



Universität Stuttgart

Evidence for the Design of Code Comprehension Experiments

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Marvin Wyrich
aus Stuttgart, Deutschland

Hauptberichter: Prof. Dr. Stefan Wagner
Mitberichter: Prof. Dr. Janet Siegmund,
Prof. Westley Weimer, Ph.D.

Tag der mündlichen Prüfung: Dienstag, 12.09.2023

Institut für Software Engineering

2023

ZUSAMMENFASSUNG

Kontext: Valide Studien schaffen Vertrauen in wissenschaftliche Erkenntnisse. Um ein Studiendesign allerdings sorgfältig beurteilen zu können, ist neben einer allgemeinen Expertise in Forschungsmethodiken auch spezifisches Fachwissen erforderlich. Beispielsweise kann in einem Experiment der Einfluss einer manipulierten Gegebenheit auf eine Beobachtung durch viele weitere Gegebenheiten beeinflusst werden. Hierbei sprechen wir von Störvariablen. Die Kenntnis möglicher Störvariablen im thematischen Kontext ist essenziell, um ein Studiendesign beurteilen zu können. Werden bestimmte Störvariablen nicht erkannt und folglich nicht kontrolliert, kann dies eine Gefahr für die Validität der Studienergebnisse darstellen.

Problem: Bisher erfolgt die Beurteilung der Validität einer Studie nur intuitiv. Die potenzielle Verzerrung von Studienerkenntnissen durch Störvariablen ist dadurch spekulativ, statt evidenzbasiert. Dies führt zu Unsicherheiten im Entwerfen von Studien sowie zu Meinungsverschiedenheiten im Peer-Review. Zwei Hindernisse erschweren derzeit jedoch die evidenzbasierte Evaluation von Studiendesigns. Erstens sind viele der vermuteten Störvariablen noch nicht ausreichend erforscht, um ihre tatsächliche Wirkung zu belegen. Zweitens fehlt eine pragmatische Methode, um die vorhandene Evidenz aus Primärstudien so aufzubereiten, dass sie Forschenden leicht zugänglich ist.

Scope: Wir untersuchen die Problemstellung im Kontext experimenteller Forschungsmethoden mit menschlichen Studienteilnehmern und im thematischen Kontext der Codeverständnis-Forschung.

Beiträge: Wir analysieren zunächst systematisch die Designentscheidungen in Codeverständnis-Experimenten der letzten 40 Jahre und die Gefahren für die Validität dieser Studien. Dies bildet die Grundlage für eine anschließende Diskussion über die große Vielfalt der Designoptionen bei gleichzeitig fehlender Evidenz zu deren Konsequenzen und Vergleichbarkeit. Daraufhin führen wir Experimente durch, die Evidenz zum Einfluss von Intelligenz, Persönlichkeit und kognitiven Verzerrungen auf das Codeverstehen liefern. Während bisher nur über den Einfluss dieser Variablen spekuliert wurde, verfügen wir nun über erste Datenpunkte zu deren tatsächlichen Einfluss. Abschließend zeigen wir, wie die Zusammenführung verschiedener Primärstudien zu Evidenzprofilen die evidenzbasierte Diskussion von experimentellen Designs vereinfacht. Für die drei am häufigsten diskutierten Bedrohungen der Validität in Codeverständnis-Experimenten erstellen wir Evidenzprofile und diskutieren deren Implikationen.

Fazit: Für häufig diskutierte Gefahren der Validität findet sich Evidenz für und gegen deren Vorhandensein. Diese widersprüchliche Evidenz erklärt sich durch die Notwendigkeit, individuelle Störvariablen im jeweiligen Kontext eines konkreten Studiendesigns zu betrachten; nicht wie oft geschehen als pauschal gültig. Evidenzprofile zeigen ein Evidenzspektrum auf und dienen Forschenden als Einstiegspunkt für eine evidenzbasierte Diskussion ihres Studiendesigns. Wie bei allen Arten systematischer Sekundärstudien beruht allerdings auch der Erfolg von Evidenzprofilen darauf, dass ausreichend viele Studien zu jeweils derselben Forschungsfrage publiziert werden. Dies stellt eine besondere Herausforderung in einem Forschungsfeld dar, in dem die Neuheit von Forschungserkenntnissen eines Manuskripts zu den Evaluationskriterien jeder größeren Konferenz gehört. Wir blicken dennoch optimistisch in die Zukunft, da auch Evidenzprofile, die lediglich aufzeigen werden, dass Evidenz zu einer bestimmten Streitfrage rar ist, einen Beitrag leisten: Sie werden meinungsstarke Beurteilungen von Studiendesigns als solche identifizieren sowie zusätzliche Studien motivieren.

ABSTRACT

Context: Valid studies establish confidence in scientific findings. However, to carefully assess a study design, specific domain knowledge is required in addition to general expertise in research methodologies. For example, in an experiment, the influence of a manipulated condition on an observation can be influenced by many other conditions. We refer to these as confounding variables. Knowing possible confounding variables in the thematic context is essential to be able to assess a study design. If certain confounding variables are not identified and consequently not controlled, this can pose a threat to the validity of the study results.

Problem: So far, the assessment of the validity of a study is only intuitive. The potential bias of study findings due to confounding variables is thus speculative, rather than evidence-based. This leads to uncertainty in the design of studies, as well as disagreement in peer review. However, two barriers currently impede evidence-based evaluation of study designs. First, many of the suspected confounding variables have not yet been adequately researched to demonstrate their true effects. Second, there is a lack of a pragmatic method to synthesize the existing evidence from primary studies in a way that is easily accessible to researchers.

Scope: We investigate the problem in the context of experimental research methods with human study participants and in the thematic context of code

comprehension research.

Contributions: We first systematically analyze the design choices in code comprehension experiments over the past 40 years and the threats to the validity of these studies. This forms the basis for a subsequent discussion of the wide variety of design options in the absence of evidence on their consequences and comparability. We then conduct experiments that provide evidence on the influence of intelligence, personality, and cognitive biases on code comprehension. While previously only speculating on the influence of these variables, we now have some initial data points on their actual influence. Finally, we show how combining different primary studies into evidence profiles facilitates evidence-based discussion of experimental designs. For the three most commonly discussed threats to validity in code comprehension experiments, we create evidence profiles and discuss their implications.

Conclusion: Evidence for and against threats to validity can be found for frequently discussed threats. Such conflicting evidence is explained by the need to consider individual confounding variables in the context of a specific study design, rather than as a universal rule, as is often the case. Evidence profiles highlight such a spectrum of evidence and serve as an entry point for researchers to engage in an evidence-based discussion of their study design. However, as with all types of systematic secondary studies, the success of evidence profiles relies on publishing a sufficient number of studies on the same respective research question. This is a particular challenge in a research field where the novelty of a manuscript's research findings is one of the evaluation criteria of any major conference. Nevertheless, we are optimistic about the future, as even evidence profiles that will merely indicate that evidence on a particular controversial issue is scarce will make a contribution: they will identify opinionated assessments of study designs as such, as well as motivate additional studies to provide more evidence.

ACKNOWLEDGEMENTS

Over the years, I have come to the equally beautiful and sad realization that I was very lucky with the people in my most immediate work environment. I have come to know many doctoral students who have told me firsthand about abusive supervisors and mentally distressing working conditions. It is not a secret that such things happen in academia. Nevertheless, it is an illusion to believe that one's own research field will be immune to bad people in positions of power. The person reading this and feeling helpless should know that you deserve to be happy, and nothing justifies the situation you are in. The vast majority of people I have met are good people at heart. Switching to their research groups is always a step forward, never a step back, even if at first it feels like giving something up.

That said, I had the pleasure of experiencing the support of some special individuals. I am grateful for this, and I would like to express my gratitude in the following lines.

Stefan Wagner. I would like to thank Stefan in particular for two things. For one, as my supervisor, he gave me every freedom in my research and choice of research topics, so that I could develop creatively and find my place in science. At the right time, he asked the right questions so that I did not wander off the path too much. For another, Stefan has shown me,

consciously or unconsciously, that as a researcher, you can hold views and ideals that, if more researchers in our field stood up for them, would do good to the current science system.

Daniel Graziotin. Daniel has been a mentor to me from the early moments of my academic career, and continues to be today. Science is a lot more political than I would have guessed a few years ago, and if there is anyone to ask for advice on how to make morally sound decisions in politically charged situations, that is you. Thank you for showing me what good science is, but even more what makes a good scientist.

Justus Bogner. Justus not only made my time as a PhD student way more fun. I have always admired the rigor and discipline with which he approaches scientific questions, so that over the years, cheeky as I am, I have tried to copy this quality. I am well on my way, but still look up to you.

Janet Siegmund. Janet has been on my radar for years as an impressive researcher whose publications I have built on more than once. I am honored that I have finally been able to get to know her as the positive person she is, and that she has agreed without reservation to officially review this thesis.

Westley Weimer. Equally, I would like to thank Westley for his support as co-examiner and for the stimulating conversations we had at Schloss Dagstuhl and in the time thereafter. Your commitment and dedication to supporting students at various stages of their careers inspires me so much that I am finding myself enjoying teaching again.

Mother, father, and brother. I thank my family for the excellent start they gave me and the values they taught me. My brother and I have always been encouraged to pursue the things that make us happy, no matter how much money or prestige is involved. When I decided to study, it was something new for all of us. That I am now even close to a doctorate is just wild. But it

makes me happy to look back, and I think I have found what will make me happy in the future.

Caroline. You have been by my side just about since the beginning of my PhD, and I hope you will be for the rest of my life. You always listened to me patiently when I fabulated with passion about nerdy scientific theories. Call yourself an expert in designing code comprehension studies; I am sure no one has heard as much as you about that topic in the past few years.

The right people at the right time. Over the past few years, I have had the privilege of working with many researchers around the world, over a thousand people have participated in my studies, and about two dozen students have had enough trust in me to complete their bachelor's and master's theses with me. Thank you all because without you this dissertation would not have turned out half as well. Lastly, I would like to thank two friends who accompanied me through the time of my PhD outside the work context. Vincent and Eddi, thank you for the conversations and the numerous amusing and exciting hours in virtual cooperative survival games.

Marvin Wyrich in April, 2023

CONTENTS

1. Introduction	15
1.1. Motivational Context	15
1.2. Research Objective and Scope	18
1.3. Research Philosophy	19
1.3.1. Experimentation and Evidence	20
1.3.2. Intuition and the Diversity of Study Designs	25
1.3.3. Validity	28
1.3.4. Measuring the Unobservable	29
1.4. Contributions	32
1.5. List of Publications	33
1.6. Outline	34
2. Source Code Comprehension	35
2.1. Program Comprehension Strategies	38
2.2. The Developer in the Spotlight	54
2.3. Context Is Everything	58
2.4. Measuring Code Comprehension	60
2.5. A Conceptual Model of Code Comprehension Experiments	63
2.6. Conclusion	70

3. Forty Years of Designing Code Comprehension Experiments	71
3.1. Context and Goals	71
3.2. A Systematic Mapping Study	73
3.2.1. Background and Related Work	73
3.2.2. Methodology	75
3.2.3. Results	84
3.2.4. Discussion	117
3.3. Conclusion	125
4. How Individual Characteristics Influence Code Comprehension	129
4.1. Context and Goals	130
4.2. A Study of Intelligence and Personality	131
4.2.1. Background and Related Work	131
4.2.2. Methodology	135
4.2.3. Results	147
4.2.4. Discussion	153
4.3. Conclusion	161
5. How Contextual Factors Influence Code Comprehension	163
5.1. Context and Goals	164
5.2. A Study of Displaying a Metric to Affect Code Comprehension	167
5.2.1. Background and Related Work	167
5.2.2. Methodology	171
5.2.3. Results	181
5.2.4. Discussion	185
5.3. A Study of Anchoring Developers Through Task Descriptions	190
5.3.1. Background and Related Work	192
5.3.2. Methodology	193
5.3.3. Results	201
5.3.4. Discussion	203
5.4. Conclusion	208

6. Evidence Profiles for Validity Threats in Code Comprehension	211
Experiments	211
6.1. Context and Goals	212
6.2. A Study of Evidence Profiles for Validity Threats	213
6.2.1. Background and Related Work	213
6.2.2. Methodology	217
6.2.3. Results	225
6.2.4. Discussion	234
6.3. Conclusion	240
7. Discussion	243
7.1. Answers to the Research Questions	243
7.2. Open Challenges and Limitations	249
7.3. Future Research Directions	255
8. Conclusion	259
Bibliography	261
List of Figures	295
List of Tables	297
List of Definitions	299
A. Primary Studies of the Systematic Mapping Study	301
B. Replication Packages	313

INTRODUCTION

1.1. Motivational Context

What does a software developer do? Many of us probably have an answer to this question. Some of us will tell anecdotal stories about exciting adventures in the software industry, such as how we used software to put people on the moon or how today's cars would no longer function without software. We do well to tell these stories, especially if we thus spark the questioner's interest in software development.

The results of scientific studies can be a bit less romantic. Developers actually spend most of their time looking at existing code and trying to understand it. Minelli et al. [MML15] suggest that about 70 percent of a developer's time is spent on code comprehension. Xia et al. [XBL+18] confirmed this finding in a large-scale field study with software professionals and found that developers spend about 58 percent of their time on code comprehension activities. However, this reality is not something that should scare off interested newcomers to software development. Code comprehension can be a fulfilling activity, and to ensure that this is the case more often than it frustrates the developer, researchers are looking into the subject. They

investigate the influences on code comprehension with scientific studies, for example to work out guidelines for more understandable code or to support developers in understanding code. Today, such studies are more common than ever [WBW23].

