conducted a calibration validation to estimate the spatial error, which showed an average error of 0.99° , or 22 pixels (horizontally) and 26 pixels (vertically). The estimated horizontal spatial error of 22 pixels during calibration validation indicates that the eye-tracker is precise enough to detect fixations on words, but not on single characters. The estimated vertical spatial error of 26 pixels during calibration validation is problematic for us as it is close to the used line height of 40 pixels. With such a spatial error, we might erroneously classify a fixation on an incorrect code line.

The calibration validation only estimates the spatial error at the beginning of the experiment. The fMRI block design prohibits us from repeating the calibration validation multiple times during the experiment. Because the experiment includes a rest condition with a centered fixation cross around every minute, we can use this condition as a continuing spatial-error estimation throughout the experiment. Our analysis shows that if the eye-tracker consistently captured frames, spatial precision was fairly stable throughout the full experiment of 30 minutes.

3.2.4.4 AOI Analysis: Overview

To analyze whether the assumed vertical spatial imprecision can lead us to incorrectly classify a fixation, we conducted an area-of-interest (AOI) analysis on the task description at the top of each presented code snippet. We added three AOIs with different heights around the instructed function call, which we visualize in Figure 3.6. The inner AOI includes only the line of the function call in question, while the extra AOIs span, respectively, 25 and 50 vertical extra pixels. Because there is nothing above or below the task line, the eye-tracker should not record more fixations in the extra AOIs than the inner AOI. Assuming there is no human error and no technical spatial error, all fixations should be on the inner AOI directly on the task line. Every fixation on one of

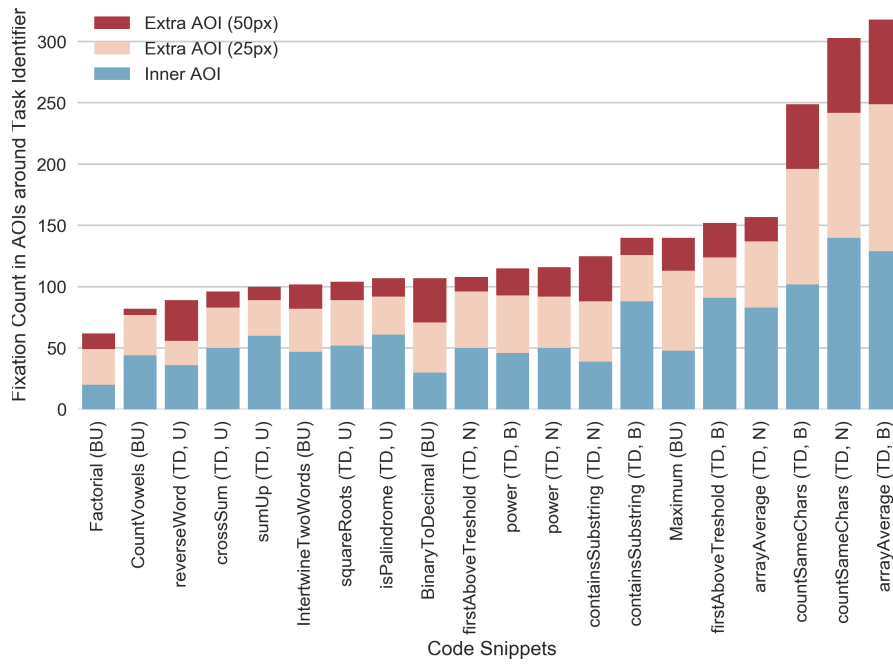


Figure 3.7: Fixation count for each AOI and code snippet

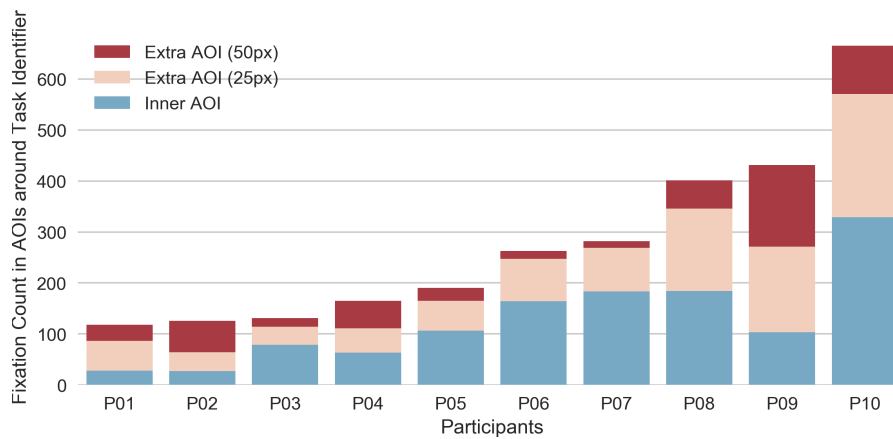


Figure 3.8: Fixation count for each AOI and participant

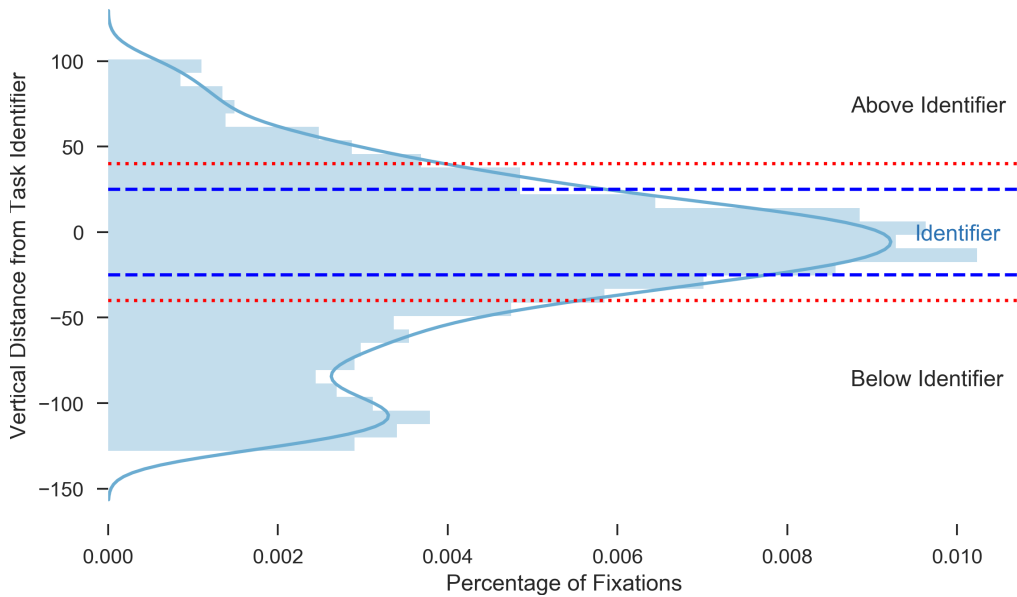


Figure 3.9: Distribution of vertical distance from fixations on task identifier. --- indicates current line height. indicates suggested line height

the extra AOIs could be later interpreted as a fixation on an incorrect line, and thus on a wrong identifier. We applied the same AOI sizes to all 20 comprehension snippets and, again, analyzed all eye-tracking data sets with, at least, 65% of recorded frames ($n = 10$).

In Table 3.2, we summarize the results of the AOI analysis. Over all comprehension snippets and participants, there are 1266 fixations, spanning, in total, over 2215 seconds on the inner, correctly sized AOI. On the two extra AOIs, which may lead to incorrectly classified fixations, there are, respectively, 2244 and 2772 fixations. Thus, only less than half of the fixations around the task identifier are actually detected on the actual task identifier line. 978 extra fixations are detected within 25 pixels, and another 528 fixations within the next 25 pixels. That is, at the used line height of 40 pixels, we cannot confidently detect fixations on an identifier level, because a vertical spatial error of 50 pixels is too high.

3.2.4.5 AOI Analysis: Snippets

To mitigate possible outliers distorting our AOI analysis result, we further divided the data for each comprehension snippet. In Figure 3.7, we show the number of fixations on all three AOIs for each snippet. While there are large differences in absolute fixation count, the relative sizes between the three AOIs is similar. We can conclude that it is a systematic error independent of the appearance of each individual snippet.

3.2.4.6 AOI Analysis: Participants

In the next step, we separately analyzed each participant to eliminate the chance that an individual participant distorts our AOI analysis result. Figure 3.8 shows that there are significant differences between our participants regarding fixation count and eye-tracking accuracy. Nevertheless, even for participants for whom we captured accurate eye-tracking data, we observe a notable amount of vertical spatial error. This supports the conclusion that 40 pixel line height is insufficient to prevent the incorrect classification of fixations.

3.2.4.7 Optimal Line Height

Lastly, after all evidence points toward a need for larger than 40 pixel high lines, we need to find the optimal line height. Larger font size and line spacing would create more vertical distance between lines and would let us be more confident when detecting fixations. However, with increasing the line height, the amount of code that we can display on the screen will be reduced (as we currently do not permit scrolling). To find a balance, we analyzed the vertical distance from each fixation around the task identifier. The maximum vertical distance considered in this analysis was three line heights in each direction (i.e., 120 pixels above and below the center of the identifier).¹⁰ We analyzed the same 10 participants with more than 65% of recorded frames. Figure 3.9 reveals that the current line height catches many fixations, but there is a significant number of detected fixations right below and above the line. Based on this result, if we would increase the line height and spacing to 80 pixels, we could be confident that fixations are classified on the right line.

RQ 3.2: Without correction, an estimated vertical spatial error of 30–50 pixels is too high with the used line height of 40 pixels to confidently detect fixations at the level of individual identifiers. An increase to 80 pixel line height would allow us to do so.

RQ 3.3: Is there a drift throughout the experiment?

To answer RQ 3.3, we analyzed the spatial error from the fixation cross during each of the 25 rest conditions throughout our experiment. To obtain an accurate picture of the drift over time, we included only participants for which the eye-tracker could consistently track the eyes ($\geq 85\%$ of frames, $n = 8$). We excluded the first half second of fixations during the rest condition, as participants were still concentrated on the previous task and needed some time to move their gaze to the fixation cross. We also excluded all fixations off the screen (e.g., participants looking above the screen at the eye-tracking camera) or with an absolute spatial error of larger than 300 pixels (e.g., participants looking around).

¹⁰Note that the increased fixations of 80 pixels and more below the identifier are due to fixations on actual code. For some snippets, there was only an 80-pixel distance between task instruction and code.

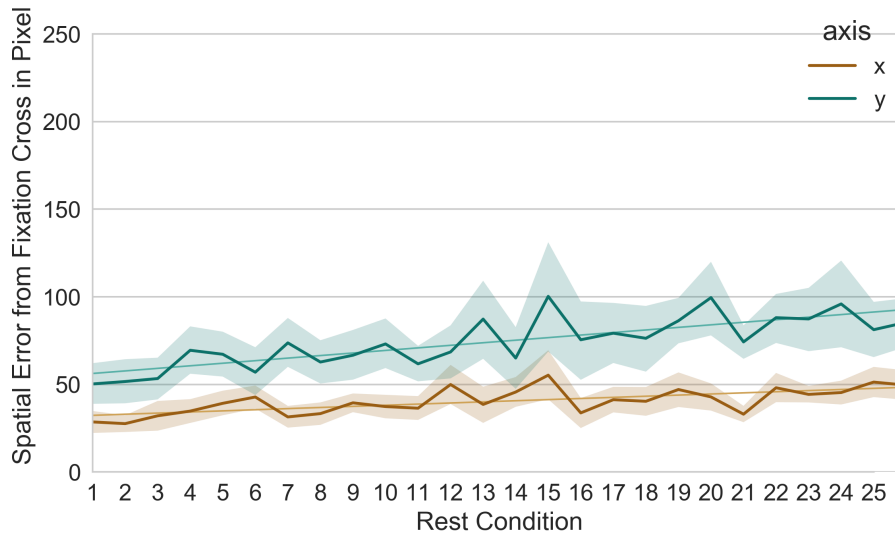


Figure 3.10: Spatial error of rest condition’s fixation cross over time. Standard deviation is shown as shades.

Figure 3.10 shows the spatial error for the x-axis and y-axis over time. The observed spatial error based on the fixation cross largely confirms the estimated error of our validation (cf. RQ 3.2). A horizontal spatial error of around 25 pixels substantiates previous estimates. The vertical spatial error of initially around 55 pixels is higher than the general validation error, but consistent with the center middle’s validation error (average error of 47 pixels). Overall, the spatial error is slowly growing, but mostly stable throughout the experiment. The estimated spatial error is slightly worse at the end of the experiment, but the eye-tracking data are still usable.

Figures 3.11 and 3.12 respectively show the spatial errors on the x-axis and the y-axis for each participant over time. For most participants, the horizontal spatial error is stable throughout the entire experiment. We believe some individual outliers (e.g., rest condition 15) can be attributed to participants looking around and not due to technical errors. However, the vertical y-axis reveals a different result. For some participants, the spatial error is consistently small enough to provide a useful data set. Nonetheless, for some participants, we observe a large spatial error of more than 100 pixels from the beginning, which was not evident during the calibration validation.

RQ 3.3: While there is a drift throughout the experiment it is negligibly small compared to the general spatial error. A split of an fMRI session in multiple sections with intermittent re-calibration is thus not required.

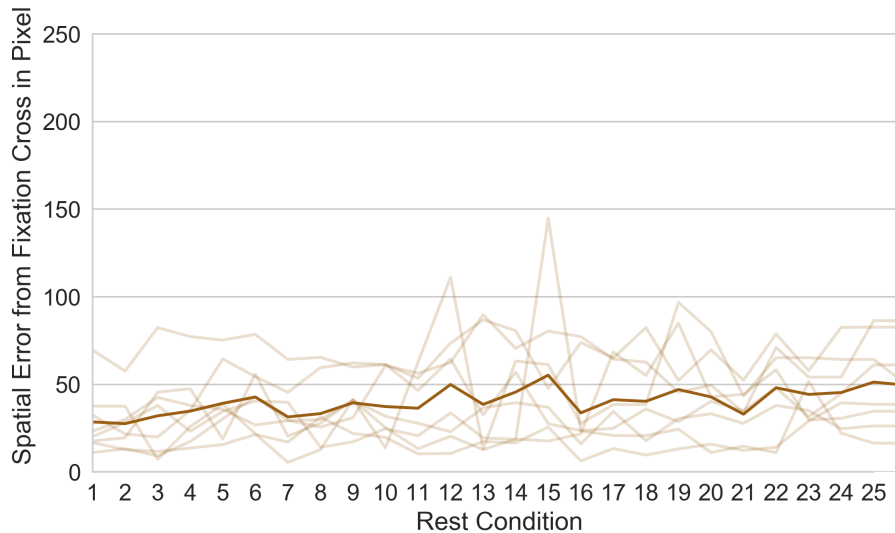


Figure 3.11: Spatial error on x-axis over time for 8 participants with complete eye-tracking data

3.2.5 Discussion

Having presented our results, we now present our lessons learned and discuss implications from the first study using simultaneous fMRI and eye tracking.

3.2.5.1 Eye-Tracking Optimizations

When we fine-tuned the eye-tracking setup in several pilot sessions, we knew that a perfect 100% output of the eye-tracker is out of reach. Nevertheless, that the eye-tracker could only reliably capture eye movements for 40% of our participants was unexpectedly low. After the study was conducted, we investigated further optimizations of the eye-tracking camera vendor, but did not find any substantial improvements.

In a later study, we investigated a fall-back solution for participants who show problems during calibration with the EyeLink 1000. We used a video-based MRC 12M camera¹¹ that is directly positioned on the head coil. Depending on the position, it can either record facial expressions (cf. Figure 3.13) or provide a close-up view of one eye. With an initial calibration, we can roughly infer the eye-gaze position. It does not provide the same accuracy, precision, or temporal resolution as the EyeLink eye-tracker, but appears to be sufficient for our AOI-based analyses.

In addition to the low recording success rate, our data showed a significant spatial error, especially on the vertical axis. In all subsequent fMRI studies, we used a refined calibration procedure of a 13-point calibration. We saw a small improvement, because spatial errors are smaller around the calibrated points [Hol+11]. This way, we were able to increase spatial precision. Moreover,

¹¹MRC Systems GmbH, Heidelberg, Germany, <https://www.mrc-systems.de>

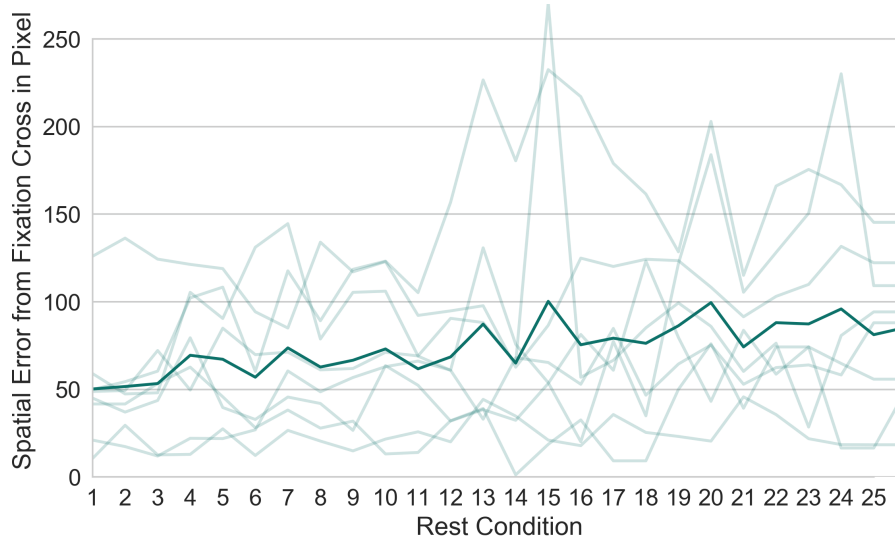


Figure 3.12: Spatial error on y-axis over time for 8 participants with complete eye-tracking data

we decreased the validation thresholds when we accept a calibration as successful to further improve spatial precision [Hol+11].

While the refinements in our setup and calibration may improve spatial precision, we still saw the need for optimizing the display of our code snippets. A significant point of action is the font size and line spacing. When we designed our study, we based it on our previous successful fMRI study design, which did not optimize the code-snippet display for the eye-tracking modality. In follow-up studies, we maximized fonts and line spacing within a reasonable size (without the need for scrolling). This way, we mitigated inevitable spatial errors and be more confident when detecting fixations, for example, on concrete identifiers.

In our study, the drift estimated with the fixation cross appears small enough to reject the need to split the fMRI run into multiple sessions. Nevertheless, if precise eye tracking is necessary (e.g., fixations are used as an input method), an intermittent re-calibration and validation may be effective.

3.2.5.2 Challenges and Benefits

Eye tracking in an fMRI scanner is challenging in multiple respects. Due to the suboptimal angle of an eye-tracker outside the bore, it is sensitive to half-closed eyes, which leads to incomplete data sets. In general, the same eye-tracking camera shows more accurate results outside of the fMRI environment [Han+16]. The increase of spatial errors in comparison to conventional eye-tracking experiments has to be kept in mind when designing code stimuli (and their font sizes, line paddings, etc). Our analysis provides a rough orientation on suitable parameters.

Our study also exemplified a typical problem of multi-modal experiments. Both of our measures, fMRI and eye tracking, have exclusion rates, where some participant's data cannot be used, for

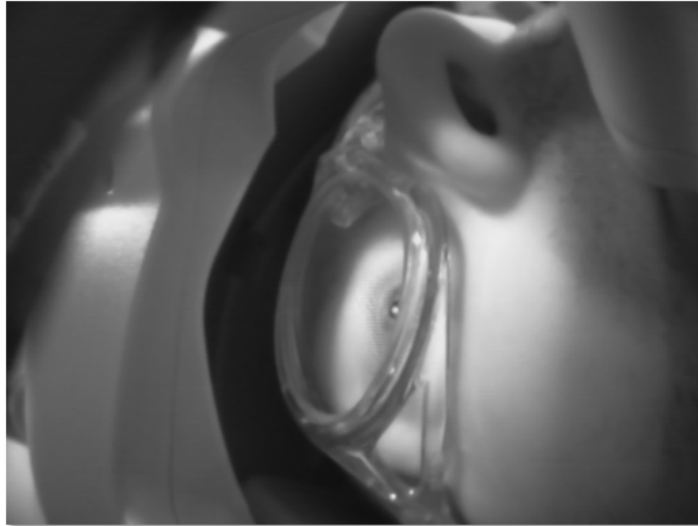


Figure 3.13: MRI-compatible video camera of a participant's face. Participant is wearing optional MRI-compatible glasses. Alternative angles, such as direct focus on the eye, are possible.

example, due to motion artifacts (fMRI) or half-closed eyes (eye tracking). In the presented study, only 17 of 22 (77%) fMRI data sets and 10 of 20 (50%) of eye-tracking data sets were usable. When considering both measures, only for 7 of 22 (32%) participants we obtained a complete data set (i.e., fMRI and eye-tracking data are both usable). Thus, for simultaneous recording (e.g., an fMRI study that uses eye tracking as input), this has to be kept in mind during planning. We may need to double the number of participants.

However, we believe the higher effort is worth it, because complete data sets, including both fMRI and eye-tracking data, provide remarkable insights. We can test individual hypotheses, for example, whether participants' fixations on a beacon (detected with eye tracking) is linked to a semantic recall (increased brain activation detected with fMRI). Specifically, we found the effect of increased brain activation in BA21 from our previous study again providing evidence for our hypothesis [Sie+17]. By adding simultaneous eye tracking, we were now also able to detect prior fixations on beacons. Initially, we replayed the eye-tracking data with Ogama to qualitatively detect fixations on task identifiers. Then, we used an AOI to identify fixations specifically for each participant and snippet. We fed the detected fixation timestamps into our fMRI analysis for a more fine-grained result. Thus, eye tracking is the basis to advance from a coarse block-based fMRI analysis to a more detailed event-related analysis where we can distinguish fine-grained effects of program comprehension (such as semantic recall after fixation on beacons, as we discuss in Section 3.3.2). Overall, we were able to confirm our hypothesis of semantic recall in BA21 in this study.

Simultaneous fMRI and eye tracking offers possibilities beyond testing particular hypotheses, such as understanding programmers' behavior better. For example, Figure 3.7 shows that, for all trained top-down comprehension conditions, participants fixate more often on the task instruction,

revealing that they focus more on the result computation than comprehending a code snippet. Thus, further supporting how beacons ease program comprehension so that participants can quickly shift their attention on result computation. Moreover, we can generate new hypotheses by exploring the data (e.g., mental loop execution leads to increased working memory activation in BA6) [Pei+18b]. Both measures combined offer a powerful way to observe and understand program comprehension, which we describe in further detail in Section 3.3.2.

3.2.5.3 fMRI Experiment Framework

Overall, our study showed that simultaneous fMRI and eye tracking is possible, but challenging. We demonstrated that it is feasible and already provides insightful data and therefore concluded that our fMRI experiment framework is within reach. Based on our study's experience, we can encourage future fMRI studies to also include eye tracking as it can notably increase insights from a study.

Nevertheless, there is a large extra effort for the principal investigator to design an fMRI study with integrated eye tracking. The selection of an appropriate eye-tracking solution, technical setup, stimuli preparation, and implementation of an analysis pipeline is time-consuming. To support future endeavors of the research community, we share all of our materials and scripts.

3.2.6 Threats to Validity

Construct Validity We operationalized eye-tracking precision by identifying fixations on code stimuli from previous fMRI studies. It is likely that a study specifically targeted at testing eye tracking in an fMRI environment, say with only small visual inputs all across the screen, would have lead to a more reliable and accurate answer. However, the results may not have been applicable to our study purposes (e.g., detecting fixations on code identifiers, finding the correct line height of code).

A post-processing correction of eye-tracking data is somewhat typical, where the fixations are manually changed to fit the stimuli. This correction is often done by hand [Coh13], even though automatic approaches are being evaluated [PS16]. We did not apply such correction, as it may substantially influence the results and insights. However, it is possible that the imprecision explored in RQ 3.2 can be reduced with such a correction, and thus our result, a recommended line height of, at least, 80 pixels, is excessive. This is a first estimation based on our data, and it may be reduced in future studies.

Internal Validity The nature of controlled fMRI program-comprehension experiments leads to a high internal validity, while reducing external validity [SSA15]. For the presented technical analysis, we only see the number of participants ($n = 22$) as a threat to internal validity, because the

eye-tracking data quality is highly dependent on an individual and we may not have captured an accurate representation of the population.

Statistical Conclusion Validity The analyses for RQ 3.2 and RQ 3.3 were conducted on a single area/point of the screen (identifier on top of the screen and fixation cross in the middle center, respectively). While the spatial precision slightly changes throughout the screen, we estimate the chances for a significantly different result on other screen parts as low.

External Validity We conducted the fMRI study at a single location. Another environment (e.g., different fMRI scanner or eye-tracking solution) may lead to better (or worse) results. For example, eye tracking built-in to the head coil¹² may offer higher precision than a long-range camera-based eye-tracker outside the scanner bore, but in turn may decrease fMRI data quality. We later used an additional MRI-compatible video camera¹³ mounted on the head coil to observe participants' eyes.

3.2.7 Related Work

Program comprehension is an established research field in software engineering. Nevertheless, neuroimaging and psycho-physiological measures are still novel methods in our field. Closest to our study is the study by Duraes et al., who observed debugging with fMRI and eye tracking [Dur+16]. It is unclear, though, what kind of eye-tracking data were recorded, and they did not appear to use the eye-tracking data. At the time, all other fMRI studies only observed brain activation data [Sie+14a; Sie+17; FSW17].

After our experiment framework was published, several studies started to use simultaneous fMRI and eye tracking. Specifically, Castelhana et al. investigated experts during debugging and (separately) analyzed fMRI and eye-tracking data [Cas+19]. Huang et al. first investigated mental rotation and data structure manipulation [Hua+19] and then code review [Hua+20], both with fMRI and eye tracking. In a follow-up paper, the team supported our view of the potential upside of a multi-modal experiment framework [Sha+20a].

3.2.8 Conclusion

Observing programmers with simultaneous fMRI and eye tracking opens the door to a more holistic understanding of program comprehension. In this section, we reported that simultaneous measurement of program comprehension with fMRI and eye tracking is challenging, but promising. In the first study of its kind, we were able to gather simultaneous fMRI and eye-tracking data,

¹²For example, Real Eye™, Avotec, Inc, <http://avotecinc.com>

¹³MRC Systems GmbH, Heidelberg, Germany, <https://www.mrc-systems.de>

although the overall success rate was relatively low. We found that the eye-tracker's spatial imprecision in the challenging fMRI environment can be controlled sufficiently to confidently detect fixations on identifiers if we design snippets properly. The drift throughout the experiment was so small that it is not an issue for our studies.

3.3 Toward Multi-Modal Data Analysis of Program Comprehension

So far, we have investigated simultaneous fMRI and eye tracking, and with the latter limited to its eye-gaze data. In this section, we integrate further eye-tracking measures as well as additional psycho-physiological modalities to our experiment framework.

3.3.1 Advanced Eye-Tracking Measures during an fMRI Study

First, we explore whether pupil dilation and blink rates offer insights in studying program comprehension. To this end, we re-analyze the collected eye-tracking data from the previous section. Our long-term goal of observing pupil dilation and blink rates in addition to brain activation via fMRI is to detect cognitive events of smaller granularity. While fMRI allows us to observe programmers' cognitive load on a larger scope (e.g., difficulty to comprehend a Java method), observing the effect of comprehending individual lines may currently be impossible with fMRI. The temporal resolution is 1 to 2 seconds, which means that we may miss short-lived cognitive events, such as a programmer stumbling over an unexpected implementation of a single line. We aim to integrate pupil dilation to detect exact lines that cause programmers to struggle. Furthermore, pupil dilation and blink rates may offer additional measures to observe cognitive load and, as such, can help us to explain some of our fMRI results.

3.3.1.1 Objectives

The literature from psychology and cognitive science suggests that pupil dilation and blink rates may be valuable measures in future studies of program comprehension (cf. Section 2.3.1.2). To ensure robust results, we focus on participants with stable and accurate eye tracking.

Our goal for the exploratory analysis is to evaluate the following two research questions:

RQ 3.4 Can we consider pupil dilation and blink rates as a measure in future experiments?

RQ 3.5 Do we find the expected correlation between cognitive load and pupil dilation?

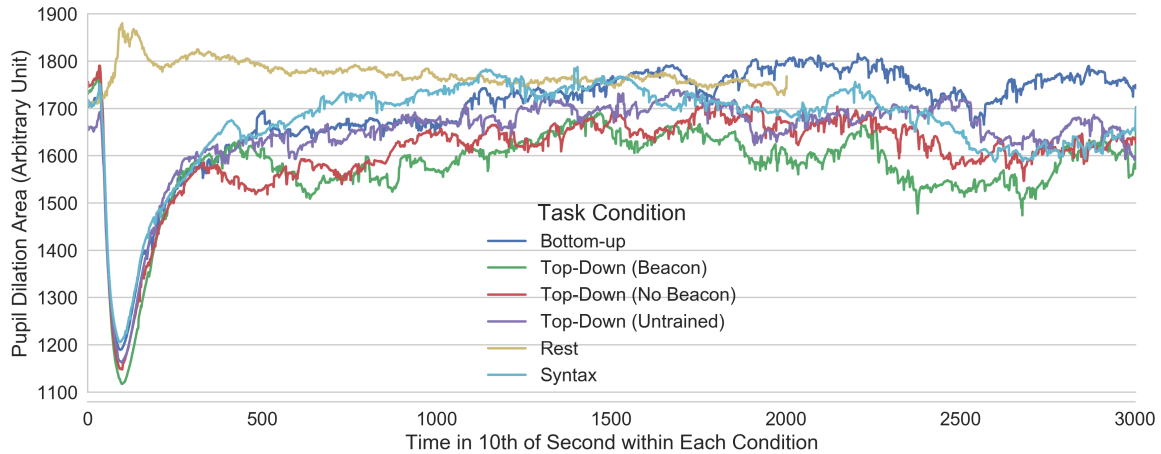


Figure 3.14: Pupil dilation over time for each condition. A large drop at the beginning of each task is a biological reaction due to the increased brightness of the screen.

3.3.1.2 Pupil Dilation

For comparison across participants, we normalized the pupil-dilation data for each participant to zero mean and unit standard deviation (z-score).

Screen-Brightness Correction Any analysis of pupil-dilation data assumes that a change in pupil dilation is only caused by a mental state transition and not by the environment (e.g., ambient light). The environment around the fMRI scanner ensures that the ambient light does not change. However, we present code snippets to the participants via a small plastic screen on which the stimuli are projected. To reduce eye strain, we use white text on a black background. As our snippets are not uniform in length, each snippet is different in its perceived brightness. Thus, the snippets may influence pupil dilation by their brightness, independent of the cognitive load change. This effect is clearly visible in Figure 3.14, where the adjustment from a dark rest condition to a brighter comprehension condition is apparent: The pupil dilation briefly drops at the beginning of the comprehension conditions as the pupils respond to the brighter light.

Because a standardized brightness of the stimuli is infeasible for us (i.e., snippets will generally differ in length), we will need to correct the baseline pupil dilation for each snippet. Figure 3.15 shows that there is a general trend to a lower pupil dilation with brighter snippets. We computed each snippet's brightness variable as the relative luminance of the RGB color space for each image (as an average of each pixel) [AF96]. The relative luminance is calculated based on the luminosity function, in which each color is weighted dependent on the human's perception (e.g., green color is perceived as much brighter than blue light):

$$\text{Relative luminance} = 0.2126 \cdot \text{red} + 0.7152 \cdot \text{green} + 0.0722 \cdot \text{blue}$$

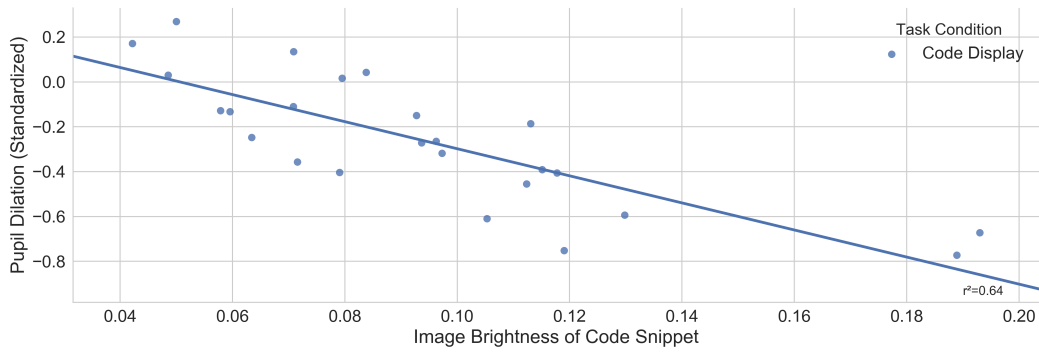


Figure 3.15: Correlation between snippet brightness and measured pupil dilation

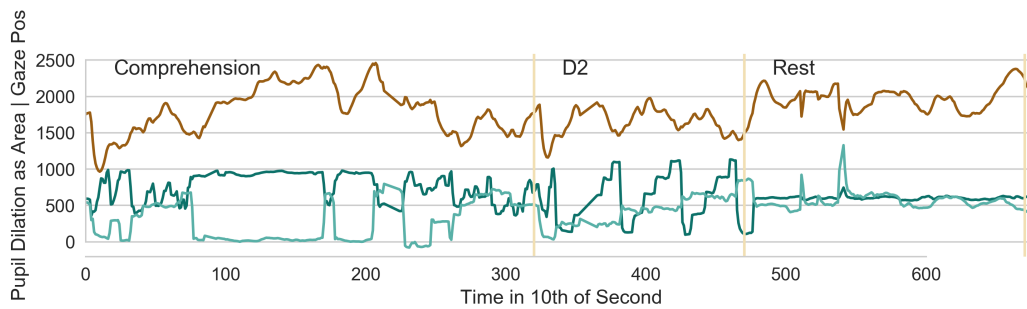


Figure 3.16: Visualization of pupil dilation, gaze position on x-axis and y-axis of participant 1 during the first minute of the experiment.

The $r^2 = 0.64$ value shows that the screen brightness largely explains the difference in baseline pupil dilation. However, there is still 36 % of variance that is not explained by screen brightness, which could include comprehension strategy (top-down versus bottom-up), individual difficulty comprehending a code snippet, or general error.

Eye-Movement Correction So far, we have considered the average pupil dilation throughout a task. To extract the maximum value from pupil dilation, we would like to evaluate pupil dilation changes within a task. Brisson et al. have shown that pupil size is overestimated for rightward and upward gaze and underestimated for leftward and downward gaze for our used EyeLink 1000 eye-tracker [Bri+13]. They recommend proceeding with methodological caution. Ideally, tasks are used that do not require any eye movement. However, this is infeasible for us, as program comprehension requires programmers to quickly move across the screen. Thus, long and fast saccades are necessary and common. The influence of eye movements on the pupil dilation questions how much we can trust the data, even if corrected for screen brightness.

In Figure 3.16, we show a pupil-dilation chart of an individual participant across a few tasks. A few significant spikes are noticeable (especially at the rest condition). The eye-tracker’s manual warns that a fast change in pupil angle can lead to a flawed pupil dilation measurement. Based

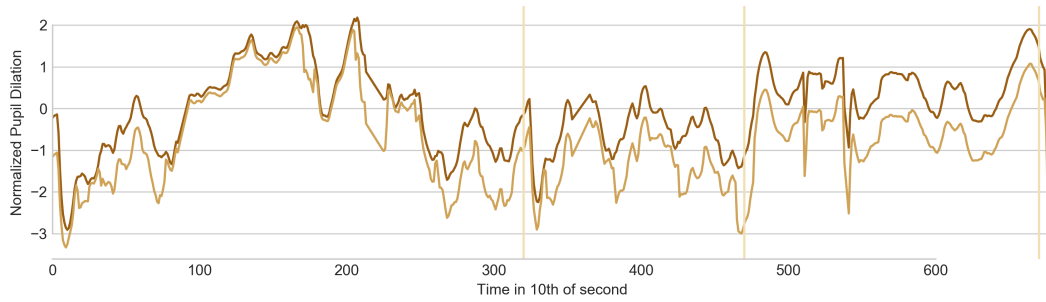


Figure 3.17: Visualization of **normalized pupil dilation** and **corrected pupil dilation** [Bri+13] of participant 1 during the first minute of the experiment.

Condition	Perceived Brightness	Pupil Dilation	Blink Rate	Blink Duration
Top-Down Comprehension (Beacon)	3.01	-0.49	12.0	245
Top-Down Comprehension (No Beacon)	3.07	-0.41	11.6	341
Top-Down Comprehension (Untrained)	2.09	-0.25	7.6	346
Bottom-Up Comprehension	2.13	-0.07	8.4	340
Locate Syntax Errors	1.89	-0.11	10.4	322
Distractor Task (d2)	2.15	0.16	10.1	327
Rest	0.01	0.31	7.9	490

Table 3.3: Mean perceived brightness, pupil dilation (z-score), blink rate (count/minute), and blink duration (ms) for each experiment condition.

on the data, we confirm that fast saccades impair the pupil-dilation accuracy. In Figure 3.16, it is visible that pupil dilation spikes correlate with eye movements, particularly on the vertical axis (e.g., at time 520 and 550). Movements on the x-axis, that is, leftward and rightward gazes, also seem to influence our pupil-dilation data. For example, the pupil dilation in Figure 3.16 for the distractor attention tasks appears (e.g., between time 320 and 480) to rhythmically move with the eye gaze. Thus, we cannot fully trust an individual raw data point of pupil dilation as it may be influenced by prior eye movements.

For future analysis, we would like to correct the pupil dilation for such eye movements. We evaluated the correction algorithm from Brisson et al. [Bri+13] to further improve the accuracy of the pupil-dilation data. However, when we applied the correcting algorithm to the previous case of participant 1, we observed a worsening of the estimated pupil dilation: the pupil-dilation offsets appear even larger in rhythm to eye movements (cf. Figure 3.17).

As we only consider average pupil dilation across an entire task as a potential approximation of cognitive load for the following sections, we did not apply the correction algorithm.

	Pupil Dilation	Blink Rate	Blink Duration
Code Complexity Metric: DepDegree	0.22	0.15	0.09
Code Complexity Metric: Halstead	0.38	0.07	0.04
Task Correctness	0.00	0.01	0.05
Response Time	0.01	0.12	0.00

Table 3.4: Correlation matrix (in r^2) of observed pupil dilation for each snippet’s code complexity and participant behavior.

Task Condition In a later chapter, we present an fMRI study that shows top-down comprehension eases programmers’ cognitive load (cf. Section 4.1, [Sie+17]). Can we support this finding purely based on pupil dilation? In Table 3.3, we show the average pupil dilation across all participants and tasks per condition. Top-down comprehension with or without a beacon reveals a difference in pupil dilation, even though the screen brightness is almost identical. This difference hints at participants using a beacon for an eased comprehension of a snippet. Bottom-up comprehension shows a slightly higher pupil dilation than untrained top-down comprehension, which may be due to the additional mental effort necessary to comprehend a snippet. The d2 attention task has a much higher pupil dilation, even though the brightness is similar to the bottom-up snippets. None of these findings are statistically significant as there were only five trials for each condition. As the other conditions are unbalanced in brightness, and we did not find a reliable way to correct for the screen brightness yet, we do not compare them.

Code Complexity Metrics Next, we correlated code complexity, measured in DepDegree [BF10] and Halstead complexity metrics [Hal77], with the observed pupil dilation. We show the results in Table 3.4. They indicate that, with increasing code complexity, the pupil dilation decreases, but only explains a part of the variance. We also must consider that code complexity is generally correlated with snippet brightness ($r = 0.56$), as longer snippets tend to be more complex. It appears that code complexity actually increases pupil dilation (when controlling for screen brightness), a result that is plausible based on the available neuro-scientific literature. In Section 5.1, we present a dedicated fMRI study to evaluate a possible relationship of program comprehension and code complexity metrics.

Task Difficulty Instead of considering the objective code complexity, we may observe a more significant difference in subjective difficulty. To this end, we correlated the averaged pupil dilation with the task correctness and response time. The results show no correlation (cf. Table 3.4).

Stepwise Regression Finally, we conducted a stepwise regression to select the significant variables that explain the variance in pupil dilation. The result revealed that screen brightness ($p < 0.001$) and response time ($p < 0.1$) significantly influence pupil dilation.

3.3.1.3 Blink Rates and Durations

Next, we analyzed the blink rates and blink duration (cf. Section 2.3.1.2). Our experiment in the fMRI scanner lasted for around 28 minutes. There was no notable increase in blink rates towards the end of the experiment. Fatigue does not seem to have set in after 28 minutes yet, and eye movements do not influence blinks, thus we can analyze the blink rates without correction.

Task Condition In Table 3.3, we show the average blink rate and blink duration for each task condition. We cannot identify any clear-cut patterns. However, it is notable that top-down comprehension with a beacon has a shorter average blink duration than any other comprehension task.

Code Complexity Metrics We correlated the observed blink rate and duration with code complexity, again measured in terms of DepDegree and Halstead complexity metrics. The results presented in Table 3.4 indicate that, with increased code complexity, the blink rate decreases, but the effect explains only a minor part of the variance. Blink duration has the same negative correlation and explains less variance.

Task Difficulty We also correlated the observed blink rate with the task correctness and response time. We found a weak negative correlation between blink rate and task correctness. The same holds for blink duration (cf. Table 3.4).

3.3.1.4 Conclusion

We set out to include pupil dilation and blink rates as promising measures to increase the reliability of measuring program comprehension with fMRI. However, our exploratory analysis showed that, so far, pupil dilation largely depends on screen brightness and that blink rates and blink durations do not seem to follow a pattern, which leads us to answer our research questions:

RQ 3.4 We generally could consider pupil dilation and blink rates as a measure for program comprehension, but need to find a way to correct for the influence of different snippet lengths on pupil dilation.

RQ 3.5

The data indicate that for different snippet versions with identical screen brightness, the pupil dilation varies (e.g., depending on meaningfulness of identifiers), which suggests an effect of cognitive load on pupil dilation.

In conclusion, our analysis confirms our belief that there is value beyond the usual eye-gaze-based analysis of eye-tracking data. Pupil dilation and blink rates may provide fascinating insight for studies of program comprehension. However, before we can use them in an fMRI scanner, we need to find ways to control for environmental factors, such as screen brightness, and develop appropriate analyses.

3.3.2 Conjoint Analysis of Multi-Modal Data

In the previous sections, we explored that simultaneous measurement of programmers with fMRI and eye tracking offers detailed insight toward a more holistic understanding of program comprehension. We described that integrating simultaneous and accurate eye tracking into our fMRI experiment framework is challenging on a technical level. We resolved the method synchronization details and successfully conducted a full study, which revealed difficulties regarding the eye-tracking data quality (cf. Section 3.2).

However, acquiring simultaneous brain and eye-movement data is only half the problem: The next challenge is to actually apply fruitful analyses to the two data streams. They could be viewed independently (e.g., observe top-down and bottom-up comprehension and separately compare brain activation strength and fixation count). However, in our view, the true value lies in integrating observations from the two separate data streams (i.e., a conjoint analysis of both data streams).

An analysis of simultaneous fMRI and eye tracking is challenging, because the two data streams have fundamentally distinct characteristics. They are not just different in temporal resolution, but fMRI relies on the haemodynamic response, which means observed brain-activation data are delayed by around five seconds [Cha+93]. A conjoint analysis of instant eye-tracking data and delayed fMRI data will need to take this into account.

In our view, the future of program-comprehension research will exemplify this struggle of creating valuable analysis as we are moving toward multi-modal experiments. While this section focuses on combining fMRI and eye-movement data, it also applies to our prior expansions of our fMRI experiment framework. Specifically, we experimented with recording pupil dilation and spontaneous blink rates (Section 3.3.1) and physiological data (i.e., heart rate, respiration, and electrodermal activity, Section 3.5.1), which also acts as an indicator of cognitive load. Thus, we will have numerous characteristically different measures observing the same cognitive process in the upcoming future. We will make the first step into aligning them in the next section for our multi-modal data exploration tool (cf. Section 3.4).

We again reconsider the data from our previous fMRI experiment with simultaneous eye tracking presented in Section 3.2.

3.3.2.1 Strategies for Data Analysis

As far as we are aware, there are no established procedures to conjointly analyze simultaneous fMRI and eye-tracking data. In this section, we present three possibilities on how we may benefit from simultaneous fMRI and eye-tracking data.

Informed fMRI Analysis We have the vision of an eye-tracking-informed fMRI analysis for program-comprehension studies. Currently, a comprehension task of 30 to 60 seconds is viewed as one black box of program comprehension. However, it actually consists of many smaller phases and cognitive subprocesses with varying intensity. In such eye-tracking-informed fMRI analysis, we would like to use eye tracking to gather information on the programmers' behavior and then feed it into a general linear model (GLM) analysis of the fMRI data. This way, we can unlock the black box of program comprehension. The fMRI data analysis could be more fine-grained and allow us to separately evaluate phases of, for example, identifier recognition, loop execution, and result computation. Such detailed analysis may help us to understand the brain activation for each phase in more detail, and in the long-term, make fMRI experiments more valuable to our community.

Hypothesis Generation We can use the two data streams to observe a programmer's behavior to generate new hypotheses. Investigating programmers individually with fMRI or eye tracking already offers an interesting perspective on programmers' minds. If we can simultaneously explore both data sets, for example, by a real-time video replay, we may generate new hypotheses of program comprehension. An intuitive exploration of fMRI and eye tracking and, if applicable, also behavioral data, pupil dilation, and physiological data, would allow us to delve into many aspects of program comprehension, such as stress levels. We show an implementation toward this goal in Section 3.4.

Hypothesis Testing A further possibility is to analyze the data to test specific hypotheses and cover another step in the scientific method. Hypothesis testing is essential, as each measure offers unique insights into program comprehension. For example, we initially found no significant effect of meaningful identifiers in our study on top-down comprehension when analyzing the averaged brain activation strength across the entire 30-second task length (cf. Section 4.1, [Sie+17]). However, Figure 3.2 illustrates a stronger activation in a language-processing brain area between 5 and 10 seconds after task onset, which fits the typical delay of the haemodynamic response [Cha+93]. With simultaneous eye tracking, we could evaluate our hypothesis that programmers fixate on a meaningful identifier and recall a matching programming plan.

In general, we may detect specific events with eye tracking indicating particular behavior (e.g., mentally executing a loop) and use that as starting point for the resulting haemodynamic response of brain activation. Such individual analysis of cognitive (sub)phases of program comprehension would allow us to test more detailed hypotheses on the programmers' brains inner workings.

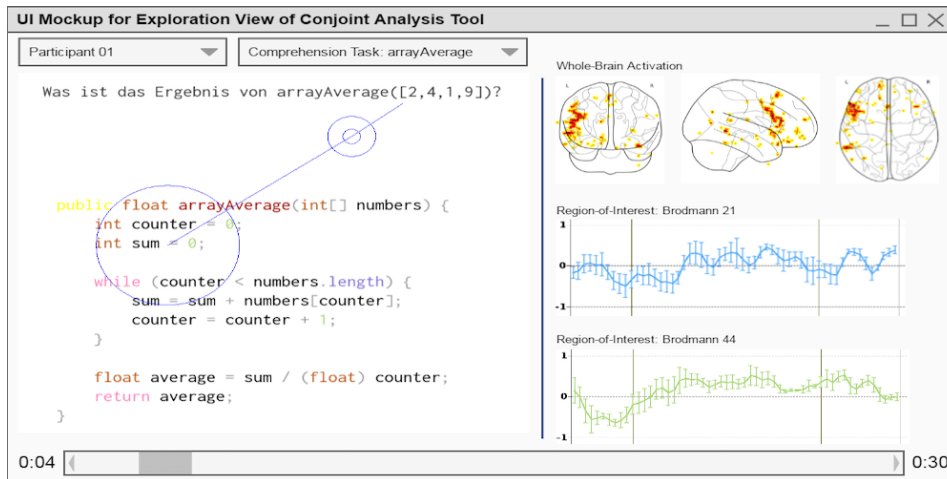


Figure 3.18: Mockup of tool user interface for data exploration

3.3.2.2 Envisioned Tool Support

After presenting our vision for multi-modal experiments and, more specifically, our ideas for conjoint analyses of fMRI and eye-tracking data, we would like to stress a clear-cut need for proper tool support for each of the three analyses. Ogama is an established eye-tracking analysis tool [Voß+08], which provides capabilities for replaying recorded data, heat maps, area-of-interest (AOI) analysis, and statistical analysis (e.g., fixation counts, saccade lengths, regressions). We see the need for a dedicated open-source tool that similarly sets the base for simultaneous fMRI and eye-tracking experiments and, in the long term, for multi-modal experiments of program comprehension.

We see the need for two different modes: Data exploration and data analysis, which we describe further in the following paragraphs.

Data Exploration To support hypothesis generation based on exploring observations of fMRI and eye tracking, the envisioned tool needs to provide a compelling data-exploration view. To implement such a view, the tool should be able to import the brain activation data, either as raw data or pre-processed, and the eye-tracking data. The tool would need to take the haemodynamic response, which delays the brain activation by several seconds, into account. We cannot directly map the instant eye-tracking observation to the delayed brain activation.

Moreover, the exploration view should be configurable and provide different views. For example, on the eye-tracking side, it may offer a scanpath or fixation view. On the brain-activation side, it may offer a whole-brain activation view or a focus on specific regions-of-interest. We created a mock-up for our vision in Figure 3.18, which presents eye-tracking replay (left side), whole-brain activation (top-right), and a selected subset of brain areas (bottom-right).

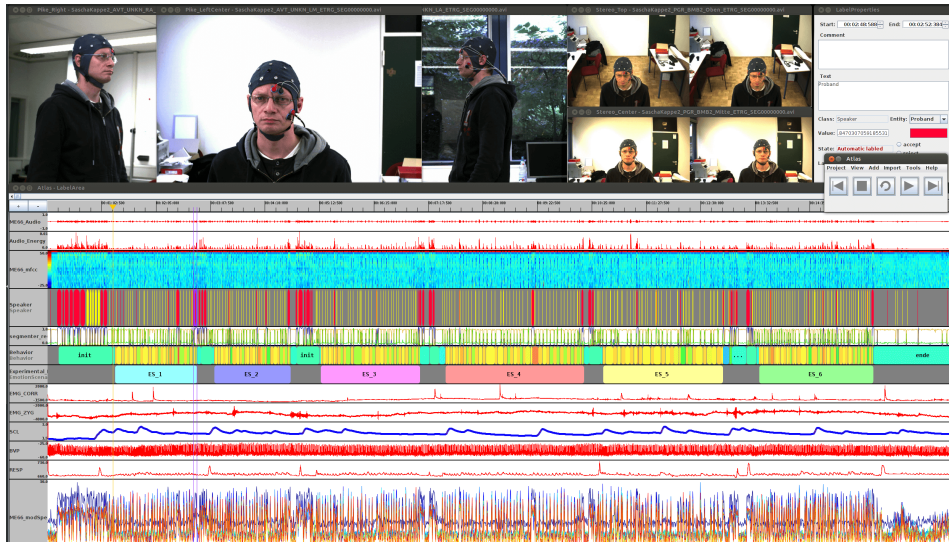


Figure 3.19: Screen of multi-modal annotation tool ATLAS [MBS12]

While we focused in this section on eye-tracking and brain-activation data, future studies will move toward multi-modal experiments. We experimented with other data streams: pupil dilation, spontaneous blink rates, and physiological data (heart rate, respiration, electrodermal activity). We potentially may integrate simultaneous EEG as well. Thus, in the long-term, the tool should be further generalized beyond fMRI and eye tracking. This way, it would be flexible and support any multi-modal view of the data, depending on the collected data set.

Data Analysis The second mode of the tool should cover hypotheses testing and informed fMRI analysis. To support these analyses, we split the requirements in two parts: (manual or automated) data annotation and informed fMRI analysis.

Manual Data Annotation To conduct an eye-tracking-informed fMRI data analysis, we would need to annotate eye-tracking events, such as a fixation on a meaningful identifier or mental loop execution. In the first stage, we could manually detect eye-tracking events. The proposed tool should allow us to manually annotate data streams. We envision a user interface similar to ATLAS, which is a tool for annotating multi-modal data streams of human-computer interaction experiments [MBS12]. Figure 3.19 shows ATLAS with multiple data streams. Initially, the tool may only support fMRI and eye tracking, but eventually could be extended to support multi-modal annotation (e.g., behavioral or physiological data).

Automated Eye-Tracking Event Detection The manual detection of events in the first stage introduces human error and inconsistencies to the data annotation. Thus, in the long term, we would prefer to automatically detect eye-tracking events. We would need to describe the event

criteria in a flexible manner and store them in a database. Similarly, much like Ogama allows researchers to define and store AOIs, we would need to describe complex eye-tracking events (e.g., fixation for at least half a second on an AOI with a meaningful identifier). As our experiments do not permit scrolling, we can still use static AOIs. Once we extend our studies to allow scrolling, and thus the code display is dynamic, we may need to implement an automated detection of AOIs similar to Barik et al. have done [Bar+17].

Informed fMRI Analysis An important feature of the envisioned tool would be connecting the annotated eye-tracking data to an fMRI data analysis tool. For example, the tool could feed the insights as a parameter input into *nipype*, a Python-based neuroimaging data processing tool [Gor+11]. *Nipype* could, with the event-enriched data, create an individualized GLM model and enable a thorough analysis of the fMRI data.

3.3.2.3 Conclusion

In this section, we have discussed why a conjoint analysis of fMRI and eye-tracking data is insightful but also challenging. Once we can conjointly analyze both data streams, we may generate new hypotheses for program comprehension, test existing hypotheses, and eventually create a more holistic theory of program comprehension.

To properly support all these goals, we see the need for extensive tool support to facilitate a more fine-grained analysis of brain activation and to increase the fMRI studies' value on program comprehension. In the following section, we present our prototype for such tool support.

3.4 Tool Support for Multi-Modal Data Exploration

After outlining the need for tool support to explore and analyze multi-modal data in the previous section, we present our open-source prototype implementation *CODERSMUSE*. The prototype is designed to allow researchers a synchronous exploration of all relevant data streams. Specifically, in addition to brain activation, we also collect behavioral data (i.e., response time and correctness), eye-tracking data, and psycho-physiological data (i.e., heart rate, respiration, electrodermal activity). With *CODERSMUSE*, we can jointly explore all data streams instead of relying on individual tools and analysis for each data stream. This way, *CODERSMUSE* enables us to generate substantially new and more holistic hypotheses for studies of program comprehension.

Data	Type	Delay	Temporal Resolution	Typical Preprocessing	Typical Measurement
	<i>Important for visualization</i>	<i>Important for data synchronization</i>		<i>Done before import to CODERSMUSE</i>	
Behavioral	Individual events	None	n/a	Coding	Response time, correctness
Eye tracking	Data stream	None	Down to less than milliseconds	Smoothing, saccade, and fixation detection	Eye gaze
Physiological	Data streams	Few seconds	Typically milliseconds	Smoothing	Stress level
fMRI	3-dimensional set of data streams	Several seconds	Typically a second	3D-motion correction, slice-scan-time correction, temporal filtering, spatial smoothing	Brain activation

Table 3.5: All data streams supported in CODERSMUSE and their characteristics, typical preprocessing, and measurements.

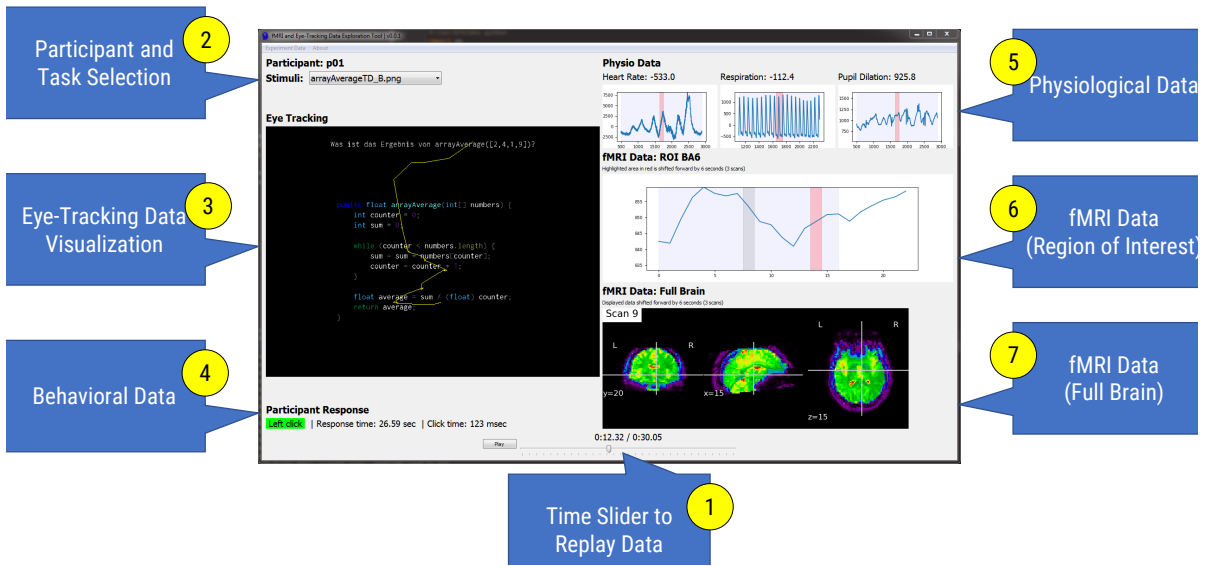


Figure 3.20: Screenshot of *CODERSMUSE*. On the top right, the user can select a specific task ②. A time slider allows the user to explore data across a task’s timeline ①. Then, several data streams will be displayed: eye-tracking data ③, behavioral data ④, psycho-physiological data ⑤, and fMRI data ⑥ ⑦.

3.4.1 Prototype Implementation of *CODERSMUSE*

In Figure 3.20, we show a screenshot with annotated feature explanations, which are numbered in yellow circles ① and which we explain in detail in this section.¹⁴

3.4.1.1 Integrated Modalities

Currently, *CODERSMUSE* supports four different modalities, of which we provide an overview in Table 3.5.

Behavioral Data ④ Behavioral data include events derived from a participant’s actions, including response correctness, response time, and click duration (i.e., time between button-down and button-up events).

¹⁴We provide a demo video and an open-source implementation with sample data on the project’s Web site.¹⁵

Eye-Tracking Data ③ Eye-tracking data are visualized on top of an image of the current task (e.g., a code snippet). We currently support static images, which is the most common type in eye-tracking experiments. Dynamic content (such as scrolling on a Web site) is not supported, but may be added in the future. Eye-tracking data are visualized as a *scanpath* [Hol+11]. Fixations and saccades are highlighted with different colors. Other visualizations, such as heat maps, can be added in the future.

fMRI Data fMRI data provide insights into the underlying cognitive processes during program comprehension. *CODERSMUSE* supports two visualizations of fMRI data. First ⑥, it shows the brain activation strength over time for a specific brain region of interest (e.g., to observe working memory load during a task). Second ⑦, *CODERSMUSE* visualizes the full-brain activation with the Python library *nipy*¹⁶ to observe higher-level patterns (e.g., to identify involvement of a brain area at a specific time).

Psycho-Physiological Data ⑤ Lastly, *CODERSMUSE* visualizes heart rate, respiration, and electrodermal activity in the form of numeric time series.

3.4.1.2 Features and Challenges

The key feature of exploring data with *CODERSMUSE* is essentially a real-time replay of collected data ①. That is, experimenters can (*re*)play data of an experiment session and simultaneously observe all data streams. The user can also use the time slider to dynamically move through a task's timeline to examine an event more closely ①. In Figure 3.20, we set *CODERSMUSE* to show the data of an experiment that are split into individual tasks. That is, we select a specific task from the entire experiment and show the data of a specific task ②. *CODERSMUSE* may also show the entire data set of all tasks, but this limits the usefulness of the eye-tracking-data view (as the presented code typically changes with each task).

The complexity of *CODERSMUSE* stems from the inherent differences in the characteristics of the integrated modalities, which poses major challenges for a proper conjoint exploration (cf. Table 3.5).

Data Preprocessing Data preprocessing is a crucial step to ensure high data quality, a prerequisite to obtaining meaningful insights. Nevertheless, the mandatory preprocessing varies between modalities. For example, eye-tracking data require fundamentally different preprocessing than fMRI data (cf. Table 3.5). There are also no golden standards for preprocessing across all modalities yet.

¹⁶<http://nipy.org/nipy/>, [MB07]

A re-implementation of every necessary preprocessing step for each modality would be inefficient and error-prone. Thus, the current prototype of *CODERSMUSE* relies on already preprocessed data before importing them into *CODERSMUSE*. We provide template scripts and guides on how to integrate your own data, including the necessary preprocessing, on the *CODERSMUSE*'s Web site.

Data Synchronization Another challenge of *CODERSMUSE* is to properly synchronize the timing of each data visualization. The integrated modalities exhibit different temporal delays. For example, fMRI measures the biological effect of cognitive processes (i.e., haemodynamic response), which means that the data stream is delayed by around five seconds [Cha+93]. To counteract this effect, the displayed fMRI data are offset by six seconds. Similarly, the underlying biological response of electrodermal activity is typically delayed by about two seconds [Bou12], so we offset the presented time frame by the same amount. Eye-tracking data have no delay. All offsets are default settings and can be customized depending on the experiment.

Data Visualization Each modality provides fundamentally different data, so we need an appropriate visualization for each modality. For example, visualizing one-dimensional eye-tracking data is different from visualizing three-dimensional neuroimaging data. Thus, each data stream implements its own visualization, which is inspired by state-of-the-art tools.

3.4.1.3 Implementation

Due to the performance requirements of handling the wealth of data, *CODERSMUSE*'s prototype is implemented as a desktop program. It is based on Python 3.6 and Qt 5.11, making it available cross-platform.

CODERSMUSE follows a plug-in architecture, such that each modality is implemented as a separate plug-in. This way, different modalities can be easily supported: For example, researchers can swap the fMRI plug-in with a new plug-in (e.g., for EEG, currently not implemented). Generally, multiple plug-ins can be active (e.g., fMRI and EEG, if the experiment setup allows for simultaneous capture). Furthermore, each plug-in can be further enhanced to individual needs, for example, with additional view options. This way, *CODERSMUSE* is also customizable.

Each plug-in creates a view for its respective data set. When the user explores data along the time slider, *CODERSMUSE*'s core triggers a view update with the current timestamp to each plug-in. When the user interacts with a specific view (e.g., to change the observed position in the brain), the respective plug-in accepts the request and renews the view. An interaction between plug-ins is currently not supported.

3.4.2 Future Work

Due to the complexity and novelty of this endeavor, there is still substantial work left. We share the current version to enable the community, which is starting to embrace multi-modal program-comprehension experiments, to shape the further development of *CODERSMUSE* or to implement their own vision. For our purposes, we foresee the inevitable need for further extensions, which we discuss next.

3.4.2.1 Additional Modalities

So far, we have focused on supporting fMRI as neuroimaging method, but there are alternatives, such as functional near-infrared spectroscopy (fNIRS) [Sch+14]. fNIRS does not require such a physically restrained setting as fMRI. fNIRS measures the same underlying biological effect as fMRI, and therefore fNIRS data are similarly delayed as fMRI data. However, fNIRS does not provide a three-dimensional data set (in the order of about 100,000 time series), but instead aggregates the measured brain activation into a handful of data streams. To support fNIRS data in *CODERSMUSE*, we could develop an according plug-in and extend the guidelines to describe how to use this plug-in.

Another extension would be to integrate electroencephalography (EEG) data, for example, to observe the cognitive load of programmers, as done by Crk et al. [CKS15], which can also be recorded simultaneously with fMRI data. EEG data differ from fMRI data in that they are not delayed, but have a higher temporal resolution (in the order of milliseconds), and provide typically 32 or 64 data streams, collected via channels located at different positions around the skull.

3.4.2.2 Data Annotation and High-Level Patterns

In Section 3.3.2, we outlined the need for data exploration and analysis. To deal with the wealth of data obtained from a multi-modal experiment, researchers should be able to annotate individual events in each individual data stream or across data streams, such as a peak in the neuronal response. At first, this may merely be a manual process, but could be extended by automatic techniques (e.g., based on a classifier) to detect similar events in other parts of an experiment (e.g., as ATLAS is offering [MBS12]). Eventually, we aim at extracting higher-level patterns across data streams. For example, *CODERSMUSE* may indicate that long fixations on loop initializations (eye tracking) trigger an activation in working memory (neuroimaging) and increased cognitive load (psycho-physiological data). Such extracted high-level patterns would allow researchers to generate new hypotheses for follow-up studies and eventually lead to a more holistic understanding of program comprehension.

3.4.2.3 Data Aggregation

In Figure 2.10, we visualized typical neuroimaging experiment designs. Currently, *CODERSMUSE* displays data per participant and per task, allowing us to perform an in-depth analysis. Of course, typical experiments use aggregated data to answer hypotheses and draw conclusions. To support this, we are currently exploring suitable aggregation methods and implement appropriate visualizations (especially, a heatmap from the eye-tracking data and aggregated neuronal responses across participants and tasks).

3.4.3 Use Cases for *CODERSMUSE*

Analyzing empirical studies of program comprehension, particularly multi-modal ones, is time-consuming. The goal of *CODERSMUSE* is to not only compute numerical results for publication, but to support insightful data exploration, which is necessary to build a holistic understanding of program comprehension. By exploring data, we can build new hypotheses to be tested in follow-up studies. *CODERSMUSE* aids both, discovery and testing of hypotheses.

Experimenters can investigate the behavior (behavioral and eye-tracking data) and measurements of program comprehension (neuroimaging, psycho-physiological) for each participant. The real-time replay of data lets researchers dive into and, hopefully, understand detailed events during an experiment. *CODERSMUSE*'s views are designed to provide a comprehensive observation of each data stream.

3.4.4 Related Work

A meaningful combination of neuroimaging and eye-tracking data, in addition to other modalities, is still in its infancy, not only in program-comprehension research but also in neuroscience. At the time of initial development, we were not aware of any commercial or scientific tool that integrates several modalities in one offline data-exploration tool that fits our needs. *ATLAS* is a multi-modal data-annotation tool [MBS12], but it does not offer a convenient integration of eye-tracking data, which is essential for understanding a programmer in action [Pei+18c].

After our initial publication of *CODERSMUSE*, two related tools were developed and presented to the community. *VITALSE* follows a similar goal as *CODERSMUSE*. It visualizes eye tracking and biometric data in the form of line graphs. However, it currently does not have support for fMRI data yet [RFA20]. Another new tool is *COGNIDE*, which is an extension to the Visual Code IDE. It annotates source code with psycho-physiological data, for example, to assist in prioritizing code reviews [VF20]. Unlike *CODERSMUSE*, *COGNIDE* is not for exploring research data, but targets practitioners.

There are numerous tools for exploring and analyzing single modalities. For example, *Ogama* is an open-source tool to record, explore, and analyze eye-tracking data [Voß+08]. *BrainVoyager*[™] and

Statistical Parametric Mapping (SPM)¹⁷ are two tools to explore and analyze fMRI data. While none of these tools combine different data streams, they inspired individual views of *CODERSMUSE* (e.g., visualizations for the eye-tracking data are inspired by Ogama).

3.4.5 Conclusion

CODERSMUSE enables researchers to explore synchronized and integrated multi-modal data. This way, we intend to unravel the mysteries of program comprehension, which has been possible only to a limited degree, so far. In the future and with the community's input, we aim to mature *CODERSMUSE*, thereby making the analysis of multi-modal experiments accessible to a wide range of users. While the future work is a long road ahead, it offers a revolutionizing perspective and, in our mind, is worth to be pursued.

3.5 Chapter Summary and Future Work

This chapter presented an experiment framework on how to study programmers with fMRI. We evaluated several improvements over the original experiment design of Siegmund et al. We also outlined a multi-modal future, in which we combine the strength of several modalities. Such multi-modal experiment framework enables software-engineering researchers to objectively study even small effects.

While we made considerable progress in this dissertation toward a multi-modal experiment framework to understand programmers' brains, several further improvements are to be implemented. In addition to further work already outlined in previous sections, we see the need for some major research on psycho-physiological measures, the topic of operationalization of programming tasks, and a suitable selection of contrast conditions.

3.5.1 Integration of Psycho-Physiological Measures

A direct avenue of future work is to integrate an analysis of psycho-physiological measures. Psycho-physiological measures have shown promise in cognitive psychology and, to a degree, in dedicated studies on program comprehension (cf. Section 2.3.2). In all fMRI studies in this dissertation, we collected psycho-physiological data. Specifically, we measured heart rate and respiration and, in the latest fMRI study, also electrodermal activity (cf. Section 5.1).

In Figure 3.21, we show collected psycho-physiological data during an fMRI experiment. However, as psycho-physiological data tends to be noisy, which might be amplified in an fMRI environment, appropriate data cleaning and analysis procedures need to be applied. In the future, we shall

¹⁷<https://www.fil.ion.ucl.ac.uk/spm/>

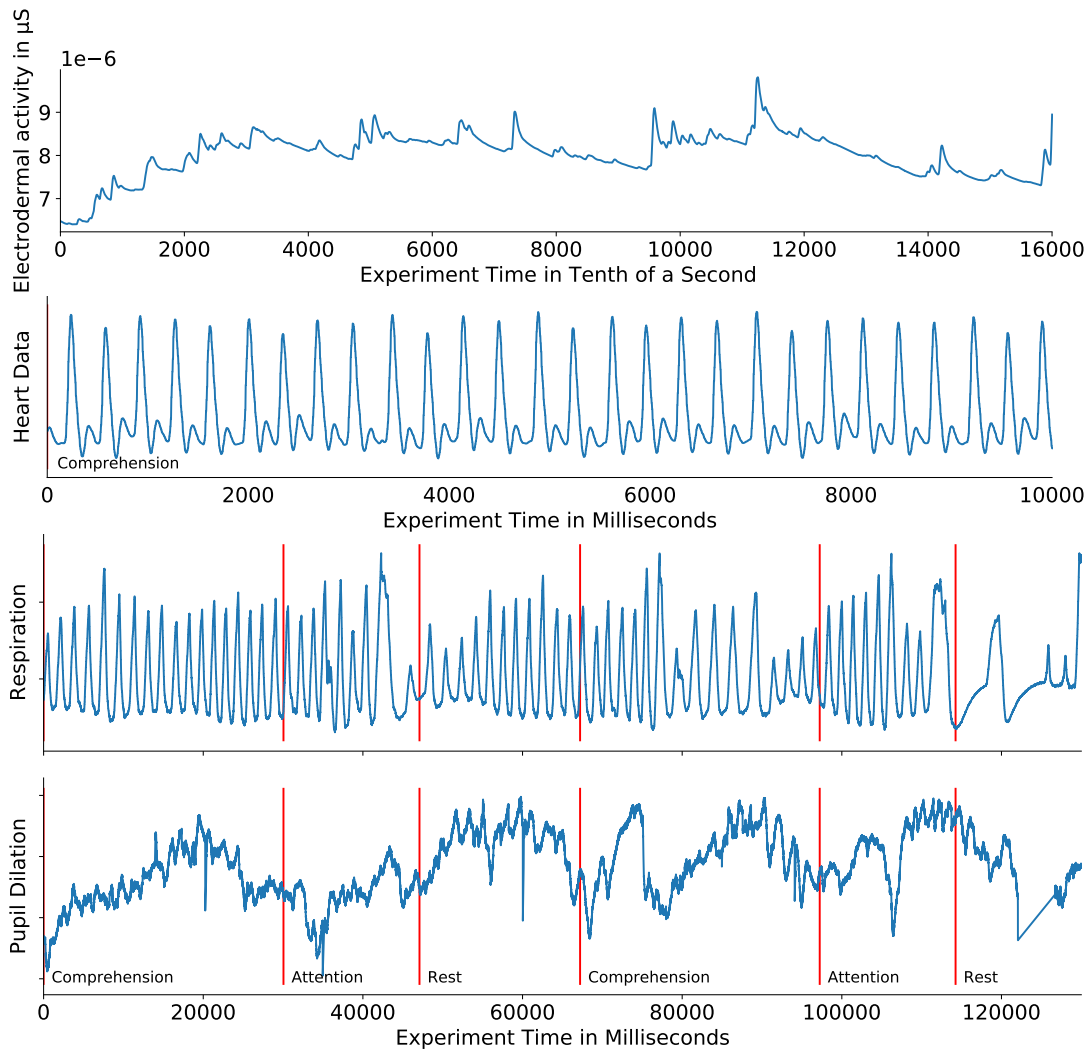


Figure 3.21: Example of psycho-physiological data (i.e., heart data, electrodermal activity, respiration, pupil dilation) during an fMRI study of a single participant. Heart data, respiration, and pupil dilation are in arbitrary units.

explore whether we can use psycho-physiological measures and pose the following research question:

RQ Can we link psycho-physiological measures to programmers' cognition in fMRI experiments?

In addition to analyzing psycho-physiological data, there are basic questions regarding tasks and contrasts that fundamentally change the outcome of an fMRI experiment. To enable comparisons across experiments, the influence of these experiment design factors need to be systematically investigated.

3.5.2 Effect of Task Design

One deciding experiment design factor is the task design. Program comprehension is a multi-faceted and multi-layered cognitive process, which can be induced through various tasks [DR00]. For example, researchers could ask programmers to simply comprehend a code snippet, or calculate input-output behavior, or consider program invariants. Many aspects from the programmer to the source code affect the observed cognitive processes (cf. Section 2.1). Over the past seven years, the early fMRI studies in a software engineering context used various approaches to operationalize program comprehension.

To put upcoming fMRI studies on a solid foundation, we need a systematic exploration of how the task design to induce program comprehension drives the observed cognitive processes and its data. Since fMRI data measures underlying cognitive processes, researchers need to be confident how the presented task directs how programmers comprehend source code.

Besides the task itself, there are further details on the operationalization that shall be investigated. For example, we always enabled syntax highlighting in our experiments, since it aligns to the programmers' natural environment based on the results of Beelders and du Plessis [BP16]. They conducted an eye-tracking study on the influence of syntax highlighting, and while there is no measurable advantage, participants preferred the snippets with syntax highlighting. However, this does not provide insights on whether it changes underlying cognitive processes and thus needs to be replicated in an fMRI study.

3.5.3 Increasing External Validity with Complex Snippets

The presented experiment framework, including all of our fMRI studies that follow in later chapters, limits the code snippets to around 30 lines of source code. This limitation is mainly driven due to the limited screen size inside the fMRI scanner. Longer snippets would require some way to scroll or navigate, which may induce unwanted head motions.

Nevertheless, longer and more complex snippets or working with multiple files would enable researchers to target a different set of cognitive processes. For example, comprehending an entire software *project* likely requires a different set of cognitive processes. To observe these aspects of program comprehension, an fMRI-compatible mouse¹⁸ can be used. Such experiments with more complex snippets would also allow us to increase external validity.

However, neuroscience currently identified an estimated upper limit of around 60 seconds for each task block [Smi+07]. If a task takes longer, the BOLD response begins to be overshadowed due to fMRI noise and participant habituation [MB15]. Thus, tasks for longer snippets must either be solvable within a relatively short time frame or use a different experiment design.

¹⁸For example, Scroll Click from Current Designs, USA: <https://www.curdes.com>

3.5.4 Reduce Task Reflection during Rest Condition

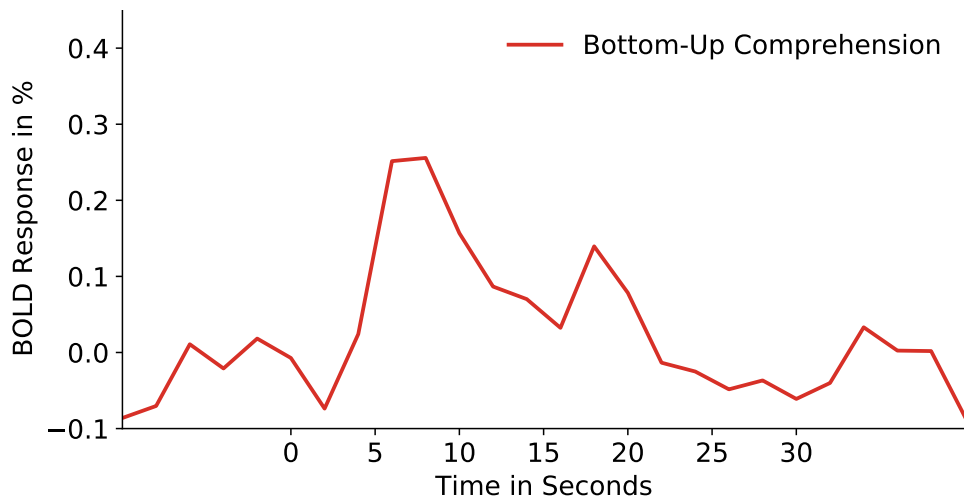


Figure 3.22: BOLD response in BA21 of bottom-up comprehension. An uptick in brain activation is visible during the rest condition between 30–35 seconds.

During data analysis of our first fMRI study, we noticed a surprising uptick in brain activation during the rest condition. We show an average BOLD response with more than expected activation during the rest condition in Figure 3.22. This effect appears to be especially pronounced when participants were unable to complete the previous task in time. In combination with participant comments, we believe that participants at times reflect on the code snippet they just had seen. While such behavior is natural, it affects our data quality, since it reduces contrast strengths.

To counteract reflective thinking during the rest condition, we took inspiration from Mallow et al. They conducted a study on superior memorizers, in which participants first had to memorize 40 digits, and then recall them after a rest condition. Mallow et al. ensured that no further memorization would occur during the rest condition by distracting them with an attention task. We adapted this approach and implemented a d2 attention task [BSAL10] directly after the comprehension task in a later fMRI study. We show an example in Figure 3.23. Participant comments indicated that it successfully removed their focus from the previous code snippet but also is somewhat strenuous. In the future, we shall explore further options to reduce task reflection during the rest condition without exerting participants.

3.5.5 Evaluation of Control Conditions

The first fMRI study of Siegmund et al. used locating syntax errors as control condition. This may be problematic as participants indicated that looking through source code still triggers some program comprehension. To maximize contrasts and thus our data quality, we need to find a better contrast condition that allows us to filter out unrelated activation, but does not remove brain activation essential to program comprehension.

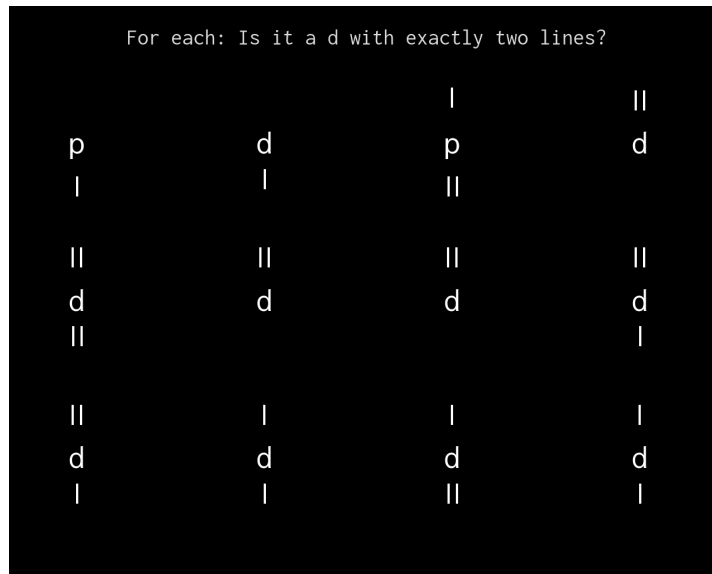


Figure 3.23: Example of a distraction task inspired by d2 attention task. Participants had to check every letter and identify whether it was a “d” with exactly two lines above or below it [BSAL10].

In our latest fMRI study on code complexity metrics, which we present in Section 5.1, we changed the contrast task. We still show code, but now participants had to only scan for opening brackets (i.e., “[” and “{”). Such simple visual scan requires less cognitive effort than locating syntax errors, in which some syntax-based checking must be done. Overall, our data hints that it may be more suited, but further work is necessary.

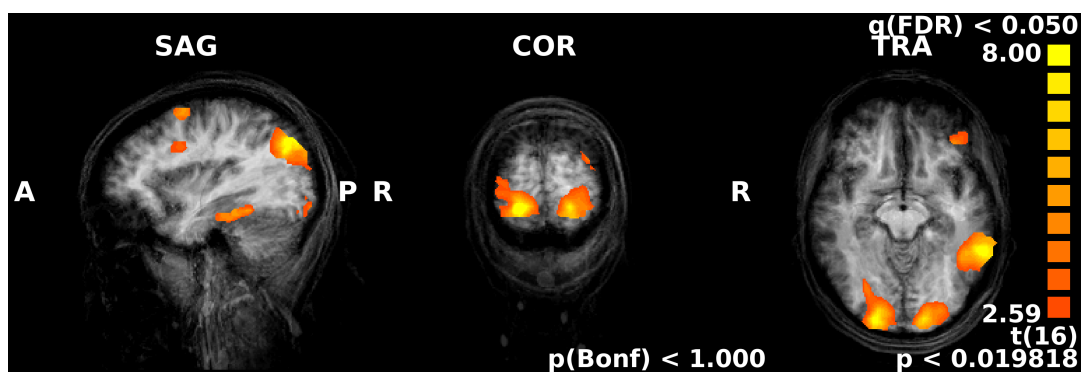


Figure 3.24: fMRI contrast between program comprehension and d2 attention task.

We also evaluated the distraction task from the previous paragraph as a suitable contrast condition. It forces participants to pay attention and triggers visual attention. However, we explored a computed contrast between program comprehension and the distraction task. As visualized in Figure 3.24, we find many brain activation clusters that seem irrelevant to the essence of program comprehension, for example, a strong activation in the visual cortex. We also observed a strong difference in the motor cortex, which is due to the difference in clicking behavior (i.e.,

one click at the end for program comprehension task or many clicks throughout the distractor task). Thus, the distraction task does not seem to be ideal. Thus, there still is an open question of what a good contrast condition for program comprehension is.

In summary, this chapter presented a robust experiment framework that allows researchers to study programmers from a neuro-cognitive perspective. While there are still improvements to be made, we use the multi-modal experiment framework to investigate several important research questions in the next two chapters.

4 Neuro-Cognitive Perspective of Program Comprehension

This chapter shares material with several prior publications [PSP17; Sie+17; Pei+20; PSA20].

In the previous chapter, we presented our state-of-the-art fMRI experiment framework. In this chapter, we apply our experiment framework to observe programmers from a *neuro-cognitive perspective*. This chapter and the subsequent chapter will demonstrate how a neuro-cognitive perspective of programmers yields novel insights into the underlying cognitive processes of program comprehension. This opening of a previously locked black box facilitates researchers and practitioners to develop more reliable code and better educate and train future programmers.

We first present our fMRI study validating the theory of top-down comprehension in the following Section 4.1. We provide objective evidence that top-down comprehension eases programmers' cognitive load, but also show that it principally shares the underlying cognitive processes with bottom-up comprehension. In Section 4.2, we present a replication of a previous eye-tracking study to understand the effects of programmer expertise, code structure, and comprehension strategy on programmers' reading order of source code. We find that the code structure substantially influences programmers' reading order. Our data also supports previous beliefs that programmers' expertise and comprehension strategy affect programmers' reading order. In Section 4.3, we outline a study to investigate the neural differences between beginning and experienced programmers. In combination, the three studies extend our knowledge on how programmers comprehend source code. It also shows the potential of fMRI studies to validate theories of program comprehension.

4.1 Neural Efficiency of Top-Down Comprehension

In this study, we contrast bottom-up comprehension with top-down comprehension with fMRI. For many decades, researchers have intensely debated these contrasting theories of program comprehension and the role of concepts such as semantic chunking, plans, and beacons without any clear consensus (cf. Section 2.1). Some researchers have argued that bottom-up comprehension cannot be avoided because meaning always needs to be extracted from perceptual and syntactical information. Others have argued that programmers actively avoid bottom-up comprehension,

```
1 public float arrayAverage(int[] array) {
2     int mgqakyy = 0;
3     int sum = 0;
4     while (mgqakyy < array.length) {
5         sum = sum + array[mgqakyy];
6         mgqakyy = mgqakyy + 1;
7     }
8
9     float average = sum / (float) mgqakyy;
10    return average;
11 }
12 }
```

Listing 4.1: Example code snippet of averaging an array with beacons and pretty-printed layout [BY, LP].

```
1 public float ayyaoAwyakyy(int[] array) {
2     int mgqakyy = 0;
3     int equ = 0;
4     while (mgqakyy < array.length) {
5         equ = equ + array[mgqakyy];
6         mgqakyy = mgqakyy + 1;
7     }
8
9     float awyyakyy = equ / (float) mgqakyy;
10    return awyyakyy;
11 }
12 }
```

Listing 4.2: Example code snippet of averaging an array without beacons but with pretty-printed layout [BN, LP].

```
1 public float ayyaoAwyakyy(int[] array) {
2     int
3     mgqakyy
4     = 0;
5     int sum = 0;
6
7     while (mgqakyy
8     < array.length) {
9         sum =
10        sum + array[mgqakyy];
11        mgqakyy
12        = mgqakyy + 1;
13    }
14
15    float average
16    = sum /
17    (float) mgqakyy;
18    return
19    average;
20 }
```

Listing 4.3: Example code snippet of averaging an array without beacons and disrupted layout [BN, LD].

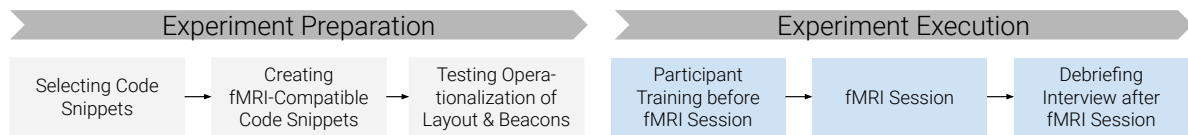


Figure 4.1: Overview of designing the snippets and conducting the fMRI study.

because it is an inherently tedious process with a high cognitive load [SE84]. Still others have debated the mechanics of how plans and beacons are used in program comprehension. This study aims to understand how top-down comprehension differs from bottom-up comprehension as cognitive processes in the programmer’s brain. One possibility is that entirely different brain areas can be activated when top-down comprehension is possible, which would provide evidence of divergent cognitive processes. Another possibility is that both processes activate similar brain areas; however, they differ in neural efficiency. *Neural efficiency* is a phenomenon where lower brain activation indicates that a cognitive process is more efficient and thereby is perceived as requiring less effort [NF09].

To investigate these hypotheses, we address two research questions:

RQ 4.1: What is the difference between bottom-up comprehension and top-down comprehension in terms of involved brain areas and their activation strengths?

Bottom-up comprehension is inherently a tedious and time-consuming process and causes a high cognitive load. In contrast, top-down comprehension is very efficient but requires prior experience and knowledge. The activated brain areas and their intensity of activation should reflect these differences. Such a result would strengthen our understanding of both bottom-up comprehension and top-down comprehension.

RQ 4.2: How do layout and beacons in source code influence program comprehension?

Different aspects of source code are believed to influence top-down comprehension, such as beacons (i.e., meaningful identifiers that indicate a program’s purpose) or the program layout (e.g., the indentation of nested loops). We test their impact in this study.

4.1.1 Experiment Design

To address our research questions, we extended Siegmund et al.’s study on bottom-up comprehension by including additional code snippets that facilitated top-down comprehension. The fMRI session was preceded by a training session, in which participants studied the code snippets to gain familiarity with them. The training ensures that participants can employ top-down comprehension. Once in the fMRI scanner, participants comprehended code snippets. For each code snippet, participants determined whether it implemented the same functionality as one of the

Top-Down	Bottom-Up	Syntax
ArrayAverage	CommonChars	Average
BinaryToDecimal	CrossSum	DoubleArray
CrossSum	DoubleArray	Power
FirstAboveThreshold	Factorial	ReverseIntArray
Power	MaxInArray	ReverseWord
SquareRoot	SumUpToN	Swap
ContainsSubstrings		
CountSameCharsAtSamePosition		
CountVowels		
IntertwineTwoWords		
Palindrome		
ReverseWord		

Table 4.1: All code snippets used in this fMRI study on top-down comprehension. Snippets in **bold** were part of the original fMRI study by Siegmund et al. [Sie+14a].

snippets they looked at in the training session. To evaluate the role of beacons and code layout on top-down comprehension, we created four snippet versions for top-down comprehension:

- Beacons and pretty-printed layout [BY, LP]
- Beacons and disrupted layout [BY, LD]
- No beacons and pretty-printed layout [BN, LP]
- No beacons and disrupted layout [BN, LD]

In the fMRI scanner, participants read snippets of all four versions, enabling us to observe how each variation affected the activated brain areas and their intensity. Based on these data, we conclude on the cognitive processes that occurred during program comprehension. Figure 4.1 provides an overview of the experiment-design process and the experiment procedure.

Next, we describe the experimental setup in detail. We include the design of the code snippets, the training session, and the imaging setup used in the fMRI scanner.

4.1.1.1 Experimental Conditions

Our experiment design contains two independent variables: Comprehension strategy (i.e., top-down comprehension vs. bottom-up comprehension), beacon (i.e., beacons vs. no beacons), and layout (i.e., pretty-printed layout vs. disrupted layout). The dependent variable was the observed brain activation in terms of brain area and activation strength.

Comprehension Strategy To influence which comprehension strategy participants used, we introduced several mechanics to control the content and presentation style of code snippets viewed during the experiment.

Bottom-Up Comprehension For bottom-up comprehension, we need to ensure that participants go through the snippets statement by statement. To this end, we use the same methodology as in the original study of Siegmund et al.: We used meaningless identifier names, such that they did not convey the meaning of a variable, but only its usage (e.g., a variable holding the result of an algorithm was named `result`, not `reversedWord`). The snippets that we used for each condition are summarized in Table 4.1. The snippets had comparable complexity and length to maintain comparability to the study by Siegmund et al. [Sie+14a].