When designing a study, researchers make sure that the study design is as valid as possible. Validity is a multifaceted construct and we will cover it in more detail later. For the moment, we use Kitchenham et al.'s brief summary that validity refers to the degree to which we can trust the outcomes of an empirical study [KBB15]. Assessing the validity of a study design, that is whether we can trust the results, requires expert knowledge. No one could make this assessment better than the researchers themselves, which is why we consider it a good development that nowadays threats to validity are discussed in almost every published code comprehension study [SKSL17a; WBW23].

Two meta-studies have categorized threats to validity discussed in code comprehension studies, coming up with over fifty different threat categories [SS15; WBW23]. At the same time, we see a substantial challenge that is the focus of this thesis: hardly anyone is sure about the actual extent of the discussed threats, and almost no paper cites evidence on the assumed threats [MWGW23; WBW23]. The list of potential threats to validity is long, but their impact in a specific study context is only speculative.

Consider the following example to illustrate this point. The most frequently discussed threat to validity in code comprehension studies is the influence of programming experience on the observed behavior of study participants during a code comprehension task [WBW23]. To researchers who have commented publicly on this topic, the assumption that a certain treatment influences novice and expert programmers differently makes intuitive sense [FZB+18]. As a result, the authors of a study often devote a paragraph in the discussion section of their paper to this potential influencing factor and mention, for example, that their sample consisted solely of students and that the study results cannot therefore be applied to more experienced developers. Is it justified to place such an emphasis on programming experience in this particular study? What about the more than fifty other potential threats to

validity, which cannot possibly all be discussed?

The last decades of code comprehension research have been characterized by intuitive study designs and speculative discussions about the validity of these studies. This leads to uncertainty for the researchers when they design their studies, and at the very least, it leads to potential conflict in peer review, when the reviewer critiques the study design based on their own different intuition. That researchers have very different views on the assessment of validity was found, for example, by Siegmund et al. [SSA15] in a survey of 79 program committee and editorial board members. These views included even those that would reject papers in principle if they attempted to maximize internal validity [SSA15]. Note that researchers should generally have their own opinions. The fundamental issue is that personal views currently determine which scientific findings are published and which are not. A decision that, apart from ethical considerations, should instead be based on an informed validity assessment. Otherwise, we run the risk that a few researchers will determine the scientific discourse according to their views, while valid and potentially influential minority views will be unfairly rejected.

Theoretically, no one can be blamed for the current situation; the research community lacks a practical solution to discuss design decisions based on evidence. The good news is that we already have some of the evidence needed to make evidence-based evaluation of study designs possible. In numerous primary studies of code comprehension, we find data that support or refute the influence of certain factors on code comprehension. For example, some of these studies show that programming experience in their study context had an effect on participants' code comprehension performance, while others found no such effect [MWGW23]. We argue in this thesis, though, that the existing evidence now needs to be synthesized so that it is more accessible to researchers when designing their primary studies. It is not feasible for authors of primary studies to look through the entire corpus of code comprehension literature each time to learn about the consequences of their design decisions. To this end, we propose to conduct meta-studies and synthesize evidence for the influence of commonly assumed threats to validity.

1.2. Research Objective and Scope

The research objective is to support researchers with synthesized evidence in the discussion of the validity of their code comprehension experiments. The central research question is as follows:

RQ How can evidence be used to evaluate the validity of code comprehension experiments?

The answer to this question needs a solid theoretical foundation. We will first look at the design landscape of code comprehension experiments. We then address individual and contextual factors that should be considered when designing code comprehension experiments. The following research questions guide us in this endeavor:

RQ1 What are differences and similarities in the design characteristics of code comprehension experiments?

RQ2 How do individual characteristics of a developer influence code comprehension?

RQ3 How do contextual factors influence code comprehension?

RQ4 What evidence can be found for frequently discussed threats to validity in code comprehension experiments?

We narrow the scope of this work by two explicit choices. First, we focus on experiments (defined later in [Section 2.5](#)). Code comprehension studies use a variety of research methods, and many of them could benefit from evidence-based design decisions. However, the methods are so different that it is more sensible to develop recommendations for the respective study designs separately. Furthermore, the entire discussion about the validity of study designs is very dependent on philosophical views, and these views are reflected in the use of different research methods. For example, one can say that validity has a different meaning in an interview study than in an experiment [[PG13](#)].

Second, we focus on experiments in which participants had to understand a set of concrete lines of code to infer the intentions behind the code at a

higher level of abstraction. This process is usually referred to as *bottom-up comprehension* [OBS04; SM79]. The counterpart to this is called *top-down comprehension*, a process in which developers refine hypotheses based on domain knowledge and documentation until the knowledge can be mapped to concrete code segments [Bro83; SV95]. There is enough reason to assume that different approaches are based on different cognitive models and, accordingly, study designs for different cognitive models are not necessarily comparable in nature. By focusing on bottom-up code comprehension studies, we can at least assume that the primary studies aim to measure the same cognitive process. We elaborate on this line of thought in [Chapter 2](#).

1.3. Research Philosophy

The core research question of this thesis makes use of at least three constructs that are frequently encountered in the everyday language of various groups: Researchers cite and discuss ‘evidence’ on specific research questions and assess the ‘validity’ of individual studies; Developers are familiar with the notion of ‘code comprehension’, in the sense that they need to understand source code to work with it. The way the terms are used in practice, in my experience, is the way they are used for the most part in this thesis. It is therefore possible to understand the remainder of the thesis without delving deeper into this section on research philosophy.

For those who seek a nuanced view of the following chapters and wish to critique or build upon the work holistically, this section will be useful. While the discoveries and reflections detailed in chapters three through six are the result of collaborative work, this section complements the work by providing a personal perspective that does not necessarily coincide with the views of my co-authors. This section will have the greatest impact on the interpretation of [Chapter 6](#), in which I make my central point about using synthesized evidence to evaluate study designs.

1.3.1. Experimentation and Evidence

“ *The principle of science, the definition, almost, is the following: The test of all knowledge is experiment. Experiment is the sole judge of scientific ‘truth.’* ”

— R.P. FEYNMAN [*FLS63*]

It might not be surprising that a physicist such as Richard Feynman puts great importance on experiments. Experiments are a good way to explore causality and I myself developed an early enthusiasm for the idea that by meticulously controlling the experimental parameters and manipulating a single variable, one might be able to attribute an observable effect to that manipulation. Even the study of *human aspects* through experimentation has long since found its way into science, and in the recent decades into a branch called behavioral software engineering [*GLFW21*; *LFW15*]. This thesis, for example, is based in part on three experiments that we conducted to investigate the behavior of developers.

Yet, I would not go so far as to say that experiments are the only way to learn about the world. I even understand if some researchers are fundamentally opposed to trying to explain human behavior with controlled experiments. One view is that each person is unique and may even perceive the world differently, which makes generalizations based on experimentally collected data on any sample at least difficult.

The value I attach to experiments for obtaining knowledge lies somewhere between these two extremes, between those who consider experiments to be the only true research method and those who are fundamentally critical of experiments for explaining human behavior. There are research questions around human aspects for which other research methods are better suited than controlled experiments; for example, those questions that seek to discover individual motivations for a particular behavior (sometimes referred to as ‘why questions’). For me, the individual characteristics of a person, innumerable in their quantity, constitute confounding factors that can, in an

ideal controlled experiment, be controlled for. Since we do not achieve this ideal state, questions arise about internal validity, that is, how certain we can be that a measured effect is really due to a manipulated variable. Regardless, I would say that even if people perceive the world differently and everyone responds differently to a treatment, an experiment at least provides evidence about how likely it is that a person will respond to a treatment and what the potential range of effect sizes is.

What Is Evidence?

Evidence, whether it comes from experiments or other research methods, can then help people in many ways. In the context of this work, for example, it should help researchers better assess the consequences of study design decisions. We argued at the beginning that without such evidence, uncertainty prevails for the most part, which can then lead to frustration when one's own study design is rejected in peer review based on the reviewer's differing intuition.

But what is evidence? When do we have enough of it to accept a theory as *truth*? When might evidence need to be re-evaluated? These questions lead us straight into the philosophy of science, and over the last centuries numerous great philosophers have argued about them [God21]. My perspective on these issues can be summarized as follows.

Foremost, I think Karl Popper made an incredibly important contribution to this discussion with his intention to distinguish science from non-science by formulating hypotheses in such a way that they have the potential to be refuted by observations at all [Pop05]. The idea of *falsificationism* can prevent statements from being formulated in such a way that observations can always be interpreted to support a certain statement. We must follow this basic principle as well when we collect and synthesize evidence to support study design decisions. Popper, however, was quite strict in his view. If a prediction were contradicted by an observation, we would reject our theory. If an observation were consistent with our prediction, then the theory would not be falsified. Up to this point, I agree. However, Popper also argues that

in the case of consistency, it cannot be concluded that we have gained in certainty and that the theory is now more likely to be true.

“ I think that we shall have to get accustomed to the idea that we must not look upon science as a ‘body of knowledge,’ but rather a system of hypotheses; that is to say, as a system of guesses or anticipations which in principle cannot be justified, but with which we work as long as they stand up to tests, and of which we are never justified in saying that we know they are ‘true’ or ‘more or less certain’ or even ‘probable.’ ”

— K.R. POPPER [POP05]

Here I consider the view of the *logical empiricists*, a group of philosophers who were rejected by Popper and subsequently by many other contemporary philosophers, which ultimately led to their irrelevance, to be quite valid: we accept that we can never be completely sure that a theory is true (a principle referred to as *fallibilism*), but individual pieces of evidence nevertheless increase our confidence in the truth of a theory [God21]. Taken to a concrete example, this would mean that observing a sample of experienced developers solve a code comprehension task efficiently provides relevant evidence on whether all experienced developers understand code efficiently. The evidence may be considered less weighty than that from a controlled experiment with explicitly rejected null hypothesis, but both cases provide us with relevant evidence that can help us in a discussion about the influence of programming experience.

When Do We Have Enough Evidence to Accept a Statement As Truth?

For our purposes, we do not necessarily need an answer whether something is *actually* as we assume or not. It is sufficient to know what speaks for and against to then be able to take an informed position. This approach aligns

with the thinking of Richard Rudner, who believed that evidence alone is not sufficient, but that it takes a decision to accept the underlying theory.

“ Since no scientific hypothesis is ever completely verified, in accepting a hypothesis the scientist must make the decision that the evidence is sufficiently strong or that the probability is sufficiently high to warrant the acceptance of the hypothesis. [...] How sure we need to be before we accept a hypothesis will depend on how serious a mistake would be. ”

— R. RUDNER [RUD53]

The topic of consequences-influenced decisions to accept hypotheses is still being debated [God21]. For me, it is intuitive that there is a gap between evidence and accepting a theory that can be closed by a conscious decision. Also playing a role for this view is the *underdetermination of theory by evidence*, which states “that for any collection of evidence we have, there is always more than one theory that can in principle account for that evidence. If so, this seems to show that our preference for a particular theory must always be influenced to some extent by factors other than evidence—by simplicity, elegance, or sheer familiarity” [God21]. Thus, while we strive to draw on more evidence when evaluating study designs, we should keep these aspects in mind and, with the evidence in hand, never argue for a single absolute truth.

Is Evidence Timeless?

Let us now assume for a moment that the evidence on a research question is reasonably conclusive. For example, there could be 42 primary studies on whether while-loops are better understood by novice programmers than for-loops, and almost all study authors conclude that a while-loop is the syntactic construct that is easier to understand of the two. Developers

form an opinion based on the evidence and program loops from now on predominantly with the more understandable control structure. Is there a point when this evidence needs to be re-evaluated and may even become obsolete?

My view of this question is influenced by the three philosophers Kuhn, Lakatos, and Laudan, who had different views, but agreed on at least one rough idea: Researchers have a set of fundamental beliefs that they question at times, but on which they have collectively agreed to conduct research.¹ Whether we believe the evidence from primary studies depends not only on how the studies themselves are designed. Each such study is based on at least one fundamental belief, and those beliefs can change over time, for example, when there is a lot of evidence that a particular theory might not hold and when we have a better one at hand. When there is a change in a core belief, we automatically look at the evidence from previous primary studies with different eyes.

Take the example of research on the comprehensibility of for- and while-loops to make this more concrete. Researchers in the code comprehension research field rely on a few basic assumptions that enough researchers agree with to not have to question or justify them often. In our example, this is at least the assumption that there is an inherent complexity to certain code and control structures. This is indeed an assumption that has motivated a number of publications in the past, for example on atoms of confusion [GIY+17a; GZFC18] or the development of code comprehensibility metrics that are purely based on the code and ignore any developer characteristics [Cam18; MWW20]. However, this fundamental assumption may be discarded in the future. What we have assumed to know about the influence of certain control structures on code comprehension, we then have to reassess in light of a

¹This quintessence summarizes some of the work of Laudan, Lakatos, and Kuhn on the evolution of scientific fields. Kuhn first came up with the idea of a *research paradigm*, and its role in what he distinguished as *revolutionary science* from *normal science* [Kuh62]. Lakatos developed a picture in which several paradigm-like units can co-exist and referred to them as *research programs* that all have a hard core and a protective belt [Lak76]. Laudan further developed the idea and called the paradigm-like units *research traditions* [Lau77]. All three had different views on how many such paradigm-like units there are in a research field, how they competed and evolve, and how critical researchers were of them at any given time [God21].

new fundamental assumption. We might then attribute the influences of an inherent complexity of code structures on code comprehension as measured in the primary studies to a variable that we do not even know about today. Kuhn went so far as to argue that it is almost inevitable that paradigms will collapse eventually as a result of the work of scientists [Kuh62]. This may sound drastic, but these are the revolutions that move a research field forward. What we learn from this is that evidence is not timeless, and a solution to our problem of helping researchers design their studies with synthesized evidence must be some kind of synthesis that can respond to paradigm shifts and new evidence with reasonable effort.