Top-Down Comprehension For top-down comprehension, we need to ensure that participants can identify beacons and generate hypotheses for the presented code snippet. *Beacons* give hints about a program's purpose and set corresponding expectations [Bro78]. For example, a method named `arrayAverage` implies that the method computes the average of an array of numbers. Prior knowledge on averaging helps programmers to quickly confirm whether the method actually computes the average. In Listing 4.1, we show a corresponding algorithm. In lines 5–7, the expected loop starts the statements to compute the sum and to increment the counter. Confirming that this method indeed computes the average of an array of numbers is then straight-forward.

We also re-used some snippets of the study by Siegmund et al. for the top-down-comprehension part of our experiment. We modified them to be more amenable to top-down comprehension (cf. Table 4.1 for the snippets that we used).

This condition, contrasted with bottom-up comprehension, enables us to address RQ 4.1.

Controlling Beacon and Layout Conditions To further discern which code aspects guide top-down comprehension, we created four different versions for each top-down-comprehension snippet. To do so, we manipulated two aspects of source code that have been the focus of early studies on top-down comprehension and are believed to have an enormous impact on the comprehension process: beacons [Bro78] and layout [Mia+83]. Beacons constitute a semantic aspect of source code, *layout* constitutes a structural one. In this study, we look at how different code layouts affect the comprehension process. If code layout considerably violates standard coding conventions, it might impair top-down comprehension. We show an example in Listing 4.2 with blank lines and line breaks in unusual locations.

We created a snippet version for each combination of the two aspects, that is: Beacons and pretty-printed layout (*BY, LP*), beacons and disrupted layout (*BY, LD*), no beacons and pretty-printed layout (*BN, LP*), and no beacons and disrupted layout (*BN, LD*). The experiment's snippets in all versions are available on the project's Web site.

These four snippet versions enable us to address RQ 4.2. If beacons and code layout affect top-down comprehension, we should see a difference in the brain activation pattern between these four conditions. For example, beacons might drive the top-down comprehension process, but not code layout. In that case, the beacon versions should lead to a different brain activation pattern than the versions without beacons, independent of the code layout. Next, we describe in detail the process of how we created the snippets.

4.1.1.2 Designing and Selecting Code Snippets

When using fMRI, it is typically necessary to compute averages over many conditions [HSM14], so we needed code snippets of similar size and complexity. Furthermore, we had to choose short snippets that fit on the screen inside the fMRI scanner to avoid scrolling. To develop suitable snippets, we followed the procedure established in the original study [Sie+14a]. First, we conducted several pilot studies, from which we requested feedback from participants, studied response times, and determined correctness. Based on these data, we selected snippets that did lead to our desired 20 to 30 second comprehension time for an optimal BOLD response, and did not cause too many incorrect answers. We excluded snippets when participants indicated that they were unsuitable (e.g., requiring mainly visual search).

In the end, we selected twelve snippets that had all similar response times, length, and complexity. The shortest snippet was eight lines long and the longest had 19 lines. The snippet in Listing 4.1, which computes the average of an array, was included in our study. The other snippets were similar in size and complexity.

We balanced the snippets' content, such that six snippets manipulated words (e.g., reverse a word) and six manipulated numbers (e.g., compute the average, factorial). This way, we ensured that a snippet's content did not overshadow the activation of comprehension (e.g., in that words might result in a specific activation pattern and not the comprehension process itself).

The largest issue we faced was ensuring that participants spend 20 to 30 seconds to understand a snippet. Understanding small code snippets with top-down comprehension is very efficient, so snippets as the one in Listing 4.1 are understandable almost instantaneously making it difficult to measure the BOLD response. Thus, we decided to obfuscate the code through word scrambling of identifiers.

To find an optimal scrambling degree (i.e., we obtain response times between 20 to 30 seconds but do not bias the comprehension process), we conducted a series of small studies with graduate students in computer science. The methodology is explained in detail on the project's Web site. In a nutshell, we first experimented with scrambling the code to use Japanese characters, but found that it interfered too much with program comprehension. Next, we tried Caesar shifting (a cipher in which each letter is replaced by another in a static scheme), which was more suitable to elicit top-down comprehension, but participants adapted to the shifting (e.g., they learned that `qom` means `int`). Thus, we used a variable Caesar shifting, such that each snippet was created

with a different scrambling scheme. Furthermore, we decided not to scramble keywords and calls to library functions, to ensure that each code snippet remained compilable.

Having found a suitable scrambling method, we evaluated the operationalization of the independent variables. To this end, we conducted two pilot studies with undergraduate computer-science students at NC State University.

Testing Beacons We conducted our first pilot study with 81 students who were all junior and senior undergraduate students in a software-engineering course. To ensure that participants can use top-down comprehension for the snippets, we trained them before the study. To this end, participants viewed, recalled, and reviewed the snippets to establish familiarity with them. Subsequently, we split the students into two groups: one worked with beacons (*BY*) and the other without (*BN*). The participants had to determine whether a code snippet fulfilled the same function as a snippet from training as correctly and quickly as possible. At the end of the study, students described their cognitive processes in an online questionnaire (see the project's Web site¹⁹). They stated that they indeed used top-down comprehension independent of whether beacons were present or not. Their response times also fit into our target interval of 20 to 30 seconds.

Testing Layout To evaluate the operationalization of code layout, we conducted a second pilot study with 12 students who did not participate in the first pilot. The pilot study's overall design was the same (train, understand, reflect), except that the tested snippets had a different layout with beacons present or not. The response times and correctness were similar to the first pilot study and were suitable for our purpose. The students reported that the cognition process is challenged differently but stayed within the understanding of top-down comprehension. Thus, also the operationalization of code layout proved suitable for our case.

4.1.1.3 Participants of the fMRI Study

We recruited 11 programmers from Otto von Guericke University Magdeburg by posting on online and local bulletin boards. Besides fulfilling the prerequisites for fMRI studies (e.g., no metallic implants), participants needed to have basic knowledge in object-oriented programming and algorithms. All participants were right-handed. Most participants were familiar with Java or C, at least, at a medium level (answer on a 3-point scale). Only one participant was rather inexperienced with both, Java and C, but had 16 years of Python experience. We designed the snippets to have minimal Java-specific features, and we clarified any questions of participants during the training sessions. We show further demographic data in Table 4.2.

3 participants agreed to a second session, which results in 14 measured fMRI sessions overall.

¹⁹<https://github.com/brains-on-code/paper-esec-fse-2017>

Characteristic		N (in %)
Participants		11
Gender	Male	9 (82%)
	Female	2 (18%)
Occupation	Computer-science student	5 (45%)
	Mathematics student	3 (27%)
	Professional programmer	3 (27%)
Age in years \pm SD		25.30 \pm 3.82
Programming experience	Experience score [Sie+14b] \pm SD	2.70 \pm 0.39
	Java experience [Sie+14b] \pm SD	3.09 \pm 1.38

Table 4.2: Participant demographics for our fMRI study on top-down comprehension.

4.1.1.4 Task Design

We used the tasks of Siegmund et al.: Participants should determine the output of code snippets with given input values. Here, we also made sure to use easily computable results (e.g., factorial of 3), to focus participants on comprehending the program, not computing the output. As control condition, participants also located syntax errors in code snippets that they had not seen before. None of the syntax errors required participants to understand the code (e.g., missing semicolons or closing brackets).

To enable participants to use top-down comprehension (RQ 4.1 and RQ 4.2), we familiarized them with the top-down snippets and programming concepts in a training session before they entered the fMRI scanner. Each code snippet had a canonical representation, as would occur in a typical program (e.g., with beacons, pretty-printed layout, syntax highlighting, not scrambled). In training, participants viewed a snippet, recalled it, and then reviewed it again. This process was repeated for each of the 12 snippets.

In the fMRI scanner, participants had to decide whether a snippet correctly implemented the same functionality as the snippets seen during training. This task involved recognizing a snippet's purpose and evaluating whether it was working as intended. Participants had to make these decisions as quickly and correctly as possible, with a time limit of 30 seconds for each snippet.

4.1.1.5 Experiment Execution

Prior to the training session, participants completed a questionnaire on their programming experience [Sie+14b]. In the fMRI scanner, the participants went through twelve trials, in randomized order for each participant. One trial is visualized in Figure 4.2.

From past experience, we know that participants have difficulty lying motionless for more than 30 minutes while concentrating on their tasks. Hence, we split the 12 trials into two sessions measured on different days, each preceded by the view-recall-review training outside the fMRI

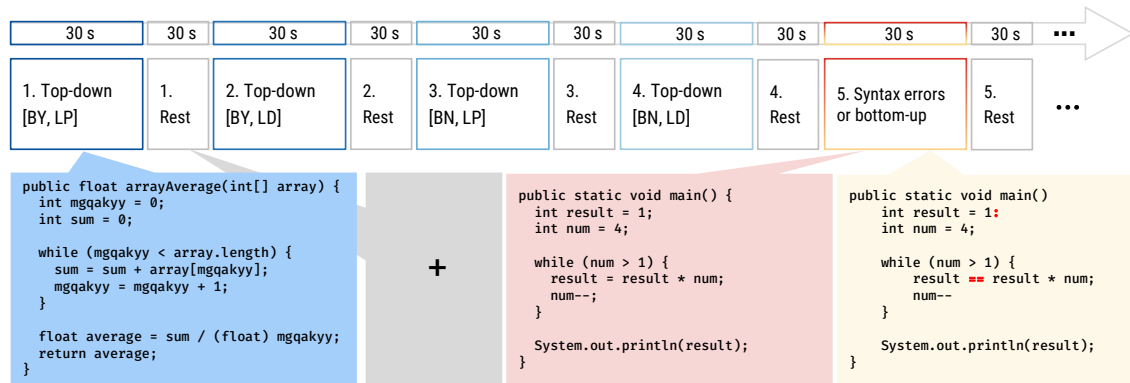


Figure 4.2: Illustration of one (out of twelve) experiment trials for our study on top-down comprehension. For half of the trials, bottom-up comprehension was the fifth task. For the other half, we presented a control condition (i.e., locating syntax errors).

scanner. When the participants entered the fMRI scanner, they spent their first 6 minutes conducting (anatomical) pre-measurements. Then, they looked at four top-down comprehension tasks (everyone used the same order) followed by alternating bottom-up and syntax-error discovery tasks. Participants could react to each task with a two-button response device. In each condition, we asked the participants to be as fast and correct as possible. After each trial, the participants rested to allow their BOLD response to return to their baseline.

Once the participants exited the fMRI scanner, we asked them to explain to us how they solved the tasks. This post-session interview provided valuable insights into their comprehension strategies, helping us interpret the results.

4.1.1.6 fMRI Data Analysis

We describe the fMRI setting and imaging sequence in the Appendix (Section 7.1). After standard fMRI data preprocessing (Section 7.1), we conducted a random-effects GLM analysis, defining one predictor for each of the comprehension tasks, and one for the syntax task. To test for differences between bottom-up and top-down comprehension (RQ 4.1), we calculated the balanced contrast between the bottom-up condition and the 4 top-down conditions. To this end, we chose the same level of significance as in the original study (i.e., $p < 0.01$, FDR corrected). The resulting clusters of voxels were defined as volumes of interest (VOI) and attributed to their respective Brodmann areas by using the Talairach daemon.²⁰ Then, we extracted the beta values of the GLM for each participant and condition to identify differences in activation for each of the program comprehension conditions within the defined VOIs.

²⁰<http://www.talairach.org>



Figure 4.3: Significant brain activation during top-down comprehension. BAs 6, 21, 44 are in the left hemisphere.

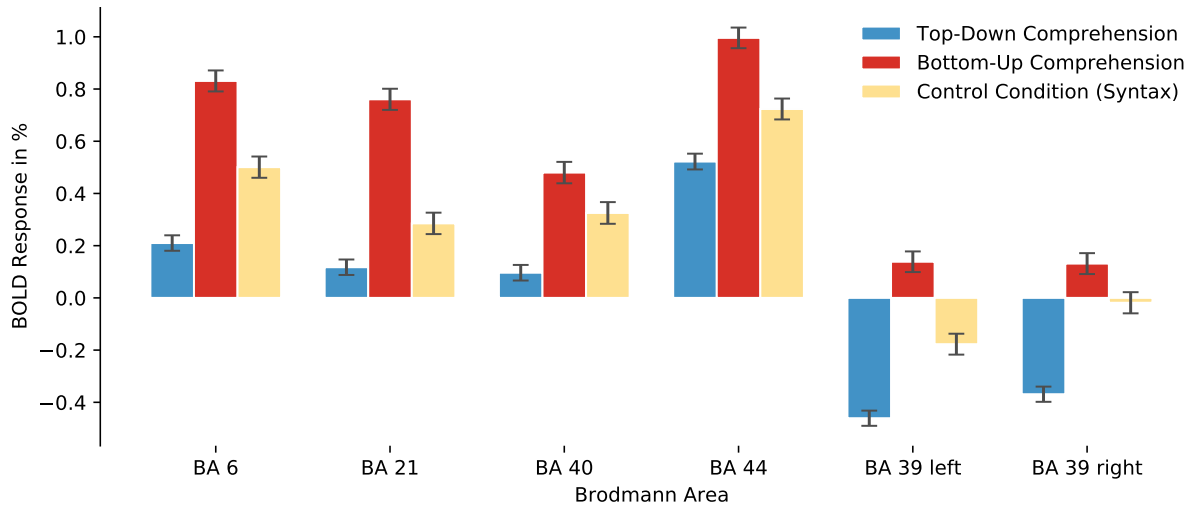


Figure 4.4: Average BOLD response. BA 39 is activated/deactivated in both hemispheres, BAs 6, 21, 40, and 44 in the left hemisphere. The whiskers indicate the standard error of the mean of the activation in the different participants.

4.1.2 Results and Discussion

In this section, we present and discuss the results of our study, structured along the two research questions.

RQ 4.1 What is the difference between bottom-up comprehension and top-down comprehension in terms of involved brain areas and their activation strengths?

Data Figure 4.3 shows the activated BAs, which contrast top-down comprehension with bottom-up comprehension. The areas largely overlap with those defined in the original study [Sie+14a]. Specifically, BAs 21 and 44 in the brain's left hemisphere are also activated. We also found a significant effect in BA 6, albeit at a slightly different location compared to the previous study.

The activation level during top-down comprehension for all three areas is, however, significantly lower than for bottom-up comprehension.

Figure 4.4 shows the activation averaged across the code snippets for each condition (cf. Section 4.1.1). For example, the left group of bars indicates the activation of BA 6 for 3 individual conditions, that is, top-down, bottom-up, and syntax errors. A positive value indicates an activation. A negative value indicates a deactivation in that area during the task, when compared to the average value across the entire session (which includes resting time).

The BOLD response for BA 39 in both hemispheres shows a *deactivation* during top-down comprehension and *activation* during bottom-up comprehension. Thus, BA 39 showed significantly less activation for top-down comprehension than for bottom-up comprehension as well as during rest (cf. Section 4.1.2).

Even though the direct contrast between bottom-up and top-down comprehension did not reveal significant activation in BA 40 at the conservative significance level, we show values for this region as defined in the original study, because we found a significant difference between bottom-up and syntax. This indicates that there is also a considerable difference between the bottom-up and the top-down conditions.

Discussion The activation of the BAs 6, 21, 40, and 44 indicates that, for top-down comprehension, all participants perform the same cognitive activities, that is, extracting the meaning of words and symbols and combining them to create a general understanding of a snippet's purpose. There is evidence of a similarity between top-down and bottom-up comprehension at a basic level, which implies a similarity between both processes. However, we expected to observe activation in memory-related areas, since theory suggests that top-down comprehension should activate programming plans. Several possibilities exist: 1) Our participants may not have formed programming plans in their minds due to low levels of expertise, 2) programming plans are embedded in the same neural circuits as comprehension so we cannot see differences in our study, or 3) the theory is simply wrong. Future studies shall dig deeper into the lack of memory retrieval during top-down comprehension.

The lower activation strength in these brain areas corresponds to better neural efficiency and indicates lower cognitive load. Thus, we assert that top-down comprehension requires less cognitive effort than bottom-up comprehension. This difference is not caused by differences in task length, since all tasks lasted for 30 seconds. Looking at the response times of participants in Figure 4.5, we see no difference between the experimental conditions. The lower activation for top-down comprehension aligns well with our understanding of both comprehension processes, suggesting that top-down comprehension is more efficient than bottom-up comprehension. Evidence from other neuroimaging studies agrees with our result. Crk and Kluthe found that expertise leads to lower EEG signals (higher neural efficiency), indicating lowered cognitive load [CK14]. In our study, we primed the participants with the code snippets (cf. Section 4.1.1), resulting in lower activation levels, and indicating lowered cognitive load.

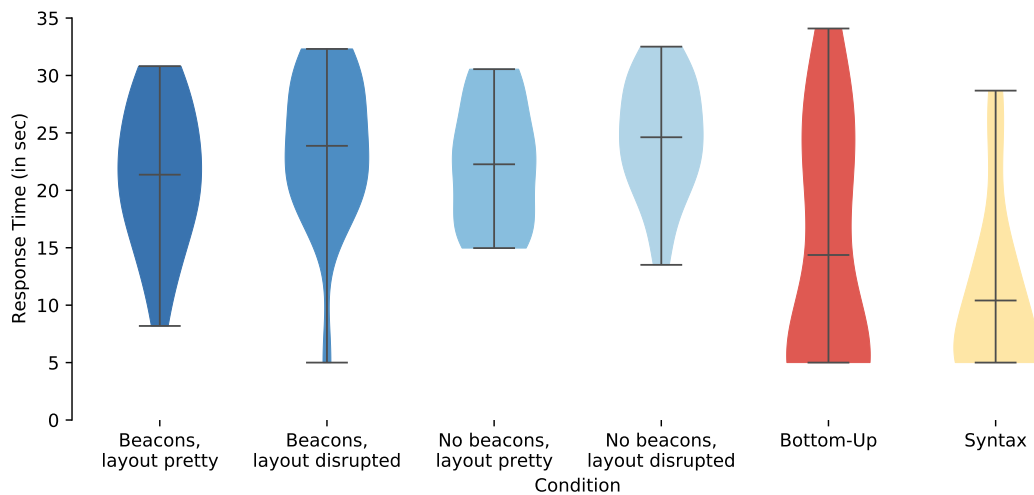


Figure 4.5: Response times in seconds per condition.

In BA 39, top-down comprehension led to less activity compared to bottom-up comprehension and even the resting condition. This particular brain area has been shown to be part of the default mode network [Rai+01], which is active during resting periods between cognitively demanding tasks (e.g., [Wir+11]). Thus, the task context of program comprehension induces cognitive processes during the resting condition (i.e., between the program-comprehension tasks). The fact that the study by Siegmund et al. did not mention this brain area can be explained by the roughly similar levels of activity in the bottom-up and the syntax condition. Underestimating which brain areas are involved in cognitive tasks is a common problem, especially for areas belonging to the default brain network [SS01]. Thus, future fMRI studies must be extra careful when devising their control conditions. Since BA 39 has been suggested to be involved in a number of cognitive processes, such as semantic processing, word reading and comprehension, number processing, memory retrieval, attention, and reasoning (which all seem relevant for all tasks of the current study), it is crucial to replace the resting condition with tasks that *distract* participants from activities related to program comprehension.

It is important to discern the degree of involvement of BA 39 in program comprehension, because it acts as a cross-modal hub, in which converging multisensory information is combined and integrated to comprehend and give sense to events, manipulate mental representations, solve familiar problems, and reorient attention to relevant information [Seg13]. Regarding deactivation relative to rest, it has been suggested that reading affords fewer semantic associations than free association [Bin+99]. This may also be true for program comprehension as applied to the current study.

Our control condition (locating syntax errors) was designed to subtract activation related to visual processing of the source code. We intended to avoid program comprehension by letting participants locate syntax errors (e.g., a missing closing bracket) that could be completed by a simple visual search. However, we did not find a difference in the activation pattern between top-down comprehension and syntax-error location. On the one hand, it could mean that top-down comprehension is very similar to locating syntax errors. Looking at the tasks, it could be argued

that deciding whether a snippet is familiar to participants requires pattern matching rather than full comprehension. On the other hand, the result could indicate that for locating syntax errors, participants at least partially comprehended the source code. Evidence from other studies shows that developers typically deploy an as-needed strategy. That is, they use understanding only as necessary to get a task done [Roe+12]. This is in line with the feedback we received from participants, who said they were not able to ignore the functionality of the code when reading it while locating errors. Syntax-error finding may mask other cognitive processes related to top-down comprehension. We will look into alternative control tasks, such as the bottom-up task, for future studies.

Thus, we can answer RQ 4.1:

RQ 4.1

Top-down comprehension activates the same regions as bottom-up comprehension, except for BA 39, which is *deactivated* during top-down comprehension, but activated during bottom-up comprehension. For all areas, the activation is significantly lower for top-down comprehension than for bottom-up comprehension.

RQ 4.2 How do layout and beacons in source code influence program comprehension?

Data We compare our 4 different top-down-comprehension conditions with one another. Overall, we found no significant differences, as supported by similar values of activation in the brain areas depicted in Figure 4.6.

Discussion We could not find any influence of beacons and layout on program comprehension. This might indicate that the role of beacons and layout for program comprehension is different than previous studies suggest [Bro78; Mia+83]. One reason may be due to our operationalization of top-down comprehension, which may have encouraged simple recognition of familiar snippets rather than comprehension, such that beacons and layout may not have played a large role. The demand of program comprehension processes may have been too low due to the intense engagement with highly comparable code snippets immediately before the fMRI experiment. Nevertheless, the results make clear that the effects resulting from different efforts to foster comprehension seem to be small.

Still, we observed activation differences at specific time points of comprehending the code snippets. For example, looking at the BOLD response for BA 21 in Figure 3.2 for each of the 4 top-down conditions, the versions with the beacons both show a peak a few seconds after task onset, independent of whether the layout was pretty-printed or disrupted. The temporal differences in activation may indicate differences in how long a programmer may spend in specific phases of a cognitive process. It would be interesting to explore whether one of these effect manifests in further studies, because BA 21 is a classical natural-language processing area. More

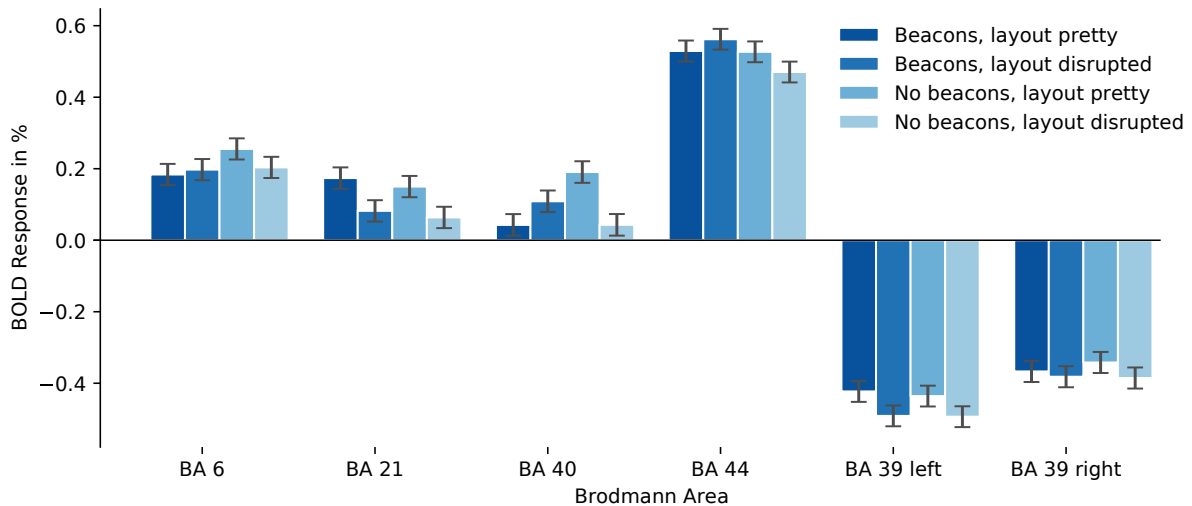


Figure 4.6: Average BOLD response. BA 39 is deactivated in both hemispheres, BAs 6, 21, 40, and 44 are activated in the left hemisphere. The whiskers indicate the standard error of the mean of the activation in the different participants.

introspective information about the dynamics during the program comprehension are needed. Eye tracking during fMRI acquisition may provide additional information and should be integrated in future studies.

RQ 4.2 Neither beacons nor program layout seem to significantly affect the program comprehension process.

4.1.3 Perspectives

4.1.3.1 Relation to Theories of Comprehension

The different activation patterns between top-down and bottom-up comprehension provide evidence in support of some aspects of theories of program comprehension, while casting doubt on others.

Semantic Chunking The theory of bottom-up comprehension suggests that developers start with details of source code and group these to semantic chunks, until they gain a high-level understanding of the program [SM79]. Based on the activation of BA 39 during bottom-up comprehension, and its role as an integration hub for semantic information, the evidence supports semantic chunking. Its absence during top-down comprehension is consistent with successive hypothesis refinement [Bro83].

Neural Efficiency Studies of brain activity find that experts demonstrate more efficient neuronal activation patterns than novices with the same tasks [NF09]. More experienced programmers also demonstrate lower activation than less experienced programmers [CK14; CKS15]. Similarly, we have observed reduced activation during top-down comprehension. Our evidence is consistent with the view that semantic cues reduce cognitive load.

Plans Software-engineering researchers have proposed that programmers use knowledge structures [Ric87] called *plans* that encode semantic and domain information about a program [Bro83]. We found evidence that is inconsistent with some theoretical aspects of plans. Unlike previous experiments that manipulated syntax layout and beacons [SE84], we could not confirm any influence on the cognitive processes developers used when comprehending source code. Even if the code is scrambled or the layout is changed, a programmer's cognitive processes may be robust enough to deal with them, just as programmers can understand code that is incomplete or has syntax errors. We did not find any specific cognitive processes that would be consistent with plan activation outside of the program comprehension process. An alternative theory may be that, if plans exist, they are embedded in the same circuits used during program comprehension. Overall reduced activation in the network of program-comprehension brain areas could then be the result of plan activation. But, it also might be that our paradigm induced more recognition rather than in-depth comprehension, in which plans may play a stronger role.

4.1.3.2 Implications

Tool Support A recent fMRI study by Sato et al. found that having access to Euler diagrams during logical-reasoning tasks allowed participants to offload the content of their working memory used to represent the logical statements onto the diagram itself [Sat+15]. As a result, this freed up resources to solve the reasoning tasks faster. The ability to support *cognitive offloading* has several important consequences. For example, the right hemisphere of the brain can take on a secondary task if both the primary task and secondary task are simple and do not require access to the same types of information [CK10]. Both hemispheres are recruited in complex tasks [Ban98]. The diagram used by Sato et al. would enable a person to perform more complex logical-reasoning tasks, because they can offload the representation onto the diagram. Likewise, if a programming tool is able to free up cognitive resources, such as a visualization or debugger tool, then programmers may be able to perform increasingly complex cognitive tasks, both that were not possible before, and with a reduced chance of mental errors. As we found in our work, the programmer is better able to integrate information when they do not have to perform semantic chunking. If a tool could be found to reduce the need to activate a particular brain region, then direct evidence could be offered about the ability of a tool to reduce cognitive load and potential mental errors. In the long run, this will help us and other researchers to develop tools (e.g., those similar to debuggers that show how values of variables change, but which are more customizable) that support developers in relieving cognitive resources. Developers would then be able to focus more on the actual task at hand, without being restricted by their own cognitive limits.

Experimental Paradigm Besides largely replicating the results of the study by Siegmund et al., we went beyond the state of the art by providing new evidence regarding the neural efficiency of top-down comprehension. This paves the way for fMRI and other neuroimaging techniques to be used in future research on program comprehension and related cognitive processes. We show that the lengthy process of conducting series of pilot studies is worth the effort, so we can add a neuro-cognitive perspective to the understanding of human factors in software engineering. With neuroimaging studies becoming more and more prevalent in software-engineering research, our study on top-down comprehension makes an important contribution toward establishing this modality as standard measurement instrument.

4.1.4 Threats to Validity

4.1.4.1 Construct Validity

The operationalization of top-down comprehension is closely linked with recognition and may not require much comprehension. This might explain why we did not observe a difference in activation between top-down comprehension and syntax-error finding. However, since recognition is also vital for top-down comprehension, and because our participants indicated that they did read the source code to locate syntax errors, we argue that our current operationalization of top-down comprehension was suitable for the study.

4.1.4.2 Internal Validity

Identifying BAs based on Talairach coordinates (used by the fMRI analysis software to identify voxels) requires a lot of expertise. Other researchers may attribute the same activated clusters of voxels to nearby Brodmann areas. However, our team has considerable experience with mapping voxels to Brodmann areas, and by double-checking all assignments the identification of BAs does not pose a threat in this study.

4.1.4.3 External Validity

Our limited experiment setup cannot generalize to program comprehension in different settings (for example, large software projects). Additionally, our operationalization of top-down comprehension captures only one aspect of this complex process, meaning we may have missed some. This is a fundamental trade-off in empirical studies. We have to control external influences as much as possible (internal validity), or strive for a more generalizable experimental setting (external validity) [SSA15].

4.1.5 Related Work

At the time of this study, a few neuroimaging studies have been employed in software-engineering research. The first fMRI study in this context was conducted by Siegmund et al. [Sie+14a], as we have discussed. Directly inspired by that study, Floyd et al. also conducted an fMRI study, but focused on a comparison of the representation of programming and natural languages [FSW17]. Our study was first to directly investigate top-down comprehension from a neuro-cognitive perspective.

4.1.6 Conclusion and Future Work

Top-down comprehension is a very efficient process for understanding source code compared with the tedious, statement-by-statement process employed during bottom-up comprehension. In this study, we replicated an fMRI study to deepen our understanding of program comprehension. First, we were able to replicate the previous fMRI study results, confirming the role of several Brodmann areas and related cognitive processes for bottom-up program comprehension. This replication strengthens the role of fMRI as a vital measurement instrument in software-engineering research. Second, we found that top-down comprehension leads to a lower activation intensity than bottom-up comprehension, increasing support from a neuroscience perspective for the hypothesis that top-down comprehension leads to neural efficiency. Finally, we found no evidence that beacons or program layout affect the comprehension process. However, it may be that the effect is just too small to detect with our experimental setup at the time.

Future Studies In the future, we would like to further explore which aspects of source code have a substantial influence on the comprehension process. Our study began this endeavor by looking at beacons and layout, which prior work focused on. In addition to the study presented here, we conducted a follow-up study on code layout. We used eye tracking to understand the effect of indentation levels on the programmers' visual effort. However, we found no significant evidence that indentation levels play a substantial role for program comprehension either [Bau+19].

In future studies, we shall also investigate other aspects of code, such as patterns [LMW79] and plans [Ric87] and various operationalizations of top-down program comprehension. This would help the community to gain a more holistic understanding of program comprehension. In the long run, this will allow us and other researchers to derive new rules for how to semantically and syntactically structure source code (for example, to highlight beacons in source code). Furthermore, we to improve programming education by helping novices focus on relevant parts of source code (e.g., to more quickly learn to identify and make use of beacons).

In the next two sections, we switch realms and shift the focus from source code to the programmer. In particular, we explore different experience levels change how programmers comprehend source code.

4.2 Reading Order of Novices and Experienced Programmers

After validating the neuronal difference between bottom-up and top-down comprehension, we tackle the next knowledge gaps: When and how can programmers apply top-down comprehension? In this section, we present an eye-tracking study that focuses on the way programmers *read* source code, which is a critical aspect of program comprehension.

Eye tracking has proved useful for observing programmers reading source code and answering such fundamental research questions on program comprehension as we introduced in Section 2.3.1. For example, Sharif and Maletic replicated a conventional study with eye tracking and found that naming style affects program comprehension in that programmers can read `under_score` style faster than `camelCase` style [Bin+09b; SM10].

Previous research suggested that the *linearity of the reading order* could be an indicator of how efficient programmers comprehend source code [Bus+15]. Busjahn et al.'s seminal study described several eye-gaze measures to gauge the linearity of reading order. They showed that programmers read source code less linear than natural text and also that expert programmers read source code less linearly than novices [Bus+15]. This study indicates that comprehending source code is a skill that needs to be developed and honed with experience. A replication of Busjahn et al.'s study by Peachock et al. supports the adequacy of the developed eye-gaze measures and partially corroborated Busjahn's study results with student participants [PIS17].

In this study, we further dig into the role of the linearity of reading order for program comprehension. Specifically, we aim at understanding how programmers' comprehension strategy and linearity of source code itself affect programmers' reading behavior. Understanding all factors that influence programmers' linearity of reading order is critical to measure program comprehension more accurately with eye tracking. To this end, we conducted a non-exact replication of the studies by Busjahn et al. and Peachock et al. with novice and intermediate programmers.

4.2.1 Original Study and Replication

Before describing our study, we briefly summarize the original study by Busjahn et al. as well as the replication study by Peachock et al.

4.2.1.1 Original Study (Busjahn et al.)

Busjahn et al. conducted a novel study on programmers' linearity of reading order [Bus+15]. They compared the linearity of reading order of novice and expert programmers as well as the novices' linearity of reading order for natural text and for source code. Busjahn et al. observed the eye movements of 14 computer-science students while reading source code as well as natural text in their weekly Java beginners course. The natural texts were short English passages of four to five lines. In addition, they asked 6 professional programmers to comprehend the source code and

observed their eye movements. Due to the novelty of this research question, they also described appropriate eye-gaze measures to quantify the *linearity of reading order*.

Experiment Design Ultimately, Busjahn et al. ran 17 trials of novices reading natural text, 101 trials of novices reading source code, and 21 trials of experts reading source code. As the novices were still learning to program, their snippets were simpler than the expert participants' snippets. Only two snippets had to be comprehended by both participant groups. For all snippets, participants were randomly asked one of three possible tasks: write a summary of the source code, compute the output, or answer a multiple-choice question. To observe eye movements, Busjahn et al. used an SMI RED-m remote eye-tracker with a sample rate of 120 Hz.

Participants 7 of the 14 novices were females. They were between 19 and 33 years old, had, at most, little programming experience, and all had, at least, medium English proficiency (while German being the native language). The experts were all professional programmers with, at least, 5 years of programming experience, and were between 26 and 49 years old. One of the 6 experts was female.

Measure	Definition	Computation
<i>Vertical Next Text</i>	% of forward saccades that either stay on the same line or move one line down	% of all F_i , where $L(F_i) - L(F_{i+1}) \in \{0, -1\}$
<i>Vertical Later Text</i>	% of forward saccades that either stay on the same line or move down any number of lines	% of all F_i , where $L(F_i) \leq L(F_{i+1})$
<i>Horizontal Later Text</i>	% of forward saccades within a line	% of all F_i , where $L(F_i) = L(F_{i+1}) \wedge W(F_i) \leq W(F_{i+1})$
<i>Regression Rate</i>	% of backward saccades of any length	% of all F_i , where $W(F_i) > W(F_{i+1})$
<i>Line Regression Rate</i>	% of backward saccades within a line	% of all F_i , where $L(F_i) = L(F_{i+1}) \wedge W(F_i) > W(F_{i+1})$
<i>Saccade Length</i>	Average Euclidean distance between every successive pair of fixations	$\frac{\sum_{i=1}^{n-1} \text{Distance}(F_i, F_{i+1})}{ F -1}$
<i>Story Order</i>	N-W alignment score of fixation order with linear text reading order	$\text{Alignment}(L(F), \text{story-order pattern})$
<i>Execution Order</i>	N-W alignment score of fixation order with the source code's execution order	$\text{Alignment}(L(F), \text{execution-order pattern})$

Table 4.3: Overview of gaze-based measures taken from Table 1 from Busjahn et al. [Bus+15]. In each trial, F is the set of all recorded fixations. F_i (where $i = 1, \dots, n$) is the fixation recorded at time index i . $L(F_i)$ is the line number of the fixation at index i . In each trial, W is the set of word indices in the text. $W(F_i)$ is the word number of the fixation at index i .

Variables The study of Busjahn et al. had two independent variables: programming experience (novice or expert, between-subject) and, for novices, whether the presented stimuli were source code or natural text (within-subject). Busjahn et al. analyzed the data in two steps: First, they contrasted how novices read source code versus natural text (within-subject). Second, they contrasted linearity of reading order between experts and novices (between-subject).

Dependent Variables To quantify the participants' linearity of reading order, Busjahn et al. describe a set of six eye-gaze measures, which we summarize in Table 4.3 and which we will also use for our data analysis.²¹ In essence, Busjahn et al.'s eye-gaze measures abstract a fixation sequence of (x,y) coordinates on the screen to higher-level concepts, such as *regressions*. Regressions are a sign that a participant had to revisit a previous part, which could be due to an insufficient understanding or following a snippet's execution (e.g., loop structures).

Operationalization of Reading Order In addition to the six fixation-based eye-gaze measures, Busjahn et al. analyzed the order in which each source code line was fixated and contrasted it with (a) the "story order" and (b) the execution order of a source code. The story order of a source code snippet is the sequence of each line from top to bottom, similar to natural text (e.g., 1, 2, 3, 4). The execution order of a source code snippet is the sequence of lines in which the code is executed, which may differ significantly from the story order (e.g., 3, 4, 2, 1, 2, 4).

The presented source code snippets contained up to 30 lines of source code. To efficiently compare long sequences of line numbers, Busjahn et al. relied on the Needleman-Wunsch (N-W) algorithm, which was originally designed for molecular comparisons of proteins [NW70] and later applied for use in eye-tracking research by Cristino et al. [Cri+10]. The N-W algorithm computes the similarity between two sequences. In this study, it can be interpreted as how similar an observed linearity of reading order is to a line-by-line reading order or a computer's execution order of the source code. Furthermore, as programmers often cannot comprehend a piece of source code in a single read, Busjahn et al. added a dynamic version of the N-W algorithm that tolerates multiple reads. In our data analysis, we also use both versions of the N-W algorithm to assess our participants' linearity of reading order.

Results Busjahn et al. reported two main findings: First, novice programmers read source code less linearly than natural text. Second, expert programmers read source code less linearly than novice programmers.

²¹Busjahn et al. also describe an *element coverage* measure, which we did not include due to our experiment setup.

4.2.1.2 Replication Study (Peachock et al.)

Experiment Design Peachock et al. replicated Busjahn et al.'s original study [PIS17]: It was also a mixed-model experiment with two independent variables (programming experience, between-subject) and stimuli (source code or natural text, within-subject). They also invited student programmers (33 overall, 18 male, 15 female) and asked them to comprehend seven short source code snippets and three pieces of natural language text. They used the same natural language snippets as Busjahn et al., but different source code snippets in C++. The source code contained some variety in complexity, but all on a rather low level. Similar to Busjahn et al., Peachock et al. asked three random comprehension questions after each task to ensure that participants fulfill the given task. Novice participants had no or only little contact with programming. The "expert" participants already had some programming experience, but were still undergraduate students.

There were some differences in their experiment design compared to the original study. Specifically, Peachock et al. used C++ (instead of Java) snippets; the natural language content was the same, but due to the different participant pool, it was in their native language (English). Unlike Busjahn et al., they did not invite expert programmers, but advanced undergraduate students referred to as "non-novice" participants. Peachock et al. used a Tobii X60 eye-tracker with a sample rate of 60 Hz.

Analysis Peachock et al. analyzed the dependent variable eye movements in terms of the Busjahn et al.'s linearity measures.

Results Peachock et al. reported the following results: First, programmers read source code less linear than natural text based on eye-gaze measures (replicated). Second, there is no significant difference in linearity of code reading order between novice and expert participants (not replicated). Third, there are significant differences between natural text and source code in terms of linearity of reading order based on the N-W score (replicated).

Implications for Future Research Busjahn et al. developed a tested way to quantify program comprehension: How *linear* do programmers read source code? With their methodology, researchers can tackle further research questions about source code readability and programmer education with affordable and reliable eye tracking. The two presented studies by Busjahn et al. and Peachock et al. consistently show that programmers read source code less linear than natural text.

As a next step, we were interested in how the linearity of the source code itself may affect programmers' linearity of reading order. Since the two studies did not explicitly manipulate the source code linearity, we conducted a non-exact replication.

4.2.2 Experiment Design

The overarching goal of our study is to gain a deeper understanding of *how source code, programming experience, and comprehension strategy affect linearity of reading order*. Specifically, we pose the following research questions:

RQ 4.4 Can we resolve the contradicting results of Busjahn et al. and Peachock et al. regarding whether more experienced programmers read source code less linear than novice programmers?

To evaluate RQ 4.4, we conducted a non-exact replication with novice programmers as well as more experienced programmers that can be categorized as *intermediate* programmers according to Dreyfus' taxonomy of skill acquisition [DD86; Mea+06]. Since we presented the same source code snippets to both groups, we can reduce the bias caused by differences between the source code snippets and increase internal validity compared to the original studies.

RQ 4.5 Does the comprehension strategy, that is, bottom-up and top-down comprehension, affect linearity of reading order?

To address RQ 4.5, we need to control the programmers' comprehension strategy. We operationalized the comprehension strategy by using meaningful versus obfuscated identifier names in the source code snippets to induce top-down or bottom-up comprehension (cf. Section 4.2.2.2, [Sie+17]).

RQ 4.6 Does the linearity of source code affect programmers' linearity of reading order?

To address RQ 4.6, we operationalize the linearity of source code. For this purpose, we have developed and validated a metric that quantifies source code execution order, which we describe next.

We provide a replication package, which includes all stimuli, acquired data, and analysis scripts on our project Web site.²²

²²<https://github.com/brains-on-code/eyetracking-linearity-replication/>

4.2.2.1 Source Code Linearity Metric i

To investigate how source code linearity affects linearity of reading order, we define source code linearity i as follows:

$$i = \frac{\Delta}{\bar{\Lambda}}, \text{ where } \Delta = \sum_{i=1}^{|M|} \delta_{m_i} \text{ and } \bar{\Lambda} = \frac{1}{|M|} \times \sum_{i=1}^{|M|} \lambda_{m_i} \text{ for } m_i \in M$$

with

$$\begin{aligned} \lambda_m &:= \text{length of a method} \\ M &:= \{m \mid m \in P, m \text{ is a method}\} \\ \delta_m &:= |C_m - D_m| \\ D_m &:= \iota_e \text{ for } e := \text{Declaration}(m) \\ C_m &:= \iota_e \text{ for } e := \text{Call}(m) \\ \iota_e &:= \text{index}(e) \text{ for } e \in P \\ P &:= \Omega_{\text{Program}} \end{aligned}$$

The linearity i of a source code snippet is the relation between the distances between jumps Δ and the average method length $\bar{\Lambda}$. A jump δ_m for a method m is the absolute distance between line ι where it is called (C_m) and the line where it is declared (D_m).

When programmers comprehend source code, they may follow its execution flow. When they encounter a method call while reading source code, their eyes may “jump” to the declaration of that method. Throughout the process of a thorough understanding of a source code snippet, over time this adds up to an overall “jump distance” Δ , which depends on the number of jumps and the distance of each jump. The distance of a jump is substantially influenced by method length, that is, when programmers have to jump to the subsequent method, they have to skip the entire length of the current method. Thus, we need to normalize the overall “jump distance” by the average length $\bar{\Lambda}$ for each method λ_m of all methods in a given source code snippet.

For example, the snippet `Calculation` in Figure 4.7 contains two methods, one with 7 and one with 4 lines. Thus, this source code snippet has an average method length $\bar{\Lambda}$ of 5.5 lines. The snippet contains only one method jump from line 11 to 2 (i.e., overall jump distance Δ is 9 lines). We divide the overall jump distance Δ by $\bar{\Lambda}$ and obtain a result of $i = 1.64$, a rather low value indicating a fairly linear snippet. We visualize the three large jumps of the less linear snippet `Student` in Figure 4.8.

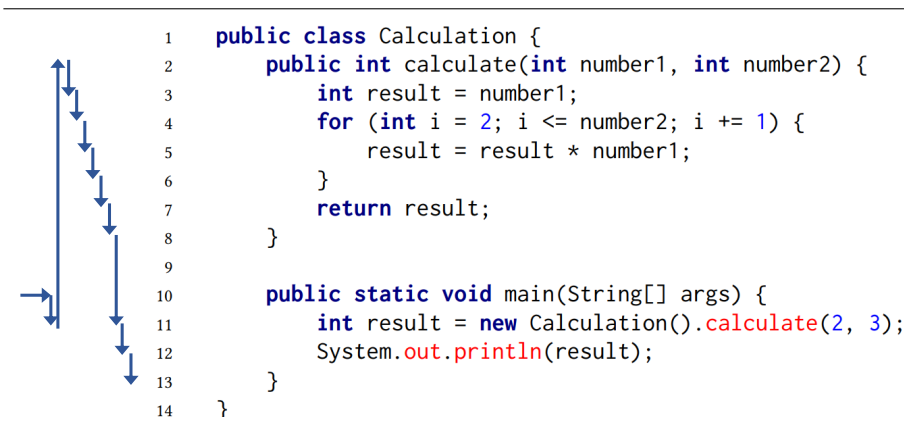
Prerequisites For these definitions to work, we assume that a given source code is a syntactically correct Java program, containing the declaration of a package, a class, and a static `main` function. Although only evaluated for Java source code, this principle can also be applied to other programming languages.

Snippet Comparability with i The source code linearity i allows us to compare source code snippets with higher sensitivity than other metrics (e.g., number or size of methods) with each other. The lower the source code linearity i , the more linear a source code snippet is (i.e., the flow of calls follows the order from the top of the screen to the bottom). The higher i is, the less linear is a source code snippet (i.e., the methods in the source code are not located in a position corresponding to the sequence of their calls).

Validation Study To evaluate whether the source code linearity i captures the intuitive notion of linearity, we conducted a validation study with ten advanced graduate students. In the validation study, participants compared two snippets regarding their perceived linearity. To this end, we selected 20 snippets with a wide range of linearity, as reflected in the source code linearity i . Then, we divided the snippets into three categories: linear, medium, and non-linear, and asked participants to compare two snippets of different categories (e.g., a linear and a medium snippet) and of the same category (e.g., two linear snippets). This way, we evaluated whether differences and similarities in linearity predicted by our source code linearity i are also reflected in programmers' intuitive notion of linearity.

We found that, for most of the snippets, the linearity metric reflected the perception of participants well. For the few snippets in which both judgments were different, we excluded such cases from the actual study to avoid having different linearity estimations. It would be interesting to refine the definition of source code linearity i in future studies.

4.2.2.2 Material



```

1  public class Calculation {
2      public int calculate(int number1, int number2) {
3          int result = number1;
4          for (int i = 2; i <= number2; i += 1) {
5              result = result * number1;
6          }
7          return result;
8      }
9
10     public static void main(String[] args) {
11         int result = new Calculation().calculate(2, 3);
12         System.out.println(result);
13     }
14 }

```

Figure 4.7: Source code snippet that elicits top-down comprehension with meaningful identifier names, which calculates the mathematical result of 2^3 . The source code can be largely read along its linear execution order from top to bottom (visualized with \rightarrow).

In line with our research goals, we used source code snippets that both novices and intermediate programmers can understand. We selected 10 snippets, 2 from each category (cf. Table 4.4). We

```
1  public class Otyrwpt {  
2      private String pckw;  
3      private int cgw;  
4  
5      public Otyrwpt(String pckw, int cgw) {  
6          this.pckw = pckw;  
7          this.cgw = cgw;  
8      }  
9  
10     public int gwtCgw() {  
11         return cgw;  
12     }  
13  
14     public int lcrHqjtlrcs() {  
15         return cgw = cgw + 1;  
16     }  
17  
18     public static void main(String[] args) {  
19         Otyrwpt iqffq = new Otyrwpt("XXXX", 25);  
20         iqffq.lcrHqjtlrcs();  
21         System.out.print(iqffq.gwtCgw());  
22     }  
23 }
```

The diagram consists of vertical blue arrows on the left side of the code snippet. An arrow starts at line 18 (the start of the main method) and points down to line 19. From line 19, an arrow points down to line 20. From line 20, an arrow points down to line 21. From line 21, an arrow points down to line 22. From line 22, an arrow points down to line 23. From line 23, an arrow points down to line 18. From line 18, an arrow points down to line 5. From line 5, an arrow points down to line 6. From line 6, an arrow points down to line 7. From line 7, an arrow points down to line 8. From line 8, an arrow points down to line 10. From line 10, an arrow points down to line 11. From line 11, an arrow points down to line 12. From line 12, an arrow points down to line 14. From line 14, an arrow points down to line 15. From line 15, an arrow points down to line 16. From line 16, an arrow points down to line 18.

Figure 4.8: Source code snippet with obfuscated identifiers that prints a student's age after a birthday. The snippet requires programmers' eyes to vertically jump between methods to follow execution flow (visualized with \rightarrow).

re-used 8 Java snippets from the original study by Busjahn et al. (including both snippets that were shown to novices and experts). In addition, we created 2 new snippets (CheckIfLetters, SumArray), comparable in complexity and content. All snippets are relatively short with, at most, 30 lines of code. The source code snippets implement algorithms commonly used in computing education (e.g., insertion sort). For example, Figure 4.7 shows a snippet that calculates the cubed number of 2.

Each snippet contains a single class with one `main` method, up to 4 helper methods, and, at least, one `System.out.print()` statement composing the snippet's result.

4.2.2.3 Independent Variables

Our study design contains three independent variables:

- Programming experience (novice vs. intermediate programmers, between-subject)
- Top-down vs. bottom-up comprehension (meaningful vs. obfuscated identifier names, within-subject)
- Linearity of snippets (5 categories of A, B, C, D, E, with A being most linear and E being least linear, within-subject, cf. Section 4.2.2.1)

```

L0 public class Calculation {
L1     public int calculate(int number1, int number2) {
L2         int result = number1;
L3         for (int i = 2; i <= number2; i += 1) {
L4             result = result * number1;
L5         }
L6         return result;
L7     }

L9     public static void main(String[] args) {
L10        int result = new Calculation().calculate(2, 3);
L11        System.out.println(result);
L12    }
L13 }

```

Figure 4.9: Visualization of the areas-of-interest (AOIs) of the snippet “Calculation”. The AOIs are wrapping individual code lines and functions, as described by Busjahn et al. [Bus+14].

To operationalize comprehension strategy, we created a second version of all 10 snippets, in which we used obfuscated (instead of meaningful) identifier names (as done in Section 4.1). The motivation to distinguish bottom-up and top-down comprehension, and to observe them separately, is two-fold: First, bottom-up comprehension reduces the advantage of prior programming experience [Pen87] as done by previous studies [Sie+14a; Bau+19; Nak+14; IU14; Lee+16; Lee+17; Iku+21]. Second, the direct contrast between meaningful and obfuscated identifier names allows us to investigate how eye movements change depending on the comprehension strategy. We show an example of an obfuscated snippet that computes the age after a birthday in Figure 4.8.

To operationalize source code linearity, we calculated i for all candidate snippets and assigned them to a category (A, B, C, D, or E). The category of a snippet indicates to which 20% percentile its linearity i belongs. For example, a linearity $i = 1.64$ belongs to the 30% percentile and thus is part of category B.

Table 4.4 lists all snippets, their linearity values, and an overview of the behavioral results. All snippets in both versions, meaningful and obfuscated, along with their solutions, are available in our replication package.

4.2.2.4 Dependent Variables

We consider two dependent variables: behavioral data (i.e., response time and correctness) and eye gaze. We define response time as the time from a participant first viewing a snippet until the time they submit their answer. The raw data of the observed eye gaze is a stream of (x,y) coordinates on the screen, which we used to compute the study’s measures by Busjahn et al. (cf. Table 4.3, [Bus+15]).

Snippet	Metrics		Linearity i (Category)	Variant		Novices		Intermediate Programmers	
	LOC	# Methods		Correctness	Response Time [s]	Correctness	Response Time [s]	Correctness	Response Time [s]
MoneyClass	8	1	0.00 (A)	Meaningful	7/7	37.6	11/12	31.0	
SumArray*	12	1	0.00 (A)	Obfuscated	5/5	55.3	7/7	44.2	
CheckIfLetters*	21	2	1.56 (B)	Meaningful	5/5	38.7	14/15	35.1	
				Obfuscated	3/4	42.7	4/4	47.3	
Calculation	14	2	1.64 (B)	Meaningful	7/7	81.2	9/10	64.9	
				Obfuscated	2/2	15.1	8/9	101.4	
InsertSort	29	3	3.24 (C)	Meaningful	6/8	72.4	12/14	47.4	
				Obfuscated	4/4	78.4	5/5	99.8	
Vehicle	26	3	4.42 (C)	Meaningful	2/9	163.0	3/12	153.9	
				Obfuscated	-	-	4/7	218.1	
Student	23	4	8.27 (D)	Meaningful	7/7	86.1	16/16	54.1	
				Obfuscated	5/5	107.4	3/3	85.8	
SignChecker	26	3	9.90 (D)	Meaningful	8/9	29.1	12/12	29.2	
				Obfuscated	2/3	64.1	7/7	46.1	
Street	21	4	10.57 (E)	Meaningful	8/8	78.6	17/17	56.7	
				Obfuscated	2/4	107.1	2/2	68.7	
Rectangle	29	5	20.00 (E)	Meaningful	6/6	26.4	14/14	30.1	
				Obfuscated	5/6	172.6	5/5	45.7	
Overall				Meaningful	9/9	94.6	11/11	59.9	
				Obfuscated	2/3	77.2	8/8	90.6	
				Meaningful	65/75 (87%)	76.3 ± 54.0	120/133 (90%)	55.0 ± 39.9	
				Obfuscated	30/36 (83%)	96.9 ± 55.7	53/57 (93%)	89.6 ± 60.4	

Table 4.4: All snippets used in the study, their metric values, and experimental results. For the metrics columns, darker shading indicates higher values. Unless noted with *, all snippets were part of the study by Busjahn et al. [Bus+15]. How often a snippet had to be solved by a group is unbalanced due to a randomized presentation of snippets regarding linearity and variant.

4.2.2.5 Task

We asked participants of both experience levels and both snippet versions to enter the result of the final `print` statement for all presented snippets. For example, for the snippet of Figure 4.7, the correct output is “8”. This task setup is a simplified version from the original study, where Busjahn et al. randomly selected between computing output, a comprehension summary, or multiple-choice questions. The rationale of fixing the task to computing the output is that we aimed to eliminate the chance that the kind of task affects participants’ comprehension strategy (in addition to the source code linearity and snippet obfuscation).

4.2.2.6 Participants

	University Passau	University Weimar	University Magdeburg
Novices	5	2	5
Intermediates	0	17	2

Table 4.5: Recruitment universities for our two participant groups with basic programming experience (Novices) and intermediate programming experience (Intermediates).

	Novices (n=12)	Intermediate Programmers (n=19)
Male	9 (75%)	18 (95%)
Female	3 (25%)	1 (5%)
Age (in Years) \pm SD	21.4 \pm 2.3	29.9 \pm 4.6
Years of Programming \pm SD	3.3 \pm 1.8	12.2 \pm 6.1
Years of Java Programming \pm SD	3.0 \pm 2.3	6.8 \pm 5.5

Table 4.6: Demographic data of our participants. Our intermediate programmers tend to be older but also have more experience specific to Java and general programming.

We recruited participants for both groups from three universities, which we detail in Table 4.5. Novice participants had a fundamental understanding of Java and object-oriented programming (i.e., passed, at least, an introductory programming class). Our intermediate programmers were advanced graduate students of computer science (i.e., higher-level master or PhD students in computer science or a related field). We verified our selection with a small programming questionnaire during the experiment [Sie+14b]. We summarize our participants’ demographics and programming experience in Table 4.6. The experience of our intermediate programmers lies between the two previous studies’ “expert” groups. We categorized our participant groups as novice and intermediate programmers according to Dreyfus’ taxonomy of skill acquisition [DD86; Mea+06].

Due to our eye-tracker's requirements, only programmers without eye-vision conditions (e.g., strabismus; corrective glasses or lenses were acceptable) were eligible to participate in our study.

4.2.2.7 Experiment Procedure

Eye-Tracker We used a Tobii EyeX eye-tracker with a sample rate of 60 Hz. Since we collected data at three different universities, different screen sizes and resolutions were in use: 1920×1200 , 1920×1080 , and 1680×1050 . We scaled all analyses according to the respective screen resolution.

Data Collection The experimenter led participants by explaining the experiment, all program comprehension tasks, and finally demographic and eligibility questions.

We assigned most participants 10 snippets, except for the first 3 novices, who got only 7 snippets. The first three participants were faster than expected and reported little exhaustion from the meaningful snippets. Thus, we increased the number of meaningful snippets from 4 to 7. For all subsequent participants, we presented 3 obfuscated (bottom-up comprehension) snippets and 7 meaningful (top-down comprehension) snippets, which led to an imbalance between the number of the two comprehension strategies (cf. Table 4.4). We pseudo-randomized the order and the selection in which snippets were presented (i.e., we ensured the split between obfuscated and meaningful snippets, but besides that, everything else was random). This way, we avoided bias due to ordering effects.

Execution We obtained 15 eye-tracking data sets from novices and 19 from intermediate programmers. We excluded three data sets from novices, because the eye-tracker failed to track more than one minute of data due to a setup issue. Therefore, all subsequent data analyses (including behavioral data) are based on 12 novice participants.

Deviation Three obfuscated snippets contained (by mistake) an error that would prevent compilation. We discuss an interesting observation on how programmers with different experience levels handle this issue in Section 4.2.5.1.

4.2.3 Data Analysis

4.2.3.1 Behavioral Data

To decide whether an answer was *semantically* correct, we manually evaluated each response. We interpreted responses with only minor formatting inaccuracies as semantically correct (e.g., if

a participant responded with a value of “1.4” instead of “1.40”).

4.2.3.2 Eye-Tracking Data: Preprocessing

The eye-tracker provides a stream of (x,y) coordinates. We applied several standard preprocessing steps to ensure high data quality and reliability.

First, we smoothed the stream of (x,y) coordinates with a Savitzky-Golay filter (window length of 5, polynomial order of 3) [NH10]. Next, we applied a velocity-based algorithm to detect fixations and saccades from the eye gaze. We used a velocity threshold of 150 pixel in 100 milliseconds. If the velocity was below the threshold, it was interpreted as a fixation, otherwise as a saccade [Hol+11].

Second, we created areas-of-interest (AOIs) for each line and block of each snippet. The AOIs allow us to compute the measures that describe the linearity of reading order as described by Busjahn et al. For all subsequent analyses based on AOIs, we filtered out all fixations outside of defined AOIs (e.g., participants looking around the room). Following Busjahn et al., we included fixations with a maximum of a 100-pixel horizontal deviation (\approx 7–8 characters), as small AOIs can otherwise be easily missed (e.g., a closing bracket) and may distort the results.

4.2.3.3 Eye-Tracking Data: Analysis Procedure

After preprocessing, we computed the measures capturing linearity of reading order developed by Busjahn et al. (cf. Table 4.3) for each participant group and experimental factor.

Since we have more than one independent variable, we compute a linear mixed regression model, which allows us to also detect possible interaction effects [LB88]. This analysis differs from Busjahn et al., who only had one independent variable and used Mann-Whitney-U tests to test for significant differences. For each of the Busjahn et al.’s measures described in Table 4.3, we computed a linear mixed regression model with three factors: programming experience (novice or intermediate programmer), comprehension mode (top-down or bottom-up), and source-code linearity (in five categories: A, B, C, D, E). We used the R `lme4` package, version 1.1.21, to compute the linear model [Bat+15]. Data across all measures yielded in a converged model, indicating that the provided factors can explain the observed variance. We subsequently tested the fitted model for statistical significance with the `car` package [FW19], version 3.0.6, which internally uses a type II Wald chi-square test.

To avoid an inflated probability of the type-I error (i.e., incorrect rejection of a null hypothesis) due to multiple statistical testing, we adjusted the significance threshold with a false-discovery-rate (FDR) correction [BH95]. This resulted in an adjusted p -level significance threshold of 0.033. Thus, we consider only results with a p -level below 0.033 as statistically significant.

4.2.4 Results

In this section, we report the results of the behavioral and eye-tracking analysis, followed by our interpretation in Section 4.2.5.

4.2.4.1 Behavioral Data

	Novices (n=12)	Intermediate Programmers (n=19)
Correct Responses (All)	86%	91%
Correct Responses (Meaningful)	87%	90%
Correct Responses (Obfuscated)	83%	93%
Response Time (All, in sec)	83.0 ± 55.1	65.4 ± 49.4
Response Time (Meaningful, in sec)	76.3 ± 54.0	55.0 ± 39.9
Response Time (Obfuscated, in sec)	96.9 ± 55.6	89.6 ± 60.4
Compiler Errors Detected	7 of 11 (64%)	2 of 7 (28%)

Table 4.7: Behavioral data separated by programming experience. Intermediate programmers are faster but miss more compiler errors. Gray font color marks non-significance.

We show a summary of the behavioral results between the two participant groups in Table 4.7 and a detailed version for each snippet, variant, and group in Table 4.4. While novices achieved a similar correctness rate as the intermediate programmers (86% vs. 91%, Mann-Whitney-U test²³: $U = 11121, p = 0.072$), they were across all snippets significantly slower, on average (83 sec vs. 65 sec, $U = 8001, p = 0.000$). Intermediate programmers showed a faster comprehension when snippets contained meaningful identifier names, which facilitate top-down comprehension (76 sec vs. 55 sec, $U = 3562, p = 0.000$). But, when we obfuscated identifier names enforcing bottom-up comprehension, intermediate programmers fall back close to the speed of novices (97 sec vs. 90 sec, $U = 889, p = 0.142$).

Unlike intermediate programmers, most novices found the accidental compiler errors in the obfuscated snippets (cf. Section 4.2.2.7). However, this difference is non-significant ($U = 52, p = 0.079$).

4.2.4.2 Eye-Tracking Data

In Table 4.8, we provide an overview of the eye-tracking results for all three RQs. In essence, we can *partially replicate* Busjahn et al.'s results that intermediate programmers read source code less linear and *contradict* Peachock et al.'s negative result (RQ 4.4). Specifically, we found

²³t tests are inappropriate as Shapiro-Wilk tests [SW65] showed non-normality for response correctness and times.

	Busjahn [Bus+15] Experience*	Peachcock [PIS17] Experience*	Experience**	Our Study Comprehension Strategy**	Source Code Linearity ⁱ (A, B, C, D, E)**
Vertical Next	$p < 0.001, E > N$	$p = 0.406$	$p = 0.031, E > N$	$p = 0.148, TD > BU$	$p = 0.000, A > B > C > D > E$
Vertical Later	$p < 0.010, E > N$	$p = 0.461$	$p = 0.003, E > N$	$p = 0.059, TD > BU$	$p = 0.000, D > E > C > B > A$
Vertical Regressions	$p < 0.001, N > E$	$p = 0.453$	$p = 0.010, E > N$	$p = 0.000, BU > TD$	$p = 0.000, D > E > C > B > A$
Horizontal Later	$p < 0.001, E > N$	$p = 0.487$	$p = 0.008, N > E$	$p = 0.001, TD > BU$	$p = 0.000, A > B > C > D > E$
Horizontal Regressions	$p = 0.970, E > N$	$p = 0.973$	$p = 0.044, N > E$	$p = 0.450, TD > BU$	$p = 0.001, A > B > C > D > E$
Saccade Length	$p < 0.001, E > N$	not provided	$p = 0.903, N > E$	$p = 0.046, TD > BU$	$p = 0.000, D > E > A > C > B$
Story Order (Naïve)	$p < 0.001$	not provided	$p = 0.909, N > E$	$p = 0.000, TD > BU$	$p = 0.000, A > D > E > B > C$
Story Order (Dynamic)	$p < 0.0001$	not provided	$p = 0.557, N > E$	$p = 0.000, TD > BU$	$p = 0.000, A > D > B > E > C$
Exec. Order (Naïve)	not provided	not provided	$p = 0.809, N > E$	$p = 0.000, TD > BU$	$p = 0.000, A > D > B > E > C$
Exec. Order (Dynamic)	not provided	not provided	$p = 0.932, N > E$	$p = 0.000, TD > BU$	$p = 0.000, A > B > D > E > C$

* Matt-Whitney U Tests ** Linear Mixed Model with Wald Chi-Square Test (Significance Threshold 0.033 after FDR Correction)
 N = Novice, E = Expert/Intermediate Programmers BU = Bottom-Up, TD = Top-Down

Table 4.8: Overview of the three studies' eye-tracking results. Inequality symbol $>$ indicates in which direction the linearity of reading order is influenced (e.g., $E > N$ would signal that experts exhibit a more linear reading order than novices). Gray font color marks non-significance. Text highlighted with green indicates replicated results, while purple indicates that our results are different from Busjahn et al. We did not find any significant interaction effects.

evidence that intermediate programmers show significantly more *vertical next* and *vertical later* eye movements, that is, their eye gaze jumps to the next or a source code line further down (cf. Table 4.3). In contrast to Busjahn et al., we observed that intermediate programmers use more *vertical regressions*, that is, that the percentage of their eye gaze movements going upwards in source code is larger than that of novices. We cannot confirm Busjahn et al.'s findings with the *N-W algorithm* in our data set, which yielded non-significant results. In other words, novice and intermediate programmers' reading order is not different for our sample.

We observed mixed results about whether and how programmers' comprehension strategy affects their linearity of reading order (RQ 4.5). First, programmers using top-down comprehension show percentage-wise less *vertical regressions*, but more *horizontal later* eye movements than in bottom-up comprehension. In other words, programmers' eyes seem to move horizontally within a line more during top-down comprehension. Although programmers use longer saccades during top-down comprehension, this effect is not significant (after FDR correction). The *N-W scores* for bottom-up comprehension are significantly smaller than for top-down comprehension, indicating that bottom-up comprehension is closer to a top-to-bottom reading order, whereas top-down comprehension expresses itself in more wandering eye movements.

In our study, source code linearity i significantly affects all observed linearity measures (RQ 4.6). In general, a higher linearity score i (indicating source code with many large vertical jumps in its execution order) leads to longer vertical eye movements, but fewer short eye movements (i.e., within one or two neighboring source code lines). While the order of the linearity categories A, B, C, D, E appear sensible for vertical and horizontal eye movements, they are mostly inconsistent for *saccade length* and the *N-W scores*. In other words, source code linearity does neither seem to influence the average eye jump distance, nor how similar the reading order is to execution or story order.

4.2.5 Discussion

4.2.5.1 Behavioral Data

We expected that more experienced programmers are generally faster [Shn77; McC11], but when bottom-up comprehension is enforced, the differences may vanish based on a contradictory result from studies of Soloway and Ehrlich [SE84] and Gilmore and Green [GG88]. We indeed observed that intermediate programmers are significantly faster than novices when meaningful identifier names facilitate top-down comprehension. While the performance gap is significantly reduced during bottom-up comprehension, novices are still slower. Our result is thus in between Soloway and Ehrlich's vanishing effect [SE84] and Gilmore and Green's result that experienced programmers stay faster [GG88].

```
1 Cjviiij cjviiij = new Cjviiij(5);  
2 cjviiij.cijTqmniv(15);  
3 System.out.print(cjviiij1.wijTqmniv());
```

Listing 4.4: Part of the obfuscated snippet *Street*, which contains a compiler error. Most intermediate programmers miss the non-initialized variable in Line 3, while novices tend to notice it.

Spotting Compiler Errors in Snippets Three of the obfuscated snippets (i.e., *Street*, *SignChecker*, *CheckIfLettersOnly*) mistakenly contained undefined function and variable identifiers (e.g., the variable *cjviiij1* in Listing 4.4 should be *cjviiij*). This results in a compiler error and therefore not to a determined output. Interestingly, many novices spotted this error, while most intermediate programmers did not.

An early study of Shneiderman and Mayer showed that experts focus on semantic aspects of source code, while novices concentrate on syntax [SM79]. We discussed this phenomenon with three of our intermediate programmers. They generally confirmed that these early results still hold true in modern times, especially with IDE support. They reported that they are used to an IDE highlighting basic compiler errors due to their daily work. One reported that “you have to look for compiler errors to find them”, which due to the missing IDE highlighting (e.g., red underline) may not be on the mind of a programmer. In addition, one reported that “at that spot, you recognize the object based on the name without checking every individual character”. Novices appear to be less likely to take such mental shortcuts and need to learn to focus on semantic aspects of source code.

4.2.5.2 Eye-Tracking Data

RQ 4.4 Can we resolve the contradicting results of Busjahn et al. and Peachock et al. regarding whether more experienced programmers read source code less linear than novice programmers?

The two studies by Busjahn et al. and Peachock et al. did not draw a clear picture of whether there is a distinguishable difference in linearity of reading order with increased programming experience: Busjahn et al.’s expert programmers were significantly different across several reading linearity measures, while Peachock et al.’s “non-novices” were not. It is interesting to learn that our data partially replicate Busjahn’s results with an intermediate programmer group, which brings us closer to learn with what experience level programmers change their reading order.

Unlike Busjahn et al., we find that intermediate programmers use significantly more eye movements across *all three vertical* measures. This result is consistent with the notion that experienced programmers’ eyes jump through source code, looking for “beacons” [Bro83; SFM12]. Intuitively, it also makes sense that all three measures capturing vertical eye movements point in the same direction (in our case, intermediate programmers use more vertical eye movements).

Busjahn et al. also observed that experts more often stay on a single line and read it from left to right, which, however, appears contradictory to the interpretation that novices read source code more “book like” from top to bottom and left to right, while experts’ eyes jump around more. Our data support the view that more experienced programmers use less horizontal eye movements than novices. This result is plausible because novices use more bottom-up comprehension, which leads to reading entire source code lines from left to right. Our result is in general agreement with the notion that experts apply a more erratic but intentional search through source code and novices a more repetitive gaze pattern [Bed12].

Again, in contrast to Busjahn et al., we did not observe significant differences in saccade length or reading order between the two participant groups. We suspected that our results diverge from Busjahn et al., because we showed both participant groups the same snippets. The average saccade length for both participant groups was the same, while Busjahn et al. showed longer snippets to experts. However, when we normalize the observed saccade length with the snippet length, this actually reverses, such that longer snippets lead to shorter (normalized) saccades. Thus, our results are inconclusive, and we cannot be confident how snippet length, saccades, and experience interact.

RQ 4.4 Overall, we were able to partially corroborate Busjahn et al.’s result and contradict Peachock et al.’s result. Intermediate programmers read source code less linear than novices.

RQ 4.5 Does the comprehension strategy, that is, bottom-up and top-down comprehension, affect linearity of reading order?

We aimed at understanding whether the comprehension strategy, that is, top-down or bottom-up comprehension, leads to a significant difference in programmers’ eye movements (RQ 4.5). We expected that programmers using top-down comprehension show more vertical eye movements and less horizontal eye movements, as top-down comprehension is instead a hypothesis-driven comprehension strategy, whereas bottom-up comprehension requires building up a source code snippet’s meaning by reading every line.

However, our eye-tracking results are inconclusive: The differences in the fixation-based linearity measures were only significant for two measures, which do not converge to an apparent interpretation. All other measures were non-significant, leading to an overall unclear picture about whether top-down comprehension entails more vertical movements.

Interestingly, the reading order based on the *N-W* scores, which did not discriminate between our novice and intermediate programmers, shows highly significant differences between bottom-up and top-down comprehension. In other words, on average, the reading patterns during bottom-up comprehension are closer to a snippet’s story order, whereas reading patterns during top-down

comprehension are closer to a snippet's execution order. But, we note that all scores are rather low (smaller than -100), indicating that there still are large differences between expected reading order and actual eye movements during program comprehension.

RQ 4.5 While we partly uncover a less linear reading order during top-down comprehension, most measures are inconclusive. Thus, a difference of eye-movement patterns between bottom-up and top-down comprehension is not supported by our data.

RQ 4.6 Does the linearity of source code affect programmers' linearity of reading order?

One goal of our study was to understand whether the linearity of source code significantly affects programmers' eye movements. The linearity of source code showed a strong effect on reading order, for both novice and intermediate programmers and both comprehension strategies. In other words, linearity of source code seems to be the major driving factor that determines the reading order, whereas experience and the comprehension strategy play a more minor role. While this may not be surprising, we have provided empirical evidence that this actually is the case.

The two original studies paved the way to study programmers' reading order. We took one further step and varied source code linearity in a systematic way to understand its influence. With our setup, we found that there are two separate effects that influence the reading order: On the one hand, efficient program comprehension by more experienced programmers or top-down comprehension leads to larger vertical eye movements, because programmers search for certain features to quickly verify their hypothesis of a snippet's purpose. These are intentional eye movements that are necessary to adeptly comprehend source code. On the other hand, less linear source code forces programmers' eyes to make large vertical jumps when they try to follow a snippet's method call chain. Thus, while the former are eye movements initiated by the programmer's internal cognitive processes to make the comprehension process efficient, the latter are eye movements that are imposed by external factors. Both types of eye movements may also interact, such that a less linear structure makes it more difficult for the programmer to make the intentional, hypothesis-confirming eye movements. In other words, linearly structured source code could reduce the required eye movements and make the comprehension process more efficient, if it supports the programmer's hypothesis-confirming eye movements.

However, our less linear snippets tend to be more complex, so we cannot draw robust conclusions from the conducted study. Instead, we call for further dedicated studies that specifically contrast source code with different internal structures, but implement the same algorithm to understand how programmers shall organize methods in a class [GM16]. Our source code linearity measure i , although incomplete for some exceptional cases, can be a starting point to systematically operationalize the linearity of source code.

RQ 4.6 The linearity of source code strongly affects programmers' reading order: Less linear source code leads to large vertical eye movements.

4.2.6 Threats to Validity

4.2.6.1 Construct Validity

Several of our used eye-movement measures are based on matching a fixation to a source code line. This leaves some room for interpretation, as participants may use peripheral vision and do not need to focus precisely on a source code line [Orl17]. Like Busjahn et al., we interpreted a fixation to be on a source code line if it was horizontally less than 100 pixels away.

Similarly, there is room for interpretation when computing the execution order of source code. For example, how to interpret lines that only contain a closing bracket (`}`) is debatable or whether class definition code (`public class Example`) should be considered as part of the execution order. We included all brackets and boilerplate code in the execution flow (as an interpreter would step through the code). This technical interpretation may divert from how humans read code and likely contributed to low N-W scores.

To operationalize linearity, we developed a source code linearity metric and validated it with intuitive notions of linearity of programmers. Although there are some deviations of the intuitive perception and the linearity metric, this did not pose a threat to construct validity, as we excluded such cases from our study.

4.2.6.2 Internal Validity

There are several threats arising from our participant sample. First, we have a skewness in gender distribution, which, however, is close to the actual population in computer science for our universities. Second, we have to question whether our participant group division was reasonable: Are our intermediate programmers actually sufficiently experienced programmers? To ensure a correct assignment, we asked participants a few questions regarding their experience, based on a questionnaire developed by Siegmund et al. [Sie+14b]. Furthermore, the behavioral data indicate that our assignment was reasonable.

Finally, regarding our eye-tracking data: We did not apply a manual correction of the scan paths, which can be error prone [PS16]. A visual exploration of the obtained data showed reasonable preciseness, so we do not consider our results threatened without a manual correction. We also did not apply a drift correction [Hol+11], as our experiment was comparably short, so we do not expect meaningful drift.

4.2.6.3 External Validity

Our study exhibits the typical threats of having small Java programs and recruited students. Our results can only be carefully generalized to other contextual factors. Reading behavior of larger snippets with higher control-flow complexity may show different results [CS90; JF17]. Nevertheless, our setting targets a critical population.

Our study used comparably small Java snippets with up to 30 lines. Program comprehension of larger systems is driven by a different cognitive process and therefore our results may not transfer to large systems.

4.2.7 Related Work

Original Study and Replications In addition to the original study and its first replication introduced in Section 4.2.1, Blascheck and Sharif conducted another replication, albeit focused on introducing a new methodology of visualizing the linearity of reading order [BS19].

Eye Tracking on Program Comprehension As introduced in Section 2.3.1, numerous studies used eye tracking to observe program comprehension besides Busjahn’s original study and Peachock’s replications. For example, Turner et al. used eye tracking to investigate the difference in bug searching tasks between C++ and Python source code, which yielded no significant difference [Tur+14]. Binkley et al. investigated the difference in identifier styles (under_score vs. camelCase) and found that it is mostly a matter of preference, with experts’ comprehension being more affected by the identifier style [Bin+13; SM10]. There are several other studies related to program comprehension covering method summarization [Abi+19; RM15], syntax highlighting [BP16], code review [Uwa+06; SFM12], and programmer education and expertise [Bus+14; Pet+19]. They all investigate one specific aspect of program comprehension, but did not focus on reading order.

Source Code Metrics and Readability While there is a plethora of source code metrics [Var+17], we are not aware of one that is designed to specifically capture the linearity of source code. We consider the linearity of source code as an element of *code readability*, which captures all syntactic factors that affect programmers. For example, Buse and Weimer asked programmers how readable source code is and, based on the results, build a predictive model to estimate source code readability [BW08; BW10]. A follow-up by Posnett et al. showed that more common metrics can similarly predict readability [PHD11]. But, Jbara and Feitelson provide evidence that common metrics overestimate repeated code constructs (e.g., `if/else`) [JF17]. Furthermore, these readability studies work on individual methods rather than (small) classes. On a class-level, many studies focus on *maintainability* [Col+94] or *complexity* [Les+08], rather than fundamental readability.

4.2.8 Conclusion

In this section, we set out to investigate the effect that source code linearity, programming experience, and comprehension strategy have on reading order of source code. Our results indicate the linearity of source code is a major driving factor that determines programmers' reading order, while experience and comprehension strategy seem to play more minor roles. With our intermediate programmers' experience level lying between the two previous studies, we seem to have found a turning point when programmers switch from a linear reading order to a reading order following the execution order. The strong effect of linearity implicates that the structure of the source code should be matched to the programmer's expectations to avoid unnecessary eye movements, which may make program comprehension more efficient.

In the next section, we dig deeper into the effects of different experience levels of programmers with the knowledge that there is a significant difference between novice and intermediate programmers in how they comprehend source code.

4.3 Brain-Activation Patterns of Novices and Experienced Programmers

Based on our fMRI study on top-down comprehension (cf. Section 4.1) and our eye-tracking study on reading linearity (cf. Section 4.2), we gathered preliminary evidence that there is a measurable difference in cognitive processes of programmers with varying levels of experience. From past program-comprehension research, there are also (unvalidated) theories that experienced programmers are more likely to employ top-down comprehension than novices [SE84].

In this section, we describe our preliminary work to contrast the cognitive processes of program comprehension of novice programmers to those of experienced programmers. We aim to gather insights into how experienced programmers think differently when comprehending source code. To this end, we set up the following plan: First, we draw from literature to understand how we can measure expertise from a neuro-cognitive perspective (Section 4.3.1). Second, we explore our previous fMRI data set to generate hypotheses (Section 4.3.2). Third, we outline an fMRI study to objectively observe the brain-activation patterns of novice programmers to those of experienced programmers (Section 4.4.1).

4.3.1 Literature Review

In this section, we draw from literature to bridge together several insights into programmer expertise. To this end, we view expertise from three perspectives: cognitive psychology, neuroscience, and past software-engineering theories and pose the following research question:

RQ 4.7 How can expertise modulate programmers' cognitive processes based on insights from cognitive psychology, neuroscience, and software engineering?

4.3.1.1 Insights from Cognitive Psychology

Research in cognitive psychology revealed that experts demonstrate a mastery of skills obtained through many years of *deliberate practice* [EL96]. In understanding what makes an expert an expert, scientists have studied the training strategies, cognitive representations, and problem-solving strategies used when performing tasks with exceptional skill. Initial research focused on the training strategies and researchers found that there is a consistent difference between experience (in terms of years spent) and expertise (performance levels) [CJ91; SS92]—the primary difference in performance arises from *how* experts were trained and not necessarily how *long*. For example, when comparing chess experts who have spent equal time in gaining experience, the consistently best performers are the ones who repetitively studied specific chess positions and scenarios, as opposed to lower performers who just practiced in tournaments [CKM96].

Cognitive expertise involves chunking of information, or organizing a stream of perceptual cues into a more meaningful pattern [DG78]. Experts use more effective *problem representations* and generate better “next steps or moves” (in chess) [Sim90] or select the best diagnostic option (in medicine) [ESS90]. Experts differ from novices in how they process information and arrive at an answer, such that they look a bit deeper and process next steps faster [Hol92], resulting in improved qualities of answers [ESS90].

4.3.1.2 Insights from Neuroscience

Neuroimaging studies found that experts demonstrate more efficient neuronal activation patterns than novices with the same tasks [NF09]. Expert brains work differently than non-expert brains. When novices are compared with experts performing the same kinds of tasks, the differences can be remarkable. When novice golf players try to perform a golf swing, their brains are alight with activity throughout many areas of the brain as they clumsily try to coordinate the swing in their mind, whereas experts have conceptualized the movements of a golf swing into a simple, focused, and energy-efficient action in the brain [Mil+07]. Not only does an expert's brain act more efficiently, experts sometimes also have a larger brain mass in these areas. A larger right posterior parietal cortex is seen in expert video game players [Tan+13]. Experienced London taxi drivers have larger parahippocampal regions with size correlated with years of experience [MWS06].

Although the brain appears to have specialized areas for specialized tasks, when humans develop new skills, there is often no specific area of the brain that supports that skill. Instead, learning processes often recruit existing information-processing networks of the brain in support of the new skill. For example, the fusiform face area is strongly associated with face perception, but it also gets recruited when identifying a specific object, such as for bird experts who can distinguish

between a vast variety of bird species or for car experts who can identify distinct differences between many models and makers of cars. Interestingly enough, bird experts who are not car experts do not use the fusiform face area when observing cars, and vice versa [Gau+00]. Other studies also show the involvement of the fusiform face area in experts of tasks that require visual perception, including the categorization of chest radiographs [Har+09a] or understanding chess positions [Bil+11].

4.3.1.3 Insights from Software Engineering

Syntax vs. Semantic Processing Software-engineering researchers have proposed that programmers use knowledge structures that encode semantic [SM79] and domain information [Bro83] about a program as well as *prime structures* [LMW79], which include elements of syntax, control-flow and data-flow [Pen87] of the program. These knowledge structures [Ric87] have been formalized referred to as *programming plans*. Motivation for programming plans was inspired from theoretical constructs in text comprehension, such as *scripts*, which are mental representations of common activities (e.g., eating in a restaurant) and can aid humans in understanding and remembering narrative text [BBT79]. Programming plans act like schemas that are first instantiated and then its slots are filled with concrete values as a programmer builds an understanding of the code [SEB82]. Plans may help programmers fill in the “gaps” when trying to understand code. Finally, it was proposed that programs follow basic rules of discourse and that any violation to “accepted conventions of programming” should as a result hamper an expert’s ability to use programming plans [SE84].

Evidence that expert programmers have different mental representations from novices has been described in several studies, however not all evidence is consistent with the theory of programming plans. In a series of studies, participants were asked to understand a piece of code and later recall text of the program. Experts recall programs better than novices when the order of presentation is correct [Shn76], but performance difference disappears when programs are presented in random order. Further, when examining the details of what is recalled [SM79], researchers found that experts could recall semantic information about source code, but incorrectly recalled the exact details. Novices did the opposite: They could more accurately replicate the source code syntax, but often mistook the meaning of the source code. In another study, when categorizing related code snippets, experts and novices differed in their organization (procedural similarity vs. syntax similarity) [Ade81]. Finally, Soloway and Ehrlich [SE84] evaluated the theory of rules of discourse by varying the style of the snippets, such that there were versions that followed typical coding conventions (plan-like) and versions that explicitly violated such conventions (unplan-like). For example, they changed the variable naming, such that in the violated version, the naming did not convey the purpose of the variable, but rather the opposite (`max` was renamed to `min`). The results showed that novice programmers were not affected by the violated coding conventions. However, experts were significantly slower with these version and made significantly more errors—specifically, experts became as slow and as incorrect as novices.

However, another series of studies cast doubt on the nature of programming plans. Gilmore and Green failed to replicate Soloway and Ehrlich’s previous results [SE84] when using programming

plans from Pascal programs with Basic programmers [GG88]. They suggested that programming plans may not generalize across different languages, and that plans cannot represent the underlying deep structure of programs. Bellamy and Gilmore [BG90] examined the protocols generated from experts in different languages as they created programs. Using two different models of programming plans, they found neither model was well supported by protocols; further, different programming language experts generated different types of representations. Finally, Pennington [Pen87] theorized that if programmers form plan-based mental representations, then they should recognize lines faster when preceded by lines from the same plan structure. Unfortunately, in the study, stronger priming effects were observed from syntax structure vs. plan structure. Subjects also made fewer errors on control-flow questions, compared with data-flow and functional questions. Pennington concluded that:

“While plan knowledge may well be implicated in some phases of understanding and answering questions about programs, the relations embodied in the proposed plans do not appear to form the organizing principles for memory structures.”

In summary, expertise is not as simple as we might sometimes think. Although high-level, efficient representations of programming knowledge develop with experience, it seems that this knowledge is not the sole determinant of programming success. Besides chunking of knowledge structures, experts seem to acquire a collection of strategies for performing programming tasks, and these may determine success more than does the programmer’s available knowledge. Programming may be rather like riding a bike, or some other motor skill, without practice it cannot be mastered.

Ikutani et al. conducted an fMRI study to contrast different levels of programmer expertise. Unlike previous studies, they used a program *categorization* task. They trained a classifier to distinguish between different programming categories, which achieved a higher accuracy on expert programmers’ brain activation patterns than for novices, confirming that expertise leads to a fine-tuning of programmers’ brains [Iku+21]. While the classifier could detect expertise differences based on the observed brain activation, they could not be causally connected to the underlying differences in cognition (e.g., mental representations).

RQ 4.7

Literature from cognitive psychology has identified deliberate practice and different problem representations as key differentiator between different expertise levels. Neuroimaging studies have confirmed that these different cognitive processes show measurable difference in brain activation patterns. Program-comprehension studies have provided several possible explanations for similar effects on programmers, but are rather inconclusive.

4.3.2 Data Exploration

After our literature review confirmed that we can expect a measurable difference in cognitive processes between programmers of different experience levels, we re-analyze data from Siegmund

et al.'s first fMRI study on bottom-up comprehension. Its simple experiment design offers the most robust data from the available studies. While the experiment was not designed for this research question, it should enable us to generate hypotheses for our dedicated follow-up study. Thus, we pose the following research question:

RQ 4.8 When exploring an existing dataset, does programming experience correlate with brain activation strength during bottom-up program comprehension?

4.3.2.1 Method

Siegmund et al. found five activated brain areas (i.e., BAs 6, 21, 40, 44, 47) during program comprehension. For RQ 4.8, we computed the Kendall correlation between the brain activation strength in the five activated brain areas during program comprehension and a participant's experience and Java knowledge, which was collected based on a validated questionnaire [Sie+14b]. We use Kendall's τ (rather than Pearson's correlation coefficient) because of its robustness with repeated values and against outliers [CC83].

4.3.2.2 Results

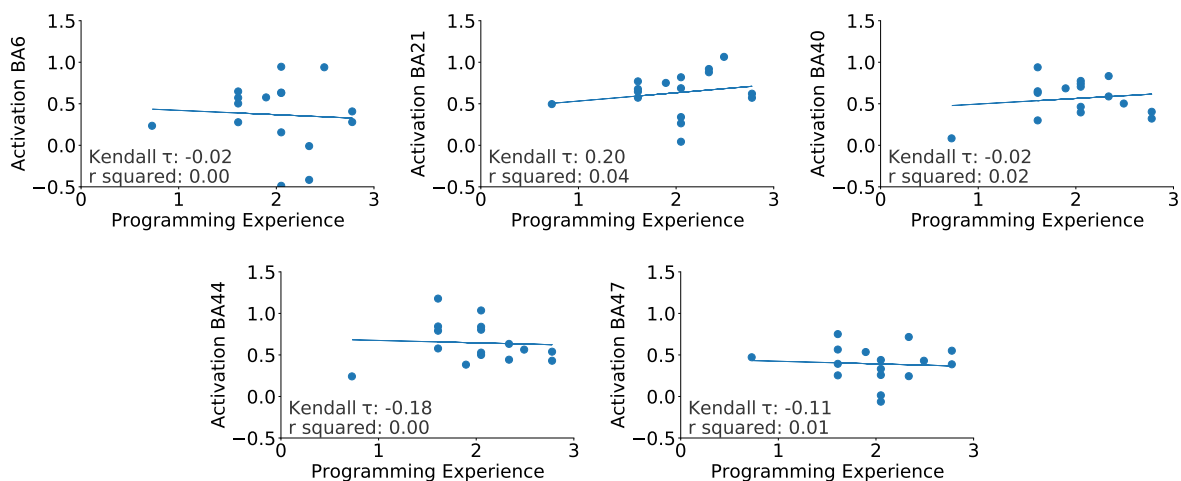


Figure 4.10: Scatterplot of programming experience and brain activation strength of BA 6, BA 21, BA 40, BA 44, and BA 47. Each dot represents one participant. The strength of brain activation for each cluster is the average beta value across all snippets.

We visualize the correlations between programming experience and Java knowledge with the observed brain activation strength in Figures 4.10 and 4.11).