1.3.2. Intuition and the Diversity of Study Designs

The pleasure of science is not the least driven by the challenge of finding innovative solutions to complex problems. Many research questions require creative and ingenious study designs and human intuition should not be underestimated in this process. While the goal of this thesis is to bring evidence into the design and evaluation of study designs, we need to clarify two points.

First, intuition has and will have its place also in the design and discussion of future code comprehension experiments. Intuition is what leads us to the hypothesis for which we collect evidence, ideas for future directions to look into, and sometimes the one methodological idea that nobody has thought about before. Evidence will not replace intuition, but it can help a researcher defend their intuition against unwarranted criticism.

Second, the collection of evidence that this thesis aims for is *not* driven by a goal of coming up with the one and only true study design for a specific research question. This line of reasoning may be intuitive, since at some point one would know which design decisions make the most sense based on evidence. Evidence will allow researchers to have meaningful discussions about their study design and how valid it is. However, we should not forget that the validity of a study design is not binary: There is no evidence that would suggest that a study design is valid or invalid. Evidence might tell us

that a research design might be limited in some ways, and sometimes the researcher has to choose which facet of validity is more important to them. This is an intuitive process of compromise, and we will discuss some such scenarios later in this thesis. In the remainder of this section, I would like to make the point that it is even beneficial if different authors make such a decision differently, and that it is important that as many different study designs as possible exist for the same research question.

On the Construction of Knowledge

“ *Science is a human activity, and all human activities are guided by values of some sort. But this might be a desire to understand, a desire to resolve questions about how the world works. If so, that is not ‘value-free.’* ”

— P. GODFREY-SMITH [[God21](#)]

Latour and Woolgar [[LW13](#)] once conducted a field study in which Latour observed scientists working in a molecular biology laboratory over several years. They concluded that observations and raw empirical data themselves would not naturally result in scientific papers. Instead, scientists would make a series of decisions, interpret observations, and support claims with a number of assumptions.

What may sound trivial to the experienced scientist is related to a philosophical view essential to this work: A large part of what we take as scientific facts is knowledge that researchers constructed, rather than natural elements that are just there independent of thought. This can be problematic. When humans are interpreting observations, they are subject to bias, “because observation is affected by the theoretical beliefs of the observer” (this is known as *theory-ladenness of observation*) [[God21](#)]. Further, science is not value-free. A specific study design may be consciously or unconsciously shaped by one’s own beliefs and goals. For example, the decision to take code snippets for an experiment from open-source projects instead of creating them oneself

could be shaped by placing more weight on the generalizability of the study results than on controlling for possible confounding variables, which in turn could come from being driven by the prospect of industrial impact.

This bias, inherent in any primary study design, is a major argument that meta-studies are needed to draw conclusions about the evidence for a hypothesis. In doing so, if enough different study designs by enough different authors contribute to answering the same research question, we argue that bias will become less influential.¹

Accordingly, intuition and the resulting diversity of study designs is of great importance to obtain a comprehensive picture of the complex reality. One question remains: how much may a researcher rebel against existing evidence and follow their intuition?

“ *The only principle that does not inhibit progress is: anything goes.* ”

— P. FEYERABEND [Fey93]

As pointedly as Feyerabend may have formulated the answer, it sums up my view. I have previously argued that evidence helps in discussions of evaluating study designs, and that every bit of evidence contributes to informed debate. I stand by this, and this stance will be apparent in all chapters of the thesis. At the same time, in this and the previous section, we have gained a more nuanced view of what evidence is and that it is neither timeless nor objective. Everyone should therefore be allowed to follow their intrinsically motivated beliefs at any time; “We may advance science by proceeding counterinductively” [Fey93]. Evidence complements our research without replacing anything of intrinsic value. Openness to critical thought and new theories, possibly contrary to any evidence, should remain with us.

¹As a personal concern, it should be noted that this is also a weighty argument that the software engineering research field should consider replication and reproduction studies, especially in empirical research and research with human study participants, to be more important than it currently does.

1.3.3. Validity

Since throughout this thesis we will be discussing threats to validity, even categorizing them comprehensively in [Chapter 3](#) and investigating their evidence in [Chapter 6](#), we should establish in advance what this is all about.

In everyday language, *validity* can denote “the quality of being well-grounded, sound, or correct” [[Merb](#)]. In this work, validity is considered a multi-faceted construct, and the degree of validity of a study design can be assessed separately for each facet. The facets into which validity is usually divided depend on the underlying research method of a study [[PG13](#)]. In controlled experiments in software engineering, a classification into internal, external, construct, and conclusion validity in the reporting of validity threats has become widely accepted. This classification was particularly successfully promoted by Wohlin et al. [[WRH+12](#)] and Jedlitschka et al. [[JCP08](#)], but in both cases the idea is based on the work of Cook and Campbell [[CC79](#)]. We will use this classification and define each facet in [Chapter 3](#) when we work with them for the first time.

In the context of research philosophy, validity has a direct relationship to the previous two sections: both intuition and evidence are necessary for its evaluation. Evidence about which threats to validity affect a particular study design should complement the researcher’s intuition and be cited in the reporting of threats to validity. If authors of primary studies already cite evidence in their discussions of threats to validity, secondary studies can also benefit: contradictory findings can be better explained, and more informed and nuanced conclusions can be made.

In principle, researchers should aim for study designs that have a high degree of validity in those facets that are most relevant to the researcher’s particular principles and goals. “Some ways of increasing one kind of validity will probably decrease another kind” [[CC79](#)], making this form of prioritization often necessary. As a consequence, a study design will always have limitations and inherent bias (see [Section 1.3.2](#)), which is why a single primary study will theoretically never be sufficient to satisfactorily shed light on a complex research question. With a sufficient variety of study

designs and authors on the same question, a holistic and less biased picture eventually emerges.

1.3.4. Measuring the Unobservable

One reason that designing code comprehension studies is so difficult is that we cannot measure how well somebody understood a certain piece of code *directly*. Code comprehension is a *construct*, a concept created by humans to classify and assign meaning to observed behaviors. If such a meaning can be agreed upon within a research community, a construct offers added value (see [RT18a; SB22] for an introduction with examples from software engineering).

Imagine the following scenario. A developer, Sofia, is known for being able to see any codebase that is previously unknown to her, and add new functionality to the codebase on the same day. In contrast, developer Favian sees himself as someone who usually takes a little longer to become proficient enough in an unfamiliar codebase to extend it in a meaningful way. We find that there are more developers like Sofia and more like Favian, all of whom can be assigned to one of these two groups or something in between. To theorize about the behaviors and discuss the phenomenon, we define the construct *code comprehension*: Sofia is better at code comprehension than Favian, or, if we do not emphasize performance differences in our theory, we could say Sofia understands code differently than Favian.

We can measure a developer's code comprehension, as in Sofia and Favian's scenario, by the time it takes them to add functionality to previously unknown code. Numerous other ways to operationalize the construct exist, however, and we explore the variety in detail in [Chapter 3](#). In the context of research philosophy, we want to highlight one point in more detail, as it is relevant to this thesis.

There are researchers who would at best cautiously agree with the opening statement that code comprehension cannot be measured directly. The underlying assumption is that there is a physical representation of what we mean by code comprehension that can be directly observed or measured,

though perhaps not yet with the methods we know. There would then be a kind of biomarker, so to speak, that could be used to *objectively* determine how well someone understood code. Indeed, this may be less science fiction than it is a combination of a certain philosophical view of what constitutes a mental state and the underlying definition of code comprehension. Some contemporary code comprehension researchers, looking for example at the brain activity of developers, seek direct access to comprehension right there, in the brain and its activity. Modern technology like fNIRS and eye tracking would promise “direct and objective measures of comprehension” [Fak18] as opposed to traditional methods of self-reporting or summarization tasks that only serve as “indirect measures” [Fak18].

This view mixes two aspects. One is the hope for direct access to comprehension and thus, most likely, to the mind of humans. A discussion of this would open the Pandora’s box of *philosophy of mind* [Jaw11; Kin20], which we refrain from doing here for pragmatic reasons. We can simply accept in the context of this work that some believe code comprehension is something *physical*, i.e. something that occurs in nature, and that others believe it is a conceptual idea by which different researchers potentially understand different things. The latter is true for me.

The second aspect is whether there are ‘objective’ or ‘more objective’ measurement methods for how well someone has understood code. This is the question we need to address at this point, as in [Chapter 5](#), we recommend certain measures based on evidence from our own studies, which are referred to within the community as ‘objective measures’ (in contrast to ‘subjective measures’). It is important to emphasize that ‘objective’ in this context does not refer to whether understanding is (believed to be) measured directly. Objectivity refers to the absence of bias that influences either the study participant or the researcher in the measurement of code comprehension. We aim to make reliable statements about a developer’s understanding of code that do not depend on factors other than the understanding itself. For example, we can imagine that developers’ self-assessments of how well they understood a particular code snippet depend on their current confidence in themselves or the standing of the person to whom they are reporting. Self-

assessments could be subject to bias and hence less objective. We provide more examples and evidence for similar hypotheses in [Chapter 5](#).

Overall, it is simpler to agree that we aim to obtain reliable (i.e., *objective*) measurement results than it is to agree on the nature of code comprehension. We will nevertheless present in [Chapter 2](#) a framework that attempts to explain code comprehension at a conceptual level. Generally, we would argue that the authors of every primary study should at least define, better yet try to explain, what they mean when they talk about code comprehension. Today, this is still the exception, as we will see, and an implicit definition by task prevails: code comprehension is what the experimental tasks measure. However, this makes it difficult to conduct meta-studies, since it is not clear whether different primary studies are intended to measure the same thing. Further, the issue for authors of primary studies is that without an explanation of what one is actually trying to measure, it becomes difficult to justify design decisions. And it is precisely these justifications that we seek to support in this thesis.

1.4. Contributions

The main contributions of this work can be summarized as follows:

1. We provide a systematic review of the state of the art in designing code comprehension experiments and of the more than 40-year history of this research field. This provides the basis for our subsequent discussion of the considerable diversity of design options in the face of a lack of basic research on their consequences and comparability. We highlight what, we believe, are the five most important action items that the code comprehension research community should address moving forward.
Relevant chapter: [3](#) | Relevant publication: [[WBW23](#)]
2. We conduct primary research and provide empirical evidence on the influence of intelligence facets and personality traits on a developer's performance in understanding source code.
Relevant chapter: [4](#) | Relevant publication: [[WW22](#)]
3. We conduct primary research and provide empirical evidence on the influence of hints about the difficulty of a code snippet on a developer's perception of the code's understandability.
Relevant chapter: [5](#) | Relevant publications: [[WMG22a](#); [WPGW21](#)]
4. We show that evidence profiles can be used to evaluate design decisions in scientific studies in an evidence-based way. For the three most frequently discussed threats to validity in code comprehension experiments, we create evidence profiles and discuss their implications.
Relevant chapter: [6](#) | Relevant publication: [[MWGW23](#)]
5. We publish data for all of our studies on which this thesis is based on Zenodo. The data will allow future studies to perform additional analyses, replicate findings and reproduce our studies.
Relevant chapter: [Appendix B](#)

1.5. List of Publications

The contributions of this work have been introduced into the scientific discourse in the form of five publications, one of which is currently undergoing the peer-review process:

1. **Wyrich, M.**, Bogner, J., & Wagner, S. (2023). 40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study. *ACM Computing Surveys* (to appear). [[WBW23](#)]
2. Wagner, S., & **Wyrich, M.** (2022). Code Comprehension Confounders: A Study of Intelligence and Personality. *IEEE Transactions on Software Engineering*, 48(12), 4789-4801. [[WW22](#)]
3. **Wyrich, M.**, Preikschat, A., Graziotin, D., & Wagner, S. (2021). The mind is a powerful place: How showing code comprehensibility metrics influences code understanding. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (pp. 512–523). IEEE. [[WPGW21](#)]
4. **Wyrich, M.**, Merz, L., & Graziotin, D. (2022). Anchoring Code Understandability Evaluations Through Task Descriptions. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)* (pp. 133-140). [[WMG22a](#)]
5. Muñoz Barón, M., **Wyrich, M.**, Graziotin, D., & Wagner, S. (2023). Evidence Profiles for Validity Threats in Program Comprehension Experiments. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (pp. 1–12). IEEE. [[MWGW23](#)]

The Wagner and Wyrich [[WW22](#)] paper was initiated by the first author. Together, we designed the study, conducted the study, analyzed the data and reported the results. The paper by Muñoz Barón et al. [[MWGW23](#)] is the result of a Master's thesis written by the first author, who I supervised. Already during the realization of the master's thesis, I was involved in parts of the data collection and data interpretation. Together with the other co-authors, we extended the thesis into a paper.