The distribution of programming experience scores [Sie+14b] is clustered around a low score value of 2.0. This is due to the target of a homogeneous participant group of computer science

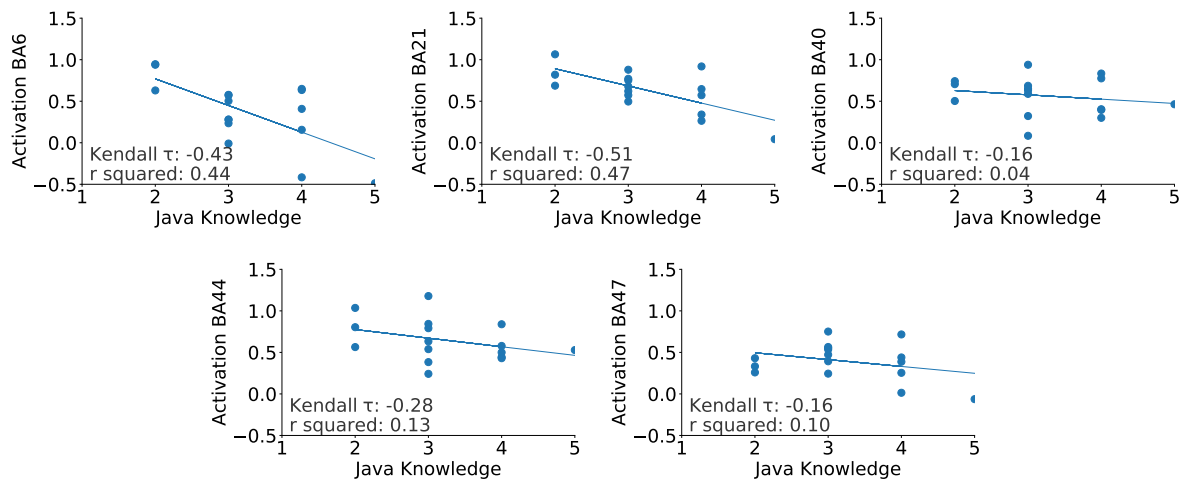


Figure 4.11: Scatterplot of Java knowledge and brain activation strength of BA 6, BA 21, BA 40, BA 44, and BA 47. Each dot represents one participant. The strength of brain activation for each cluster is the average beta value across all snippets.

students, which we classify between novice and intermediate programmers according to Dreyfus' taxonomy of skill acquisition [DD86; Mea+06]. Nevertheless, it is noteworthy that the correlations are different between the five brain areas. BA 21 shows a weak *positive* correlation (0.211). BA 6, BA 40, BA 44, and BA47 show a weak *negative* correlation (-0.089 , -0.027 , -0.202 , and -0.148 , respectively).

Self-estimated Java knowledge provides a more varied distribution. That means while the participants are overall relatively inexperienced, their individual Java knowledge is diverse. The correlation between the Java knowledge and the brain activation strength is *negative* for all five activated brain areas. Hence, participants with more Java experience tend to have a lower activation strength. The data indicate that programmers familiar with Java require less cognitive effort to understand Java source code. In particular, BA 6 and BA 21 show a strong and significant negative correlation (-0.514 , and -0.601 , respectively). The activation strength of BA 40, BA 44, and BA 47 is also negatively correlated with the Java knowledge (-0.219 , -0.379 , and -0.21 , respectively).

4.3.2.3 Discussion

The expected, but missing correlation between programming experience (based on the experience score) and brain activation strength indicates that a higher general programming experience does not seem to lead to a reduced cognitive effort. Earlier in this chapter, we showed that top-down comprehension leads to a lower activation strength (*neural efficiency*) than bottom-up comprehension. However, our analysis here indicates that experienced programmers do not automatically show higher neural efficiency for a comprehension task in any programming language. Only experience in the specific programming language leads to a lower cognitive effort,

as indicated by the negative correlation between knowledge of the Java programming language and brain activation. In other words, programming skills might not be efficiently transferred [PS92] to any domain, so an expert programmer might fall back to the neural efficiency of a novice when working with an unfamiliar language or domain. Floyd et al. found a similar result in their fMRI study to analyze the difference in brain activation across programming-experience levels. Their results show that program comprehension becomes increasingly similar to prose reading with higher programming experience [FSW17].

The strength of the correlations, especially for BAs 6 and 21, is surprising, as the experimental design was not targeting this research question. This indicates that it is a promising direction to further look into the role that the familiarity of a programming language plays for neural efficiency. The reduced activation strength in BA 21 indicates that being familiar with Java allowed our participants to be more efficient in analyzing the words and symbols of the source code. Consequently, the number of values the participants had to keep in their working memory was reduced as well, which would explain the lower activation in BA 6.

These results are based on a small and rather homogeneous sample. Nevertheless, based on these results, we can formulate a set of hypotheses for our dedicated fMRI study of programming experience:

- (H1) Specific programming knowledge has a strong effect on neural efficiency of program comprehension.
- (H2) General programming experience has, at most, a small effect on neural efficiency of program comprehension.
- (H3) Higher-level semantic (represented in BAs 40, 44, and 47) is only moderately affected by familiarity with the specific programming languages.

The dedicated follow-up study that we outline in the future-work section will build on the literature review and the exploratory results to fully understand the relationship between brain activation strength, cognitive effort, and programming experience.

4.3.3 Summary

In this section, we outlined results from a literature review and a data exploration for a planned study on programmers with different levels of experience. From cognitive psychology, we transferred possible explanations how experienced programmers employ different cognitive processes during program comprehension. With a future fMRI study, we will be able to gather evidence to better understand the minds of experienced programmers.

These insights will enable us to improve training and education for programmers. For example, should (student) programmers deliberately practice their skills through daily exercises, such as code kata [TH19] or performance checks [HJP16]? Alternatively, if specific knowledge is more important, should educators focus on teaching domain knowledge as much as on general

programming concepts? With our planned neuro-cognitive perspective of programmer expertise, we can contribute answers to such fundamental questions.

In the future, we may be objectively measure programming experience with fMRI, which has substantial implications for the hiring process of programmers. The current style of technical interviews for programmers are heavily debated for the objective evaluation accuracy [Beh+20]. Moreover, the process could be severely biased [BPB19]. A future in which applicants can demonstrate their skill level with a single fMRI would remove barriers from the hiring process and allow organizations to draw better candidates from a wider pool. However, there are concerns regarding privacy that need to be addressed (e.g., how to proceed with applicants that are medically or personally unable to safely participate in such fMRI test?).

4.4 Chapter Summary and Future Work

In this chapter, we presented several studies that reveal a neuro-cognitive perspective of programming. With an fMRI study, we could validate a decade-old theory on top-down comprehension and could objectively show that it requires less cognitive effort. However, we were also able to show that – from a cognitive perspective – top-down comprehension is similar to bottom-up comprehension but exhibits higher neural efficiency. In an eye-tracking study, we identified several factors that influence the reading order of programmers. This included an already suspected effect of programmer expertise, but we uncovered that the code structure plays a dominant role as well. Finally, we outlined another fMRI study that investigates the neuronal basis of programmer expertise in further detail. While our studies shed light on several key factors of program comprehension, there are many further research questions to be answered. With our framework presented in Chapter 3 and the conducted studies in this chapter, we provide a template for future studies investigating further important topics of program comprehension. We delve into three concrete topics next.

4.4.1 fMRI Study on Programming Experience Levels

In this planned fMRI study, we go beyond the influence of comprehension strategy (i.e., bottom-up vs. top-down comprehension) and activated brain areas: We seek to understand how a programmer's experience, measured with a suitable questionnaire, influences the *strength* of brain activation and deactivation. Based on the literature review and the data exploration (cf. Section 4.3), we expect that novices need to concentrate more than experienced programmers showing stronger brain activation. In particular, we expect that general programming experience will be less influential than specific programming knowledge. We also expect that novices display a stronger deactivation in brain areas of the default mode network as they must concentrate more to solve the tasks.

One advantage of identifying such measurable differences is that we could estimate the effect of a programmer's experience as a confounding variable. Furthermore, objectively evaluating a programmer's experience based on their brain activation and structure may offer future applications beyond research (e.g., teaching).

Challenge of Measuring Programming Experience For the success of this fMRI study, it is critical to select a representative sample for novice and experienced programmers. This way, we can clearly distinguish cognitive differences between the groups. However, it is a challenge to establish a sound measure of programming experience. Many studies in software engineering used simplistic measures, for example, Soloway and Ehrlich categorized undergraduate students as novices and graduate students as experts [SE84]. Siegmund et al. developed and validated a questionnaire to measure programmers experience [Sie+14b]. While this questionnaire was useful for our previous studies, it is limited here, since Siegmund et al. only investigated undergraduate students (and graduate students as validation). This raises doubts of the robustness for professional programmers with many years of experience.

We therefore extended Siegmund et al.'s questionnaire to capture elements we drew from the literature review, that is elements of deliberate practice, specific domain knowledge, and work distribution. Siegmund et al.'s basic questionnaire and our extended version are in the Appendix (Section 7.2 and Section 7.3, respectively).

Challenge of Materials and Task The previous fMRI studies presented in this dissertation used source code with comparatively lower complexity due to the restrictions of the fMRI scanner. For this study, we cannot simply re-use the same snippets. On the one hand, code snippets that are too simple may be insufficient in triggering enough cognitive demands to distinguish novices from experienced programmers. On the other hand, code snippets that are too complex may be too difficult to solve for novice programmers in the given time frame of an fMRI experiment (i.e., one minute per task). We therefore conducted an online pilot study to test various snippet candidates.

All snippets contain semantic cues, such as beacons, to allow top-down comprehension, because we expect experienced programmers to be employing top-down comprehension more than novices.

4.4.2 Experiment Execution

The experiment will be conducted in line with our previous experiments. We use a similar fMRI protocol (cf. Section 7.1). Similarly to the study on code complexity described in Section 5.1, we will use a semi-structured interview after the fMRI session to gather qualitative insights in the participants' problem-solving strategies.

4.4.3 Neural Representations of Programming Constructs

In our fMRI study on top-down comprehension, we considered two code factors: beacons and layout. But, there are other code aspects that affect program comprehension. Specifically, the programming constructs, such as if-then-else, iteration, and recursion may substantially change how programmers think through source code. In a future study, we aim to uncover the neural representations of if-then-else, iteration and recursive constructs. This way, we can understand the influence of such programming constructs for program comprehension. Novices often struggle with the rather unfamiliar concept of recursion [Gin05]. Recursive methods call themselves (unlike iterative methods), making it difficult to grasp the single components of such methods. Understanding the neural correlates of if-then-else statements, iteration, and recursion provides insights into the cognitive processes that are part of understanding and learning such code constructs.

In an exploratory study with remote eye tracking, we translated our previously used snippets into iterative and recursive versions and contrasted behavior and visual effort [Aqe+21]. 117 undergraduate students participated in the study and, on a high level, showed no significant differences regarding their behavior or visual effort. It might be that the participants were too inexperienced to develop a preference yet. For example, in our study on code complexity metrics, which we present in Section 5.1, we also used recursive and iterative snippets. Interestingly, these more experienced participants held strong opinions on their preference for iterative or recursive code.

In the future, we shall conduct an fMRI study that investigates this topic further along the following guiding research question:

RQ What are the neural representations of if-then-else statements, iteration, and recursion?

4.4.4 Neural Representations of Data Types

Similar to programming constructs, source code can manipulate different types of data such as words and numbers. In this study, we will look deeper into the neuronal representation of types of data, which leads us to the following research question:

RQ Do words and numbers, when manipulated in source code, have different neuronal representations?

In previous studies, our code snippets manipulated words (e.g., reversing a word) and numbers (e.g., compute factorial), which may have different neuronal representations (i.e., visual word form area [Sok+17] and number form area [ND09; Han+15]). Can we find this difference in representation also during program comprehension? Understanding whether the neuronal representation of

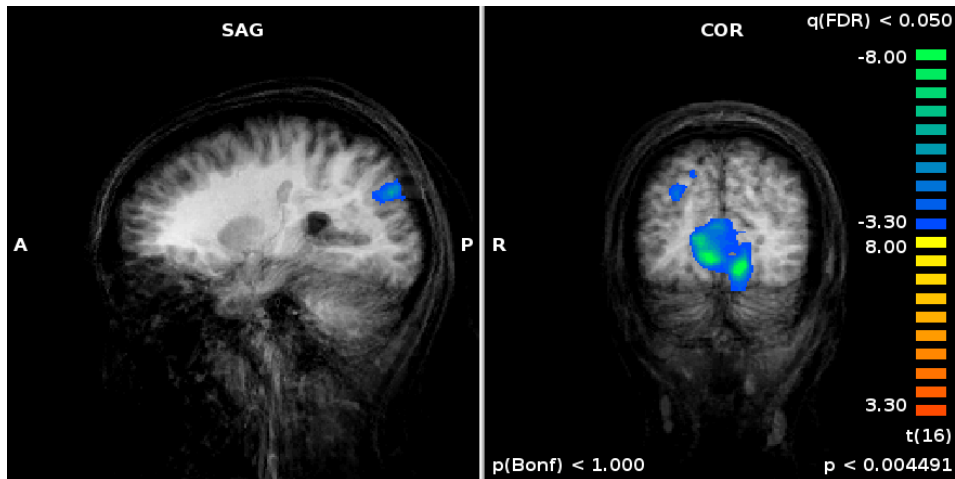


Figure 4.12: fMRI study on simultaneous fMRI and eye tracking (cf. Section 3.2): 2 clusters with stronger activation for snippets containing words, 12066 voxels, TAL -1, -76, 3 (BA18); 1285 voxels, TAL 22, -76, 32 (BA19)

different data types during program comprehension is the same as during non-programming activities helps us to identify all relevant cognitive processes for program comprehension. If we cannot find a difference in activated areas between word or number processing, this would indicate that they are handled differently during programming compared to when we are not programming. This could help us to explain the steep learning curve, as novices cannot rely on their existing cognitive processes to manipulate words or numbers.

In a preliminary data exploration, we re-analyzed our previous fMRI studies with the same procedures. It is important to note that the experiments were not designed for this research question, but by design had a balanced distribution of snippets containing either numbers or words. We directly contrasted the brain activation between snippets containing numbers against snippets containing words. We visualize the results in Figures 4.12 to 4.14. Overall all three fMRI studies, there is no consistent result. It appears that the brain activation is largely the same, except some difference in the visual cortex (BA 18 and BA 19). This is likely due to the (on average) larger size of snippets containing words, which require more visual effort. This result indicates that in the context of programming typical brain structures for words and numbers does not play a role, but a dedicated study is necessary to provide a definite answer.

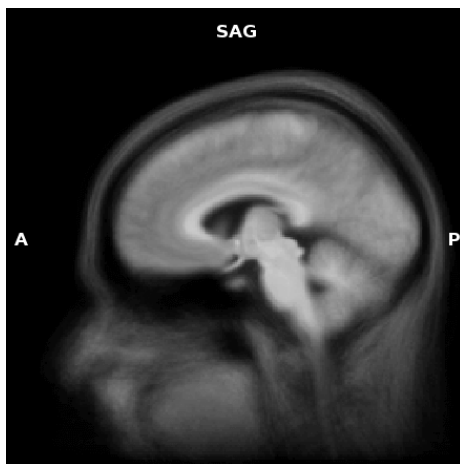


Figure 4.13: fMRI study on top-down comprehension (cf. Section 4.1): No statistically significant differences

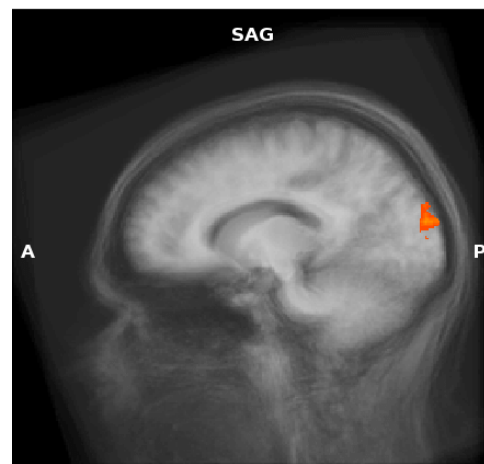


Figure 4.14: fMRI study on code complexity metrics (cf. Section 5.1): 1 cluster with stronger activation for snippets containing words, 1408 voxels, TAL -10, -91, 13 (BA18)

5 Applications of fMRI Research in Software Engineering

This chapter shares material with several prior publications [Pei+20; Pei+21; Sie+21; Neu+21].

After introducing our experiment framework in Chapter 3 and presenting conducted studies with the framework in Chapter 4, we apply our framework to further practical issues in this chapter. First, in the following Section 5.1, we evaluate commonly used code complexity metrics with an fMRI study and investigate their unclear link to programmers' cognition. Second, in Section 5.2, we re-use data from three fMRI experiments to prove how different levels of data aggregation significantly influences results. Thus, program-comprehension researchers need to carefully consider this effect when balancing costs and reliability of experiments. Third, we introduce an alternative approach to analyze fMRI data in Section 5.3. We demonstrate how using anatomical information of participants can increase signal strength for an analysis of fMRI data.

5.1 Code Complexity Metrics and Program Comprehension

In the past 40 years, the software engineering community has been using various complexity metrics to predict how programmers understand code [Cur+79; Zus93; Sne95], implement quality gates in continuous integration [FP14; GMGVEB16], and predict the likelihood of defects [OSH76; MPS08; Hud+17]. For example, from 2010 to 2015, a total of 226 studies proposed or analyzed nearly 300 code metrics alone, with code complexity being one of the most frequent categories of study [Var+17]. Many of these metrics are widely used in software analysis tools. For example, SONARQUBE, a popular static analysis tool used in continuous integration, supports a multitude of metrics, such as cyclomatic complexity, across dozens of programming languages. For this particular metric, the tool will report a "Methods should not be too complex" violation for any method that exceeds the default threshold of 10.

While code metrics help to describe properties of code, they are notoriously limited in capturing human cognition and behavior: Already 15 years ago, Kaner and Bond warned that too simplistic software metrics can do more harm than good, because it is doubtful whether they actually measure what we think they measure [KB04]. Several studies underline this point. For example,

Scalabrino et al. found in an empirical study, at most, minuscule correlations between complexity metrics and the observed code understanding [Sca+17]. Ajami et al. found that complexity metrics fail to consider how humans process code, for example, that flat structures are easier to comprehend than nested ones [AWF17]. In the same vein, Jbara and Feitelson provide evidence that complexity metrics miss the increased ease of comprehension of repeated code constructs [JF17]. Despite these warnings, it is tempting to use complexity metrics to predict how complex programmers perceive code or how much cognitive load it would require understanding a piece of code. With our fMRI experiment framework, we are able to objectively test whether (and which) source code complexity metrics are suitable as a proxy for program comprehension by examining cognitive load, operationalized by (de)activation patterns in the brain. We start by examining data of the original fMRI study.

5.1.1 Data Exploration

5.1.1.1 Method

We selected code complexity metrics among four major classes: *code size*, *vocabulary size*, *control-flow complexity*, and *data-flow complexity*. For each, we picked a commonly used representative metric. The underlying idea is that, the more code lines (code size, LOC as representative) or vocabulary (vocabulary size, Halstead) to understand, or the more possible execution paths (control flow, McCabe) or data dependencies to keep track of (data flow, DepDegree), the higher the programmers' cognitive load is.

In our previous fMRI studies, we focused only on brain activation, which revealed a network of several areas. Another approach is to measure the deactivation in the *default mode network*, which can be used as an indicator of cognitive load [McK+03]. The default mode network comprises several brain areas (e.g., cingulate cortex, prefrontal midline regions) and is related to self-referential processing [Rai+01; GIM13]. When left to think about nothing specific (e.g., in the rest conditions), we often think about self-related aspects, for example, our plans for after the fMRI scanner session or previous experiences. This is reflected in an increased blood flow within the default mode network, that is, the default mode network shows high activation during rest states. When we concentrate on tasks, the default mode network deactivates, so that this self-referential processing does not interfere with the task. Hence, with the level of deactivation of the default mode network, we can measure the cognitive load of participants: The stronger the deactivation, the higher the cognitive load.

For our exploration, it seems plausible to expect that programmers exhibit higher cognitive load with increasing complexity of source code.

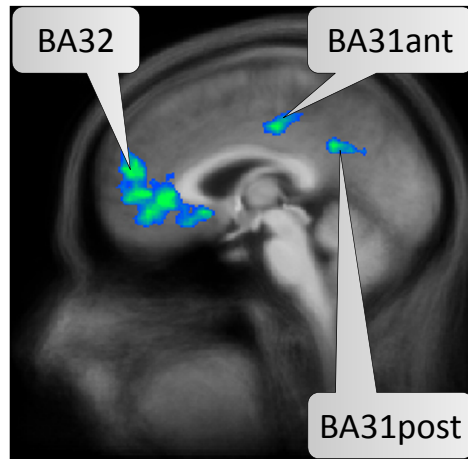


Figure 5.1: Visualization of significant deactivation during program comprehension in the default mode network.

5.1.1.2 Results

In Figure 5.1, we show the significantly deactivated brain areas, that is, areas with less activation compared to the rest condition (color-coded with blue). In the reanalysis, we found deactivation in brain areas that are key components of the default mode network (DMN).

Next, we looked at the correlation between the strength of deactivation and code complexity for the four metrics: LOC, McCabe, Halstead, DepDegree. Figure 5.2 visualizes the correlation of the code complexity metrics with the beta value for each deactivated areas averaged across participants. By using the mean, we can reduce the influence of peculiarities of individual participants. Each dot in the plot indicates one comprehension task. We found that all deactivated areas correlate negatively with Halstead and DepDegree; that is, the higher the value for these complexity measures, the lower the level of the beta value for the deactivated areas (indicating higher cognitive load). One correlation, that is, the correlation of BA 32 with DepDegree (-0.591 , bottom right) is strong. Given our small sample size, we can actually expect that, although some of the correlations have a high value, these are not necessarily true relationships. Interestingly, the correlation with McCabe is positive, indicating that with a higher control-flow complexity, the deactivation of the found areas is less pronounced, or in other words, requires lower cognitive load. The weakest correlations are with LOC, indicating that there is no relationship between lines of code and cognitive load in this sample. However, it is important to note the code snippets were designed to be similar in length and complexity and do not have a high variation in the values of the metrics. Thus, while we demonstrated how we can analyze the data, the code snippets need to show a higher variation in length and complexity to reliably evaluate whether a relationship between cognitive load and complexity exists.

Despite this, the results motivate us to design a dedicated experiment to evaluate the relationship between complexity metrics and cognitive load: The high correlation values with DepDegree and Halstead indicate that data-flow complexity and vocabulary size as operationalized by these

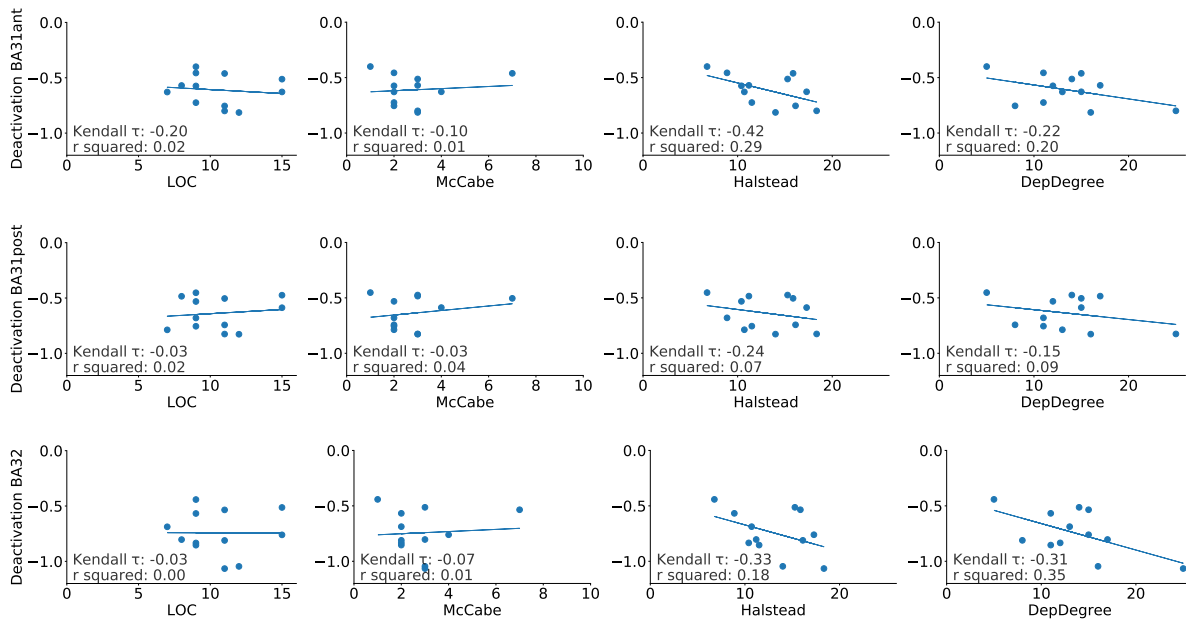


Figure 5.2: Scatterplot of code complexity metrics and strength of deactivation of BA 31ant, BA 31post, and BA 32. Each dot represents one comprehension task. The strength of deactivation is the average beta value across all participants.

measures modulate cognitive load of participants, which is in line with McKiernan’s result of a stronger deactivation during more difficult tasks [McK+03]. This also fits well to bottom-up comprehension, because there are no beacons to act as cues that could relieve cognitive load during comprehension. Instead, variable names remain rather abstract, and data-flow cannot easily be associated with certain variables.

5.1.2 fMRI Study

After our data exploration confirmed the feasibility of a dedicated fMRI study, we can go one step further and start with the *overarching question* that drives this work:

RQ Can variations in (classes of) code complexity metrics explain differences in programmer cognition during program comprehension?

To address this question, we first draw on insights from neuroscience studies on cognitive load and from linguistic studies on sentence complexity. An extensive body of previous research has examined how syntax complexity in natural language can influence brain activation and processing difficulty. For example, complex sentences such as “The girl that the boy is tickling is happy” can be difficult to process for patients with Broca’s aphasia [CZ76] (a cognitive deficiency due to an acute brain lesion); these patients cannot reliably distinguish between the girl or the boy being happy. In healthy patients, such sentences will cause distinct activation of specialized

language processing regions in the brain not seen with simpler grammatical structures. To perform these studies, neuro-linguists typically conduct a parameterized analysis of sentences, where they intentionally construct sentences that vary along several metrics of interest (such as left-branching complexity [Udd+19], movement distance [SG07], or filler-gap dependencies [FSF01]) observing differences in behavioral measures and brain activation. Such studies provide deep insights into *why*—not only *whether*—comprehension of certain language constructs can be difficult, gaining insights into the inner processing of language cognition.

Inspired by parameterized analysis of sentence complexity, we conducted a study, in which 19 participants comprehended 16 code snippets. To this end, we explore the relation of code complexity metrics to behavioral and cognitive correlates of program comprehension. We use the same four representatives for different classes of code complexity (i.e., LOC, Halstead, McCabe, DepDegree) as a baseline before exploring 37 further complexity metrics. We also investigate the reliability of subjective perception of code complexity to include the programmer’s perspective.

Based on their aim and definition as well as prior neuro-linguistic studies, we expect different outcomes for different kinds of complexity metrics:

- (a) a higher number of symbols (as measured by vocabulary-size metrics) induces higher cognitive processing demands, as seen when increasing the number of words in a sentence [Sch+20]. This is also supported by early studies on the relationship of Halstead’s complexity and comprehensibility [Cur+79];
- (b) an increased number of control paths (as measured by control-flow metrics) increases activation of areas associated with rule-guided conditional reasoning, such as reading conditional propositions [Liu+12] and counterfactual [Kul+13] sentences: “If Mike pressed the brake pedal, then the car would have stopped”;
- (c) a higher number of data-flow dependencies (as measured by DepDegree) increases activation of Broca’s area (syntactic working memory) [FSF01], as seen with sentences where assignment of values must occur later in the sentence (i.e., filler-gap dependencies), “Which cowgirl did Mary expect to have injured herself due to negligence?”.

5.1.3 Experiment Design

Our fMRI study of neuronal and behavioral correlates of code complexity metrics builds on the data exploration that we presented in Section 5.1.1. We specifically implemented this study with our multi-modal experiment framework to unveil this relationship. We selected proper code snippets that vary across different metrics, and analyzed a total of 41 metrics regarding their predictive power.

We provide an online replication package²⁴ to share experiment design, tasks, and analysis protocols.

²⁴<https://github.com/brains-on-code/fMRI-complexity-metrics-icse2021>

5.1.3.1 Research Goals

In this study, we aim at answering the following research questions:

- RQ 5.1** Do different (classes of) code complexity metrics correlate with programmers' behavior during program comprehension?
- RQ 5.2** Do different (classes of) code complexity metrics correlate with programmers' cognitive load in terms of brain (de)activation during program comprehension?
- RQ 5.3** Do different (classes of) code complexity metrics correlate with programmers' subjective perception of code complexity?

5.1.3.2 Pilot Studies

To answer our research questions, we carefully designed our experiment. First, we compiled a set of 50 Java code snippets by obtaining snippets from previous studies on program comprehension [Sie+14a; Sie+17; Bus+15] and by augmenting this set by searching for code snippets with similar complexity in public code repositories. Second, from this pool of code snippets with a wide range of complexities, we selected the most suitable snippets for an fMRI study by running two pilot studies. We asked 7 pilot-study participants (2 professional programmers and 5 PhD students) to understand the snippets as fast and accurately as possible and to verbally share their thoughts afterwards. Unlike in the fMRI scanner, we did not set a time limit per snippet, because the actual comprehension time is an important factor to select appropriate snippets for an fMRI study. In the pilot studies, we also asked for a subjective evaluation of each snippet's complexity, but, unlike in the subsequent fMRI study, we did not ask them to rank all presented snippets. Third, we selected snippets for the fMRI study that were associated with a range of complexity metrics values balancing our final selection across the four classes of complexity metrics.

For illustration, we show one of the snippets in Listing 5.1, which computes the length of the last word in a string: The code exhibits large values for some of the metrics, not being trivial to solve (pilot participants took about 45 seconds), while still staying within the 60 seconds limit allowed in the fMRI study. Feedback from participants indicated that they needed a high level of cognitive effort to understand the snippet, but could still succeed. In particular, they noted that the snippet did not allow them to take a “mental break”. That is, it was difficult to analyze individual statements while keeping other statements in mind, and that they were unable to match the code to any known algorithm.

Snippet	Complexity Metrics				Experiment Results		
	Code Size (LOC) ¹	Vocabulary (Halstead) ¹	Control Flow (McCabe) ¹	Data Flow (DepDegree) ²	Correctness (in %)	Time (in sec.)	Subjective Complexity Low → Medium → High
Average of array	17	12.64	3	17	47%	49.0	
Contains substring	26	25.50	7	29	32%	50.2	
Count vowels in string	19	13.00	5	22	89%	30.4	
Greatest common divisor	24	26.63	5	33	47%	50.0	
h index	21	16.25	4	20	53%	46.0	
Length of last word	22	18.58	8	17	58%	43.3	
Palindrome check	17	16.72	4	17	79%	38.4	
Square root of array	23	39.83	5	27	68%	40.1	
Binary to decimal	17	16.75	4	10	68%	42.3	
Cross sum	12	14.66	3	4	84%	27.6	
Factorial	12	16.50	3	4	95%	22.3	
Fibonacci variation	12	10.88	3	4	84%	35.6	
Power	17	15.38	4	8	79%	33.3	
Contains yes or no	22	6.13	6	7	95%	23.4	
Hurricane check	17	9.00	7	13	100%	21.5	
Sort four elements	17	30.30	7	61	79%	41.4	

¹<https://github.com/BasLeijdekkers/MetricsReloaded/> ²<https://www.sosy-lab.org/~dbeyer/DepDigger/>

Table 5.1: Code snippets used in the study with four selected complexity metric scores and experimental results: behavioral data (correctness, time) and subjective complexity. A higher intensity of a cell's background color indicates a higher complexity. The histogram plots show the distribution of subjective complexity; skewedness to the right represents higher subjective complexity.

```
1 public static void main(){
2     String text = "The quick brown fox jumps";
3     System.out.print(compute(text));
4 }
5
6 static int compute(String text){
7     int result = 0;
8     boolean flag = false;
9     for (int i = text.length() - 1; i >= 0; i--){
10        char c = text.charAt(i);
11        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')){
12            flag = true;
13            result++;
14        } else {
15            if (flag)
16                break;
17        }
18    }
19    return result;
20 }
21
```

Listing 5.1: Complex code snippet that computes the length of the last word in a string. The output for this snippet is “5”.

Finally, consistent with previous studies [Sie+14a; Sie+17], we excluded all context information from snippets to enforce bottom-up comprehension. This way, we reduce the influence of previous experience on cognitive load, because employing a more efficient top-down approach is impeded, and because expertise can moderate the relationship between complexity and performance [Cur+79]. In Table 5.1, we provide information on the code snippets and code complexity metrics that we selected for the study in the fMRI scanner (see the supplementary Web site for all code snippets and all 41 metrics).

In Table 5.3, we show the correlations among the four metrics for all code snippets. Although we designed the snippets to vary in complexity, we were restricted by requirements of the fMRI scanner (especially the limited screen size to show a maximum of 30 lines of code and that each snippet is comprehensible within 60 seconds). Thus, a certain correlation among the metrics is unavoidable.

5.1.3.3 Experiment Design & Execution

Participants Participants were 19 (including one whose fMRI data had to be excluded from analysis due to excessive head movements) late undergraduate or graduate students at the University Magdeburg. We determined their programming experience based on a validated questionnaire [Sie+14b]. The participants are intermediate programmers according to the Dreyfus’ taxonomy of skill acquisition [DD86; Mea+06]. All participants had normal or corrected-to-normal vision and were right-handed. We show further demographic data in Table 5.2.

Characteristic		N (in %)
Participants		19
Gender	Male	17 (89%)
	Female	2 (11%)
Pursued academic degree	Bachelor	9 (41%)
	Master	13 (59%)
Age in years \pm SD		26.47 \pm 2.68
Programming experience	Years of experience \pm SD	6.79 \pm 4.96
	Experience score [Sie+14b] \pm SD	2.83 \pm 0.53
	Java experience [Sie+14b] \pm SD	3.47 \pm 1.09

Table 5.2: Participant demographics for our fMRI study on code complexity metrics.

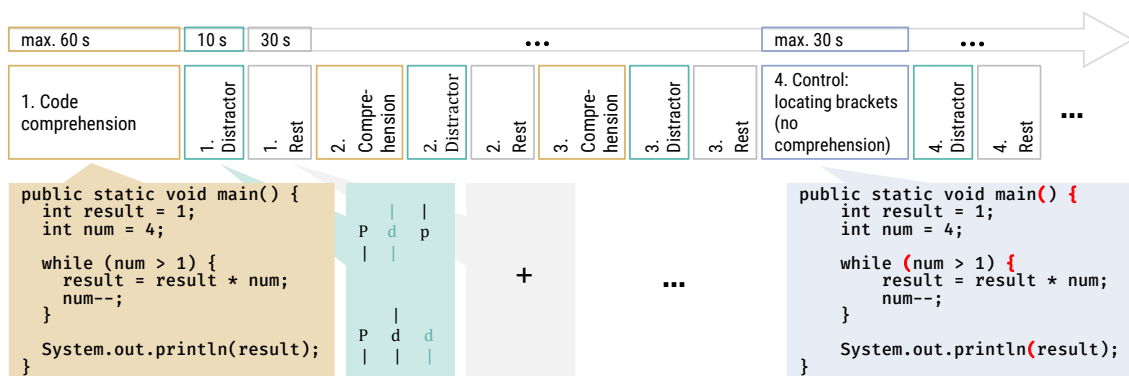


Figure 5.3: Illustration of one (out of five) experiment trials for our study on code complexity metrics.

Design and Tasks The study design builds on our developed framework. We presented three tasks in the fMRI scanner. First, participants should comprehend a code snippet. To this end, they should determine what would be printed on screen if the snippet was executed. The experiment moved on when they responded, but there was an upper limit of 60 seconds for a comprehension task. This was followed by a 10 second distractor task (cf. Section 3.5.4). Finally, a 30 second rest condition followed. After three comprehension snippets, a control condition followed, in which participants saw another snippet and should click whenever they spotted an opening bracket. This was repeated until the participant completed all 16 comprehension snippets. In Figure 5.3, we illustrate one out of five experiment trials.

Data Collection We describe the fMRI setting and imaging sequence in the Appendix (Section 7.1). During the fMRI session, we collected behavioral data with an fMRI-compatible two-button response device. Participants indicated whether they could compute the result of a snippet. We showed a warning after 58 seconds that the time was almost up.

After the fMRI session, we conducted a semi-structured interview with each participant, which

was based on the results of the pilot studies. In addition to open-ended questions investigating the participants' individual perception of snippet complexity, we showed them the code snippets again and asked them to order the snippets regarding complexity. Such categorization tasks can produce insights into how participants approach a task (e.g., physics novices and experts categorize problems based on different aspects [Sny00]). We allowed participants to self-choose the number of categories, because we noticed in the pilot runs that participants had difficulties when a fixed number of piles did not match their expectation of different complexity levels. As in the pilot studies, this helps us to understand what made a snippet difficult or easy to comprehend. If participants made only two piles ("simple" and "complex"), we encouraged them to distinguish it further, often leading to three or four piles (see Section 5.1.4.3).

fMRI Data Analysis After the standard fMRI preprocessing described in Section 7.1, we conducted a random-effects general linear model (GLM) analysis, defining one predictor per condition: program comprehension, distraction, control, and rest. To identify brain areas related to program comprehension, we filtered the data to all voxels that showed a positive deflection of the BOLD response during the comprehension condition. We computed the contrast between program comprehension and control condition ($p < 0.05$, false discovery rate (FDR) corrected [BH95], minimum cluster size: 27 mm^3). To identify deactivated brain areas, we obtained all voxels that show a negative BOLD deflection when contrasting comprehension and rest ($p < 0.001$, FDR corrected, minimum cluster size: 27 mm^3). For correlation analysis, we computed the mean amplitude of positive or negative BOLD deflection in percent for each cluster, task snippet, and participant, and correlated with the complexity metrics.

In addition to brain activation, we collected two further dependent variables: First, we observed participant behavior (response time, response correctness). This helps us to evaluate whether participants actually worked on understanding the snippets. We excluded snippets where participants (accidentally) responded too fast (response time less than 3 seconds), which happened in 3 out of 304 comprehension tasks. Second, we recorded subjective complexity in a post-interview: We converted the piles into equidistant values between 0 and 100. For example, if a participant created 4 piles, we assigned snippets of the first pile a complexity score of 0, the second pile a score of 33, the third 66, and the fourth 100. 3 piles translate to 0, 50, and 100, and 2 piles to 0 and 100.

To analyze the relationship between complexity metrics and cognitive processes, we use Kendall's τ to compare each metric's complexity value with the observed brain (de)activation across all snippets. We use Kendall's τ (rather than Pearson's correlation coefficient) because of its power with interval data and robustness against outliers [CC83].

5.1.4 Results

In this section, we present the results of our data analysis, including behavioral, fMRI, and subjective complexity data. We summarize the correlation results in Table 5.3. We separate results

Complexity Metrics	Complexity Metrics				Activation			Deactivation		Subjective Complexity	
	LOC	Halstead	McCabe	DepDegree	BA 6	BA 21	BA 39	Broca	BA 31		BA 32
Complexity Metrics	.32	.57	.25	.59							.16 (.04)
	.32	.57	.25	.59							.20 (.04)
	.59	.50	.49								-.07 (.01)
	.59	.50	.49								.16 (.01)
Activation	.26 (.05)	.38 (.20)	.04 (.00)	.32 (.11)							.20 (.09)
	.43 (.39)	.32 (.27)	.09 (.04)	.41 (.24)							.18 (.09)
	.17 (.10)	.40 (.18)	.07 (.01)	.36 (.21)							.17 (.10)
	.15 (.04)	.17 (.06)	-.04 (.00)	.22 (.09)							.22 (.06)
Deactivation	-.30 (.21)	-.30 (.11)	.05 (.00)	-.24 (.04)							-.44 (.44)
	-.39 (.24)	-.42 (.22)	.04 (.00)	-.29 (.04)							-.69 (.69)
Behavioral Data	-.46 (.29)	-.45 (.26)	-.09 (.02)	-.41 (.22)	-.63 (.53)	-.63 (.67)	-.58 (.47)	-.34 (.13)	.59 (.58)	.71 (.62)	-.77 (.71)
	.22 (.10)	.24 (.09)	.06 (.00)	.26 (.07)	.29 (.19)	.22 (.10)	.31 (.22)	.24 (.13)	-.22 (.11)	-.21 (.10)	.34 (.18)

Table 5.3: Kendall's τ and the explained variance (r^2 , in brackets) of the dependent variables. A darker cell shading indicates a stronger correlation: none ($\tau < 0.1$), small ($0.1 < \tau < 0.3$), medium ($0.3 < \tau < 0.5$), and strong ($0.5 < \tau$) [Coh69].

from discussion (see Section 5.1.5) to prevent mixing interpretation with data. To streamline the presentation, we concentrate here on the four representative metrics. In Section 5.1.5.1, we consider further metrics providing more evidence on the link between complexity metrics and cognitive processes.

5.1.4.1 Complexity Metrics and Behavioral data

On average, participants needed 32 seconds to solve a task, and solved 72 % of the tasks correctly (cf. Table 5.1). All participants were able to complete all tasks before the maximum experiment time was reached.

We present the observed correlations between behavioral data and complexity metrics in Table 5.3. They let us answer our first research question:

RQ 5.1 McCabe has no correlation to neither response time nor correctness. LOC, Halstead, and DepDegree all show a small correlation with response time and a medium correlation with correctness.

5.1.4.2 Complexity Metrics and fMRI Data

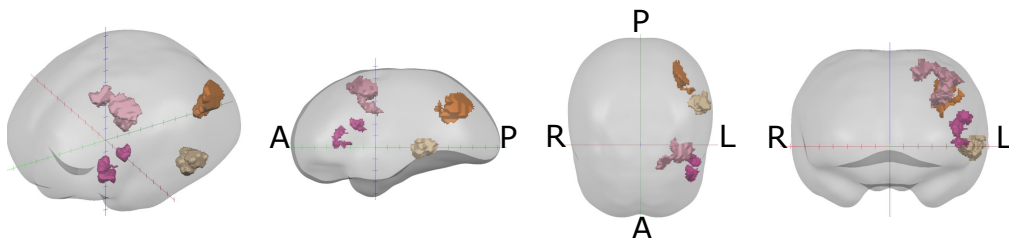


Figure 5.4: Visualization of the activated brain areas (all in the left hemisphere): **BA 6** (4315 voxel, TAL: -35, 10, 47), **BA 21** (3306 voxel, TAL: -57, -39, 1), **BA 39** (3527 voxel, TAL: -37, -65, 35), and **Broca's area** (1618 voxel, TAL: -49, -22, 14). A: anterior, P: posterior, L: left, R: right.

Brain Activation In Figure 5.4, we visualize the activated brain clusters. In our study, four brain areas are significantly activated during program comprehension: BA 6, BA 21, BA 39, and Broca's area (BAs 44 and 45). Notably, these activation clusters were found also in previous fMRI studies of program comprehension [Sie+14a; Sie+17; Cas+19; Iku+21].

The relationship of the complexity metrics and activation strength of the four Brodmann areas corroborates the behavioral data, but provides a stronger and more nuanced view (cf. Table 5.3). Again, McCabe shows no correlation. LOC has a medium correlation with BA 21, and a small

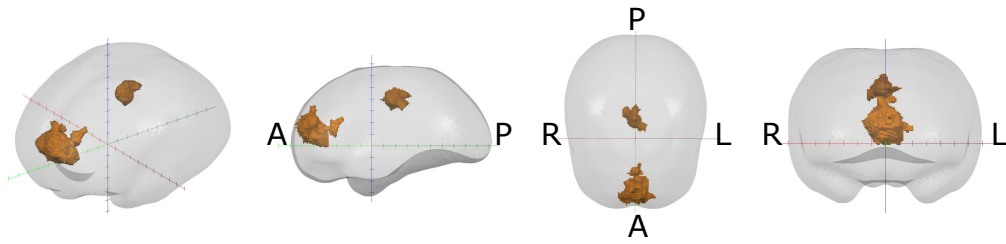


Figure 5.5: Visualization of the deactivated brain areas: **BA 32** (10'651 voxels, TAL: -1, 51, 14) and **BA 31** (2672 voxels, TAL: 3, -20, 40), which both are part of the default mode network. A: anterior, P: posterior, L: left, R: right.

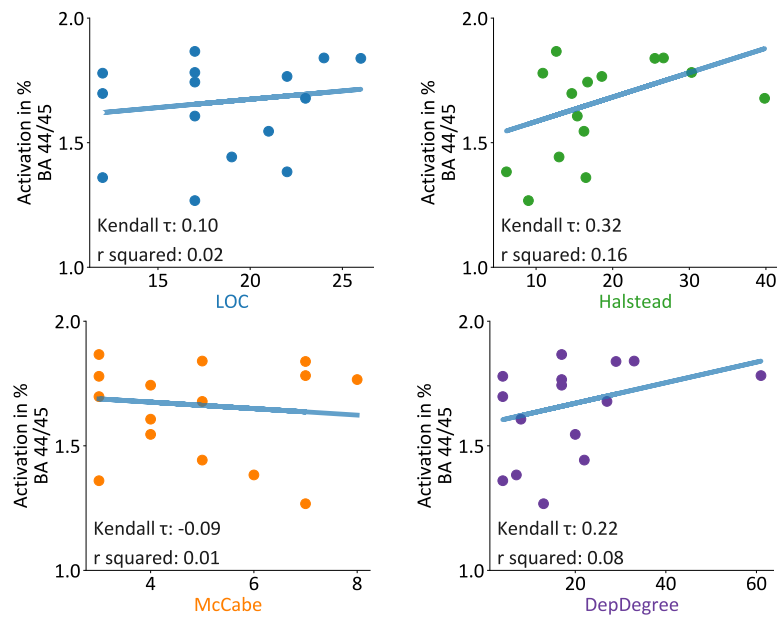


Figure 5.6: Relationship of the four complexity metrics with BA44/45 (Broca's area). Each dot represents the mean activation for a single snippet. McCabe shows no correlation. LOC, Halstead, and DepDegree show small positive correlations, meaning that higher complexity values increase load in BA44/45. However, each metric only explains part of the observed variance in activation.

correlation with BA 6, BA 39, and Broca. Halstead and DepDegree show a small correlation with Broca (cf. Figure 5.6) and consistent medium correlations across BA 6, BA 21, and BA 39.