1.6. Outline

The thesis is structured such that each of the chapters is self-contained, so that one can jump into any chapter depending on one's interests and previous knowledge. Each chapter begins with a reference to the overarching research context to fit into the continuous storyline of this thesis. The only exception is [Chapter 7](#), in which the results of all previous chapters are discussed in the larger context of the thesis, and for which it is therefore advisable to have read at least [Chapter 6](#) beforehand. The contents of the work are organized as follows:

[Chapter 2](#) can be seen as a general background chapter that introduces the reader to the research area of code comprehension. The contents form the scientific foundation for all subsequent chapters.

[Chapter 3](#) extends this foundation with a systematic mapping study that analyzes in detail design characteristics of code comprehension experiments and concludes from the results how the research field may evolve in the future. One key finding is the lack of support for evidence-based study designs.

Chapters [4](#) and [5](#) present three primary studies that investigate the influence of individual characteristics and contextual factors, respectively, on code comprehension. The evidence obtained from all these studies can help in designing future code comprehension experiments.

[Chapter 6](#) contributes considerably to answering the primary research question of how evidence can be used in the design of code comprehension experiments. We will introduce the methodology of evidence profiles and demonstrate their practical use in synthesizing evidence from primary studies to support researchers in designing future code comprehension experiments.

[Chapter 7](#) summarizes all contributions and critically examines how this thesis advances the code comprehension research field. We address the research questions introduced in [Section 1.2](#) and turn to open challenges and future research directions before we conclude the thesis in [Chapter 8](#).

CHAPTER



SOURCE CODE COMPREHENSION

The cognitive process of code comprehension is still fairly unexplored. Researchers in the field have a rough idea of what is meant when talking about comprehensible code or performance in code understanding. However, a definition of the construct, let alone a comprehensive cognitive model, has not yet been established to the extent that primary studies would rely on it in their design. As a result, what is measured is often implicitly defined by how it is measured, e.g., by the number of correctly answered questions about a previously considered C function or the time required to find a bug.

Nonetheless, there are starting points on which we can build to define the scope of this work. For example, ISO/IEC 25010 describes a quality model for defining individual quality characteristics of a software product. The quality attribute most related to code comprehensibility in this model is perhaps *analyzability*, which is defined as: “Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose

a product for deficiencies or causes of failures, or to identify parts to be modified” [ISO11]. This definition has some overlap with frequently used participant tasks in code comprehension studies. However, it remains at a level of granularity that does not explicitly address that source code may need to be understood for all described intentions and what it means to have understood source code.

Boehm et al. [BBL76] provide a more concrete definition of *understandability* in this regard in their early work on quantitative evaluation of software quality: “code possesses the property of understandability to the extent that its purpose is clear to the reviewer. This implies that variable names or symbols are used consistently, code modules are self-describing, control structure is simple or conforms to a prescribed standard, etc”. The first sentence of Boehm et al.’s definition is one that can serve us as a basis for discussion in the following lines. The understandability of source code can be measured by how clear the purpose of the code is to the reader. The second part of the definition provides some examples of code characteristics that would help improve comprehensibility. These are common hypotheses that are the subject of code comprehension experiments, whose design we want to support with evidence.

We consider *comprehensibility* and *understandability* as synonyms, i.e., choosing one over the other is “purely a matter of linguistic variation” [Kin98]. Note that both terms describe a property of the code. However, studies that intend to provide insights into the influence of a treatment on the comprehensibility of code usually first measure *code comprehension*, i.e., how well a participant understood source code under certain conditions. The data collected can then be used to draw conclusions about the comprehensibility of the code.

So, what does *code comprehension* mean? The landscape of prominent definitions for code comprehension is fairly limited to the early days of the research field. At that time, the term *program comprehension* was mainly used, which today covers a broader range of research topics, of which code comprehension is one [WBW23].

For Shneiderman [Shn77], for example, program comprehension is “the

recognition of the overall function of the program, an understanding of intermediate level processes including program organization and comprehension of the function of each statement in a program”. Similarly, Letovsky and Soloway [LS86] state that “the goal of program understanding is to recover the intentions behind the code” and Pennington [Pen87] elaborates that “comprehension involves the assignment of meaning to a particular program, an accomplishment that requires the extensive application of specialized knowledge”.

These basic ideas that understanding code means being able to see through and grasp the semantics of the code are not very controversial, neither then nor today. Such definitions have since provided a common research foundation for primary research. Yet, it is also clear that the discussion about the meaning of code comprehension cannot end here because a proper *explanation* of what code comprehension could be had not been provided so far.

For example, a valuable contribution regarding our goal of better understanding what code comprehension is comes from Gilmore, who distinguishes between the comprehension process and the state of comprehension:

“ This paper will make a distinction between the comprehension process and the state of comprehension. The former involves the mobilisation of cognitive resources and processes in some particular configuration, with the goal of constructing some mental representation of the program code. It is this mental representation of the code which is the comprehension state. The possibility that the cognitive processes may be used in different ways in order to achieve a comprehension state gives rise to the importance of comprehension strategies.

— D.J. GILMORE [Gil91]

In the context of this work, both the comprehension process and the state of comprehension are relevant. If we could better understand the comprehension process, we could, for example, design comprehension tasks and measures in experiments in such a way that they can be justified with validated code comprehension models and thus strengthen our confidence that the tasks actually measure code comprehension. The state of comprehension is relevant at the moment when we attempt to measure how well somebody has understood code, since most common code comprehension measures assess the comprehension of a study participant at one given point in time.

The remainder of this chapter is organized as follows. In [Section 2.1](#) we will review elements of program comprehension strategies. Knowledge about these strategies helps us to better define the scope of this thesis and to come closer to an explanation of the construct of code comprehension. We will learn, moreover, that the strategy itself has an impact on a developer's state of comprehension. [Sections 2.2](#) and [2.3](#) describe, respectively, research on developer characteristics and contextual factors that have been shown to influence code comprehension. [Section 2.4](#) will briefly discuss the role of a construct definition for the reasonable measuring of code comprehension. In [Section 2.5](#), we bring together all these aspects from sections 2.1–2.4 and create a conceptual model for code comprehension experiments. From this model, we highlight the focus of this thesis and derive some propositions about what code comprehension can be and what meaning different comprehension measures have within this model.

2.1. Program Comprehension Strategies

We travel a bit in time, more precisely to the 1980s, a decade that introduced almost all the relevant concepts that are still shaping our current picture of program comprehension. Program comprehension research of that time is in many aspects comparable to contemporary research. However, the period was particularly characterized by the development of models and theories to explain behavioral processes of developers. Although many of the articles

of that time resemble essays in which new ideas appear to be supported more argumentatively than empirically, experiments to test the hypotheses presented were already common at that time.

We start in 1978, when Ruven Brooks [Bro78] lays the first milestone of our journey with an essay about a behavioral theory of program comprehension. He assumes that developers switch between ‘knowledge domains’ in program understanding, which, roughly summarized, represent different levels of abstraction of reality. According to Brooks, a developer has understood a program when not only information about objects and their relationships within one abstraction level are known, but also their relationship in a nearby abstraction. Brooks’ most influential idea, however, was that the process of understanding is characterized by the “successive refinement of **hypotheses** about the program’s operation” [Bro78]. If a developer takes a particularly long time to understand a certain code snippet, this can be explained by the difficulty of finding correct hypotheses at that moment. Brooks already speculated at this time that there were elements in the program, such as comments or variable names, that represented *cues* in understanding the program.

Five years later, in 1983, Brooks [Bro83] elaborated his thoughts and the model that we know today as the ‘**top-down model**’ of program comprehension emerged: Hypotheses about knowledge domains are generated, tested, and refined until their relationship to the code becomes apparent. A hypothesis describes for example the basic function of a component. The first hypothesis is generated as soon as the developer receives the first information about the program; how appropriate this hypothesis is depends on the skill level and the experience of the developer with the problem domain. The verification and the establishment of alternative hypotheses are supported by certain indicators in the code. Brooks calls these indicators ‘**beacons**’ here for the first time.

Between these two publications by Brooks, in 1979 Shneiderman and Mayer [SM79] came forward with their model for programmer behavior. In their work, they did not limit themselves exclusively to program comprehension. Five programming tasks were studied, one of which constitutes

comprehension. Nevertheless, we can also call this work a seminal work because it resulted in what we know today as the ‘**bottom-up model**’ of program comprehension: “Instead of absorbing the program on a character-by-character basis, programmers recognize the function of groups of statements, and then piece together these **chunks** to form even larger chunks until the entire program is comprehended”. Shneiderman and Mayer [SM79] further emphasized the role of syntactic knowledge (programming language dependent) and semantic knowledge (general programming concepts) in a developer’s long-term **memory**, as well as the role of short-term and working memory for the construction of a “multileveled internal semantic structure to represent the program [. . .] The central contention is that programmers develop an internal semantic structure to represent the syntax of the program, but that they do not memorize or comprehend the program in a line-by-line form based on the syntax”.

Both Brooks [Bro83] and Shneiderman and Mayer [SM79] have supported their presented concepts with initial experiments, and it did not take long for independent studies to follow that put the models to the test. Basili and Mills [BM82] observed themselves comprehending and judging the correctness of a program, providing early anecdotal evidence that bottom-up comprehension practically takes place. Adelson [Ade81] found experts to have larger recall chunks than novices, and that expert chunks contain more semantically rather than syntactically related information. Wiedenbeck [Wie86] conducted two experiments on the differing influence of beacons on experienced and inexperienced developers, and found that supposedly useful indicators in the code primarily help experienced developers understand it. Wiedenbeck discusses the role of beacons and considers them to be in line with the bottom-up comprehension strategy, since beacons can represent “the link between the top-down hypothesis formation stage and the data of the program text”. Moreover, what Shneiderman calls a chunk could also represent a beacon, provided that the chunk represents a stereotypical part of the program [Wie86]. Soloway and Ehrlich [SE84], at the same time as Brooks [Bro83], investigated top-down comprehension assumptions and provided evidence that *programming plans* and *rules of programming*

discourse would help experienced developers with program comprehension. Programming plans represent stereotypical program fragments, similar to a kind of beacon. Rules of programming discourse represent intuitive expectations of developers, e.g. that a function name matches the content of the function [SE84]. Rist et al. [Ris+86] refinded Soloway and Ehrlich's plan idea and provided additional experimental evidence for its positive influence on program comprehension.

In 1987, the focus of research seemed to shift minimally from cognitive comprehension strategies to the nature of mental representations of source code (mental models).

Pennington [Pen87] shows with an example of a code snippet that a program can be abstracted in at least four different ways: (1) by the goals of the program, (2) as a data flow abstraction, (3) as a control flow abstraction, and (4) of 'conditionalized action', i.e. under certain conditions the program performs actions and enters another state. Pennington calls the control-flow representation '**program model**' and the combination of data-flow and goal hierarchy representation '**situation model**', drawing on theories from earlier work in the field of text comprehension [VK83]. In her first study, Pennington shows that procedural (control flow) rather than functional units (goal hierarchy) "form the basis of expert programmer's mental representations", which implies that even experts initially apply bottom-up comprehension strategies [Pen87]. Her second study addresses the limitation of small code snippets. Once again, an understanding of program control flow and procedures preceded an understanding of program functions.

Letovsky [Let87] propose a cognitive model that is about a 'knowledge-based program understander' who is made of three components, namely a knowledge base, a mental model of their current and evolving understanding, and an assimilation process interacting with the stimulus materials. His position is that different types of knowledge play the central role in the comprehension process and that the understander "is best viewed as an opportunistic processor capable of exploiting both bottom-up and top-down cues as they become available". Letovsky elaborates on some details and examines his assumptions using video recordings of six professional developers

who were asked to add a new feature to a project while thinking out loud. What is interesting for us at a later point of this chapter is what Letovsky believes a mental model should contain: a description of the goals of the program (specification), a description of actions and data structures in the program (implementation), and an explanation of the links between goal and implementation (annotation) [Let87].

Finally, Littman et al. [LPLS87] distinguish between static knowledge (what the program does functionally) and causal knowledge (interactions of functional components during execution). Both types of knowledge are required for comprehension, which is why they call a mental model that contains both types of knowledge a ‘strong mental model’. In contrast, a mental model of the program that only contains static knowledge is called a ‘weak mental model’. They found some developers to use a *systematic* comprehension strategy, i.e. “the programmer performs extensive symbolic execution of the data flow paths between subroutines”, and some to use an *as-needed* strategy, meaning that they attempt to only focus on those parts of a program relevant for the experimental task, which resulted in either a weak or a strong mental model. As a consequence, developers following the systematic strategy were more successful at the requested modification task [LPLS87]. Koenemann and Robertson [KR91] provided further evidence a few years later that developers follow an ‘as-needed’ strategy when they need to understand or modify a program, and that *need* is primarily based on the developer’s goals. They further conclude that program comprehension is mainly a top-down process and bottom-up is used “in cases of missing or failing hypotheses and locally for directly relevant code units” [KR91].

The 1990s and the Integrated Model

We see that there has been neither a lack of concepts that contribute to understanding program comprehension nor a lack of empirical investigations of them. In the 1990s, we also find good examples of work that derives implications for practice based on the basic research on program comprehension described so far. For example, Storey et al. [SFM99] address the

design of software exploration tools to support developers and use theory to explain what design principles the tools would need to follow to facilitate the construction of a mental model for program understanding.

What was missing, however, was some kind of metamodel that coherently integrated the various concepts of the 1980s. Von Mayrhauser and Vans [VV95] took on this task and in 1995 set another milestone in our history of program comprehension strategy models with their “integrated metamodel”.