Brain Deactivation Figure 5.5 visualizes the specific positions of the two clusters that we found in BA 31 and BA 32, which belong to the default mode network. Regarding the relationship with the metrics, McCabe again shows no correlation with brain deactivation. DepDegree shows a small correlation that explains almost none of the observed variance in brain deactivation. LOC and Halstead have medium correlations with both deactivated areas (cf. Table 5.3).

Behavioral Data and fMRI Data We analyzed the relation of the behavioral data with the (de)activated areas: We found strong correlations with the response correctness for BA 6, BA 21, BA 39, and a medium correlation with Broca. For response time, we found mostly small correlations, and one medium correlation (BA 39).

RQ 5.2 McCabe shows no correlation with the strength of brain activation or generic cognitive load. LOC, Halstead, and DepDegree show small to medium correlations with brain activation and cognitive load.

5.1.4.3 Subjective Complexity

In addition to objective measures of brain activation, we investigated the relationship between code complexity metrics and subjective complexity based on the participants' rating. Two participants created two piles (cf. Section 5.1.3.3), nine participants created three piles, and eight participants created four piles of complexity. Participants generally balanced the size of each pile. We transformed the piles into numerical values (42.42 ± 40.92). Regarding the relationship to the metrics, we observe only low correlations, with McCabe showing no correlation and the other metrics a small correlation only.

Subjective Complexity and Behavioral Data We observe a medium correlation between response time and the subjective complexity rating. With number of correctly solved tasks, the subjective rating shows the strongest correlation, in general.

Subjective Complexity and fMRI Data All (de)activated areas show mostly stronger correlations with the subjective complexity ratings than with the code complexity metrics. The deactivated areas of BA 31 and BA 32 show a medium and strong correlation, respectively. The four activated brain areas show only small correlations with subjective complexity.

RQ 5.3 While subjective complexity has, at best, small correlations with complexity metrics, it accurately depicts whether participants were able to solve a task. Subjective complexity also strongly correlates with our measure of cognitive load.

5.1.5 Discussion

We start the discussion by answering our overarching research question, followed by a detailed discussion of the relationship of code complexity metrics with cognitive processes. We conclude by formulating a set of hypotheses and outlining perspectives that arise from our study.

5.1.5.1 Overarching Research Question

RQ Can variations in (classes of) code complexity metrics explain differences in programmer cognition during program comprehension?

Yes, and no. Based on plausibility and prior neuro-linguistic studies, we hypothesized some definitive relationships between code complexity metrics and programmers' cognition. For example, we expected that code with more data flow shows a direct positive correlation with Broca's area due to increased memory load [FSF01]. While we indeed observe a positive correlation between DepDegree (as an indicator for data-flow complexity) and Broca's area, we also observe small to medium-strength correlations with all other cognitive processing measures (i.e., all activated and deactivated brain areas and their levels) as well as behavioral data (i.e., response time and correctness). Similarly, LOC and Halstead exhibit small to medium-strength correlations with all behavioral and cognitive processing measures. McCabe, however, consistently lacked any significant correlation with our observed measures.

All of the observed relationships between complexity metrics and cognitive processing measures are nuanced and contextual, into which we delve in more detail next.

Deactivated Areas: Cognitive Load When we must allocate attention to a task with perceived difficulty, areas of the brain that are associated with wandering and reflective thinking (default mode network) [McK+03] are deactivated. The level of deactivation is an indicator for cognitive load [McK+03]. Two classes of complexity metrics exhibited medium correlations with this shutdown: code size (LOC) and vocabulary (Halstead). In other words, the size of code, either as pure textual length or in terms of vocabulary size, drives cognitive load. We also observed a small correlation of data flow (DepDegree), but no correlation of control flow (McCabe) with deactivation. Perhaps, size-based metrics naturally interact with a programmer's sense of gestalt [Gol02], and thus induce stronger anticipation for higher cognitive load.

The results of our study on complexity metrics substantiates our possibly spurious findings from the previous exploration, but differ in a few key ways: We previously also observed a default mode network deactivation relationship with vocabulary size, but not code size. In contrast, now, with more varied snippets and complexity values, we found that code size exhibits also a medium correlation, in the same range as vocabulary size.

Activated Areas: Link to Cognitive Processes When faced with solving a complex task, we perform additional cognitive operations (e.g., extracting the meaning of identifiers) and recruit additional cognitive resources and processes to fulfill the extra demands of the task. The areas of the brain that are stronger activated in a complex task (as compared to a simple task) indicate increased demands for cognitive processes and resources hosted by these particular areas.

The LOC metric hints at an increased demand on a single area (as indicated by a medium-strength correlation), the middle temporal gyrus (BA 21). BA 21 is typically associated with semantic processing during language comprehension [Bin+09a; Dro+04] and program comprehension [Sie+14a]. Its role for program comprehension is interpreted as extracting the meaning of individual identifiers and symbols from code [Sie+14a; Sie+17; FSW17; Cas+19]. When processing complex sentences, an increased activation of BA 21 indicates higher grammatical processing load [KS02]. So, we conclude that simply increasing LOC increases the cognitive processing of identifiers and symbols, but otherwise does not necessarily pose a strong demand on other cognitive resources.

Halstead and DepDegree hint at increased demands (as indicated by medium-strength correlations) across three areas (BA 21, BA 6, and BA 39). The middle frontal gyrus (BA 6) is activated when attention and working memory is required [Neb+05]. Several fMRI studies on program comprehension found strong activation, albeit with slightly changing location in the brain [Sie+14a; Sie+17; Cas+19]. The angular gyrus (BA 39) is a part of the brain that is associated with complex cognitive processes; it acts as a hub for integrating incoming information. Like a reservoir, as processing areas are filled to capacity, other areas are recruited to share the load. Collectively, this result indicates that increasing the number of symbols and the number of data dependencies requires a broader network of processing than with increases in other factors. McCabe showed no relationship with an increased demand in any brain area.

Our results align well with the study by Schuster et al. [Sch+20], who found that an increasing number of words in a sentence leads to higher activation in BA 21. Furthermore, we found support for early results of Curtis et al. [Cur+79] that program size relates to whether programmers successfully comprehend a piece of code. However, we did not find a link to McCabe, as Curtis et al. did.

Metric	BRANCH	NP	NBD	D	LOOP	EXP	N	V	CRCT	MAINT	iv(G)	RLOC	n	CALL	RLBTY	E	IF_NEST	CogCompl	CONTR	STAT	ev(G)	v(G)
BA 6	.50	.40	.36	.51	.36	.45	.41	.41	.41	.40	.34	.37	.37	.27	.34	-0.06	.31	.30	.27	.29	.23	.21
BA 21	.56	.27	.54	.41	.48	.34	.36	.36	.35	.33	.38	.37	.35	.37	.27	-0.07	.27	.24	.21	.23	.27	.10
BA 39	.29	.56	.29	.45	.22	.32	.34	.26	.31	.33	.10	.32	.17	.21	.27	-0.01	.31	.14	.13	.25	.19	.12
Broca	.16	.50	.20	.41	.12	.16	.18	.14	.15	.17	.06	.16	.12	.16	.11	.03	.27	.04	.01	.07	.08	.00

Table 5.4: Kendall's τ correlation between brain activation and the unique, differentiating top 20 of the 37 explored complexity metrics (based on average correlation). The metric's text color indicates its class (size, vocabulary, control flow, data flow, other). The cell shading highlights strong correlations (cf. Table 5.3). All metrics, raw data, and results are available on our supplementary Web site.

Exploration of Further Complexity Metrics So far, our analysis concentrated on commonly used representatives of each complexity metric class. However, while widely used, some of the selected representatives exhibit alleged limitations regarding human cognition. For example, the control-flow metric McCabe fails to consider the added complexity of nested structures, such as nested loops or recursion with complex break conditions. More recent control-flow metrics, such as “cognitive complexity” [Cam18], take code repetition, layout, and modern program constructs into account and promise relief of such weaknesses.

To understand whether more advanced (or any) metric predicts cognitive load better, we computed our snippets’ complexity values of 43 further metrics provided by our analysis tools (i.e., MetricsReloaded and SONARQUBE). We included metrics that target the method level. Then, we excluded all complexity metrics that were unable to differentiate between our snippets (e.g., a metric counting the number of TODOs would yield 0 for all snippets and have no differentiating value for our analysis), which left 37 metrics, partially shown in Table 5.4. We categorized the metrics into: 2 size metrics, 8 vocabulary metrics, 12 control-flow metrics, 0 data-flow metrics, and 15 others.

Overall, the 37 complexity metrics show a wide range of correlations with the observed brain activation. Interestingly, some simple control-flow metrics, such as the number of branching statements or the maximum loop depth, correlate more strongly than McCabe. SONARQUBE’s “cognitive complexity” shows an improvement over McCabe, but only a small correlation, at best. This corroborates a prior meta-analysis on the limitations of the cognitive complexity regarding physiological data [BWW20]. In addition to Halstead, the number of parameters is a second vocabulary-based metric that shows a strong correlation with brain activation in Broca’s area.

These findings corroborate that program comprehension is a complex cognitive process. While some simple metrics are well-suited to predict cognitive load in some brain areas, there is no single metric that predicts the overall cognitive effort. Advanced methods that try to capture all aspects are not an accurate predictor for cognitive load. Rather than trying to devise complex metrics with sophisticated and all-encompassing views on complexity, it may instead be worthwhile to use a basket of simple, but targeted metrics, with well-understood relationships with code and cognitive effort. For example, a continuous integration process could check simple metrics with well-understood cognitive relationships: maximum loop depth for constraints on programmers’ working memory (BA6) or the number of parameters as indicator for load on semantic processing (BA39 and Broca).

Summary Considering the four representative complexity metrics, the vocabulary-size-based metric Halstead’s complexity, followed by the data-flow-based metric DepDegree, have shown the most consistent relationship with the various measures of cognitive effort. Control-flow complexity, as measured by McCabe, consistently lacked any relationship with cognitive effort. An exploration of other method-level metrics did not reveal any individual metric that accurately predicts cognitive load. However, beside DepDegree, none considered data-flow despite promising results. Future research shall consider data flow as a predictor for cognitive effort. While no single metric of complexity is sufficient for comprehensively explaining all observed data, we

conclude that programmers should aim at minimizing the number of variables, branching depth, and amount of data flow within methods to reduce cognitive load when comprehending code.

Size-based metrics (Halstead and LOC) align well with the anticipation of work, and therefore with the amount of attention needed to allocate toward a task. Data-flow-based metrics (DepDegree) were less likely to capture this anticipation, yet demonstrated a link to increased cognitive demands during program comprehension. These results imply that complexity estimates that use an assessment of the appearance of code, as is common in readability studies, rather than requiring actual comprehension of code, could misrepresent complexity.

Finally, a useful metric may involve the subjective complexity rating after the completion of a comprehension task. In other words, when participants struggled in grasping the meaning of a code snippet, they find it more complex, and they seem to be well aware of their struggle. They are likely to predict their correctness ($\tau = -.77$), and their rating often matches their level of concentration ($\tau = -.69$), and relates to increases in cognitive effort.

5.1.5.2 Hypotheses

During the analysis and interviews, we found some further interesting insights that we formulate in terms of hypotheses to be addressed in future studies.

Size is a Preattentive Indicator for Cognitive Load We found that, when a snippets consisted of more lines of code, the deactivation of the default mode network was rather strong. Thus, participants might use simply the amount of material to comprehend as heuristic of how much cognitive load they expect. This is an easy to assess property, which might not even require attention and might be a feature of perception [Gol02]. However, this could also lead to overestimation as the comprehension process progresses.

Novel Metric of Data Flow versus Control Flow Although DepDegree builds on McCabe, McCabe shows no relationship to cognitive effort, whereas DepDegree does. Thus, everything that distinguishes DepDegree from McCabe might be responsible for the small to medium correlations. Thus, a metric capturing only the part that differs between DepDegree and McCabe might be a good predictor for cognitive load.

5.1.5.3 Perspectives

Mental Shortcuts during Program Comprehension Programmers tend to minimize efforts for program comprehension by actively looking for efficient ways to solve a task [Roe+12]. In the context of our experiment, this means that participants try to find the simplest path to solve the task. For example, a participant reported, upon recognizing a list of square numbers, they expected an algorithm dealing with square roots (see top-down comprehension in Section 4.1, [Bro78;

[Sie+17]). Complexity metrics seem to neglect that programmers try to take such mental shortcuts. For example, McCabe counts the number of *all* possible execution paths, but programmers often do not have to consider all paths; just enough to solve the current task. For example, in our “hurricane check” snippet, participants could skip some `if` statements after they have found the solution. Further work shall investigate this phenomenon by carefully controlling which mental shortcuts during comprehension are available. This would produce further insights into the context sensitivity of complexity metrics.

The Effects of Extreme Complexity Some metrics may not have a relationship with human cognition, until the values exceed some extreme threshold. Although a higher number of possible execution paths did not increase the cognitive load of participants, there are only 6 different values for McCabe, and the largest (i.e., 8) might not even be considered sufficiently complex. Still, we selected them for their widespread use in practice. Some static analysis tools, such as SONARQUBE, consider a McCabe value of 10 to be so complex that it should not even be checked into a repository. Unfortunately, it is almost impossible to vary the code along all 4 classes of metrics while also adhering to presenting the full code on one screen within the fMRI scanner. Thus, future studies could either use longer code, which would require the participants to scroll up and down, or could focus more on control flow to have more extreme ranges so that the relationship of complexity and cognitive load can be observed in more depth.

Activation of Broca’s Area for High Performers We confirmed the activation of the inferior frontal gyrus (Broca’s area), which is crucially involved in establishing a unified understanding between alternatives (e.g., combining the meaning of words to a sentence) [Hag05] and which was consistently activated in previous fMRI studies on program comprehension [Pei+20; Sie+17; Iku+21] and sentence complexity [KS02].

Correlations between our complexity metrics and the observed activation in Broca’s area are mostly small. One reason could be that activation of Broca’s area was modulated by individual performance. That is, some participants, finding the code too complex, did not activate later stages of language processing. Another possibility could be that the code snippets were more mentally challenging than in previous studies, and as a result, some participants needed to recruit Broca’s area as syntactic working memory [FSF01]. Both explanations are consistent with our data, as we found stronger activation in Broca’s area among the *high-performing participants* (i.e., who correctly solved, at least, 13 of the 16 comprehension tasks, $n = 9$). Studies examining complex sentence comprehension of individuals have also observed activation differences between high-performing individuals with good comprehension (increased activation of Broca’s area) and poor comprehension (decreased activation of Broca’s area) [VEV+16].

Future experiments shall examine when and why Broca’s area is activated by high performers, increasing our understanding of expertise. Furthermore, if we want code to be understandable by *everyone*, then we can use these methods to design metrics that predict truly simple code, code that does not require the brain circuitry associated with complexity.

Overcoming Shortcomings of Complexity Metrics Based on the participants' feedback and the observed relationships to various measures of programmer cognition, we found that popular complexity metrics fail to capture some comprehension aspects that participants used in their subjective rating:

- Long, diffuse code lines can cause particular difficulties. For example, Line 11 of Listing 5.1 obfuscates the intention (is it a non-letter?), which needs to be extracted from the code.
- Identifiers with similar names lead to confusion (e.g., `number1`, `number2`, `numbers` in one snippet), likely because participants have to pay specific attention not to confuse these. Thus, code *readability* may be just as important as structural complexity, as suggested in previous studies [BW08; BW10; PHD11].
- DepDegree fails to consider the “distance” or the locality of data-flow relationships. For example, in an unrolled sort algorithm the swap operations are localized to each line and can be abstracted away once a line has been processed. Incorporating other factors, such as the variable lifetime or lexical distance, would be worthwhile to explore.

These and other aspects shall be considered when using code complexity metrics to describe human cognition (e.g., Jbara and Feitelson consider repeated statements [JF17]). Our experiment design provides a structured way to test and refine code complexity metrics to make them a more accurate proxy for program comprehension and elevate them beyond simple code size predictors [GL17].

5.1.6 Threats to Validity

5.1.6.1 Construct Validity

We carefully designed our experiment to limit threats to validity. With regard to construct validity, we operationalized code complexity with four widely used metrics covering different concepts of code complexity (e.g., control-flow or data-flow complexity). An exploration into 37 further metrics did not reveal any candidates with consistently strong correlations. The four selected metrics correlate with each other even with our carefully designed experiment. Nevertheless, in addition to our conceptual insights, our study outlined how to investigate a possible *cognitive* complexity metric with a multi-modal experiment.

Another threat arises from our operationalization of program comprehension, which is a multi-faceted phenomenon in which the chosen experiment task is decisive for observed cognitive processes [DR00]. In our study, we asked programmers to evaluate snippets regarding input and output, and we found, at most, a medium correlation with code complexity metrics. An experiment with another type of task (e.g., deriving program invariants) may emphasize a different facet of program comprehension and thus may show stronger or weaker correlations with complexity metrics. However, our task operationalization is typical to specify program behavior and in line with previous fMRI studies on program comprehension [Sie+14a; Sie+17; FSW17; Dur+16; Cas+19].

While no single metric of complexity is sufficient for comprehensively explaining all observed data, we can conclude that programmers should aim at minimizing the number of variables, branching depth, and amount of data flow within methods to reduce cognitive load when comprehending the code.

5.1.6.2 Internal and External Validity

Several threats to validity arise from our participant sample.

Participants First, we have a skewed gender distribution, which, however, is close to the population in computer science for most universities. Second, participants may have encountered algorithms used in our snippets before. However, we mitigated this threat by enforcing bottom-up comprehension.

Code Snippet Selection Due to the nature of controlled fMRI experiments, we intentionally focused on high internal validity to control for confounding parameters as much as possible. Our snippets are rather small, in one programming language, and we selected a homogeneous sample in terms of programming experience. Thus, our results apply only to similar circumstances and cannot easily be generalized, for example, to expert programmers or large code bases. This is an unavoidable trade-off between targeting either high internal or high external validity [SSA15].

Code Complexity Granularity We need to be aware that, in our experiment, we studied program comprehension at the method level, but software systems consist of many methods and higher-level components. Nevertheless, our results still have practical impact: When we know that intermediate programmers work at the method level, code complexity metrics can help to predict their cognitive effort and that they might need longer than expected. Furthermore, different complexity metrics have been devised beyond the method level (e.g., Weighted Methods per Class [CK94], Lack of Cohesion in Methods [HS95; HM95]), which shall be addressed in future research. Our study provides a starting point for dedicated follow-up studies that shall investigate these metrics and associated cognitive processes.

5.1.7 Related Work

Beside the work that evaluates how software metrics are related to human cognition (cf. Section 5.1), several neuroimaging studies exist that shed light on how programmers work with code [Sie+14a; Sie+17; Yeh+17; CK14; Nak+14; FSW17; Fak+18; Kos+18; Iku+21; Hua+19; Kru+20]. While all these studies considered neuronal correlates of processes related to program comprehension, none establishes a link to software metrics.

5.1.8 Conclusion

Code complexity metrics are relevant for researchers and practitioners alike, especially as proxy for difficulty during comprehending code. Despite their widespread use, the validity of code complexity metrics is debated and, despite substantial research, their usefulness is still unclear. To shed light on this issue, in an fMRI study, we investigated 41 complexity metrics and their behavioral and neuronal correlates during program comprehension. We found corroborating evidence with mostly weak to medium correlations with programmers' correctness and response time. More importantly, since we observed participants' brain activation with fMRI, we enriched previous research by offering a novel perspective and explaining *why* code and certain aspects of it are difficult to comprehend. In particular, we found that the code's textual size drives cognitive load due to programmers' expected attention and that vocabulary size of code particularly burdens programmers' working memory.

Despite these encouraging results, further studies need to dig deeper to better understand the suitability of code complexity metrics as a proxy for programmers' cognition. Data-flow-based metrics, such as DepDegree, showed promise and need further investigation. Future work shall also address the gap on how individual programmer behavior and knowledge enables mental shortcuts, which likely reduce generic precision of complexity metrics.

5.2 The Role of Aggregation in Human Studies

Besides the dedicated experiment to understand the link between code complexity metrics and programmers' cognition, we used our gathered data in two projects to further contribute to using neuroimaging methods in software-engineering research. Due to space considerations, we only provide a high-level overview and refer to the publications for full details (Section 5.2: [Sie+21] and Section 5.3: [Neu+21]).

In the first project, we tackled the prevalent issue of optimally analyzing human-response data [Sie+21]. Human studies often fail to use the full potential of analysis methods by combining analysis of individual tasks and participants with an analysis that aggregates results over tasks and/or participants. This may hide interesting insights of tasks and participants and may lead to false conclusions by overrating or underrating single-task or participant performance. Here, we show that studying multiple levels of aggregation of individual tasks and participants allows researchers to have both, insights from individual variations, and generalized, reliable conclusions based on aggregated data. First, we conduct a literature survey to reveal that most human studies perform either a fully aggregated analysis or an analysis of individual tasks. To show that there is important, non-trivial variation when including human participants, we re-analyze 12 published empirical studies, thereby changing the conclusions or making them more nuanced. Moreover, we demonstrate the effects of different aggregation levels by answering a novel research question on four sets of fMRI data. We show that, when more data are aggregated, the results become

more accurate. This proposed technique can help researchers to find a sweet spot in the tradeoff between cost of a study and reliability of conclusions.

We support replication by providing relevant material.²⁵

5.2.1 Literature Analysis

	Total Number	Percentage
Published Papers	1584	100%
... include empirical study	1579	99%
... with human participants	397	25%
... with defined tasks	165	10%

Table 5.5: Overview of number of papers that include an empirical study with human participants, in which tasks were defined in ICSE, ESEC/FSE and EMSE between 2011 and 2018.

We first gained an overview of software-engineering literature with human experiments. The human factor in software-engineering research has grown more and more important: Between 1993 and 2002, only 1.9 % of all empirical studies included human participants [Sjø+05]. In 2010, this number has increased to 18 % [BSW11], and between 2011 and 2018, it increased further to 25 %. We conducted a literature survey of the past eight instances of the International Conference on Software Engineering (ICSE), the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), and the Empirical Software Engineering Journal (EMSE) as primary venues for empirical research on software engineering. We show a high-level overview in Table 5.5.

We further examined how the empirical studies are typically analyzed, that is, per task or aggregated over (categories of) tasks. We illustrate different aggregation methods of task-wise, participant wise, and both in Table 5.6. Of 165 studies that have defined tasks with human participants, 22 studies used a task-wise analysis, 71 conducted an aggregated analysis, and 51 used a combined approach. Thus, the question of whether to aggregate or not to aggregate is relevant.

5.2.2 Re-Analysis of Behavioral Studies

Second, we re-analyzed studies that conducted a task-wise analysis to understand how aggregation can affect the results. To this end, we aggregated the response data of 12 papers and reran the analysis on the data that we aggregated. In a nutshell, we found instances where the aggregation changed the conclusions (6) and where it led to the same conclusions (6). So, in

²⁵<https://github.com/brains-on-code/conducting-and-analyzing-human-studies>

Participant	Task 1	Task 2	Task 3	Total
Expert 1	5	2	8	5
Expert 2	4	4	4	4
Expert 3	2	3	4	3
Expert 4	5	4	6	5
Experts	4	3.25	5.5	4.25
Novice 1	3	6	9	6
Novice 2	3	3	3	3
Novice 3	6	4	5	5
Novice 4	8	6	10	8
Novices	5	4.75	6.75	5.5

Table 5.6: Illustration of aggregation per participant (i.e., participant-wise), per task (i.e., task-wise), and a combination of both. The values represent fictional response times [min] of four novice and four expert participants to three tasks.

half of the cases analyzed, aggregation changed at least partially the conclusion, indicating that researchers need to be aware of the effects.

5.2.3 Re-Analysis of fMRI Studies on Program Comprehension

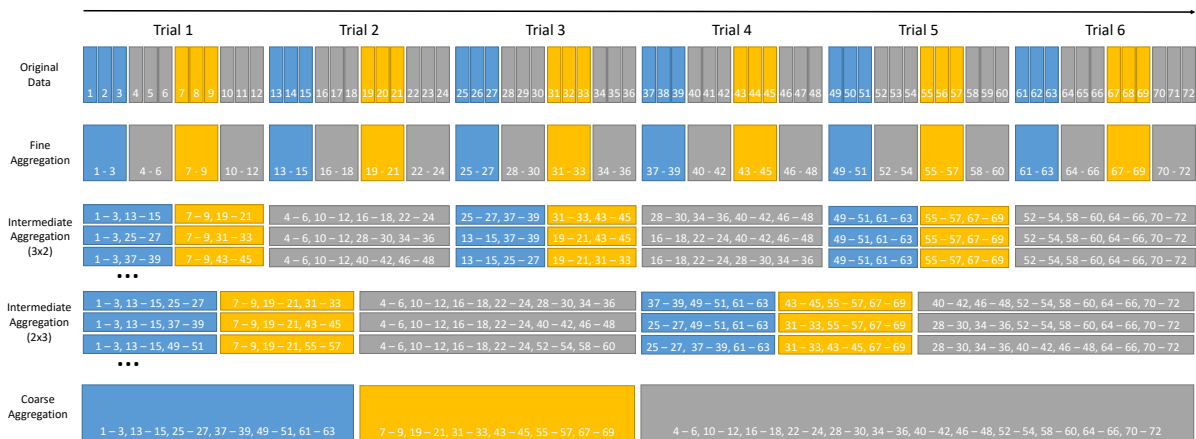


Figure 5.7: Visualization of fine, intermediate, and coarse aggregation levels. Blue boxes represent brain scans of comprehension tasks, yellow boxes represent brain scans of syntax tasks, and gray boxes represent brain scans of rest.

Third, we trained a classifier on the data of three previous fMRI studies in which participants completed different kinds of tasks [Sie+14a; Sie+17; Pei+18c]. The classifier should predict the

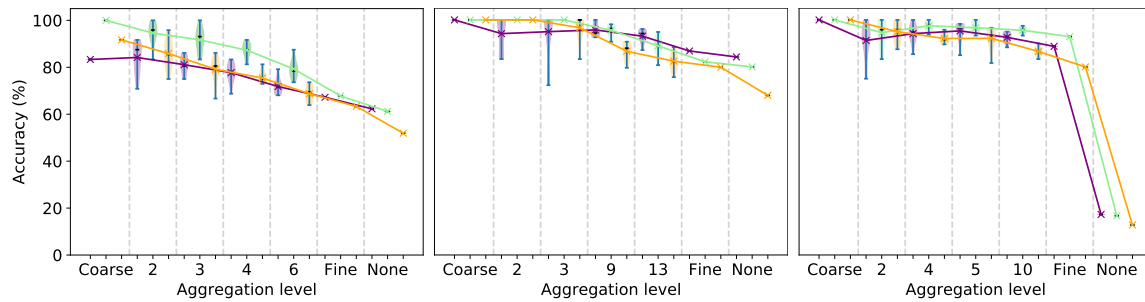


Figure 5.8: Accuracies of the classifier for the ICSE (left), FSE (center), and ESEM study (right). Each figure shows different aggregation variations, from a coarse aggregation on the left to no aggregation on the right. The numbers denote to how many groups data were aggregated. The colors denote the set of voxels; purple: activated voxels, yellow: deactivated voxels, green: union.

kind of tasks that participants were completing, similar to the study by Floyd et al. [FSW17]. To evaluate how aggregation affects the accuracy of the classifier, we applied an aggregation function to the data and fed both, the aggregated data and raw data, to the classifier. With the aggregated data, the classifier performed better, which contradicts the assumption that more data means higher accuracy. In other words, aggregation also affects results in this case. Furthermore, we defined intermediate aggregation levels, such that a subset of tasks is aggregated (cf. Figure 5.7). As visualized in Figure 5.8, the more tasks are aggregated, the better the prediction accuracy.

Our re-analysis of fMRI data demonstrates how aggregation of several tasks in can improve reliability and validity of measurement. If many task repetition are an unfeasible effort for a software-engineering study, we propose using intermediate aggregation levels. They are compromise between study effort (in terms of task creation and duration) and reliability and validity of measurement.

5.2.4 Conclusion

In this project, we combined three methods for a triangulation approach. We show the prevalence of how often researchers face the decision of whether to aggregate or not to aggregate. Subsequently, with the re-analysis of human studies and re-using the data of our fMRI studies, we demonstrate the effect of aggregation on the results and conclusions. In one case, we obtained even a reversed result. Thus, we make the community aware that the decision of whether and how data should be aggregated is not trivial and should be considered carefully.

5.3 fMRI Analysis with Participant-Specific Brain Parcellation

In the second project, we also re-analyzed of a previous fMRI study but with a different objective. Here, we use the fMRI data from our study on top-down comprehension (cf. Section 4.1) to demonstrate how using participant-specific brain parcellation can fundamentally change how we analyze fMRI data [Neu+21]. In conventional fMRI data analysis, voxels can be grouped into regions of interest on which we carry out tests for statistical significance. Grouping the signal of many voxels into a region typically leads to a higher sensitivity when compared with voxel-wise multiple testing approaches. In the case of a multi-subject study, we propose to define the regions for each subject separately based on their individual brain anatomy, represented, for example, by so-called Aparc labels. The aggregation of the subject-specific evidence for the presence of signals in the different regions is then performed by means of a combination function for p -values. We apply this methodology to our fMRI data on top-down comprehension and demonstrate that our approach can perform comparably to a two-stage approach.

Siegmund et al. initiated using fMRI as measurement method for program comprehension and located a network of activated brain areas based on a voxel-wise analysis [Sie+14a]. To confirm their findings, we conducted a follow-up experiment and tested the identified brain areas (cf. Section 4.1, RQ 4.1). This process to establish brain areas that are activated for a cognitive process requires two independent studies. The first study defines the regions of interest with a voxel-wise analysis and the second study confirms the findings with a regions-of-interest analysis.

5.3.1 Method

In this project, we propose an alternative analysis method which only requires one study. Instead of a voxel-wise analysis, we group the functional data based on a participant-specific parcellation of the anatomical data. To this end, we chose the Harvard-Oxford brain atlas [Mak+06; Des+06] that provides a parcellation based on gross anatomical landmarks and delivers an Aparc label for each voxel of each individual brain space. Next, we conduct statistical tests on the group data for each anatomical Aparc label. We illustrate this process in Figure 5.9. Such a priori definition increases the statistical power [STD16].

5.3.2 Example Use Case

We demonstrate this analysis method with the fMRI data of the top-down comprehension study. We do not use the defined regions of interest by Siegmund et al. Instead, we only use the anatomical parcellation of each participant. We used FreeSurfer to segment and parcellate the brain of each participant based on their anatomical scan [Fis+02; Fis+04]. We used the Destrieux' cortical atlas to assign Aparc labels on an individual participant basis [Des+10]. We then conduct a statistical test for each Aparc label outlined in the original paper [Neu+21].

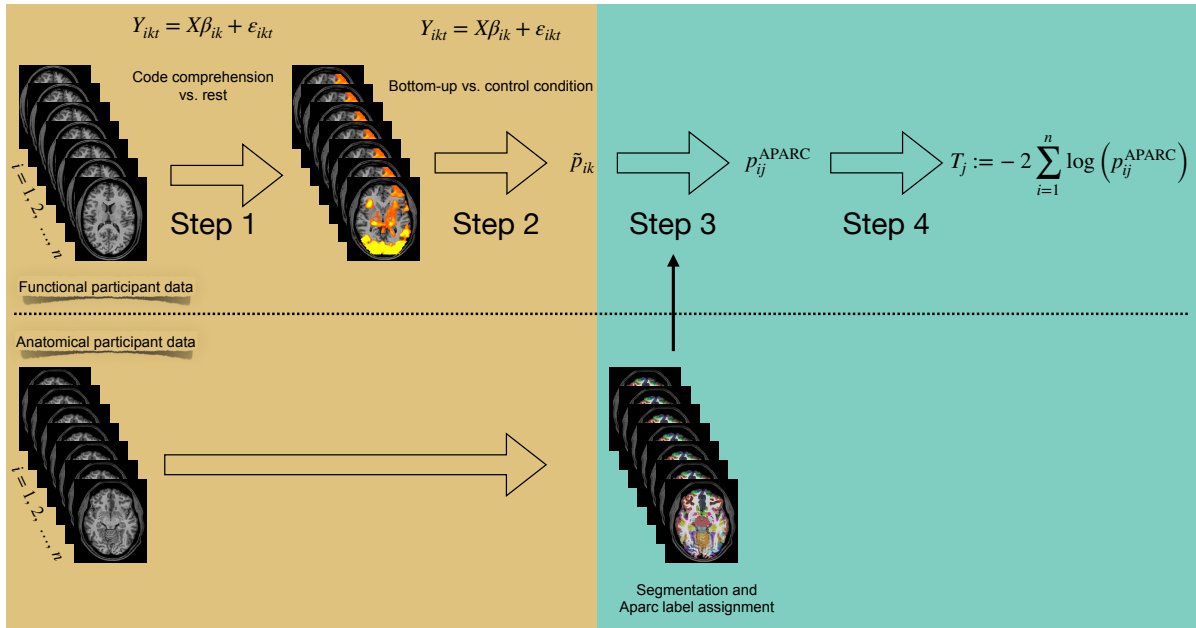


Figure 5.9: Illustration of processing of the fMRI data. The box in Harvest Gold indicate analysis steps that have already been performed in Section 4.1. The box in Monte Carlo indicates the processing steps conducted for this project.

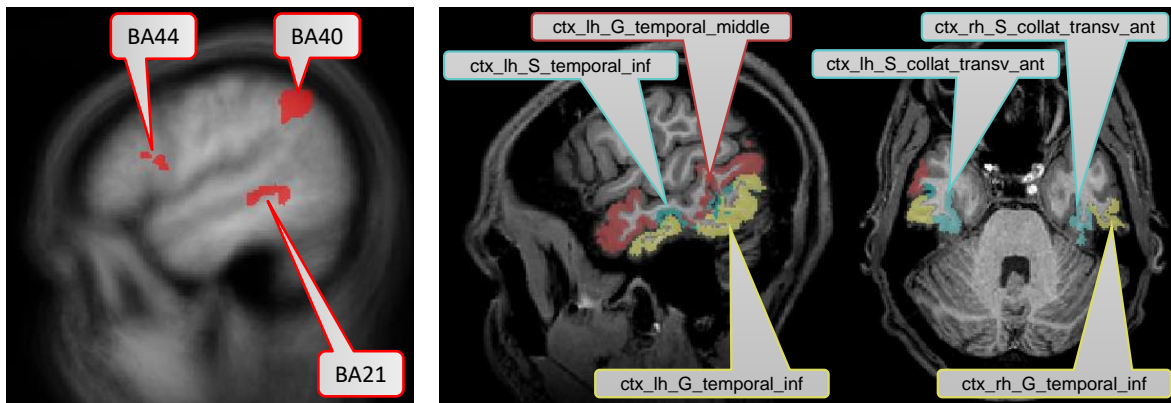


Figure 5.10: Comparative visualization of significantly activated brain areas. On the left are the left-lateralized brain areas (i.e., BAs 21, 40, 44) activated during program comprehension confirmed in Section 4.1. On the right are the significantly activated Aparc labels identified in this analysis, particularly in the middle and inferior temporal lobe.

We compare the found network of brain areas to our previous results in Figure 5.10. While they are not directly comparable due to the differences in their atlases, the advantage of the proposed method is clear. It found activation clusters in similar areas, in particular in the middle and inferior temporal lobe without prior knowledge. These areas have been found in many other fMRI studies as well [Pei+18c; Cas+19; Iku+21].

The presented analysis method is not limited to fMRI studies of programmers. Rather, this project illustrated how our neuro-cognitive studies of programmers can help to improve neuroscience methods.

5.4 Chapter Summary

In this chapter, we presented three applications of our neuro-cognitive experiment framework. First, we conducted a dedicated fMRI study on code complexity metrics. In a nutshell, we found that changes in complexity metrics can explain differences in programmer cognition to different extents. Size-based and vocabulary-based metrics correlate with the anticipation of work, whereas data-flow metrics correlate with higher cognitive demands in a network of brain areas activated during program comprehension. Simple control-flow metrics (e.g., number of branches) predict cognitive load better than more complex control-flow metrics (e.g., McCabe). In general, not many code complexity metrics incorporate data flow, which seem to capture a distinct aspect of cognition, showing a promising avenue of further work. From a more abstract view, this study showed how researchers can use fMRI to tackle critical, open questions in software engineering. The framework presented in this dissertation and conducted studies demonstrate the value of a neuro-cognitive perspective.

In addition to direct contributions to software engineering, our interdisciplinary projects provided further insights for empirical software-engineering research with human participants as well as analysis of any fMRI studies with dedicated tasks. We re-analyzed several of our fMRI studies to show how different aggregation levels strongly affect the robustness of drawn conclusions. We also re-analyzed our fMRI study on top-down comprehension to demonstrate the feasibility of an alternative fMRI analysis method. While the neuro-cognitive perspective of programmers and its methodology are still in the early stages, this chapter showed how we already can make a practical impact on industry and research.

6 Conclusion and Future Work

Programming is on the verge of becoming the fourth literacy. Understanding the underlying cognitive processes of programming is therefore fundamentally important to shape future education and practice. While understanding program comprehension as a core activity of programming has been the focus of many decades of research, conventional methods have limited researchers. In this dissertation, we introduced a novel neuro-cognitive perspective of program comprehension that establishes methodological standards of using fMRI in combination with other modalities. We demonstrated the potential of our framework with several studies shedding new light on our existing understanding of program comprehension. Overall, the research in this dissertation made progress in key areas:

Multi-Modal Experiment Framework Adopting fMRI for software-engineering research is still in its infancy. But, researchers are quickly adapting it with an ever-increasing number of studies. While these studies already provided plenty of new insights, methodological standards are necessary to establish fMRI as a widely accepted measure. We explored several aspects on how to properly design fMRI studies and outlined further topics.

In Chapter 3, we showed how fMRI provides objective evidence on cognitive processes of programmers. We then discussed a limitation regarding fast cognitive processes, such as recognizing beacons. Thus, we evaluated a combination of fMRI with simultaneous eye tracking to further understand program comprehension and its phases. Our experiment showed that, despite some technical limitations in our setup regarding eye-tracking data quality, it is feasible and insightful to combine fMRI with eye tracking. In an exploratory data analysis, we evaluated whether secondary eye-tracking measures, such as pupil dilation or blink behavior, can provide an additional perspective to a programmer's mind. While generally pupil dilation and blink behavior showed promise, the fMRI environment introduces confounding factors that must be controlled. We identified value in further testing experiment designs and analysis procedures that enable researchers to use secondary eye-tracking measures.

In a follow-up step, we pushed for a multi-modal experiment framework that combines fMRI and eye tracking with further modalities. Specifically, we added psycho-physiological measures to capture high-level physiological and psychological states, such as stress. In combination, all these measures provide a more comprehensive view on a programmer's mind. Finally, we also outlined several methodological refinements that shall be explored in future research.

Throughout Chapter 3, we showed that multi-modal experiments are highly promising. However, they require sound analysis procedures and, to amplify their benefits, effective tool support. We

described how to maximize our insights from an experiment with a set of multi-modal analysis strategies. To further support future efforts, we developed a prototype of *CODERSMUSE* that enables researchers to explore multiple multi-modal data streams at once and generate new specific hypotheses for future work. Due to its multi-modality, *CODERSMUSE* allows identifying complex relationships across modalities, such as a programmer's focus on a loop structure (eye tracking) leads to an increase in stress (psycho-physiological measures) due to a high load of working memory (fMRI). Overall, our multi-modal framework provides a novel neuro-cognitive perspective of program comprehension. This way, we now have a new methodology to objectively study a programmer's mind for a wide range of questions in research and practice.

Neuro-Cognitive Perspective of Program Comprehension Following the establishment of our experiment framework, we tackled open issues in our understanding of program comprehension in Chapter 4. In particular, we investigated a neuro-cognitive view of the historical theories of top-down comprehension and bottom-up comprehension. In an fMRI experiment, we found that top-down comprehension largely activates the same network as bottom-up comprehension, but with a much higher neural efficiency. This experiment demonstrates how our experiment framework provides a way to validate and refine theories on how a programmer's mind works.

In a second research avenue, we started to demystify programming expertise. For decades, software-engineering research has observed substantial differences in productivity and skill level, even between programmers of comparable backgrounds. In an eye-tracking study, we showed how programmers with different levels of experience use different reading patterns when understanding source code. Next, we explored insights from psychology, neuroscience, and software engineering on expertise to understand potential factors in programming expertise. In a data exploration of an fMRI experiment on bottom-up comprehension, we identified that knowledge in a programming language appears to reduce cognitive load. General programming experience, however, does not show such a relationship. With our experiment, we paved the way for future studies on programmer experience with a neuro-cognitive perspective.

Overall, Chapter 4 showed how insightful fMRI experiments are in software engineering. This further substantiates the value of our experiment framework. In the near future, many open questions in program-comprehension research and software engineering can be tackled and provide novel insights.

Feedback to Software-Engineering Research This dissertation focused on program comprehension. But, a neuro-cognitive perspective is applicable to further fields of software engineering. In another fMRI study, we demonstrated one important use case by linking programmers' cognition to code complexity metrics, which have been widely debated. Our experiment revealed that widely used complexity metrics are limited regarding how much they can predict programmers' cognitive load. Through interviews, we identified several shortcomings of existing complexity metrics and outlined a way to develop a set of more accurate complexity metrics. Our experiment framework can here also be valuable in follow-up experiments to test and refine complexity metrics that precisely reflect programmers' cognitive load.

In conclusion, software-engineering researchers gained a valuable method to objectively measure programmers' minds. The research in this dissertation made important contributions in establishing this new methodology of a neuro-cognitive perspective. We also demonstrated its potential across several studies.

Future Work

The research presented in this dissertation laid the foundation for future work on the methodology and application. We opened up a multitude of further research avenues which we briefly discuss next.

Further Establish Methodology The work in this dissertation facilitated fMRI studies in software-engineering research. But, our work has uncovered several further improvements that must be made to further establish fMRI as a standard measurement in software engineering. Thus, we need to continue our research efforts to further establish and refine our experiment framework.

Specifically, we underline that fMRI studies require a well-designed experiment with proper tasks and control conditions. Throughout our fMRI studies, we observed the importance of a suitable task instruction and control condition. They play a major role in clearly isolating the correct cognitive process. Our framework enabled future studies that explore various options for tasks and control conditions. This is critically important as a growing number of fMRI studies on programmers show a wide range of experiment designs. Without a common standard, it becomes difficult to compare and synthesize results across these studies. We continue to investigate how, depending on the research question, to correctly operationalize program comprehension in the context of fMRI studies.

While this dissertation concentrated on fMRI along with eye tracking, there are other neuroimaging measures that have been established in neuroscience. Specifically, EEG and fNIRS offer alternatives to measure programmers' minds. Software-engineering researchers already have experimented with EEG and fNIRS, but there is a need to establish methodological guidelines on choosing the correct measure along with a proper design for the research question. Further, a sound combination of several measures, such as fMRI with simultaneous eye tracking, is an option that shall be established.

Advance Multi-Modal Research of Program Comprehension This dissertation started a push towards a multi-modal experiment design benefiting from the strengths of several modalities. We established a combination of fMRI and eye tracking. We also outlined an option to include psycho-physiological measures. With these first steps already made, future work can further develop an all-encompassing experiment framework that provides a comprehensive view on a programmers' mind.

Although there is a large potential for such multi-modal research, it comes with its own set of challenges. In particular, ensuring high data quality across all modalities requires a fine-tuned setup. Further, exploiting the gathered multi-modal data to its full potential requires new analysis procedures including proper tool support. In this dissertation, we already started the development that shall be fine-tuned to the needs of future research.

Continue Exploring Program Comprehension In this dissertation, we presented one study to contrast top-down comprehension and bottom-up comprehension. But, there are many further facets of program comprehension that would prosper under a neuro-cognitive perspective. With our framework and conducted studies, we provide a template for future studies. We already drafted an experiment on the differences of underlying cognitive processes of programmers with various experience levels. But, this is only the beginning. There are many theories that have been formulated in the past and could be validated and refined with our experiment design. Ultimately, the work in this dissertation helps us to work toward a comprehensive theory of program comprehension.

Examine Additional Applications of fMRI in Software Engineering We have demonstrated in Chapter 5 that, beyond just the field of program comprehension, there are other areas of software engineering that could benefit from our experiment framework. Neuroscience has made substantial progress in their theories of the human brain with consistent use of fMRI. By applying the same methods, we can make similar progress in understanding the programmers' minds in the context of software-engineering practice. Our study on complexity metrics serves only as a starting point.