Figure 2.1 depicts the metamodel. It is a re-creation of the graphical illustration of Von Mayrhauser and Vans, where we have simplified the knowledge base in its level of detail to make the illustration more concise and understandable. Apart from that, all components are included that are also contained in the original model: The top-down model based on Soloway and Ehrlich [SE84], the program model and situation model based on Pennington [Pen87], and the knowledge base, as the node that contains the information to build and switch between the respective cognitive process models. The integrated metamodel also incorporates elements from the work of Brooks [Bro83], Letovsky [Let87], and Shneiderman and Mayer [SM79], so that we find again, e.g., the concepts of beacons, memory, chunking, and various kinds of knowledge. All five incorporated models also describe a matching process between the developer’s knowledge and the object to be understood (‘match comprehension process’), for example through the systematic verification of hypotheses according to Brooks [Bro83] or via a more opportunistic approach according to Letovsky [Let87].

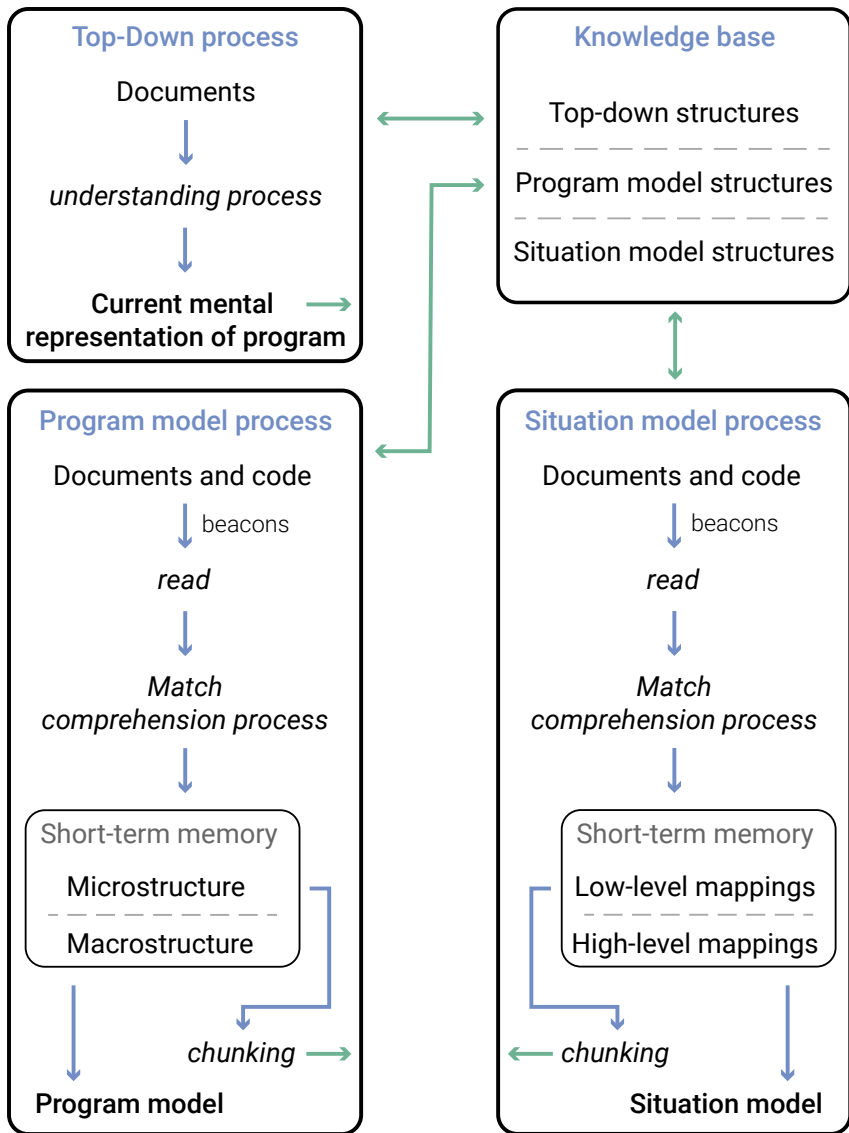


Figure 2.1.: The “integrated metamodel” by Von Mayrhauser and Vans. Figure based on Figure 6 in [VV95].

Von Mayrhauser and Vans [VV95] then summarize in their paper which components of their model and its underlying theory have been empirically investigated up to that point. Thereby they make the following remark, which will be of particular importance in [Chapter 3](#) of this thesis:

“ *Many experiments are designed to measure specific conditions (for example, whether or not programmers use plans) but the experimental hypotheses (for example, that programmers use plans when understanding code they have never seen before) are not always based on a well-defined program-comprehension theory.* ”

— VON MAYRHAUSER AND VANS [VV95]

They consider the development of program comprehension models to be “in the theory-building phase” and theories on large-scale program comprehension to be “in their infancy” [VV95]. Von Mayerhauser and Vans could not have known that the end of the 1990s also marked the end of two decades of research that produced the most influential models of program comprehension to date. Is the theory behind program understanding still in its infancy today? Very likely, yes. We will see in a moment that there were some notable research contributions and efforts to better understand program comprehension in the 2000s and the 2010s as well. Code comprehension experiments, however, are still not designed based on a lot of theory, or only to a minimal extent because too little can be explained with existing theory. We will return to this issue in [Chapter 3](#).

Before we turn to the progress towards theory building in the 2000s, we would like to highlight an aspect that was increasingly made explicit in papers of the 1990s and seems still a consensus today [MTRK14]: program comprehension is usually not an end in itself, but a (necessary) means to accomplish a maintenance task successfully. We might do well, therefore, to also develop models that are specific to particular development activities and incorporate program comprehension as one aspect of such activity.

Gilmore [Gil91] presents an example of such a model that connects the debugging and comprehension processes. According to Gilmore, the view that a mental model resulting from the comprehension process would be used in a subsequent and independent step by the developer for the debugging activity is not accurate. Instead, he argues that these are two distinct processes, of which comprehension is part of a larger debugging loop: “The fact that, in both cases, bugs were more easily found when they were deeper in the structures contradicts the traditional interpretation of propositional analyses of comprehension. Both groups of programmers were more likely to find the deeper bugs, suggesting that they were being detected during an ongoing comprehension process, rather than by comparison with a comprehension state” [Gil91].

Gilmore’s resulting model is shown in [Figure 2.2](#). Besides embedding the comprehension process in a higher-level debugging activity, it also models the interesting aspect of problem comprehension and places mismatch detection between the mental representation of the program and the mental representation of the problem at the center of debugging.

We will see in [Section 3.2](#) that debugging and maintenance tasks are used in code comprehension experiments, but constitute a minority in the choice of experimental tasks. It is much more common in code comprehension experiments to try to isolate the comprehension process, which is why we will not discuss the broader task context in more detail at this point. However, we will go into quite some detail about contextual factors in general that have an impact on code comprehension (see [Section 2.3](#) and [Chapter 5](#)).

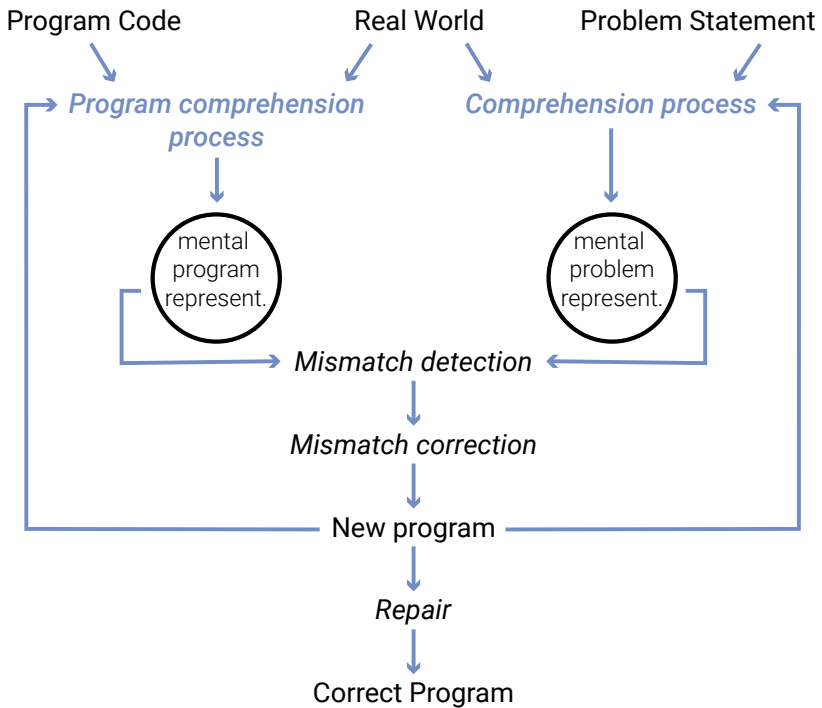


Figure 2.2.: The “schematic view of debugging” by Gilmore. Figure based on Figure 4 in [Gil91].

The 2000s: Model Refinements, Model Extensions, and... Teaching

We enter the decade in which the International Workshop on Program Comprehension transitioned into a full conference. The research field has matured and continued to grow in importance. In the context of program comprehension strategies and theory, there were two things happening in the 2000s: existing models were refined or extended, and a new branch of research emerged that dealt with the education of program comprehension.

Crosby et al. [CSW02] conducted further research on the role of beacons. Among other methods, they used eye-tracking to see what developers fixated

on and for how long. They found that there are at least two types of beacons. One is single lines of code, which contain mnemonic cues about functionality, and the other is more complex beacons, which contain cues about main goals of the program. Also, experts seem to rely on beacons, while novices do not.

At the same time, Rajlich and Wilde [RW02] discuss the role of **‘concepts’** for understanding and learning about unknown and large programs. They define concepts as “units of human knowledge that can be processed by the human mind (short-term memory) in one instance”. Examples of concepts can be single features of the software, which can be found as such in the code. There may be an overlap here with the concept of complex beacons described by Crosby et al. [CSW02] and there is definitely some relationship to the idea of chunking parts of the code into larger segments [SM79]. In any case, Rajlich and Wilde [RW02] consider domain concept knowledge essential for program comprehension.

Building on the works of Brooks [Bro83] and Soloway and Ehrlich [SE84], O’Brien et al. [OBS04] refine the idea of top-down comprehension into expectation-based and inference-based comprehension, where the difference can be roughly summarized as whether beacons [Ris+86; Wie86] in the code confirm or trigger a developer’s (pre-generated) hypotheses. In an experiment with eight industrial developers, they were able to show empirically that for expert programmers both strategies play a role together with the bottom-up comprehension strategy. The preferred approach depends on the familiarity with the application domain [OBS04].

Guéhéneuc [Gué09] extends existing theories on program comprehension by adding vision science, i.e. the *recognition* of items during program comprehension. The work brings an interesting new dimension into play: modality. In the code comprehension experiment landscape, this aspect has so far been somewhat neglected, perhaps because a homogeneous picture of the *code-reading* developer prevails in the minds of the researchers, perhaps also because modality is assumed to have little influence on the comprehension of information in general [RCT16]. We will discuss individual developer characteristics in the following [Section 2.2](#), including how blind developers

understand source code.

In addition to these theoretical extensions and refinements, the theory of program comprehension strategies of the 80s and 90s also increasingly found its way into teaching. A workshop working session on teaching program comprehension was organized with the goal of establishing “a repository of teaching resources, to identify the lessons learned and future directions” [DFKR03]. Then, for example, Exton [Ext02] reviews existing program comprehension strategies and its relation to the constructivist learning theory, and Schulte [Sch08] proposes a model of core aspects of understanding a program text that should guide instructors in designing teaching lessons. The interested reader will find a good introduction to how program comprehension models can enrich the computer science educational perspective in Schulte et al. [SCT+10].

The 2010s: What Happens in Practice

A few refinements of theoretical program comprehension models also entered the scientific literature in the 2010s [BDA14; NP15; SAA+15], but the decade was marked in particular by studies that examined program comprehension in practice.

Maalej et al. [MTRK14] conducted a mixed methods study in which they observed and interviewed 28 software professionals and surveyed 1477 others about their comprehension strategies, the nature of the knowledge relevant to understanding, and the use of comprehension tools. The work is a goldmine of qualitative and quantitative findings. Some of the key insights are that developers use a recurring, structured comprehension strategy depending on the task context, the starting point for comprehension depends on experience, hypotheses are indeed generated and tested as described by the theoretical models, and that developers sometimes take notes to reflect on their mental model. Source code is more trusted than documentation, and naming conventions and common architecture simplify program comprehension. Maalej et al. [MTRK14] also found that dedicated program comprehension tools are not used. Developers would instead prefer to use

basic tools such as compilers and text editors for program comprehension.

Minelli et al. [MML15] suggest that about 70 percent of a developer's time is spent on program comprehension. They quantitatively examined an IDE interaction dataset of 740 development sessions from 18 developers (mouse and keyboard events together with contextual information). A development activity was defined as a sequence of mouse and keyboard sprees. Development activities were then categorized into understanding, navigation, editing, and UI interactions, where comprehension consisted of basic understanding, inspection, and mouse drifting (i.e. mouse-supported reading). The results show that by far the most time is spent on understanding (70%), followed at a large distance by UI interactions (14%), editing (5%), and navigation (4%), with the remaining 7–8% being time spent outside the IDE. The authors highlight the importance of not getting interrupted during the development activity, as it is for the most part characterized by mental processes.