In summary, this dissertation already made the first steps, but there are many open challenges in software engineering that shall be explored with neuroimaging. We are convinced that the neuro-cognitive perspective will further emphasize its value to software-engineering research and practice.

7 Appendix

7.1 fMRI Scanner Setting and Analyses Procedures

We conducted all fMRI experiments described in this dissertation at the Leibniz Institute for Neurobiology in Magdeburg, Germany. When the participants arrived, they gave their consent to participate in the study. We then conducted the study described in the respective sections.

fMRI Scanner Setting

We carried out the imaging sessions on a 3-Tesla scanner,²⁶ equipped with a 32-channel head coil. The participants' heads were fixed with a cushion with attached ear muffs containing fMRI-compatible headphones.²⁷ Participants wore earplugs to further reduce scanner noise by 40 to 60 dB. We obtained a T1-weighted anatomical 3D data set with 1 mm isotropic resolution of the participant's brain before the functional measurement.

To capture a whole-head fMRI, we acquired functional volumes using a continuous echo planar imaging (EPI) sequence. The study on code complexity metrics used a multi-band EPI sequence. The number of volumes and scan time depended on the study which we present along with further details in Table 7.1.

fMRI Data Preparation

For all fMRI studies, we used BrainVoyager™ QX 2.8.4²⁸ to process the data. We preprocessed the functional data with 3D-motion correction, slice-scan-time correction, and temporal filtering (high-pass GLM Fourier, 2 cycles). We transformed each participant's anatomical scan into the standard Talairach brain [TT88] (to account for anatomical differences between participants' actual brains). Before group analysis, we spatially smoothed each participant's functional data with a Gaussian filter (FWHM=4 mm). Based on this transformation and smoothing, we could then analyze the fMRI data with the specific needs of each study.

²⁶Philips Achieva dStream, Best, The Netherlands

²⁷MR Confon GmbH, Magdeburg, Germany, <http://www.mr-confon.de>

²⁸Brain Innovation BV, Maastricht, The Netherlands, <http://brainvoyager.com>

Study	# of Volumes	Scan Time	EPI Sequence Configuration
fMRI & Eye Tracking (Section 3.2, [Pei+18c])	878	28 min	EPI sequence echo time [TE]: 30 ms; repetition time [TR]: 2000 ms; flip angle, 90°; matrix size, 80 x 80;
Top-Down Comprehension (Section 4.1, [Sie+17])	930	31 min	field of view, 24 cm x 24 cm; 35 slices of 3 mm thickness with 0.3 mm gaps
Code Complexity Metrics (Section 5.1, [Pei+21])	Dynamic	Dynamic (up to 30 min)	multi-band EPI sequence echo time [TE]: 30 ms repetition time [TR]: 1200 ms flip angle [FA]: 60° multi-band acceleration factor: 2 36 slices of 3 mm thickness with 0.3 mm gaps

Table 7.1: fMRI scanner configuration for EPI sequence of the three fMRI studies.

7.2 Programmer Experience Questionnaire of Siegmund et al.

D1: What is your age (in years)?

D2: What is your gender?

Male Female

D3: Which year did you start your current degree?

D4: What is your major?

E1: On a scale of 1–10, what do you estimate is your programming experience?

E2: How long have you been programming (in years)?

E3: How many courses did you take in which you had to program?

E4: How experienced are you with the following programming languages?

Java: Very inexperienced Inexperienced Medium

Experienced Very experienced

C: Very inexperienced Inexperienced Medium

Experienced Very experienced

Haskell: Very inexperienced Inexperienced Medium

Experienced Very experienced

Prolog: Very inexperienced Inexperienced Medium

Experienced Very experienced

E5: How many further programming languages do you know with on at least a medium level?

E6: How experienced are you with the following programming paradigms?

Logical: Very inexperienced Inexperienced Medium

Experienced Very experienced

Functional: Very inexperienced Inexperienced Medium

Experienced Very experienced

Imperative: Very inexperienced Inexperienced Medium

Experienced Very experienced

Object-oriented: Very inexperienced Inexperienced Medium

Experienced Very experienced

E8 Have you contributed to a large software project at a company or your university?

Yes No

If yes, for how long (in years)? _____

If yes, in which domain? _____

If yes, how many lines of code did the projects have on average?

<900 LOC 900–40 000 LOC ≥40 000 LOC

E9: How do you estimate your programming experience with other students in your study year and programmers with 10 years of experience?

Other students: Much worse Worse Identical

Better Much better

Programmers: Much worse Worse Identical

Better Much better

7.3 Programmer Experience Questionnaire (Extended for Professionals)

Demographics

D1: What is your age (in years)?

D2: What is your gender?

Male Female Diverse Prefer not to answer

D3: What is your handedness?

Left Right Noth

D4: Do you have a color vision deficiency?

Yes No Prefer not to answer

D5: Do you have trouble concentrating?

Very little Little Sometimes Often Very often

D6: Do you have trouble reading?

Very little Little Sometimes Often Very often

D7: What is currently your profession?

University student (undergraduate, bachelor) University student (graduate, master)
 University employee (PhD, postdoc, professor) Professional programmer, developer
 Other: _____

D8: What is your formal education in computer science, software engineering?

No formal qualification, self-taught Vocational training program, course
 University degree (undergraduate, bachelor level) University degree (graduate, master level)
 University degree (graduate, PhD level) Other: _____

Programming Experience

Programming Education

Q1.1: How many years have you worked at a company or studied in a tech-related field?

0–5 years 5–10 years 10–15 years \geq 15 years

Q1.2: How long have you been learning programming (in years)?

Q1.3: How long have you been programming professionally (in years)?

Programming Languages and Paradigms

Q2.1: Which programming languages are you familiar with?

Q2.2: Which programming languages are you most experienced in?

Q2.3: Which programming language(s) are you currently working with?

Q2.4: How many years have you been programming in Java?

D2.5: How experienced are you with the following programming paradigms?

Logical: Very inexperienced Inexperienced Mediocre
 Experienced Very experienced

Functional: Very inexperienced Inexperienced Mediocre
 Experienced Very experienced

Imperative: Very inexperienced Inexperienced Mediocre
 Experienced Very experienced

Object-oriented: Very inexperienced Inexperienced Mediocre
 Experienced Very experienced

Q2.6: How do you estimate your programming experience with other students in your study year and programmers with 10 years of experience?

Other students: Much worse Worse Identical
 Better Much better

Programmers: Much worse Worse Identical
 Better Much better

General Interest in Programming

Q3.1: Which of the following technical/programming-specific content do you regularly consume?

Online courses/seminars Video tutorials (Youtube, ...) Articles/books
 Tech news Other: _____ None

Q3.2: Which of the following technical/programming-specific content have you produced in the past?

Online courses/seminars
 Video tutorials (Youtube, ...)
 Do you produce video (tutorial) content on a regular basis? Yes No

Articles/books
 Contribution to open-source projects
 How many open-source projects have you worked on so far? _____
 How many lines of code did these projects have on average? _____

Other: _____
 None

Professional Programmers

Q4.1: How much time do you typically spend on the following activities per week?

Total work time per week (in hours): _____ h

Meetings _____ h

Code reviews/process _____ h

Programming _____ h

Tests _____ h

Deploy/Operations _____ h

Mentoring _____ h

Learning/Training _____ h

Other: _____ h

Q4.2: Your primary development activities include (check all that apply)?

APIs Front-end UI design Automation testing Database

Data analysis Configuration management and orchestration CI/CD

Security Other: _____

7.4 Comprehension Snippets

In Table 7.2, we provide an overview over all used snippets for comprehension tasks across the studies presented in this dissertation. The full versions with all variations of the used snippets are provided on their respective project Web sites.^{29 30 31 32}

²⁹<https://github.com/brains-on-code/simultaneous-fmri-and-eyetracking/tree/master/images/code>

³⁰<https://github.com/brains-on-code/paper-esec-fse-2017/tree/master/snippets>

³¹<https://github.com/brains-on-code/fMRI-complexity-metrics-icse2021/tree/master/replication/task-comprehension>

³²https://github.com/brains-on-code/eyetracking-linearity-replication/tree/master/1_stimuli/Snippets

Modality	fMRI			Eye Tracking
Study Chapter	fMRI & Eye Tracking Section 3.2	Top-Down Comprehension Section 4.1	Code Complexity Metrics Section 5.1	Reading Order Section 4.2
Number	ArrayAverage	ArrayAverage <i>Listings 4.1, 4.2, 4.3</i>	ArrayAverage	Calculation InsertSort MoneyClass Rectangle SignChecker Student SumArray Vehicle
	BinaryToDecimal	BinaryToDecimal	BinaryToDecimal	
	CrossSum	CrossSum	CrossSum	
	DoubleArray	DoubleArray		
	Factorial	Factorial	Factorial	
			FibonacciVariation	
	FirstAboveThreshold	FirstAboveThreshold		
			GreatestCommonDivisor	
			hIndex	
			Power	
Word				CheckIfLetters Street <i>Listing 4.4</i>
	ContainsSubstrings	CommonChars ContainsSubstrings	ContainsSubstrings ContainsYesOrNo	
	CountSameChars	CountSameChars		
	CountVowels	CountVowels	CountVowels	
	IntertwineTwoWords	IntertwineTwoWords		
			HurricaneCheck	
			LengthOfLastWord <i>Listing 5.1</i>	
	Palindrome	Palindrome	Palindrome	
	ReverseWord	ReverseWord		

Table 7.2: Overview of all snippet algorithms used in studies of this dissertation. While the algorithm name may be identical across studies, the implementation can vary (e.g., different identifier naming, recursion instead of iteration).

7.5 Informed Consent and Participant Safety Questionnaire

For our studies, we used the informed consent and participant questionnaire presented in Figure 7.1.

Fragebogen für Teilnehmer/innen an Magnetresonanztomographieuntersuchungen an der Klinik für Neurologie der OVG-Universität Magdeburg oder am Leibniz-Institut für Neurobiologie Magdeburg

Name:.....
 Vorname:.....Geschlecht:.....Gewicht:.....
 Geburtsdatum:.....Größe:.....
 Straße und Hausnummer:.....PLZ:.....
 Wohnort:.....eMail:.....
 Telefon:.....
 Beruf:.....IBAN:.....
 Bank:.....BIC:.....

Beantworten Sie bitte folgende Fragen zu möglichen Gegenanzeigen für Ihre Teilnahme an den Untersuchungen (Zutreffendes unterstreichen):

Sind Sie Träger eines Herzschrittmachers oder anderer elektrischer Geräte?	ja	weiß nicht	nein
Tragen Sie metallische Implantate (zum Beispiel Zahnschrauben, künstliche Gelenke, Knochennägel oder metallische mechanische Verhütungsmittel)?	ja	weiß nicht	nein
Befinden sich an oder in Ihrem Körper andere metallische Fremdkörper (z.B. Piercing) ?	ja	weiß nicht	nein
Haben Sie Tattoos?	ja	weiß nicht	nein
Wurde bei Ihnen eine Gefäßoperation durchgeführt?	ja	weiß nicht	nein
Haben oder hatten Sie einen Tinnitus?	ja	weiß nicht	nein
Haben Sie ein Anfallsleiden (Fallsucht, Epilepsie)?	ja	weiß nicht	nein
Leiden Sie unter Platzangst?	ja	weiß nicht	nein
Besteht die Möglichkeit, dass Sie Schwanger sind?	ja	weiß nicht	nein

Beantworten Sie bitte folgende für unsere Untersuchungen wichtigen Fragen:

Sind Sie Brillenträger/in?	ja	weiß nicht	nein
Tragen Sie Kontaktlinsen?	ja	weiß nicht	nein
Haben Sie Hörprobleme?	ja	weiß nicht	nein
Sind Sie linkshändig oder rechtshändig?	links	weiß nicht	rechts

Ich habe alle Fragen auf dieser Seite wahrheitsgemäß und nach bestem Wissen beantwortet.

Ort	Datum	Unterschrift der Probandin/des Probanden
-----	-------	--

Informationsblatt

Richtung zurückdreht so drehen sich die Atomkerne des Wasserstoffs nach Abschalten der Radiowellen zurück in ihre ursprüngliche Richtung. Dabei senden sie nun ihrerseits Radiowellen zurück, deren Stärke und zeitliches Verhalten vom Gewebetyp abhängt. Zudem kann man durch das zusätzliche Schalten von sogenannten Magnetfeldgradienten den räumlichen Ursprung der Radiowellen eindeutig festlegen. Ein Computer errechnet schließlich aus den aufgezeichneten Radiowellen die Schnittbilder, welche man fachlich auch als *Tomogramme* bezeichnet.

Ziele der Untersuchungen

Mit einer Magnetresonanztomographieuntersuchung können verschiedene Ziele verfolgt werden. Beispielsweise kann ein Bild aufgenommen werden, mit dem Abstände und Volumina von unterschiedlichen Strukturen vermessen werden sollen. Eine andere Variante, nämlich die sogenannte funktionelle Magnetresonanztomographie (fMRT), misst die Vorgänge im Gehirn begleitenden Durchblutungsänderungen mit hoher räumlicher Auflösung. Sie werden vor jeder Untersuchung ausführlich über das konkrete Ziel der Messung informiert.

Mögliche Risiken der Methode

Der Magnetresonanztomograph hält alle für die Sicherheit des Betriebes und insbesondere die Sicherheit der Probanden oder Patienten erforderlichen Grenzwerte ein. Er wurde vom TÜV einer Sicherheitsprüfung unterzogen und wird darüber hinaus in den vorgeschriebenen Intervallen überprüft. Dennoch müssen die nachfolgenden Punkte beachtet werden:

- Herzschrittmacher können im Magnetfeld ihre Funktionsfähigkeit verlieren. Deshalb dürfen Personen mit Herzschrittmachern nicht an den Untersuchungen teilnehmen.
- Personen mit Cochlea-Implantaten, Neurostimulatoren, Defibrillatoren oder Pumpensystemen sollten nicht einem hohen Magnetfeld ausgesetzt werden, da es auch in diesen Fällen zu Risiken durch magnetische Kräfte oder Felder kommen kann.
- Metallische Implantate und andere Fremdkörper wie Geschossteile können ebenfalls ferromagnetisch sein, durch magnetische Kräfte ihre Position im Körper verändern und dadurch innere Verletzungen hervorrufen.
- Auf ferromagnetische Gegenstände (z. B. Gegenstände, die Eisen oder Nickel enthalten) im Bereich des Magneten (z. B. Messer, Schraubenzieher, Münzen, Haarspangen, ...) wird eine starke Anziehungskraft ausgeübt. Die Gegenstände werden mit großer Geschwindigkeit in den Magneten gezogen und können Versuchspersonen erheblich verletzen.
- Kleine Metallsplitter im Auge können durch magnetische Kräfte bewegt oder gedreht werden und das Auge verletzen.
- Bei einer Messung mit der Magnetresonanztomographie kommt es zur Abstrahlung von hochfrequenter elektromagnetischer Strahlung, wie sie z. B. bei Radiosendern und Funktelefonen auftritt. Dies kann zu einer geringfügigen, aber nicht spürbaren Erwärmung des untersuchten Gewebes führen.
- Bei großflächigen Tattoos kann es zu starken Erwärmungen kommen.
- Das Schalten der Magnetfeldgradienten erzeugt als unerwünschten Nebeneffekt Lärm, der Schalldruck von über 100 dB(A) erreichen kann. Deshalb müssen Sie bei allen Messungen entweder schallabsorbierende Kopfhörer oder Lärmschutzohrenstöpsel tragen, die von uns zur Verfügung gestellt werden. Bei Einhaltung dieser Vorsichtsmaßnahme kann eine Schädigung des Hörsystems ausgeschlossen werden.

3

Sehr geehrte Probandin, sehr geehrter Proband,

wenn Sie Ihr Einverständnis erklären, nehmen Sie an Untersuchungen teil, bei denen die Methode der Magnetresonanztomographie (MRT) angewandt wird. Im Folgenden erhalten Sie einige Informationen zu derartigen Messungen. Selbstverständlich können Sie sich mit allen Fragen zu diesem Thema jederzeit auch nach Beginn der Untersuchungen an die Mitarbeiter des Labors für Magnetresonanztomographie wenden.

Allgemeine Informationen

Vor Beginn einer Untersuchung werden Sie vom Untersuchungsleiter ausführlich über die für den Tag geplanten Messungen und deren Zielstellung informiert. Sie haben das Recht, ohne Angabe von Gründen die Teilnahme an der Messung abzulehnen. Auch während der gesamten Untersuchung werden Sie vom Untersuchungsleiter jederzeit gehört und können ohne Angabe von Gründen den Abbruch der Untersuchung verlangen.

Die Untersuchungen dürfen erst beginnen, wenn Sie den Probandenfragebogen und die Einverständniserklärung ausgefüllt und unterschrieben haben.

Die bei den Untersuchungen mit Ihnen gewonnenen Daten werden mit Computern weiterverarbeitet und sollen eventuell für wissenschaftliche Veröffentlichungen verwendet werden. Die Verarbeitung und Veröffentlichung erfolgt in anonymisierter Form, damit ist eine Zuordnung zu Ihrer Person nicht möglich.

Für Ihren Weg zur und von der Untersuchung besteht kein Unfallversicherungsschutz.

Ablauf einer Untersuchung

Für die Untersuchungen müssen Sie sich auf eine Liege legen. Bei einigen Messungen wird in der Nähe des zu untersuchenden Körperteils eine Spule angebracht. Auf der Liege werden Sie dann langsam in die Röhre des Magnetresonanztomographen geschoben oder gefahren. Dort befinden Sie sich während der gesamten Untersuchung, die normalerweise 60 bis 90 Minuten dauert, in einem starken Magnetfeld, das für die Untersuchung benötigt wird. Während der eigentlichen Messung werden zusätzliche Hochfrequenzfelder, die Sie weder spüren noch hören können, und sogenannte Magnetfeldgradienten, die sich als klopfendes oder piepsendes Geräusch bemerkbar machen, eingeschaltet. Während der gesamten Untersuchungen sollen Sie versuchen, möglichst ruhig liegen zu bleiben. Bei Untersuchungen mit der funktionellen Magnetresonanztomographie müssen Sie zusätzlich einige Aufgaben erfüllen, die Ihnen zuvor vom Untersuchungsleiter erklärt werden. Auch bei diesen Untersuchungen ist es von großer Bedeutung, dass Sie sich wenig bewegen. Um dies zu erleichtern, wird Ihr Kopf während einer funktionellen Magnetresonanztomographieuntersuchung mit Polstern und anderen Hilfsmitteln schmerzfrei fixiert.

Methode der Magnetresonanztomographie (MRT)

Die MRT ist ein weitverbreitetes Standardverfahren der bildgebenden Diagnostik, das bei Einhaltung der Sicherheitsvorschriften nach heutigem Wissensstand keine schädigenden Nebenwirkungen verursacht. Sie nutzt den Effekt, dass die Atomkerne des Wasserstoffs, aus dem wir zu einem ganz großen Teil bestehen, magnetisch sind. Bildlich kann man sich diese als winzig kleine Kompaßnadeln vorstellen. In einem starken *Magnetfeld* orientieren sich die Atomkerne des Wasserstoffs nun so, wie sich eine Kompaßnadel im Erdmagnetfeld ausrichtet. Mittels geeigneter Antennen, die hier Spulen genannt werden, strahlt man für Bruchteile von Sekunden Radiowellen mit geeigneter Frequenz, der sogenannten *Resonanzfrequenz*, aus, die den Orientierungszustand der Wasserstoffatomkerne stören. Veranschaulicht drehe man mit dem Finger die Kompaßnadel in Ost-West-Richtung. So wie beim Loslassen der Kompaßnadel diese sich wieder in Nord-Süd-

2

Einwilligungserklärung

Name der Probandin/des Probanden _____

Ich bin über Wesen, Bedeutung und Tragweite der geplanten Untersuchungen mit der Magnetresonanztomographie eingehend unterrichtet worden. Dazu lagen mir ein entsprechender Fragebogen zu Kontraindikationen sowie ein Informationsblatt vor. Zu dem Ablauf und den möglichen Risiken konnte ich Fragen stellen; die mir erteilten Informationen habe ich inhaltlich verstanden. Ich willige hiermit in die Teilnahme an den Untersuchungen ein. Mir ist bekannt, dass ich meine Einwilligung jederzeit ohne Angabe von Gründen widerrufen kann.

Ich weiß, dass die bei Untersuchungen mit mir gewonnenen Daten mit Computern weiterverarbeitet und eventuell für wissenschaftliche Veröffentlichungen verwendet werden sollen. Hiermit bin ich einverstanden, wenn die Verarbeitung und Veröffentlichung in einer Form erfolgt, die eine Zuordnung zu meiner Person ausschließt. Auch diese Einwilligung kann ich jederzeit ohne Angabe von Gründen widerrufen.

Ich weiß, dass die Erstellung einer individuellen Diagnostik nicht das Ziel der hier durchgeführten Messungen ist und dass die aufgenommenen Bilder nicht systematisch auf Auffälligkeiten untersucht werden. Sollten dennoch zufällig in den erhobenen Daten Besonderheiten bemerkt werden, bin ich damit einverstanden, dass die Bilder zur Beurteilung an einen Arzt weitergeleitet werden. Erscheint es nach dieser Beurteilung sinnvoll und notwendig, werde ich über den Befund informiert und beraten.

Mir ist bekannt, dass für meine Wege zur und von der Untersuchung kein Unfallversicherungsschutz besteht.

Ort	Datum	Unterschrift der Probandin/des Probanden
-----	-------	--

Ort	Datum	Unterschrift der Mitarbeiterin/des Mitarbeiters, die/der das Informationsgespräch geführt hat ihrt hat
-----	-------	---

4

Figure 7.1: Pages 1–4 of the informed consent and participant questionnaire (in German) for our fMRI studies.

Bibliography

- [Abi+19] Nahla Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan Maletic. "Developer Reading Behavior While Summarizing Java Methods: Size and Context Matters." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2019, pp. 384–395.
- [Ade81] Beth Adelson. "Problem Solving and the Development of Abstract Categories in Programming Languages." In: *Memory & Cognition* 9.4 (1981), pp. 422–433.
- [AF96] Theodore Anderson and Jeremy Finn. *The New Statistical Analysis of Data*. Springer, 1996.
- [Agu11] Geoffrey Aguirre. "Experimental Design and Data Analysis for fMRI." In: *Functional Neuroimaging*. Springer, 2011, pp. 321–330.
- [And13] John Andreassi. *Psychophysiology: Human Behavior & Physiological Response*. Psychology Press, 2013.
- [Aqe+21] Arooba Aqeel, Norman Peitek, Sven Apel, Jonas Mucke, and Janet Siegmund. "Understanding Comprehension of Iterative and Recursive Programs with Remote Eye Tracking." In: *Annual Conf. Psychology of Programming Interest Group (PPIG)*. 2021, pp. 1–17.
- [AWF17] Shulamyt Ajami, Yonatan Woodbridge, and Dror Feitelson. "Syntax, Predicates, Idioms - What Really Affects Code Complexity?" In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. 2017, pp. 66–76.
- [AZD98] Geoffrey Aguirre, Eric Zarahn, and Mark D'esposito. "The Variability of Human, BOLD Hemodynamic Responses." In: *NeuroImage* 8.4 (Nov. 1998), pp. 360–369.
- [BA95] Martin Bland and Douglas Altman. "Multiple Significance Tests: The Bonferroni Method." In: *Bmj* 310.6973 (1995), p. 170.
- [Ban98] Marie Banich. "The Missing Link: The Role of Interhemispheric Interaction in Attentional Processing." In: *Brain and Cognition* 36.2 (1998), pp. 128–157.
- [Bar+17] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. "Do Developers Read Compiler Error Messages?" In: *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2017, pp. 575–585.
- [Bat+15] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. "Fitting Linear Mixed-Effects Models Using lme4." In: *Journal of Statistical Software* 67.1 (2015), pp. 1–48.
- [Bau+19] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes Hofmeister, and Sven Apel. "Indentation: Simply a Matter of Style or Support for Program Comprehension?" In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. ACM, 2019, p. 11.

- [BBT79] Gordon Bower, John Black, and Terrence Turner. "Scripts in Memory for Text." In: *Cognitive Psychology* 11.2 (1979), pp. 177–220.
- [BDA14] Javier Belmonte, Philippe Dugerdil, and Ashish Agrawal. "A Three-layer Model of Source Code Comprehension." In: *Proc. India Software Engineering Conf. ACM*, 2014, pp. 1–10.
- [Bed12] Roman Bednarik. "Expertise-Dependent Visual Attention Strategies Develop over Time during Debugging with Multiple Code Representations." In: *Int'l Journal of Human-Computer Studies* 70.2 (2012), pp. 143–155.
- [Beh+18] Mahnaz Behroozi, Alison Lui, Ian Moore, Denae Ford, and Chris Parnin. "Dazed: Measuring the Cognitive Load of Solving Technical Interview Problems at the Whiteboard." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2018, 4 pages.
- [Beh+20] Mahnaz Behroozi, Shivani Shirolkar, Titus Barik, and Chris Parnin. "Does Stress Impact Technical Interview Performance?" In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. 2020, pp. 481–492.
- [Ben+11] Simone Benedetto, Marco Pedrotti, Luca Minin, Thierry Baccino, Alessandra Re, and Roberto Montanari. "Driver Workload and Eye Blink Duration." In: *Transportation Research Part F: Traffic Psychology and Behaviour* 14.3 (2011), pp. 199–208.
- [Ber+07] Chris Berka, Daniel Levendowski, Michelle Lumicao, Alan Yau, Gene Davis, Vladimir Zivkovic, Richard Olmstead, Patrice Tremoulet, and Patrick Craven. "Eeg Correlates of Task Engagement and Mental Workload in Vigilance, Learning, and Memory Tasks." In: *Aviation, Space, and Environmental Medicine* 78.5 (2007), B231–B244.
- [Ber29] Hans Berger. "Über das Elektroenkephalogramm des Menschen." In: *Archiv für Psychiatrie und Nervenkrankheiten* 87.1 (1929), pp. 527–570.
- [BF10] Dirk Beyer and Ashgan Fararooy. "A Simple and Effective Measure for Complex Low-Level Dependencies." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 2010, pp. 80–83.
- [BG90] RKE Bellamy and DJ Gilmore. "Programming Plans: Internal or External Structures." In: *Lines of Thinking: Reflections on the Psychology of Thought* 2 (1990), pp. 59–72.
- [BH95] Yoav Benjamini and Yosef Hochberg. "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing." In: *Journal of the Royal Statistical Society. Series B (Methodological)* 57.1 (1995), pp. 289–300.
- [Bil03] Francis Billson. *Strabismus*. BMJ Books, 2003.
- [Bil+11] Merim Bilalić, Robert Langner, Rolf Ulrich, and Wolfgang Grodd. "Many Faces of Expertise: Fusiform Face Area in Chess Experts and Novices." In: *The Journal of Neuroscience* 31.28 (July 2011), pp. 10206–10214.
- [Bin+09a] Jeffrey Binder, Rutvik Desai, William Graves, and Lisa Conant. "Where Is the Semantic System? A Critical Review and Meta-Analysis of 120 Functional Neuroimaging Studies." In: *Cerebral Cortex* 19.12 (2009), pp. 2767–2796.

-
- [Bin+09b] David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. "To CamelCase or Under_score." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 2009, pp. 158–167.
- [Bin+13] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan Maletic, Christopher Morrell, and Bonita Sharif. "The Impact of Identifier Style on Effort and Comprehension." In: *Empirical Software Engineering* 18.2 (2013), pp. 219–276.
- [Bin+99] Jeffrey Binder, Julia Frost, Thomas Hammeke, Patrick Bellgowan, Stephen Rao, and Robert Cox. "Conceptual Processing During the Conscious Resting State: A Functional MRI Study." In: *Journal of cognitive neuroscience* 11.1 (1999), pp. 80–93.
- [BK66] Jackson Beatty and Daniel Kahneman. "Pupillary Changes in Two Memory Tasks." In: *Psychonomic Science* 5.10 (1966), pp. 371–372.
- [BLW00] Jackson Beatty and Brennis Lucero-Wagoner. *The Pupillary System, Handbook of Psychophysiology, Cacioppo, Tassinary & Berntson*. 2000.
- [BMW09] Craig Bennett, Michael Miller, and George Wolford. "Neural Correlates of Interspecies Perspective Taking in the Post-Mortem Atlantic Salmon: An argument for Multiple Comparisons Correction." In: *Neuroimage* 47.Suppl 1 (2009), S125.
- [Bou12] Wolfram Boucsein. *Electrodermal Activity*. Springer Science & Business Media, 2012.
- [BP16] Tanya Beelders and Jean-Pierre du Plessis. "The Influence of Syntax Highlighting on Scanning and Reading Behaviour for Source Code." In: *Proc. Annual Conf. of the South African Institute of Computer Scientists and Information Technologists (SAICSIT)*. ACM, 2016, pp. 1–10.
- [BPB19] Mahnaz Behroozi, Chris Parnin, and Titus Barik. "Hiring Is Broken: What Do Developers Say About Technical Interviews?" In: *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2019, pp. 1–9.
- [Bre+01] Matthew Brett, Kalina Christoff, Rhodri Cusack, Jack Lancaster, et al. "Using the Talairach Atlas with the MNI Template." In: *Neuroimage* 13.6 (2001), pp. 85–85.
- [Bri+13] Julie Brisson, Marc Mainville, Dominique Mailloux, Christelle Beaulieu, Josette Serres, and Sylvain Sirois. "Pupil Diameter Measurement Errors as a Function of Gaze Direction in Corneal Reflection Eyetrackers." In: *Behavior Research Methods* 45.4 (2013), pp. 1322–1331.
- [Bro06] Korbinian Brodmann. *Brodmann's Localisation in the Cerebral Cortex*. Springer, 2006.
- [Bro09] Korbinian Brodmann. *Vergleichende Lokalisationslehre der Großhirnrinde in ihren Prinzipien dargestellt auf Grund des Zellbaues*. Leipzig: Barth. 1909.
- [Bro78] Ruven Brooks. "Using a Behavioral Theory of Program Comprehension in Software Engineering." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 1978, pp. 196–201.

- [Bro83] Ruven Brooks. "Towards a Theory of the Comprehension of Computer Programs." In: *Int'l Journal of Man-Machine Studies* 18.6 (1983), pp. 543–554.
- [BRW94] Richard Backs, Arthur Ryan, and Glenn Wilson. "Psychophysiological Measures of Workload During Continuous Manual Performance." In: *Human Factors* 36.3 (1994), pp. 514–531.
- [BS19] Tanja Blascheck and Bonita Sharif. "Visually Analyzing Eye Movements on Natural Language Texts and Source Code Snippets." In: *Proc. Symposium on Eye Tracking Research & Applications*. ETRA. ACM, 2019, 14:1–14:9.
- [BSAL10] Rolf Brickenkamp, Lothar Schmidt-Atzert, and Detlev Liepmann. *Test d2-Revision: Aufmerksamkeits-und Konzentrationstest*. Hogrefe Göttingen, 2010.
- [BSP15] Omar Benomar, Houari Sahraoui, and Pierre Poulin. "A Unified Framework for the Comprehension of Software's Time Dimension." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2015, pp. 603–606.
- [BSW11] Raymond Buse, Caitlin Sadowski, and Westley Weimer. "Benefits and Barriers of User Evaluation in Software Engineering Research." In: *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2011, pp. 643–656.
- [BT06] Roman Bednarik and Markku Tukiainen. "An Eye-Tracking Methodology for Characterizing Program Comprehension Processes." In: *Proc. Symposium on Eye Tracking Research & Applications*. ETRA. ACM, 2006, pp. 125–132.
- [Bus+14] Teresa Busjahn, Carsten Schulte, Bonita Sharif, Andrew Begel, Michael Hansen, Roman Bednarik, Paul Orlov, Petri Ihantola, Galina Shchekotova, and Maria Antropova. "Eye Tracking in Computing Education." In: *Proc. Conf. on International Computing Education Research*. ACM, 2014, pp. 3–10.
- [Bus+15] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. "Eye Movements in Code Reading: Relaxing the Linear Order." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, May 2015, pp. 255–265.
- [Bus+18] Sara Busechian, Vladimir Ivanov, Alan Rogers, Ilyas Sirazitdinov, Giancarlo Succi, Alexander Tormasov, and Jooyong Yi. "Understanding the impact of pair programming on the minds of developers." In: *Proc. Int'l Conf. Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE. 2018, pp. 85–88.
- [BW08] Raymond Buse and Westley Weimer. "A Metric for Software Readability." In: *Proc. Int'l Symposium on Software Testing and Analysis*. ACM, 2008, pp. 121–130.
- [BW10] Raymond Buse and Westley Weimer. "Learning a Metric for Code Readability." In: *IEEE Trans. Softw. Eng.* 36.4 (2010), pp. 546–558.
- [BWW20] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. "An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability." In: *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*. 2020, p. 12.

-
- [Cam18] Ann Campbell. *Cognitive Complexity: A New Way of Measuring Understandability*. Tech. rep. SonarSource S.A., 2018, p. 21.
- [Cas+19] Joao Castelhana, Isabel Duarte, Carlos Ferreira, Joao Duraes, Henrique Madeira, and Miguel Castelo-Branco. "The Role of the Insula in Intuitive Expert Bug Detection in Computer Code: An fMRI Study." In: *Brain Imaging and Behavior* 13.3 (2019), pp. 623–637.
- [CC83] Jacob Cohen and Patricia Cohen. *Applied Multiple Regression: Correlation Analysis for the Behavioral Sciences*. Addison Wesley, 1983.
- [Cha+93] B. Chance, Z. Zhuang, C. UnAh, C. Alter, and Lipton L. "Cognition-Activated Low-Frequency Modulation of Light Absorption in Human Brain." In: *Proc. Nat'l Academy Sciences of the United States of America (PNAS)* 90.8 (1993), pp. 3770–3774.
- [Che+11] Siyuan Chen, Julien Epps, Natalie Ruiz, and Fang Chen. "Eye Activity as a Measure of Human Mental Effort in HCI." In: *Proc. Int'l Conf. Intelligent User Interfaces*. ACM. 2011, pp. 315–318.
- [CJ91] Colin Camerer and Eric Johnson. In: Cambridge University Press, 1991. Chap. The Process-Performance Paradox in Expert Judgment: How Can Experts Know So Much and Predict So Badly? Pp. 195–217.
- [CK10] Sylvain Charron and Etienne Koechlin. "Divided Representation of Concurrent Goals in the Human Frontal Lobes." In: *Science* 328.5976 (2010), pp. 360–363.
- [CK14] Igor Crk and Timothy Kluthe. "Toward Using Alpha and Theta Brain Waves to Quantify Programmer Expertise." In: *Int'l Conf. Engineering in Medicine and Biology Society*. IEEE. 2014, pp. 5373–5376.
- [CK94] Shyam Chidamber and Chris Kemerer. "A Metrics Suite for Object Oriented Design." In: *IEEE Trans. Softw. Eng.* 20.6 (1994), pp. 476–493.
- [CKM96] Neil Charness, Ralf Krampe, and Ulrich Mayr. In: Lawrence Erlbaum Associates, Inc, 1996. Chap. The Role of Practice and Coaching in Entrepreneurial Skill Domains: An International Comparison of Life-Span Chess Skill Acquisition, pp. 51–80.
- [CKS15] Igor Crk, Timothy Kluthe, and Andreas Stefik. "Understanding Programming Expertise: An Empirical Study of Phasic Brain Wave Changes." In: *ACM Trans. Comput.-Hum. Interact.* 23.1 (2015), 2:1–2:29.
- [Coh13] Andrew Cohen. "Software for the Automatic Correction of Recorded Eye Fixation Locations in Reading Experiments." In: *Behavior Research Methods* 45.3 (2013), pp. 679–683.
- [Coh69] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic Press, 1969.
- [Col89] Michael Coles. "Modern Mind-Brain Reading: Psychophysiology, Physiology, and Cognition." In: *Psychophysiology* 26.3 (1989), pp. 251–269.
- [Col+94] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. "Using Metrics to Evaluate Software System Maintainability." In: *Computer* 27.8 (1994), pp. 44–49.

- [Cou+19a] Ricardo Couceiro, Raul Barbosa, João Durães, Gonçalo Duarte, João Castelhan, Catarina Duarte, Cesar Teixeira, Nuno Laranjeiro, Júlio Medeiros, Paulo Carvalho, et al. "Spotting Problematic Code Lines using Nonintrusive Programmers' Biofeedback." In: *Proc. Int'l Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2019, pp. 93–103.
- [Cou+19b] Ricardo Couceiro, Gonçalo Duarte, João Durães, João Castelhan, Catarina Duarte, César Teixeira, Miguel Castelo Branco, Paulo de Carvalho, and Henrique Madeira. "Biofeedback Augmented Software Engineering: Monitoring of Programmers' Mental Effort." In: *Proc. Int'l Conf. on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '19. IEEE, 2019, pp. 37–40.
- [Cri+10] Filipe Cristino, Sebastiaan Mathôt, Jan Theeuwes, and Iain Gilchrist. "ScanMatch: A Novel Method for Comparing Fixation Sequences." In: *Behavior Research Methods* 42.3 (2010), pp. 692–700.
- [CS90] Martha Crosby and Jan Stelovsky. "How Do We Read Algorithms? A Case Study." In: *Computer* 23.1 (1990), pp. 25–35.
- [Cur+79] Bill Curtis, Sylvia Sheppard, Phil Milliman, M.A. Borst, and Tom Love. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics." In: *IEEE Trans. Softw. Eng.* SE-5.2 (1979), pp. 96–104.
- [CZ76] Alfonso Caramazza and Edgar B. Zurif. "Dissociation of Algorithmic and Heuristic Processes in Language Comprehension: Evidence from Aphasia." In: *Brain and Language* 3.4 (1976), pp. 572–582.
- [DBC06] Barbara DiCicco-Bloom and Benjamin Crabtree. "The Qualitative Research Interview." In: *Medical education* 40.4 (2006), pp. 314–321.
- [DD86] Hubert Dreyfus and Stuart Dreyfus. "Mind over Machine: The Power of Human Intuition and Expertise in the Era of the Computer." In: *The Free Press* (1986).
- [Des+06] Rahul Desikan, Florent Ségonne, Bruce Fischl, Brian Quinn, Bradford Dickerson, Deborah Blacker, Randy Buckner, Anders Dale, R Paul Maguire, Bradley Hyman, Marilyn Albert, and Ronald Killiany. "An Automated Labeling System for Subdividing the Human Cerebral Cortex on MRI Scans into Gyral Based Regions of Interest." In: *Neuroimage* 31.3 (July 2006), pp. 968–80.
- [Des+10] Christophe Destrieux, Bruce Fischl, Anders Dale, and Eric Halgren. "Automatic Parcellation of Human Cortical Gyri and Sulci Using Standard Anatomical Nomenclature." In: *NeuroImage* 53.1 (2010), pp. 1–15.
- [DG78] Adriaan De Groot. *Thought and Choice in Chess*. Vol. 4. Walter de Gruyter GmbH & Co KG, 1978.
- [Dou01] Michael Doughty. "Consideration of Three Types of Spontaneous Eyeblink Activity in Normal Humans: During Reading and Video Display Terminal Use, in Primary Gaze, and while in Conversation." In: *Optometry and Vision Science* 78.10 (2001), pp. 712–725.

-
- [DR00] Alastair Dunsmore and Marc Roper. *A Comparative Evaluation of Program Comprehension Measures*. Tech. rep. EFOCS 35-2000. Department of Computer Science, University of Strathclyde, 2000.
- [Dro+04] Nina Dronkers, David Wilkins, Robert Van Valin, Brenda Redfern, and Jeri Jaeger. "Lesion Analysis of the Brain Areas Involved in Language Comprehension." In: *Cognition* 92.1-2 (2004), pp. 145-177.
- [DSL18] Rodrigo Duran, Juha Sorva, and Sofia Leite. "Towards an Analysis of Program Complexity From a Cognitive Perspective." In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 2018, pp. 21-30.
- [Duc17] Andrew Duchowski. *Eye Tracking Methodology – Theory and Practice, Third Edition*. Springer, 2017.
- [Dur+16] João Duraes, Henrique Madeira, João Castelhana, Isabel C. Duarte, and Miguel Castelo-Branco. "WAP: Understanding the Brain at Software Debugging." In: *Proc. Int. Symposium Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 87-92.
- [Eck+17] Maria Eckstein, Belén Guerra-Carrillo, Alison Miller Singley, and Silvia Bunge. "Beyond Eye Gaze: What Else Can Eyetracking Reveal about Cognition and Cognitive Development?" In: *Developmental Cognitive Neuroscience* 25 (2017), pp. 69-91.
- [EL96] Anders Ericsson and Andreas Lehmann. "Expert and Exceptional Performance: Evidence of Maximal Adaptation to Task Constraints." In: *Annual Review of Psychology* 47.1 (1996), pp. 273-305.
- [End+21] Madeline Endres, Zachary Karas, Xiaosu Hu, Ioulia Kovelman, and Westley Weimer. "Relating Reading, Visualization, and Coding for New Programmers: A Neuroimaging Study." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE. 2021, pp. 600-612.
- [ERK08] Wolfgang Einhäuser, Ueli Rutishauser, and Christof Koch. "Task-Demands Can Immediately Reverse the Effects of Sensory-Driven Saliency in Complex Visual Stimuli." In: *Journal of Vision* 8 (2008), pp. 2-2.
- [ES84] Anders Ericsson and Herbert Simon. *Protocol Analysis: Verbal Reports as Data*. the MIT Press, 1984.
- [ESS90] Arthur Elstein, Lee Shulman, and Sarah Sprafka. "Medical Problem Solving: A Ten-Year Retrospective." In: *Evaluation & the Health Professions* 13.1 (1990), pp. 5-36.
- [Eva+12] Alan Evans, Andrew Janke, Louis Collins, and Sylvain Baillet. "Brain Templates and Atlases." In: *Neuroimage* 62.2 (2012), pp. 911-922.
- [Eva+93] Alan Evans, Louis Collins, SR Mills, Edward Brown, Ryan Kelly, and Terry Peters. "3D Statistical Neuroanatomical Models from 305 MRI Volumes." In: *Conf. Record Nuclear Science Symposium and Medical Imaging Conference*. IEEE. 1993, pp. 1813-1817.
- [Ext02] Christopher Exton. "Constructivism and Program Comprehension Strategies." In: *Proc. Int'l Workshop Program Comprehension (IWPC)*. IEEE, 2002, pp. 281-284.

- [Fak+18] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. "The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 2018, 11 pages.
- [Fak+20] Sarah Fakhoury, Devjeet Roy, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. "Measuring the Impact of Lexical and Structural Inconsistencies on Developers? Cognitive Load during Bug Localization." In: *Empirical Software Engineering* (25 May 2020), pp. 2140–2178.
- [FBP15] Denae Ford, Titus Barik, and Chris Parnin. "Studying Sustained Attention and Cognitive States with Eye Tracking in Remote Technical Interviews." In: *Eye Movements in Programming: Models to Data* (2015), 5 pages.
- [FF20] Fabian Fagerholm and Thomas Fritz. "Biometric Measurement in Software Engineering." In: *Contemporary Empirical Methods in Software Engineering*. Springer, 2020, pp. 151–172.
- [FF94] Andrea Fontana and James Frey. "Interviewing: The Art of Science." In: *The Handbook of Qualitative Research* 361376 (1994).
- [Fis+02] Bruce Fischl, David H Salat, Evelina Busa, Marilyn Albert, Megan Dieterich, Christian Haselgrove, Andre Van Der Kouwe, Ron Killiany, David Kennedy, Shuna Klaveness, et al. "Whole Brain Segmentation: Automated Labeling of Neuroanatomical Structures in the Human Brain." In: *Neuron* 33.3 (2002), pp. 341–355.
- [Fis+04] Bruce Fischl, André Van Der Kouwe, Christophe Destrieux, Eric Halgren, Florent Ségonne, David H Salat, Evelina Busa, Larry J Seidman, Jill Goldstein, David Kennedy, et al. "Automatically Parcellating the Human Cerebral Cortex." In: *Cerebral Cortex* 14.1 (2004), pp. 11–22.
- [FP14] Norman Fenton and Shari Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. 3rd. CRC Press, 2014.
- [Fri+14] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. "Using Psycho-Physiological Measures to Assess Task Difficulty in Software Development." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2014, pp. 402–413.
- [FSF01] Christian Fiebach, Matthias Schlesewsky, and Angela Friederici. "Syntactic Working Memory and the Establishment of Filler-Gap Dependencies: Insights from ERPs and fMRI." In: *Journal of Psycholinguistic Research* 30.3 (2001), pp. 321–338.
- [FSW17] Benjamin Floyd, Tyler Santander, and Westley Weimer. "Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2017, pp. 175–186.
- [Fuc+19] Davide Fucci, Daniela Girardi, Nicole Novielli, Luigi Quaranta, and Filippo Lanubile. "A Replication Study on Code Comprehension and Expertise Using Lightweight Biometric Sensors." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE. 2019, pp. 311–322.

-
- [FVT05] Stephen Fairclough, Louise Venables, and Andrew Tattersall. "The Influence of Task Demand and Learning on the Psychophysiological Response." In: *Int'l Journal of Psychophysiology* 56.2 (2005), pp. 171–184.
- [FW19] John Fox and Sanford Weisberg. *An R Companion to Applied Regression*. Third. Thousand Oaks CA: Sage, 2019.
- [Gau+00] Isabel Gauthier, Pawel Skudlarski, John Gore, and Adam Anderson. "Expertise for Cars and Birds Recruits Brain Areas Involved in Face Recognition." In: *Nature Neuroscience* 3.2 (Feb. 2000), pp. 191–197.
- [GG88] David Gilmore and Thomas Green. "Programming Plans and Programming Expertise." In: *The Quarterly Journal of Experimental Psychology* 40.3 (1988), pp. 423–442.
- [GIM13] Michael Gazzaniga, Richard Ivry, and George Mangun. *Cognitive Neuroscience: The Biology of the Mind*. Norton & Company, 2013.
- [Gin05] David Ginat. "The Suitable Way is Backwards, but They Work Forward." In: *Journal of Computers in Mathematics and Science Teaching* 24.1 (2005), pp. 73–88.
- [GL17] Yossi Gil and Gal Lalouche. "On the Correlation between Size and Metric Validity." In: *Empirical Softw. Eng.* 22.5 (2017), pp. 2585–2611.
- [GLN17] Daniela Girardi, Filippo Lanubile, and Nicole Novielli. "Emotion Detection Using Noninvasive Low Cost Sensors." In: *Proc. Int'l Conf. on Affective Computing and Intelligent Interaction (ACII)*. IEEE. 2017, pp. 125–130.
- [GM16] Yorai Geffen and Shahar Maoz. "On Method Ordering." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE. 2016, pp. 1–10.
- [GMGVEB16] Javier García-Munoz, Marisol García-Valls, and Julio Escribano-Barreno. "Improved Metrics Handling in SonarQube for Software Quality Monitoring." In: *Proc. Int'l Conf. Distributed Computing and Artificial Intelligence*. Springer. 2016, pp. 463–470.
- [Gol02] Bruce Goldstein. *Sensation and Perception*. Cengage Learning Services, 2002.
- [Gop+17] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin Yeh, and Justin Cappos. "Understanding Misunderstandings in Source Code." In: *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. ACM, 2017, pp. 129–139.
- [Gop+20] Dan Gopstein, Anne-Laure Fayard, Sven Apel, and Justin Cappos. "Thinking Aloud about Confusing Code: A Qualitative Investigation of Program Comprehension and Atoms of Confusion." In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, 605–616.
- [Gor+11] Krzysztof Gorgolewski, Christopher Burns, Cindee Madison, Dav Clark, Yaroslav Halchenko, Michael Waskom, and Satrajit Ghosh. "Nipype: A Flexible, Lightweight and Extensible Neuroimaging Data Processing Framework in Python." In: *Frontiers in Neuroinformatics* 5 (2011), 13 pages.

- [Gra+16] Mariel Grassmann, Elke Vlemincx, Andreas Von Leupoldt, Justin Mittelstädt, and Omer Van den Bergh. "Respiratory Changes in Response to Cognitive Load: A Systematic Review." In: *Neural Plasticity* (2016).
- [Gua+18] Drew Guarnera, Corey Bryant, Ashwin Mishra, Jonathan Maletic, and Bonita Sharif. "itrace: Eye Tracking Infrastructure for Development Environments." In: *Proc. Symposium on Eye Tracking Research & Applications (ETRA)*. 2018, pp. 1–3.
- [Hag05] Peter Hagoort. "On Broca, Brain, and Binding: A New Framework." In: *Trends in Cognitive Sciences* 9.9 (2005), pp. 416–423.
- [Hal77] Maurice Halstead. *Elements of Software Science*. Elsevier Science Inc., 1977.
- [Han+15] Thomas Hannagan, Amir Amedi, Laurent Cohen, Ghislaine Dehaene-Lambertz, and Stanislas Dehaene. "Origins of the Specialization for Letters and Numbers in Ventral Occipitotemporal Cortex." In: *Trends in cognitive sciences* 19.7 (2015), pp. 374–382.
- [Han+16] Michael Hanke, Nico Adelhöfer, Daniel Kottke, Vittorio Iacovella, Ayan Sengupta, Falko Kaule, Roland Nigbur, Alexander Waite, Florian Baumgartner, and Jörg Stadler. "A Studyforrest Extension, Simultaneous fMRI and Eye Gaze Recordings During Prolonged Natural Stimulation." In: *Scientific Data* 3 (2016).
- [Han+20] Michael Hanke, Sebastiaan Mathôt, Eduard Ort, Norman Peitek, Jörg Stadler, and Adina Wagner. "A Practical Guide to Functional Magnetic Resonance Imaging with Simultaneous Eye Tracking for Cognitive Neuroimaging Research." In: *Spatial Learning and Attention Guidance*. Ed. by Stefan Pollmann. Springer US, 2020, pp. 291–305.
- [Har+09a] Erin Harley, Whitney Pope, Pablo Villablanca, Jeanette Mumford, Robert Suh, John Mazziotta, Dieter Enzmann, and Stephen Engel. "Engagement of Fusiform Cortex and Disengagement of Lateral Occipital Cortex in the Acquisition of Radiological Expertise." In: *Cereb Cortex* 19.11 (Nov. 2009), pp. 2746–2754.
- [Har+09b] Valentina Hartwig, Giulio Giovannetti, Nicola Vanello, Massimo Lombardi, Luigi Landini, and Silvana Simi. "Biological Effects and Safety in Magnetic Resonance Imaging: A Review." In: *International Journal of Environmental Research and Public Health* 6.6 (2009), pp. 1778–1798.
- [HD17] Eric Harth and Philippe Dugerdil. "Program Understanding Models: An Historical Overview and a Classification." In: *Proc. Int'l Conf. on Software Technologies*. INSTICC. SciTePress, 2017, pp. 402–413.
- [HF14] Matthias Hartmann and Martin Fischer. "Pupillometry: The Eyes Shed Fresh Light on the Mind." In: *Current Biology* 24.7 (2014), R281–R282.
- [HH02] Anthony Hornof and Tim Halverson. "Cleaning Up Systematic Error in Eye-Tracking Data by Using Required Fixation Locations." In: *Behavior Research Methods, Instruments, & Computers* 34.4 (2002), pp. 592–604.

-
- [HH17] Elham Hosnieh and Hirohide Haga. "A Novel Approach to Program Comprehension Process Using Slicing Techniques." In: *Journal of Computers* 11.5 (2017), pp. 353–364.
- [HJP16] Emily Hill, Philip M Johnson, and Daniel Port. "Is an Athletic Approach the Future of Software Engineering Education?" In: *Software* 33.1 (Jan. 2016), pp. 97–100.
- [HM95] Martin Hitz and Behzad Montazeri. "Measuring Coupling and Cohesion in Object-Oriented Systems." In: *Proc. Int'l Symp. Applied Corporate Computing*. 1995, pp. 299–300.
- [Hol+11] Kenneth Holmqvist, Marcus Nyström, Richard Andersson, Richard Dewhurst, Halzka Jarodzka, and Joost Van de Weijer. *Eye Tracking: A Comprehensive Guide to Methods and Measures*. OUP Oxford, 2011.
- [Hol92] Dennis Holding. "Theories of Chess Skill." In: *Psychological Research* 54.1 (1992), pp. 10–16.
- [HP64] Eckhard Hess and James Polt. "Pupil Size in Relation to Mental Activity during Simple Problem-Solving." In: *Science* 143.3611 (1964), pp. 1190–1192.
- [HS95] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1995.
- [HSM14] Scott Huettel, Allen Song, and Gregory McCarthy. *Functional Magnetic Resonance Imaging*. Vol. 3. Sinauer Associates, 2014.
- [Hua+19] Yu Huang, Xinyu Liu, Ryan Krueger, Tyler Santander, Xiaosu Hu, Kevin Leach, and Westley Weimer. "Distilling Neural Representations of Data Structure Manipulation using fMRI and fNIRS." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2019, pp. 396–407.
- [Hua+20] Yu Huang, Kevin Leach, Zohreh Sharafi, Nicholas McKay, Tyler Santander, and Westley Weimer. "Biases and differences in code review using medical imaging and eye-tracking: genders, humans, and machines." In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. 2020, pp. 456–468.
- [Hud+17] Shamsul Huda, Sultan Alyahya, Md Mohsin Ali, Shafiq Ahmad, Jemal Abawajy, Hmood Al-Dossari, and John Yearwood. "A Framework for Software Defect Prediction and Metric Selection." In: *IEEE Access* 6 (2017), pp. 2844–2858.
- [IC05] Curtis Ikehara and Martha Crosby. "Assessing Cognitive Load with Physiological Sensors." In: *Proc. Hawaii Int'l Conf. on System Sciences*. IEEE. 2005, 295a–295a.
- [Ikr+19] Rustam Ikramov, Vladimir Ivanov, Sergey Masyagin, Ruslan Shakirov, Ilyas Sirazidtinov, Giancarlo Succi, Ananga Thapaliya, Alexander Tormasov, and Oydinoy Zufarova. "Initial Evaluation of the Brain Activity under Different Software Development Situations." In: *Proc. Int'l Conf, on Software Engineering and Knowledge Engineering (SEKE)*. 2019, pp. 741–777.

- [Iku+21] Yoshiharu Ikutani, Takatomi Kubo, Satoshi Nishida, Hideaki Hata, Kenichi Matsumoto, Kazushi Ikeda, and Shinji Nishimoto. "Expert Programmers have Fine-Tuned Cortical Representations of Source Code." In: *Eneuro* 8.1 (2021), pp. 1–16.
- [Ioa+] Constantina Ioannou, Per Bækgaard, Ekkart Kindler, and Barbara Weber. "Towards a Tool for Visualizing Pupil Dilation Linked with Source Code Artifacts." In: *Conference on Software Visualization (VISSOFT)*. IEEE, pp. 105–109.
- [IU14] Yoshiharu Ikutani and Hidetake Uwano. "Brain Activity Measurement during Program Comprehension with NIRS." In: *I.Í Conf. Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE, 2014, pp. 1–6.
- [IU19] Toyomi Ishida and Hidetake Uwano. "Synchronized Analysis of Eye Movement and EEG during Program Comprehension." In: *Int'l Workshop on Eye Movements in Programming (EMIP)*. IEEE. 2019, pp. 26–32.
- [Iva+20] Anna Ivanova, Shashank Srikant, Yotaro Sueoka, Hope Kean, Riva Dhamala, Una-May O'reilly, Marina Bers, and Evelina Fedorenko. "Comprehension of Computer Code Relies Primarily on Domain-General Executive Resources." In: *BioRxiv* (2020).
- [JF17] Ahmad Jbara and Dror Feitelson. "How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking." In: *Empirical Softw. Eng.* 22.3 (2017), pp. 1440–1477.
- [JH19] Phillip Johnston and Rozi Harris. "The Boeing 737 MAX Saga: Lessons for Software Organizations." In: *Software Quality Professional* 21.3 (2019), pp. 4–12.
- [JS85] Lewis Johnson and Elliot Soloway. "PROUST: Knowledge-Based Program Understanding." In: *IEEE Trans. Softw. Eng.* 3 (1985), pp. 267–275.
- [Kar+21] Zachary Karas, Andrew Jahn, Westley Weimer, and Yu Huang. "Connecting the Dots: Rethinking the Relationship between Code and Prose Writing with Functional Connectivity." In: (2021).
- [KB04] Cem Kaner and Walter P. Bond. "Software Engineering Metrics: What Do They Measure and How Do We Know?" In: *Proc. Int'l Software Metrics Symposium (METRICS)*. IEEE, 2004, p. 12.
- [Kle+09] Arno Klein, Jesper Andersson, Babak Ardekani, John Ashburner, Brian Avants, Ming-Chang Chiang, Gary Christensen, Louis Collins, James Gee, and Pierre Hellier. "Evaluation of 14 Nonlinear Deformation Algorithms Applied to Human Brain MRI Registration." In: *Neuroimage* 46.3 (2009), pp. 786–802.
- [Kli99] Wolfgang Klimesch. "EEG Alpha and Theta Oscillations Reflect Cognitive and Memory Performance: A Review and Analysis." In: *Brain Research Reviews* 29.2-3 (1999), pp. 169–195.

-
- [Kos+18] Makrina Kosti, Kostas Georgiadis, Dimitrios Adamos, Nikos Laskaris, Diomidis Spinellis, and Lefteris Angelis. "Towards an Affordable Brain Computer Interface for the Assessment of Programmers' Mental Workload." In: *Int. J. Human-Computer Studies* 115 (2018), pp. 52–66.
- [KR91] Jürgen Koenemann and Scott Robertson. "Expert Problem Solving Strategies for Program Comprehension." In: *Proc. Conf. Human Factors in Computing Systems (CHI)*. ACM, 1991, pp. 125–130.
- [Kre+16] Krzysztof Krejtz, Andrew Duchowski, Izabela Krejtz, Agnieszka Szarkowska, and Agata Kopacz. "Discerning Ambient/Focal Attention with Coefficient K." In: *Transactions on Applied Perception (TAP)* 13.3 (2016), p. 11.
- [Kru+20] Ryan Krueger, Yu Huang, Xinyu Liu, Tyler Santander, Westley Weimer, and Kevin Leach. "Neurological Divide: An fMRI Study of Prose and Code Writing." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2020, pp. 678–690.
- [KS02] Edith Kaan and Tamara Swaab. "The Brain Circuitry of Syntactic Comprehension." In: *Trends in Cognitive Sciences* 6.8 (2002), pp. 350–356.
- [KSS19] Mariska Kret and Elio Sjak-Shie. "Preprocessing Pupil Size Data: Guidelines and Code." In: *Behavior Research Methods* 51.3 (2019), pp. 1336–1342.
- [Kul+13] Eugenia Kulakova, Markus Aichhorn, Matthias Schurz, Martin Kronbichler, and Josef Perner. "Processing Counterfactual and Hypothetical Conditionals: An fMRI Investigation." In: *NeuroImage* 72 (2013), pp. 265–271.
- [LB88] Mary Lindstrom and Douglas Bates. "Newton-Raphson and EM Algorithms for Linear Mixed-Effects Models for Repeated-Measures Data." In: *Journal of the American Statistical Association* 83.404 (1988), pp. 1014–1022.
- [Lee+16] Seolhwa Lee, Andrew Matteson, Danial Hooshyar, SongHyun Kim, JaeBum Jung, GiChun Nam, and Heuseok Lim. "Comparing Programming Language Comprehension between Novice and Expert Programmers Using EEG Analysis." In: *Int'l Conf. on Bioinformatics and Bioengineering (BIBE)*. IEEE, 2016, pp. 350–355.
- [Lee+17] Seolhwa Lee, Danial Hooshyar, Hyesung Ji, Kichun Nam, and Heuseok Lim. "Mining Biometric Data to Predict Programmer Expertise and Task Difficulty." In: *Cluster Computing* (2017), pp. 1–11.
- [Les+08] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings." In: *IEEE Trans. Softw. Eng.* 34.4 (2008), pp. 485–496.
- [Let87] Stanley Letovsky. "Cognitive Processes in Program Comprehension." In: *Journal of Systems and Software* 7.4 (1987), pp. 325–339.
- [Lik32] Rensis Likert. "A Technique for the Measurement of Attitudes." In: *Archives of Psychology* 22.140 (1932), pp. 1–55.
- [Lit+87] David Littman, Jeannine Pinto, Stan Letovsky, and Elliot Soloway. "Mental Models and Software Maintenance." In: *Journal of Systems and Software* 7.4 (1987), pp. 341–355.

- [Liu+12] JiMei Liu, Meng Zhang, Jerwen Jou, Xin Wu, Wei Li, and Jiang Qiu. "Neural Bases of Falsification in Conditional Proposition Testing: Evidence from an fMRI Study." In: *Int'l Journal of Psychophysiology* 85.2 (2012), pp. 249–256.
- [Liu+20] Yun-Fei Liu, Judy Kim, Colin Wilson, and Marina Bedny. "Computer Code Comprehension Shares Neural Resources with Formal Logical Inference in the Fronto-Parietal Network." In: *Elife* 9 (2020), e59340.
- [LM10] Thomas LaToza and Brad Myers. "Developers Ask Reachability Questions." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2010, pp. 185–194.
- [LMW79] Richard Linger, Harlan Mills, and Bernard Witt. "Structured Programming: Theory and Practice." In: (1979).
- [LSG12] Bruno Laeng, Sylvain Sirois, and Gustaf Gredebäck. "Pupillometry: A Window to the Preconscious?" In: *Perspectives on Psychological Science* 7.1 (2012), pp. 18–27.
- [LST78] Bennet Lientz, Burton Swanson, and Gail Tompkins. "Characteristics of Application Software Maintenance." In: *Communications of the ACM* 21.6 (1978), pp. 466–471.
- [LT93] Nancy Leveson and Clark Turner. "An Investigation of the Therac-25 Accidents." In: *Computer* 26.7 (1993), pp. 18–41.
- [LVD06] Thomas LaToza, Gina Venolia, and Robert DeLine. "Maintaining Mental Models: A Study of Developer Work Habits." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2006, pp. 492–501.
- [Mak+06] Nikos Makris, Jill Goldstein, David Kennedy, Steven Hodge, Verne Caviness, Stephen Faraone, Ming Tsuang, and Larry Seidman. "Decreased Volume of Left and Total Anterior Insular Lobule in Schizophrenia." In: *Schizophr. Res.* 83.2-3 (Apr. 2006), pp. 155–71.
- [Mal+15] Johannes Mallow, Johannes Bernarding, Michael Luchtman, Anja Bethmann, and André Brechmann. "Superior Memorizers Employ Different Neural Networks for Encoding and Recall." In: *Frontiers in Systems Neuroscience* 9.128 (2015).
- [MB07] Jarrod Millman and Matthew Brett. "Analysis of Functional Magnetic Resonance Imaging in Python." In: *Computing in Science & Engineering* 9.3 (2007), pp. 52–55.
- [MB15] Bärbel Maus and Gerard van Breukelen. "Optimal Design for Functional Magnetic Resonance Imaging Experiments." In: *Zeitschrift für Psychologie* (2015).
- [MBS12] Sascha Meudt, Lutz Bigalke, and Friedhelm Schwenker. "Atlas - Annotation Tool Using Partially Supervised Learning and Multi-View Co-Learning in Human-Computer-Interaction Scenarios." In: *Int'l Conf. Information Science, Signal Processing and their Applications (ISSPA)*. IEEE, July 2012, pp. 1309–1312.
- [McC11] Steve McConnell. "What Does 10x Mean? Measuring Variations in Programmer Productivity." In: *Making Software*. O'Reilly & Associates, Inc., 2011, pp. 567–574.
- [McK+03] Kristen McKiernan, Jacqueline Kaufman, Jane Kucera-Thompson, and Jeffrey Binder. "A Parametric Manipulation of factors Affecting Task-Induced Deactivation in Functional Neuroimaging." In: *J. Cognitive Neuroscience* 15.3 (2003), pp. 394–408.

-
- [Mea+06] Jerry Mead, Simon Gray, John Hamer, Richard James, Juha Sorva, Caroline St Clair, and Lynda Thomas. "A Cognitive Approach to Identifying Measurable Milestones for Programming Skill Acquisition." In: *ACM SIGCSE Bulletin* 38.4 (2006), pp. 182–194.
- [Med+19] Julio Medeiros, Ricardo Couceiro, João Castelhan, M Castelo Branco, Gonalo Duarte, Catarina Duarte, Joo Dures, Henrique Madeira, P Carvalho, and C Teixeira. "Software code complexity assessment using EEG features." In: *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE. 2019, pp. 1413–1416.
- [Med+21] Jlio Medeiros, Ricardo Couceiro, Gonalo Duarte, Joo Dures, Joo Castelhan, Catarina Duarte, Miguel Castelo-Branco, Henrique Madeira, Paulo de Carvalho, and Csar Teixeira. "Can EEG Be Adopted as a Neuroscience Reference for Assessing Software Programmers' Cognitive Load?" In: *Sensors* 21.7 (2021), p. 2338.
- [MF15] Sebastian Mller and Thomas Fritz. "Stuck and Frustrated or in Flow and Happy: Sensing Developers' Emotions and Progress." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. Vol. 1. IEEE. 2015, pp. 688–699.
- [MF16] Sebastian Mller and Thomas Fritz. "Using (Bio)Metrics to Predict Code Quality Online." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE. 2016, pp. 452–463.
- [Mia+83] Richard Miara, Joyce Musselman, Juan Navarro, and Ben Shneiderman. "Program Indentation and Comprehensibility." In: *Communications of the ACM* 26.11 (1983), pp. 861–867.
- [Mil+07] John Milton, Ana Solodkin, Petr Hluřtk, and Steven Small. "The Mind of Expert Motor Performance is Cool and Focused." In: *NeuroImage* 35.2 (2007), pp. 804–813.
- [MML15] Roberto Minelli, Andrea Mocci, and Michele Lanza. "I Know What You Did Last Summer-An Investigation of How Developers Spend Their Time." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE. 2015, pp. 25–35.
- [MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. 2008, pp. 181–190.
- [MV95] Anneliese von Mayrhauser and Marie Vans. "Program Comprehension During Software Maintenance and Evolution." In: *Computer* 28.8 (1995), pp. 44–55.
- [MW01] Russell Mosemann and Susan Wiedenbeck. "Navigation and Comprehension of Programs by Novice Programmers." In: *Proc. Int'l Workshop Program Comprehension (IWPC)*. IEEE, 2001, pp. 79–88.
- [MWS06] Eleanor Maguire, Katherine Woollett, and Hugo Spiers. "London Taxi Drivers and Bus Drivers: A Structural MRI and Neuropsychological Analysis." In: *Hippocampus* 16.12 (Dec. 2006), pp. 1091–1101.

- [Nak+14] Takao Nakagawa, Yasutaka Kamei, Hidetake Uwano, Akito Monden, Kenichi Matsumoto, and Daniel M. German. "Quantifying Programmers' Mental Workload During Program Comprehension Based on Cerebral Blood Flow Measurement: A Controlled Experiment." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2014, pp. 448–451.
- [ND09] Andreas Nieder and Stanislas Dehaene. "Representation of Number in the Brain." In: *Annual Review of Neuroscience* 32 (2009), pp. 185–208.
- [Neb+05] Katharina Nebel, Holger Wiese, Philipp Stude, Armin de Greiff, Hans-Christoph Diener, and Matthias Keidel. "On the Neural Basis of Focused and Divided Attention." In: *Cognitive Brain Research* 25.3 (2005), pp. 760–776.
- [Neu+21] André Neumann, Norman Peitek, André Brechmann, Karsten Tabelow, and Thorsten Dickhaus. "Utilizing Anatomical Information for Signal Detection in Functional Magnetic Resonance Imaging." In: *WIAS Preprints* (2021).
- [NF09] Aljoscha Neubauer and Andreas Fink. "Intelligence and Neural Efficiency." In: *Neuroscience & Biobehavioral Reviews* 33.7 (2009), pp. 1004 –1023.
- [NH10] Marcus Nyström and Kenneth Holmqvist. "An Adaptive Algorithm for Fixation, Saccade, and Glissade Detection in Eyetracking Data." In: *Behavior Research Methods* 42.1 (2010), pp. 188–204.
- [NMB15] Keith Nolan, Aidan Mooney, and Susan Bergin. "Examining the Role of Cognitive Load When Learning to Program Program." In: January (2015), pp. 2–4.
- [NP15] Milan Nosal and Jaroslav Poruban. "Program Comprehension with Four-Layered Mental Model." In: *Int. Conf. Engineering of Modern Electric Systems (EMES)*. IEEE, 2015, pp. 1 –10.
- [NR68] Peter Naur and Brian Randell. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee*. Scientific Affairs Division, NATO, 1968.
- [NW70] Saul Needleman and Christian Wunsch. "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins." In: *Journal of Molecular Biology* 48.3 (1970), pp. 443–453.
- [OAH18] Unaizah Obaidallah, Mohammed Al Haek, and Peter C.-H. Cheng. "A Survey on the Usage of Eye-Tracking in Computer Programming." In: *ACM Comput. Surv.* 51.1 (Jan. 2018), 5:1–5:58.
- [OB17] Pavel Orlov and Roman Bednarik. "The Role of Extrafoveal Vision in Source Code Comprehension." In: *Perception* 46.5 (2017), pp. 541–565.
- [Orl17] Pavel Orlov. "Ambient and Focal Attention During Source-Code Comprehension." In: *Proc. Int'l Workshop on Eye Movements in Programming (EMIP)*. 2017, pp. 12–13.
- [OSH76] Linda Ottenstein, Victor Schneider, and Maurice Halstead. *Predicting the Number of Bugs Expected in a Program Module*. Tech. rep. 76-205. Department of Computer Science, University of Purdue, 1976.

-
- [OST57] Charles Egerton Osgood, George Suci, and Percy Tannenbaum. *The Measurement of Meaning*. 47. University of Illinois press, 1957.
- [Pei+18a] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, and André Brechmann. "Beyond Gaze: Preliminary Analysis of Pupil Dilation and Blink Rates in an fMRI Study of Program Comprehension." In: *Proc. Int'l Workshop on Eye Movements in Programming*. ACM, 2018, 4:1–4:5.
- [Pei+18b] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, and André Brechmann. "Toward Conjoint Analysis of Simultaneous Eye-Tracking and fMRI Data for Program-Comprehension Studies." In: *Proc. Int'l Workshop on Eye Movements in Programming*. ACM, 2018, 1:1–1:5.
- [Pei+18c] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, Johannes Hofmeister, and André Brechmann. "Simultaneous Measurement of Program Comprehension with fMRI and Eye Tracking: A Case Study." In: *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*. ACM, 2018, 24:1–24:10.
- [Pei+19] Norman Peitek, Sven Apel, André Brechmann, Chris Parnin, and Janet Siegmund. "CodersMUSE: Multi-Modal Data Exploration of Program-Comprehension Experiments." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. ACM, 2019, p. 4.
- [Pei+20] Norman Peitek, Janet Siegmund, Sven Apel, Christian Kästner, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. "A Look into Programmers' Heads." In: *IEEE Trans. Softw. Eng.* 46.4 (2020), pp. 442–462.
- [Pei+21] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. "Program Comprehension and Code Complexity Metrics: An fMRI Study." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2021.
- [Pen87] Nancy Pennington. "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs." In: *Cognitive Psychology* 19.3 (1987), pp. 295–341.
- [Per98] Thomas Perneger. "What's Wrong with Bonferroni Adjustments." In: *Bmj* 316.7139 (1998), pp. 1236–1238.
- [Pet+19] Cole Peterson, Jonathan Saddler, Tanja Blascheck, and Bonita Sharif. "Visually Analyzing Students' Gaze on C++ Code Snippets." In: *Proc. Int'l Workshop on Eye Movements in Programming (EMIP)*. IEEE. 2019, pp. 18–25.
- [PHD11] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. "A Simpler Model of Software Readability." In: *(MSR) Proc. Int'l Conf. Mining Software Repositories*. ACM, 2011, pp. 73–82.
- [Phi+97] James Phillips, Richard Leahy, John Mosher, and Bijan Timsari. "Imaging neural activity using MEG and EEG." In: *Engineering in Medicine and Biology Magazine* 16.3 (1997), pp. 34–42.

- [Pik+14] Matthew Pike, Horia Maior, Martin Porcheron, Sarah Sharples, and Max Wilson. "Measuring the Effect of Think Aloud Protocols on Workload Using fNIRS." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2014, pp. 3807–3816.
- [PIS17] Patrick Peachock, Nicholas Iovino, and Bonita Sharif. "Investigating Eye Movements in Natural Language and C++ Source Code - A Replication Experiment." In: *Augmented Cognition. Neurocognition and Machine Learning*. Springer International Publishing, 2017, pp. 206–218.
- [PS16] Christopher Palmer and Bonita Sharif. "Towards Automating Fixation Correction for Source Code." In: *Proc. Symposium on Eye Tracking Research & Applications*. ACM. 2016, pp. 65–68.
- [PS92] David Perkins and Gavriel Salomon. "Transfer of Learning." In: *International Encyclopedia of Education 2* (1992), pp. 6452–6457.
- [PSA20] Norman Peitek, Janet Siegmund, and Sven Apel. "What Drives the Reading Order of Programmers? An Eye Tracking Study." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. ACM, 2020, 342–353.
- [PSB17] Norman Peitek, Janet Siegmund, and André Brechmann. "Enhancing fMRI Studies of Program Comprehension with Eye-Tracking." In: *Proc. Int'l Workshop on Eye Movements in Programming*. Freie Universität Berlin, 2017, pp. 22–23.
- [PSP17] Chris Parnin, Janet Siegmund, and Norman Peitek. "On the Nature of Programmer Expertise." In: *Annual Conf. Psychology of Programming Interest Group (PPIG)*. 2017, pp. 109–118.
- [Rai+01] Marcus Raichle, Ann MacLeod, Abraham Snyder, William Powers, Debra Gusnard, and Gordon Shulman. "A Default Mode of Brain Function." In: *Proc. Nat'l Academy of Sciences* 98.2 (2001), pp. 676–682.
- [Ray78] Keith Rayner. "Eye movements in reading and information processing." In: *Psychological bulletin* 85.3 (1978), p. 618.
- [Ray98] Keith Rayner. "Eye Movements in Reading and Information Processing: 20 Years of Research." In: *Psychological Bulletin* 124.3 (1998), p. 372.
- [RFA20] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. "VITALSE: visualizing eye tracking and biometric data." In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 2020, pp. 57–60.
- [RHM15] Stevche Radevski, Hideaki Hata, and Kenichi Matsumoto. "Real-Time Monitoring of Neural State in Assessing and Improving Software Developers' Productivity." In: *Int'l Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE. 2015, pp. 93–96.
- [Ric87] Charles Rich. "Inspection Methods in Programming." MA thesis. Massachusetts Institute of Technology, 1987.

-
- [RM15] Paige Rodeghero and Collin McMillan. "An Empirical Study on the Patterns of Eye Movement During Summarization Tasks." In: *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–10.
- [Roe+12] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. "How Do Professional Developers Comprehend Software?" In: *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2012, pp. 255–265.
- [RW02] Václav Rajlich and Norman Wilde. "The Role of Concepts in Program Comprehension." In: *Proc. Int'l Workshop Program Comprehension (IWPC)*. IEEE, 2002, pp. 271–278.
- [Sak17] Siavash Sakhavi. "Application of Deep Learning Methods in Brain-Computer Interface Systems." PhD thesis. National University of Singapore (Singapore), 2017.
- [Sat+15] Yuri Sato, Sayako Masuda, Yoshiaki Someya, Takeo Tsujii, and Shigeru Watanabe. "An fMRI Analysis of the Efficacy of Euler Diagrams in Logical Reasoning." In: *Symp. on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2015.
- [SBS94a] Maarten Someren, Yvonne Barnard, and Jacobijn Sandberg. *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes*. Academic Press, 1994.
- [SBS94b] John Stern, Donna Boyer, and David Schroeder. "Blink Rate: A Possible Measure of Fatigue." In: *Human factors* 36.2 (1994), pp. 285–297.
- [Sca+17] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. "Automatically Assessing Code Understandability: How Far Are We?" In: *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE, 2017, pp. 417–427.
- [Sch+14] Felix Scholkmann, Stefan Kleiser, Andreas Jaakko Metz, Raphael Zimmermann, Juan Mata Pavia, Ursula Wolf, and Martin Wolf. "A review on continuous wave functional near-infrared spectroscopy and imaging instrumentation and methodology." In: *Neuroimage* 85 (2014), pp. 6–27.
- [Sch+20] Sarah Schuster, Stefan Hawelka, Nicole Alexandra Himmelstoss, Fabio Richlan, and Florian Hutzler. "The Neural Correlates of Word Position and Lexical Predictability during Sentence Reading: Evidence from Fixation-Related fMRI." In: *Language, Cognition and Neuroscience* 35.5 (2020), pp. 613–624.
- [SE84] Elliot Soloway and Kate Ehrlich. "Empirical Studies of Programming Knowledge." In: *IEEE Trans. Softw. Eng.* 10.5 (1984), pp. 595–609.
- [SEB82] Elliot Soloway, Kate Ehrlich, and Jeffrey Bonar. "Tapping into Tacit Programming Knowledge." In: *Proc. Conf. Human Factors in Computing Systems*. CHI '82. Gaithersburg, Maryland, USA: ACM, 1982, pp. 52–57.
- [Seg13] Mohamed Seghier. "The Angular Gyrus: Multiple Functions and Multiple Subdivisions." In: *The Neuroscientist* 19.1 (2013), pp. 43–61.

- [SFM12] Bonita Sharif, Michael Falcone, and Jonathan Maletic. "An Eye-Tracking Study on the Role of Scan Time in Finding Source Code Defects." In: *Proc. Symposium on Eye Tracking Research and Applications (ETRA)*. ACM, 2012, pp. 381–384.
- [SG07] Andrea Santi and Yosef Grodzinsky. "Working Memory and Syntax Interact in Broca's Area." In: *Neuroimage* 37.1 (2007), pp. 8–17.
- [Sha+08] Shi-Yun Shao, Kai-Quan Shen, Chong Jin Ong, Einar Wilder-Smith, and Xiao-Ping Li. "Automatic EEG Artifact Removal: A Weighted Support Vector Machine Approach with Error Correction." In: *Transactions on Biomedical Engineering* 56.2 (2008), pp. 336–344.
- [Sha+15a] Timothy Shaffer, Jenna Wise, Braden Walters, Sebastian Müller, Michael Falcone, and Bonita Sharif. "itrace: Enabling eye tracking on software artifacts within the ide to support software engineering tasks." In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. 2015, pp. 954–957.
- [Sha+15b] Amal Shargabi, Syed Ahmad Aljunid, Muthukkaruppan Annamalai, Shuhaida Mohamed Shuhidan, and Abdullah Mohd Zin. "Program Comprehension Levels of Abstraction for Novices." In: *Int. Conf. on Computer, Communications, and Control Technology*. IEEE, 2015, pp. 211–215.
- [Sha+20a] Zohreh Sharafi, Yu Huang, Kevin Leach, and Westley Weimer. "Towards an objective measure of developers' cognitive activities." In: *ACM Trans. Softw. Eng. & Methodology* (2020).
- [Sha+20b] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha Crosby. "A practical guide on conducting eye tracking studies in software engineering." In: *Empirical Software Engineering* (2020), pp. 1–47.
- [Shn76] Ben Shneiderman. "Exploratory Experiments in Programmer Behavior." In: *International Journal of Computer & Information Sciences* 5.2 (1976), pp. 123–143.
- [Shn77] Ben Shneiderman. "Measuring Computer Program Quality and Comprehension." In: *Int'l J. Man-Machine Studies* 9.4 (1977), pp. 465–478.
- [Sie+12] Janet Siegmund, André Brechmann, Sven Apel, Christian Kästner, Jörg Liebig, Thomas Leich, and Gunter Saake. "Toward Measuring Program Comprehension with Functional Magnetic Resonance Imaging." In: *Proc. Int'l Symposium Foundations of Software Engineering—New Ideas Track (FSE-NIER)*. ACM, 2012, 24:1–24:4.
- [Sie+14a] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. "Understanding Understanding Source Code with Functional Magnetic Resonance Imaging." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2014, pp. 378–389.
- [Sie+14b] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. "Measuring and Modeling Programming Experience." In: *Empirical Softw. Eng.* 19.5 (2014), pp. 1299–1334.

-
- [Sie16] Janet Siegmund. "Program Comprehension: Past, Present, and Future." In: *Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 13–20.
- [Sie+17] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. "Measuring Neural Efficiency of Program Comprehension." In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 140–150.
- [Sie+20] Janet Siegmund, Norman Peitek, André Brechmann, Chris Parnin, and Sven Apel. "Studying Programming in the Neuroage: Just a Crazy Idea?" In: *Commun. ACM* 63.6 (2020), pp. 30–34.
- [Sie+21] Janet Siegmund, Norman Peitek, Norbert Siegmund, and Sven Apel. "Conducting and Analyzing Human Studies: The Role of Variation and Aggregation." In: *ACM Trans. Softw. Eng. & Methodology* 30.1 (Dec. 2021). To appear.
- [Sim90] Herbert Simon. "Invariants of Human Behavior." In: *Annual Review of Psychology* 41.1 (1990), pp. 1–20.
- [Sjø+05] Dag Sjøberg, Jo Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette Rekdal. "A Survey of Controlled Experiments in Software Engineering." In: *IEEE Trans. Softw. Eng.* 31.9 (2005), pp. 733–753.
- [SM10] Bonita Sharif and Johnathon Maletic. "An Eye Tracking Study on camelCase and under_score Identifier Styles." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 2010, pp. 196–205.
- [SM79] Ben Shneiderman and Richard Mayer. "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results." In: *Int'l J. Parallel Programming* 8.3 (1979), pp. 219–238.
- [Smi+07] Stephen Smith, Mark Jenkinson, Christian Beckmann, Karla Miller, and Mark Woolrich. "Meaningful Design and Contrast Estimability in fMRI." In: *Neuroimage* 34.1 (2007), pp. 127–136.
- [Sne95] Harry Sneed. "Understanding Software Through Numbers: A Metric Based Approach to Program Comprehension." In: *Journal of Software Maintenance: Research and Practice* 7.6 (1995), pp. 405–419.
- [Sny00] Jennifer Snyder. "An Investigation of the Knowledge Structures of Experts, Intermediates and Novices in Physics." In: *Science Education* 22.9 (2000), pp. 979–992.
- [Sok+17] Moriah Sokolowski, Wim Fias, Ahmad Mousa, and Daniel Ansari. "Common and Distinct Brain Regions in Both Parietal and Frontal Cortex Support Symbolic and Nonsymbolic Number Processing in Humans: A Functional Neuroimaging Meta-Analysis." In: *NeuroImage* 146 (2017), pp. 376–394.

- [SS01] Craig Stark and Larry Squire. "When Zero is Not Zero: The Problem of Ambiguous Baseline Conditions in fMRI." In: *Proc. National Academy of Sciences* 98.22 (2001), pp. 12760–12766.
- [SS92] James Shanteau and Thomas Stewart. "Why Study Expert Decision Making? Some Historical Perspectives and Comments." In: *Organizational Behavior and Human Decision Processes* 53.2 (1992), pp. 95 –106.
- [SSA15] Janet Siegmund, Norbert Siegmund, and Sven Apel. "Views on Internal and External Validity in Empirical Software Engineering." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. Vol. 1. IEEE. 2015, pp. 9–19.
- [SSG15] Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. "A Systematic Literature Review on the Usage of Eye-Tracking in Software Engineering." In: *Information and Software Technology* 67 (2015), pp. 79–107.
- [Sta84] Thomas Standish. "An Essay on Software Reuse." In: *IEEE Trans. Softw. Eng.* SE–10.5 (1984), pp. 494–497.
- [STD16] Konstantin Schildknecht, Karsten Tabelow, and Thorsten Dickhaus. "More Specific Signal Detection in Functional Magnetic Resonance Imaging by False Discovery Rate Control for Hierarchically Structured Systems of Hypotheses." In: *PLOS ONE* 11.2 (Feb. 2016), pp. 1–21.
- [Sto05] Margaret-Anne Storey. "Theories, Methods and Tools in Program Comprehension: Past, Present and Future." In: *Proc. Int'l Workshop on Program Comprehension (IWPC)* (2005), pp. 181–191.
- [SV95] Teresa Shaft and Iris Vessey. "The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension." In: *Information Systems Research* 6.3 (1995), pp. 286–299.
- [SW65] Samuel Shapiro and Martin Wilk. "An Analysis of Variance Test for Normality (Complete Samples)." In: *Biometrika* 52.3/4 (1965), pp. 591–611.
- [Tan+13] Satoshi Tanaka, Hanako Ikeda, Kazumi Kasahara, Ryo Kato, Hiroyuki Tsubomi, Sho Sugawara, Makoto Mori, Takashi Hanakawa, Norihiro Sadato, Manabu Honda, and Katsumi Watanabe. "Larger Right Posterior Parietal Volume in Action Video Game Experts: A Behavioral and Voxel-Based Morphometry (VBM) Study." In: *PLoS ONE* 8.6 (June 2013), e66998+.
- [TH19] David Thomas and Andrew Hunt. *The Pragmatic Programmer: Your Journey to Mastery*. Addison-Wesley Professional, 2019.
- [Tia11] Rebecca Tiarks. "What Programmers Really Do: An Observational Study." In: *Softwaretechnik-Trends* 31.2 (2011), pp. 36–37.
- [Tri17] Tricentis. *Software Fail Watch: 5th Edition*. Tech. rep. 2017.
- [TT88] Jean Talairach and Pierre Tournoux. *Co-Planar Stereotaxic Atlas of the Human Brain*. Thieme, 1988.

-
- [Tur+14] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. "An Eye-Tracking Study Assessing the Comprehension of C++ and Python Source Code." In: *Proc. Symposium on Eye Tracking Research and Applications (ETRA)*. ACM, 2014, pp. 231–234.
- [Udd+19] Julia Uddén, Annika Hultén, Jan-Mathijs Schoffelen, Nietzsche Lam, Karin Harbusch, Antal van den Bosch, Gerard Kempen, Karl Magnus Petersson, and Peter Hagoort. "Supramodal Sentence Processing in the Human Brain: fMRI Evidence for the Influence of Syntactic Complexity in More Than 200 Participants." In: *bioRxiv* (2019), p. 576769.
- [Uwa+06] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. "Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement." In: *Proc. Symposium on Eye Tracking Research & Applications (ETRA)*. ACM, 2006, pp. 133–140.
- [Var+17] Alberto Salvador Núñez Varela, Hector G. Pérez-González, Francisco E. Martínez-Perez, and Carlos Soubervielle-Montalvo. "Source Code Metrics: A Systematic Mapping Study." In: *Journal of Systems and Software* 128 (2017), pp. 164–197.
- [VEV+16] Helene Van Ettinger-Veenstra, Anita McAllister, Peter Lundberg, Thomas Karlsson, and Maria Engström. "Higher Language Ability is Related to Angular Gyrus Activation Increase During Semantic Processing, Independent of Sentence Incongruency." In: *Frontiers in Human Neuroscience* 10 (2016), p. 110.
- [VF20] Roger Denis Vieira and Kleinner Farias. "CognIDE: A Psychophysiological Data Integrator Approach for Visual Studio Code." In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. 2020, pp. 393–398.
- [VG98] JA Veltman and AWK Gaillard. "Physiological workload reactions to increasing levels of task difficulty." In: *Ergonomics* 41.5 (1998), pp. 656–669.
- [VMV96] Anneliese Von Mayrhauser and A. Marie Vans. "Identification of Dynamic Comprehension Processes During Large Scale Maintenance." In: *IEEE Trans. Softw. Eng.* 22.6 (1996), pp. 424–437.
- [Voß+08] Adrian Voßkühler, Volkhard Nordmeier, Lars Kuchinke, and Arthur Jacobs. "OGAMA (Open Gaze and Mouse Analyzer): Open-Source Software Designed to Analyze Eye and Mouse Movements in Slideshow Study Designs." In: *Behavior Research Methods* 40.4 (2008), pp. 1150–1162.
- [WF95] Keith Worsley and Karl Friston. "Analysis of fMRI Time-Series Revisited—Again." In: *Neuroimage* 2.3 (1995), pp. 173–181.
- [Wie86] Susan Wiedenbeck. "Processes in Computer Program Comprehension." In: *Workshop on Empirical Studies of Programmers*. 1986, pp. 48–57.
- [Wie91] Susan Wiedenbeck. "The Initial Stage of Program Comprehension." In: *International Journal of Man-Machine Studies* 35.4 (1991), pp. 517–540.

- [Wir+11] Miranka Wirth, Kay Jann, Thomas Dierks, Andrea Federspiel, Roland Wiest, and Helge Horn. "Semantic Memory Involvement in the Default Mode Network: A Functional Neuroimaging Study Using Independent Component Analysis." In: *Neuroimage* 54.4 (2011), pp. 3057–3066.
- [WS89] Susan Wiedenbeck and Jean Scholtz. "Beacons and Initial Program Comprehension." In: *SIGCHI Bull.* 21.1 (Aug. 1989), 90–91.
- [Xia+17] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed Hassan, and Shanping Li. "Measuring Program Comprehension: A Large-Scale Field Study with Professionals." In: *IEEE Trans. Softw. Eng.* 44.10 (2017), pp. 951–976.
- [Yeh+17] Martin Yeh, Dan Gopstein, Yu Yan, and Yanyan Zhuang. "Detecting and Comparing Brain Activity in Short Program Comprehension Using EEG." In: *Frontiers in Education Conference*. IEEE, 2017, pp. 1–5.
- [Züg+18] Manuela Züger, Sebastian Müller, André Meyer, and Thomas Fritz. "Sensing Interruptibility in the Office: A Field Study on the Use of Biometric and Computer Interaction Sensors." In: *Proc. Conf. Human Factors in Computing Systems*. ACM, 2018, pp. 1–14.
- [Zus93] Horst Zuse. "Criteria for Program Comprehension Derived from Software Complexity Metrics." In: *Workshop on Program Comprehension*. IEEE, 1993, pp. 8–16.