On a side note, both Maalej et al. [MTRK14] and Minelli et al. [MML15], arrive at a similar picture of program comprehension as we discussed earlier, namely as an embedded activity. They conclude that “program comprehension is considered a necessary step to accomplishing different maintenance tasks rather than a goal by itself” [MTRK14] and “that base understanding is prevalent inside activities, that is, inside conceptually related sequences of keyboard or mouse sprees. In other words, the process of program understanding is not really an activity per-se, but it is interleaved with other activities like editing” [MML15].

Just three years after Minelli et al.'s research, the key message that program comprehension accounts for the majority of developmental activity was confirmed by Xia et al. [XBL+18]. They conducted a large-scale field study with 78 software professionals from two IT companies in China and found that developers spend about 58 percent of their time on program comprehension activities. Since, according to the authors, program comprehension does not only take place in the IDE, Xia et al. [XBL+18] tracked the activities of developers across different applications. They were able to show that only about 20% of program comprehension takes place in the IDE,

27% in web browsers and about 10% in document editors. There is frequent switching between these applications.

The 2020s: Dawn of the Neuro Age

Up to this point, we have observed that much has been done in terms of fundamental theories around program comprehension, especially in the 1980s and 1990s, but these efforts have since given way to other efforts, such as integrating the theories into teaching or comparing them to practitioners' experiences. Toward the middle and end of the 2010s, a new trend emerged that we predict will dominate the 2020s and may revive basic research: the application of psycho-physiological measurement methods in the context of code comprehension.

While its realization brings some new challenges, the idea is simple to explain: Researchers use eye tracking, functional magnetic resonance imaging (fMRI), functional near-infrared spectroscopy (fNIRS), electroencephalography (EEG), or combinations of these devices to gain insights into the cognitive processes associated with software engineering tasks (neural correlates) to better understand and support such software engineering tasks [SHLW21; SPB+20; VF21]. A recent systematic mapping study in software engineering found that most papers that make use of psycho-physiological data explore program comprehension and debugging strategies [VF21].

Except for eye tracking, which was already used in program comprehension studies two decades ago (e.g. [CSW02]), the number of studies using, for example, fMRI and fNIRS to study program comprehension is now also increasing. Some position papers promisingly outline the possibilities of gaining more objective insights into program comprehension with neuroimaging devices [Fak18] or even an entirely new, neurocognitive perspective on program comprehension [Pei18; SPB+20]. However, this development of the research field is still at a stage where some researchers would agree that the achieved reliability of the related research setups is probably the greater achievement to date than initial findings on brain activities in program understanding [SPB+20]. In a recent Dagstuhl seminar on the use of

biometrics and neuroimaging devices in software engineering, the methodological challenges of the new technologies were still clearly the focus of discussion¹.

Nonetheless, there are initial findings from psycho-physiological research that strengthen and expand our understanding of program comprehension.

Siegmund et al. [SKA+14] conducted the first fMRI study in which participants had to understand code while positioned in a scanner. Measured blood oxygenation levels can be used to determine brain regions that are active at the time of a code comprehension task. To identify the regions that are responsible for comprehension in isolation, participants also had to perform a contrast task of finding syntax errors. The data show increased activity in several Brodmann areas during code comprehension. The authors identified five of these as relevant for code comprehension (BAs 6, 21, 40, 44, 47), since it is known from previous studies that these areas are related to working memory, attention, and language processing. The findings form a basis for the development of a cognitive model of bottom-up code comprehension that could enrich the previously discussed theory [SKA+14].

Further studies followed that at least partially confirmed these initial findings and provided some additional insights [FSW17; LMH+16; PSP+18; SPP+17]. For example, another study by Siegmund et al. [SPP+17] showed that the presence of semantic beacons in code leads to lower activity in the relevant brain regions during comprehension, and Floyd et al. [FSW17] found that developer expertise affects how accurately a classifier can distinguish code and prose review based on brain activity. Also, worth mentioning is Peitek et al.'s study [PSP+18], in which, for the first time, fMRI and eye tracking were tested in combination to investigate code comprehension. Such a combination has the advantage that fast cognitive subprocesses can be captured due to a higher temporal resolution of the eye tracker. This allowed them to show, for example, that fixations on a beacon leads to semantic recall [PSP+18]. We refer the interested reader to Vieira and Farias's mapping study [VF21] as well as to the introduction of the paper by

¹The thesis author was present at the Dagstuhl seminar: <https://www.dagstuhl.de/de/seminars/seminar-calendar/seminar-details/22402>

Sharafi et al. [SHLW21] for further pointers to recent psycho-physiological studies in software engineering.

Concluding Remarks on Program Comprehension Strategies

We would like to conclude our time travel through the literature on program comprehension strategies and theory with two notes before we continue to build on the findings right away in [Section 2.5](#).

First, program comprehension theories have adopted some of their core elements from the somewhat older research field of reading and text comprehension [PC15]. For example, the situation model, as well as bottom-up and top-down comprehension, are concepts that existed before they were introduced into the program comprehension context. Pennington [Pen87] is, to our impression, the only one who really emphasizes that some of her inspiration comes from theories of text comprehension. While the appreciation for our related field of research leaves much to be desired, one should also not immediately fall into the belief that findings from reading comprehension studies can be directly transferred to code comprehension. For example, we know that code reading and text reading have different eye movement patterns [BBB+15; PSA20] and that developers with and without dyslexia can probably understand code equally well (which is not true for text) [MB19]. However, while the evidence on the comparability of text and code comprehension is otherwise sparse, differences can already be identified at the conceptual level: for example, one element of code comprehension could be that the developer understands what the code is supposed to do and compares this with what the code actually does for certain input values. This matching of a specification with dynamic execution behavior is an element of code comprehension that is hard to find in comparable form in text comprehension. So while we should always be open to learning from the related discipline of text comprehension and should continue to empirically investigate comparability in the future, we probably do well to further develop our own models of code comprehension.

Second, this section already helps us a lot to better define the scope of this

thesis. We will focus on supporting studies that investigate the bottom-up code comprehension process. [Chapter 3](#), for example, systematically examines primary studies that have enforced such bottom-up code comprehension via their task design. Primary studies usually do this to achieve high internal validity by focusing on a particular aspect of the comprehension process. However, we know at this point that bottom-up comprehension is only part of the overall comprehension process, which in practice is part of a higher-level activity such as debugging. Program comprehension today also involves much more than understanding source code; for example, developers need to gain an overview of the architecture, or identify developers who are responsible for a particular component to solicit their help if necessary [[Sie16](#)]. All these aspects make it important, in our opinion, to distinguish between code comprehension (the focus of this thesis) and the broader concept of program comprehension. Whatever our findings, though, they then also play an important role in the overall process of program comprehension because of the direct dependency.

2.2. The Developer in the Spotlight

Code comprehension is a cognitive psychological process in which, in addition to the characteristics of the code to be understood, the capacities of the person who wants to understand the code play a role. We already know that different types of knowledge are involved in program comprehension (see [Figure 2.1](#)). But is a developer's knowledge base the only characteristic that influences their approach or performance in code comprehension? Already in 1978, Brooks [[Bro78](#)] noted that developer characteristics other than knowledge may play a role in program comprehension.

Even though such individual characteristics were not part of the program comprehension models and theories presented in the previous section, Brooks was not the last to speculate on the influence of individual parameters on program comprehension. Siegmund and Schumann [[SS15](#)] compiled a catalog of confounding parameters discussed in program comprehension studies.

Their survey includes papers from 13 journals and conferences published between 2001 and 2010, and results in 39 confounding parameters. In the category of ‘individual confounding parameters’ we find 14 variables that relate to the person who needs to understand the program and that other researchers in the field assume have an impact on program understanding. These variables are: color blindness, culture, gender, intelligence, ability, domain knowledge, education, familiarity with study object, familiarity with tools, programming experience, reading time, fatigue, motivation, and treatment preferences [SS15].

We can well imagine that these variables influence code comprehension in one way or another. But is there evidence on their actual influence? In [Chapter 6](#) of this thesis, we address this question by looking for evidence for frequently discussed confounders. At this point, we can already spoil that, depending on the study context, some of the confounding parameters compiled by Siegmund and Schumann [SS15] either do not have the assumed impact on program comprehension, or empirical evidence on them is scarce.¹

Yet explicit research on the presumed influencing factors would be important for at least two reasons: first, researchers could design better experiments because they could make more informed decisions about which confounding factors to control and how to control them. Second, such research would contribute to theoretical models, ultimately better supporting developers in their comprehension strategy. We address the first aspect in detail in this thesis. We will illustrate the second aspect, supporting developers by researching the influence of individual characteristics, with a concrete example in the following lines.

The Stack Overflow Developer Survey 2022 received 73,268 responses

¹Note that for some of these individual characteristics, it may be considered unethical to conduct explicit research on differences in performance, for example, if the goal is solely to investigate whether a particular cultural background or gender accounts for better or worse code comprehension. Such research can rarely be carefully enough framed to avoid inviting others to exclude certain people in certain contexts and to point fingers at supposed evidence. Gender may be among the more commonly discussed confounders [SS15]. However, the real added value of research in this area will likely be to determine the origin of this assumption and whether there are gender differences in code comprehension strategies due to, for example, gender-related hurdles that then need to be addressed.

(73% of participants identified themselves as developer by profession). 2,547 respondents reported having a physical difference, the majority of which indicated being blind or having difficulty seeing (1.7% of all survey participants) [Sta22].

When we theorize about how developers understand source code, does it make a difference whether a developer is sighted or, for example, fully blind? While there is some empirical evidence and some field reports that blind and sighted developers understand code equally well [ARM18; MrV22], blind developers face many hurdles that they must overcome and work around.

Mealin and Murphy-Hill [MM12] interviewed 8 blind developers on challenges they face during software development. With similar intentions, Albusays and Ludi [AL16] collected 69 survey responses from blind developers and, in a follow-up study [ALH17], observed and interviewed 28 blind developers as they navigated a codebase using their preferred coding tool. The three studies came to similar results: all participants regularly use screen readers for development work, some additionally resort to Braille displays, for example, for more details on punctuation. A recurring theme in all studies is the poor accessibility of developer tools, which is why blind developers prefer printf to debugging tools, for example, or copy code snippets into separate scratchpads and editors to be able to navigate through that section of code more easily (it is easier to jump to the beginning and end of the snippet when it is viewed isolated) [AL16; ALH17; MM12]. In general, many features in IDEs are visual supports for which there is no accessible equivalent yet. For example, the usefulness of autocompletion is undisputed among sighted developers, but when such a feature is implemented as a pop-up, it cannot be recognized by most screen readers [ALH17].

Armaly et al. [ARM18] found that blind and sighted programmers overall focus on similar parts of the code. Both groups would read method signatures more often than other parts of the code, and would be less concerned with control flow than other parts of the code.

Yet, fully blind developers must intuitively and evidently deviate from sighted developers in some aspects of their comprehension strategy. It is difficult to get a high-level overview of the code when screen readers work

linearly and braille displays only show a single line of code. Therefore, blind developers rely heavily on API documentation [MM12], which at least raises whether sighted and blind developers have a slightly different idea of what constitutes a helpful beacon. Then, since diagrams are mostly inaccessible [AL16; MM12], knowledge sources in addition to code are limited. Blind developers must also sometimes seek assistance from their sighted peers [AL16; ALH17], raising the as-yet unexplored question of how cooperative code comprehension affects the construction of mental models. Speaking of mental models, the blind outperform sighted peers on serial memory tasks [RSP+07], which may be why they presumably have a more detailed mental model for coding tasks that are difficult to visualize [MM12].

We see that the single characteristic of being blind can have a major impact on the approach to code understanding. At the same time, the aspect of being blind, as well as the influence of numerous other individual characteristics, has hardly been reflected in code comprehension models. For example, personality plays a major role in software engineering [CSC15], but has barely been studied in the context of code comprehension [WW22]. It would be naive to believe that all people understand code in the same way and that differences in performance are solely due to expertise and knowledge. We will therefore map the influence of individual characteristics on code comprehension as part of the experimental variables in our conceptual model (Section 2.5) and provide empirical evidence on the influence of personality and intelligence on code comprehension in Chapter 4 ourselves.

For the moment, we note that experiment participants differ in other individual characteristics beyond their prior knowledge, and that it is important to understand the influence of these differences on code comprehension strategies and performance better than we currently do. The findings then help, as previously explained, in the better design of experiments. Firstly, because one then knows which confounders need to be controlled in which way. Secondly, because experiments themselves can then be designed in such a way that they are accessible to certain groups in the first place, for example in online experiments by inserting code as text instead of as an image so that the code can be read out by screen readers. Further, research findings

benefit the developers themselves eventually: In the context of blindness, corresponding research findings can guide and motivate the development of tools and IDE plugins that use audio, e.g., for improving the accessibility during debugging [PVI+18; SHM+09] and for assisting vision-impaired students in learning how to program [SFM00].

2.3. Context Is Everything

Imagine an IDE without syntax color highlighting, without autocompletion, and without all your beloved plugins that make programming and debugging easier. After reading the previous section, which primarily dealt with the hurdles that blind developers have to deal with, it is probably no longer that difficult to imagine such a scenario. In everyday development, however, the omission of such features would mean major productivity losses for most developers.

Similar to individual characteristics, *contextual factors* such as syntax color highlighting are potential influencing factors on code comprehension, on which there is a little research, but which have hardly been integrated into theoretical comprehension models so far. Yet, that would also be quite worthwhile for reasons that will become evident shortly.

Have you ever tried to understand code in virtual reality? There are now numerous tools that display code in virtual reality or even allow it to be manipulated. The tools make use of various spatial metaphors [AAV+19]: some VR tools display the program as a city [CERS17; FKH17; MLMD01; RCE+19; SKR19], others as an island [SM18], and still others build a virtual desk [DTR+20; OMP18; SMHB21] that can, for example, display multiple screens and exploit the potentially infinite space of a virtual reality.

Most virtual reality tools for visualizing program code are intended to support developers at least in program comprehension (some additionally serve to simplify programming or debugging). The results of empirical studies on the benefits of VR tools vary. A general trend seems to be that developers work or perceive to work slower in VR [CLB20; DTH+20; OMP18], but this

may as well depend on the concrete task [MMV+21; RCE+19]. Among the positive findings is that fewer mistakes are made, and more bugs can be found through appropriate visualizations [DTR+20; FKH17]. However, some studies also observed that compared to using established (non-VR) software visualization tools, there are no significant performance differences when solving comprehension or related programming tasks [MMV+21; OMP18; SKR19]. Yet, for some, it is at least fun to use VR tools during development [CLB20].

Virtual reality is an example of a contextual factor that we can assume has an impact on a developer's code understanding process. However, evidence for this has largely only been collected since 2017 and studies are primarily interested in the performance of developers rather than their comprehension processes. Compared to the program comprehension theory presented in [Section 2.1](#) what could be different when one needs to understand code in VR?

For example, it could be argued that the metaphor adopted by the VR tool and the associated visualization approach of the structure of a program influences which mental representation of the program emerges in the mind of a developer (a central aspect of program comprehension theories). The VR tool can be designed in such a way that it shows only hierarchies of components, packages and classes, but visualizes no control flow. With such a tool, a developer could not create a program model at all. The understanding of the program would be restricted intentionally to certain other aspects of the program. The bottom line is that the example of understanding code in VR demonstrates well that we should incorporate contextual factors into code comprehension models, as they can be decisive for how a developer proceeds in understanding a program and which aspects of the program are perceived in principle. When these connections become clearer, researchers and tool developers may be able to help developers understand programs more effectively.

Let us take another look at the work of Siegmund and Schumann [SS15], who have systematically collected even more contextual factors that, according to scientific literature, are suspected of influencing code comprehension.

Examples of contextual factors that become relevant in code comprehension studies include learning effects, operationalization of the study object, order and difficulty of tasks, evaluation apprehension, and time pressure. Further, at least two psychological effects may be at play: The ‘Hawthorne effect’ [MWI+07; RD03], which describes that participants in experiments would behave differently because they were observed, as well as the ‘Rosenthal effect’ [RJ66], which today generally stands for the idea that the expectation of the experiment leader could influence the behavior of the participants and thus the study results.

In [Chapter 5](#), we will empirically test the influence of a particular psychological effect, the anchoring effect, on experiment participants’ subjective code comprehension. And we will find out that developers are influenced by subtle cues in the environment at least in their subjective perception of code comprehensibility, but likely also in their concrete code comprehension process. In that chapter, we will then explain in more detail that the anchoring effect is a cognitive bias, and that cognitive biases play a much bigger role in software engineering than most people realize [MST+18]. Thus, because code comprehension plays such a central role in software engineering, we would do well to assume that the previous paragraphs have only sketched the tip of an iceberg of contextual factors that might have an impact on code comprehension.

2.4. Measuring Code Comprehension

When researchers in an experiment seek to investigate the influence of a treatment on the code understanding of the study participants, they eventually face the question of how to measure the code understanding of an individual person at a particular point in time. Many design decisions play a role in this measurement, such as the criteria according to which the code snippets to be understood are selected, how the comprehension task is designed, and, as a central aspect, which comprehension measures are used. These design decisions result in reflective indicators that operationalize the

construct of code comprehension [RT18a].

There are more diverse options available today than ever before to measure a person's code comprehension performance [WBW23]. At the same time, there is a lack of research on the comparability of different comprehension measures [MWGW23]. The research community was aware of this lack of comparability as early as 40 years ago, when the first studies examined the comparability of specific task designs [CBF84; HZ86] and others examined the comparability of various comprehension measures [RC97; Shn77]. Not much has happened since then, and the few existing studies on the comparability of different comprehension measures overwhelmingly conclude that different common ways of measuring code comprehension do not correlate with each other [AWF18; BLMM09; FRM+20; Ise88; YYZD21]. All of this not only potentially leads to uncertainty in the design of new primary studies, but also currently leads to each code comprehension experiment developing its own methodology for measuring the code comprehension performance of its participants [WBW23]. What was a neglected issue of comparability of comprehension measures is now becoming a much larger issue of comparability of study results in the face of increasing and barely surveyable publication volumes, as authors of potential meta-studies cannot know which primary studies intend to measure the same construct.

In [Chapter 3](#), we discuss in detail the variety by which code comprehension is measured in experiments, and there we discuss in more depth the consequence of the increasing diversity of study designs. In [Chapter 6](#), we present a methodology for conducting meta-studies that can be used despite a lack of evidence on the comparability of different study design characteristics. The variety of ways to measure code comprehension and the difficulty in comparing primary studies both stem from potential differences in how different researchers conceptualize code comprehension. At this point, we will therefore discuss the role of the construct definition for a meaningful measurement of code comprehension.

What Makes Sense?

The justification of a particular operationalization depends, among other things, on how one defines code comprehension at the conceptual level. Based on such a definition, it can then be judged whether “an operationalization seems intuitively reasonable” (*face validity*) and whether “an operationalization encompasses all aspects of a construct” (*content validity*) [RT18a]. The assessment of face and content validity are the first of several steps in the evaluation of construct validity [RT18a].

A general threat to construct validity is that a definition of the construct is missing or insufficient [SB22]. An obvious problem that arises is that if it is not clear what is being measured, it cannot be assessed whether *it* is being measured validly in terms of face and content validity. Sjøberg and Bergersen [SB22] note that in software engineering research, most of the concepts are often not theoretically defined. We ourselves have confirmed this observation within the literature on code comprehension experiments [WBW23]. Hardly any study defines code comprehension. As a consequence, in most cases, an operationalization of a previously inadequately described construct occurs. This then leads to limitations in evaluating construct validity, and further to meta-studies facing difficulties in deciding on the inclusion of certain primary studies [WBW23].

The intriguing aspect of the discussion about face and content validity is that the question of a meaningful operationalization of a construct relies primarily on intuition and consensus among experts [RT18a; SB22]. It is an example of how intuition enriches the evidence for designing experiments sought in this thesis (see also [Section 1.3.2](#)). Nevertheless, before we can discuss what constitutes a meaningful measure of code comprehension, we first need to define code comprehension. We will do this in the following [Section 2.5](#).

2.5. A Conceptual Model of Code Comprehension Experiments

“ *The goal was always the same: to develop models and tools to help developers with program understanding during program maintenance. However, few authors targeted the more fundamental question: “what is program understanding?”* ”

— HARTH AND DUGERDIL [HD17]

There is a sentiment in Harth and Dugerdil’s quote that we can relate to a certain extent. Especially in contemporary code comprehension literature, there are hardly any explanations of what the authors mean by code comprehension. To be fair, though, it is precisely the empirical studies of the 80s and 90s that have made a fairly solid contribution to our understanding of program comprehension, both in terms of observable comprehension strategies of developers and in terms of how mental models are formed. Presumably, empirical studies simply reach a limit of the definable with their methodology, at which point we must proceed with philosophical discussions. The question of when a developer has understood source code is at least a question of definition.

We will define code comprehension in this section and then explain the construct in the context of experiments at a particular level of abstraction that allows the majority of existing code comprehension experiments to be mapped to it. In other words, we now design a conceptual model of code comprehension experiments. The model will immediately help us to clarify the scope of this thesis. In the medium term, it can be extended to a useful framework in which new code comprehension experiments can be anchored and thus assure secondary research a minimum of comparability with other code comprehension experiments.

Definition 2.1 (Source Code Comprehension)

Source code comprehension describes a person’s intentional act and degree of accomplishment in inferring the meaning of source code.

This theoretical definition overlaps with some of the previously presented definitions from the early days of the field (see introduction to this [Chapter 2](#)), in particular [Pennington](#)’s description that “comprehension involves the assignment of meaning to a particular program” [[Pen87](#)]. The definition further includes the distinction between the comprehension process and the comprehension state, as described by [Gilmore](#) [[Gil91](#)], and it deliberately uses the term *source code* rather than *program* to explicitly distinguish code comprehension from comprehension of other objects in the context of the diversified research field of program comprehension that exists today. Accordingly, we consider the introduction of a new definition of the construct to be justified by the way in which it unifies historically evolved views in a contemporary manner.

Accompanying this definition are some propositions that explain the construct in more detail:

- The *degree of accomplishment* represents a spectrum. Colloquially, we tend to speak of someone having understood or not understood something, thus treating the construct as a binary variable. Researchers are free to do the same in their operationalization of the construct, but in doing so, they would miss out on potential in the finer-grained analysis of performance differences of their experiment participants, and this should be justified argumentatively.
- The *degree of accomplishment* can be expressed by contrasting a developer’s mental model with external reality in terms of the developer’s goals. Such a comparison suggests intuitively that one measures how accurately and completely the mental model reflects reality. However, it should be kept in mind that, depending on the goal of the developer, it is not always favorable to aim for a complete mental model of the

code¹. We learned in [Section 2.1](#) that the formation of mental models depends on both the comprehension strategy and the comprehension task. For example, the task of understanding source code in detail and the task of finding a bug in the code will lead to different mental models of the code. When designing the concrete way to measure the success of code comprehension, this aspect must be considered.

- *Inferring the meaning of source code* can be discussed and assessed on at least three dimensions: what the code does (functionality), what it should do (specification), and what the intention of the code author was (context). These aspects can be found in similar forms in various program comprehension models and are surveyed to varying degrees in contemporary code comprehension experiments. Again, there is no single approach here: The concrete task design depends on the reasoning of the researcher regarding the goal scenario of the developer.
- There are a variety of ways to observe and measure source code comprehension. The definition applies to both assessments in the form of administered comprehension tests and self-assessments of the subject.

The question of what ‘understanding’ means conceptually is not a simple one. Often, the answer also depends on the specific context, for example, whether it is a question of implementing an AI with the capability of understanding user needs or whether it is a question of explaining neuroscientifically how information processing takes place at different levels of the nervous system². For our context in which developers need to understand source code, the above [Definition 2.1](#) and the associated propositions provide a starting point to work with. Before we delve a little deeper into the notion of understanding code in an experimental context, we provide a definition of the term experiment.

¹Credit where credit’s due: This idea about considering the goal when matching the mental model with reality comes from Stefan Wagner, during one of our many fruitful meetings.

²Kevin Mitchell, in a blog post about whether an AI could cook meth, conveys the potential diversity of views well: <http://www.wiringthebrain.com/2019/02/understanding-understanding-could-ai.html>

Definition 2.2 (Experiment)

An operation or procedure carried out under controlled conditions in order to discover an unknown effect or law, to test or establish a hypothesis, or to illustrate a known law. [Mera]

We are aware that the term experiment is used differently [SF18] or even incorrectly [ATFJ22] within the software engineering research field, and that there are different types of experiments. The kind of experiments we deal with in this thesis is closest to what Stol and Fitzgerald classify as ‘Laboraty Experiment’, the purpose of which is “to study with a high degree of precision relationships between variables, or comparisons between techniques; may allow establishment of causality between variables” [SF18]. These can then be further subdivided into, for example, randomized controlled experiments and quasi experiments.

Our restriction to a specific research methodology is motivated by the need to ensure that the associated primary studies are comparable in their design and the resulting threats to validity so that we can make targeted and concrete suggestions for improvement. According to Biffel et al. [BKE+14], the vast majority of threats are too specific to be generalized outside the particular research area they occur in. We are interested in the type of experiment in which some degree of causal investigation is sought that is influenced by confounding variables, which in turn will play a larger role in this thesis. However, just as important as the research methodology is that the experiments measure code comprehension with human participants (more on this in Section 3.2.2).

Now that we have a definition of source code comprehension as well as experimentation, we bring these two concepts together in the conceptual model depicted in Figure 2.3. The model consists of three layers: mental model, mental state and experimental variables. These play a role at two points in time t and $t + x$ with $x > 0$.

Let us first look at the top two layers and their interaction. At each point in time, a developer is in a mental state S , which is backed up by a mental model M that contains a representation of the source code to be understood. The

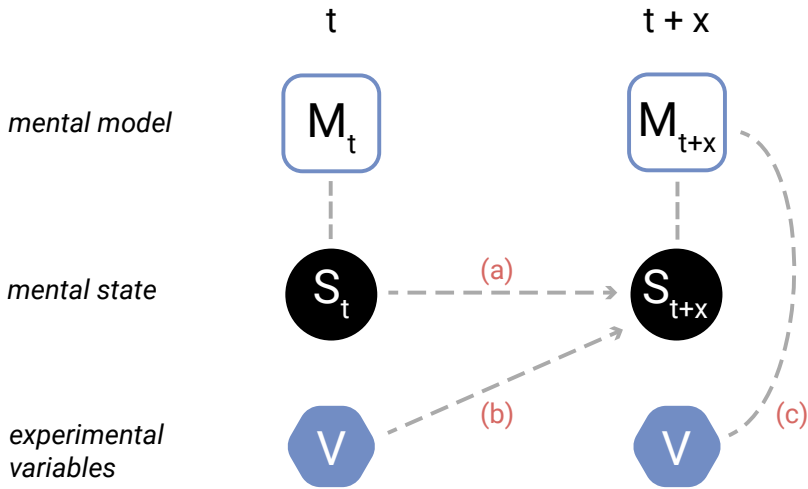


Figure 2.3.: Conceptual model for the design of code comprehension experiments

beginning of an experiment may represent time t ; at this time, a developer's mental model of a particular code snippet (depending on prior knowledge of the code) probably does not look too extensive. At a later point in the experiment ($t + x$), after the developer has completed a code comprehension task, the developer is in a new mental state and accordingly has a changed mental model of the code.

A lot can happen between these two points in time. Developers generate hypotheses about how the code works, reject or confirm them, and understand the code better over time. This process generates any number of mental states and changes in the mental model between the two explicitly depicted mental states. The two mental states S_t and S_{t+x} are particularly crucial for designing code comprehension experiments because they represent the initial state and the state in which a researcher tries to access the mental model to infer how well a developer has understood the source code presented (this may happen several times during the experiment).

The third layer, experimental variables, includes all experimental design

decisions (e.g. the comprehension task), contextual factors, individual characteristics of the developer, and other variables that can influence the mental state of a developer. In [Figure 2.3](#) we see that the mental state S_{t+x} is influenced by both the previous mental state S_t and a set of experimental variables V . Part of the experimental variables, namely the statistical tests and comprehension measures, operationalize in which way the mental model M_{t+x} is accessed at the time $t + x$.

The term ‘mental model’ is used in this context as it was introduced in the 1990s in the context of program comprehension strategies, i.e. as “a maintainer’s mental representation of the program to be understood” [[SFM99](#)] and “an internal, working representation of the software under consideration” [[VV95](#)]. According to a recent meta-study on mental representations during program comprehension, this description captures the essence on which the community could agree [[HLHF22](#)]. At the same time, it should be mentioned that it would be difficult to find controversies at present anyway, since research in the field of mental representations during program comprehension has unfortunately declined considerably [[BAV20](#); [HLHF22](#)].

The meaning of the term ‘mental state’ is quite diverse due to different streams in the philosophy of mind. For example, for followers of physicalism mental states are equal to brain states (followers of this theory are in the majority today), for followers of dualism theory mental and physical are two different things [[Jaw11](#); [Kin20](#)]. The topic is as exciting as it is complex because between and alongside these positions there are many variants, all of which sound plausible at first, but over time all of which have at least been confronted with thought experiments, and from some people’s point of view also refuted. Further elaboration is beyond the scope of this thesis, so we handle the matter pragmatically as follows. Understanding is directly related to the mind, and a mental state can be considered as the mind of a person at a particular time. Therefore, we would like to reflect this connection in the conceptual model, in that mental states represent the bridge between the environment (experimental variables) and the mental model a developer has about the code. However, the conceptual model allows for different views on the nature of mental states, so that in the first place we only have to agree

on the idea that at any point in time such a state exists, which will transition into another state (similar to a state machine). This perspective allows us to model a progression in the comprehension process of a developer.

With this, we have a rough understanding of the elements of the conceptual model for the design of code comprehension experiments. We will now go into more detail on the three depicted links (a), (b), and (c).

Link (a) represents the transition from one mental state to another and thus constitutes the *cognitive model*, i.e., “the cognitive processes and information structures used to form the mental model” [SFM99]. There is research on this, as we saw in [Section 2.1](#), but it is far from sufficient to adequately characterize the processes and information structures involved in code comprehension. In this thesis, however, the link is *not* the focus of our investigation.

Link (b) is the focus of this thesis. This link is about the (assumed) influence of experimental variables on the code comprehension process and accomplishment. Every experiment that investigates the influence of a variable on code comprehension is interested in this link (b). It is important to understand that each such experiment creates its own implicit or explicit causal model. While link (b) is drawn as a single line with an arrowhead at the level of abstraction of our model ([Figure 2.3](#)), one would see at a level of detail below that it is actually a complex network of interdependent variables. We have made such a causal model explicit in two of our studies and return to it in [Section 4.2.2.7](#) and [Section 5.3.2.4](#).

Link (c) deals with the concrete capturing of the mental model, which becomes relevant whenever we need to measure a person’s understanding about a code snippet. In most experiments, this happens once per experimental task, for example when comprehension questions are asked to the developer after understanding a code snippet. Link (c) will also play an important role in this thesis, as there are more or less valid ways to measure comprehension, which can also be supported with evidence.

2.6. Conclusion

We traveled through the history of program comprehension strategy and theory research in [Section 2.1](#), learned in [Sections 2.2](#) and [2.3](#) that individual characteristics and contextual factors, respectively, influence code comprehension, and got a teaser in [Section 2.4](#) that code comprehension can be measured in many ways. In [Section 2.5](#), we brought all of this together in a conceptual model for the design of code comprehension experiments. At this point, we have established a good theoretical foundation on which to build in the following sections. We begin with a systematic mapping study that looks in detail at how code comprehension experiments of the past 40 years have been designed and what kind of difficulties they face.

FORTY YEARS OF DESIGNING CODE COMPREHENSION EXPERIMENTS

This chapter will provide an answer to **RQ1** on what the differences and similarities in the design characteristics of code comprehension experiments are. It extends our journal publication [WBW23]. The results and their discussion serve to establish the foundation for and motivate our overall objective of bringing more evidence into the evaluation of study design decisions. Furthermore, it will become clear what currently makes it difficult to synthesize evidence from code comprehension experiments in meta-studies.

3.1. Context and Goals

The relevance of code comprehension in a developer's daily work was recognized more than 40 years ago. Estimates of the average working time invested in source code comprehension range from 30 to 70% [MML15;

XBL+18]. Consequently, many studies were conducted to find out how developers could be supported during code comprehension and which code characteristics contribute to better comprehension. Today, such experiments are more common than ever. While this is great for advancing the field, the number of publications makes it difficult to keep an overview. Additionally, designing rigorous experiments with human participants is a challenging task, and the multitude of design decisions and options can make it difficult for researchers to select a suitable design. Even though some recommendations exist, they are largely a collection of arguments and considerations based on personal experience [Fei21] or refer on a more abstract level to guidelines for general research methods like controlled experiments [JM01; WRH+12].

As a result, every researcher currently designs, conducts, and reports their code comprehension study quite differently. For example, the differences in design begin with the concrete tasks given to the participants in a study, i.e., whether they only need to read code, fix a bug, or even extend the code. Differences continue with the way in which the success of code comprehension is quantified, e.g., via the time required to solve a task, correctness in comprehension questions, subjective self-assessments by the participants, or psycho-physiological measurements [OBMC20].

We consider a certain diversity in study designs to be essential in good scientific practice, as complex research questions should be approached from different angles. Currently, however, this leads to two major issues: first, it creates uncertainty when designing a new study, as it is not clear from the multitude of code comprehension studies what the majority of the program comprehension community currently agrees on as valid study designs [Sie16; SSA15]. This makes it especially difficult for novice researchers to get familiar with this field. Second, different study results are difficult to compare, let alone to incorporate into meta-analyses, in which study results are to be synthesized [KMB20; Woh14b].

To address these issues, we conducted a systematic mapping study of 95 source code comprehension experiments published between 1979 and 2019. By systematically structuring the design characteristics of code com-

prehension studies, we provide a basis for a subsequent discussion of the huge diversity of design options in the face of a lack of basic research on their consequences and comparability. We describe what topics have been studied, as well as how these studies have been designed, conducted, and reported. Frequently chosen design options and deficiencies are pointed out. We conclude with five concrete action items that we as a research community should address moving forward to improve publications of code comprehension experiments.

3.2. A Systematic Mapping Study

3.2.1. Background and Related Work

In 2007, Di Penta et al. [DSK07] proposed a working session to collaboratively launch empirical program comprehension studies. Their motivation was that the design of such studies was complex due to the ‘large space of decisions’ and that the community had to discuss what the best practices were.

In recent years, the debate about the design of program comprehension studies, which had subsided meanwhile, gained momentum again. In 2016, Siegmund [Sie16] suggested, based on a survey of program committee members at major software engineering venues, that the reason for the few new program comprehension studies may still be that “empirical evaluations of program comprehension (and human factors in general) are difficult to conduct”.

In 2021, Feitelson [Fei21] published a series of ‘considerations and pitfalls in controlled experiments on code comprehension’. For different design decisions like selecting code to be comprehended or the concrete comprehension tasks, the paper describes how these decisions can affect the results of a study, and exemplarily points to studies that implemented specific design decisions. We believe that Feitelson’s work [Fei21] is a valuable pragmatic approach for guiding researchers in the design of code comprehension experiments by compiling intuitive knowledge. Our work will provide the

necessary empirical data on how prevalent certain design decisions, and thus certain pitfalls, actually are.

Several other publications on the analysis of existing program comprehension studies exist. Oliveira et al. [OBMC20] model program comprehension as a learning activity and map typical tasks of empirical human-centric studies on readability and legibility to an adaptation of Bloom’s taxonomy of educational objectives. Their goal is to identify the cognitive skills that are most frequently tested in readability and legibility studies. We are particularly interested in their classification of commonly used tasks and measures, as we use it as a basis for our own classification of comprehension tasks. Their data show that participants in readability and legibility studies most often have to provide information about the code, and correctness is favored as a response variable.

Schröter et al. [SKSL17b] conducted a literature review of 540 ICPC papers published between 2006 and 2016. They addressed three questions, namely what the primary studies examined in the context of program comprehension, what terminology they used to describe the evaluation (e.g., empirical study, user study, etc.), and whether they reported threats to validity. Their investigation revealed that *source code* and *program behavior* are the most frequently addressed parts of program comprehension studies at ICPC. Moreover, researchers would use “a diverse and often ambiguous terminology to report their evaluation”. Finally, they found that the number of empirical studies reporting threats to validity has increased in recent years. This study complements our work very well, since it deals with program comprehension studies in a broader scope and conducts similar investigations as we do. Yet, we analyze the topic more deeply and consider numerous other study design features, as – according to their comment on potential future work – Schröter et al. [SKSL17b] themselves intended “to analyze in more detail how evaluations on software comprehension are performed, for example, regarding typical evaluation tasks and comprehension measurement”.

Related to the analysis of threats to validity is a literature survey by Siegmund and Schumann [SS15] that compiles a catalog of confounding parameters discussed in program comprehension studies. Their survey

includes papers from 13 journals and conferences over a period of 10 years and results in 39 confounding parameters. In our systematic mapping study, we will use the resulting categories of confounders as a basis to categorize discussed threats to validity in code comprehension studies. We will come back to this in more detail in [Section 3.2.3.8](#).

In addition to the meta-studies described above, several primary studies address specific aspects of the design of code comprehension studies and provide evidence for consequences of certain design decisions. This started almost 40 years ago with studies on the differences of certain task designs [[CBF84](#); [HZ86](#)]. Over the years, this line of research was extended with studies on the comparability of different comprehension measures [[AWF18](#); [RC97](#); [Shn77](#); [YYZD21](#)], the influence of cognitive biases on subjective and objective code comprehension measures [[WVG22a](#); [WPGW21](#)], and the actual influence of suspected confounding variables like intelligence, expertise, or code length [[BP16](#); [FWS93](#); [Tea94](#); [WW22](#)]. All of these studies finally culminated in the current discussions about capturing psycho-physiological data to better understand program comprehension [[Fak18](#); [SPB+20](#); [VF21](#)]. While these primary studies all contribute to informed decision-making on individual design aspects, our mapping study reviews multiple design characteristics in their entirety and examines interactions between individual aspects.

In summary, designing code comprehension studies has interested the research community for many decades, and yet a detailed, comprehensive study to capture the actual state of past and current study characteristics is lacking. We fill this gap with a systematic mapping study.

3.2.2. Methodology

A systematic literature review (SLR) focuses on synthesizing the available evidence for a topic [[KC07](#)]. Conversely, a systematic mapping study (SMS) is a different type of secondary study with a focus on summarizing the conducted research on a certain topic to provide structure [[PFMM08](#)]. Unlike in SLRs, quality-based exclusion is therefore usually not essential in an SMS [[PVK15](#)],

as the goal is to describe and map what topics were investigated in what way. Since this is well suited to our objective, we decided to follow the SMS methodology.

The research questions that guided us in our study on experiments about bottom-up source code comprehension are as follows:

RQ1.1 How can the studies be classified in terms of their relationship to code comprehension?

RQ1.2 What are differences and similarities in the design characteristics of the studies?

RQ1.3 What are differences and similarities in the reporting characteristics of the studies?

RQ1.4 Which issues and opportunities for improvement are evident in the design and reporting of the studies?

The answer to RQ1.1 is intended to provide insights into what research concerns the primary studies address. The focus, however, is not on a possible synthesis of evidence, but on the classification of the research topics as well as their relationship to the code comprehension construct, e.g., whether a certain influence on code comprehension was investigated or, conversely, whether the influence of code comprehension on a certain construct was investigated.

RQ1.2 forms the core of this systematic mapping study. The answer to this question provides comprehensive insights into the unclear status quo of the design of code comprehension studies. RQ1.1–RQ1.3 were systematically addressed using the methodology described in the following subsections. The data obtained is analyzed in [Section 3.2.3](#) and forms the basis for the subsequent discussion of RQ1.4 in [Section 3.2.4](#). This discussion should improve the quality of code comprehension studies in the medium term and should immediately result in concrete proposals for improving the design and reporting of code comprehension studies.

The following subsections explain our approach to searching and selecting relevant literature, as well as to data extraction and analysis. An overview of the methodology is depicted in [Figure 3.1](#).

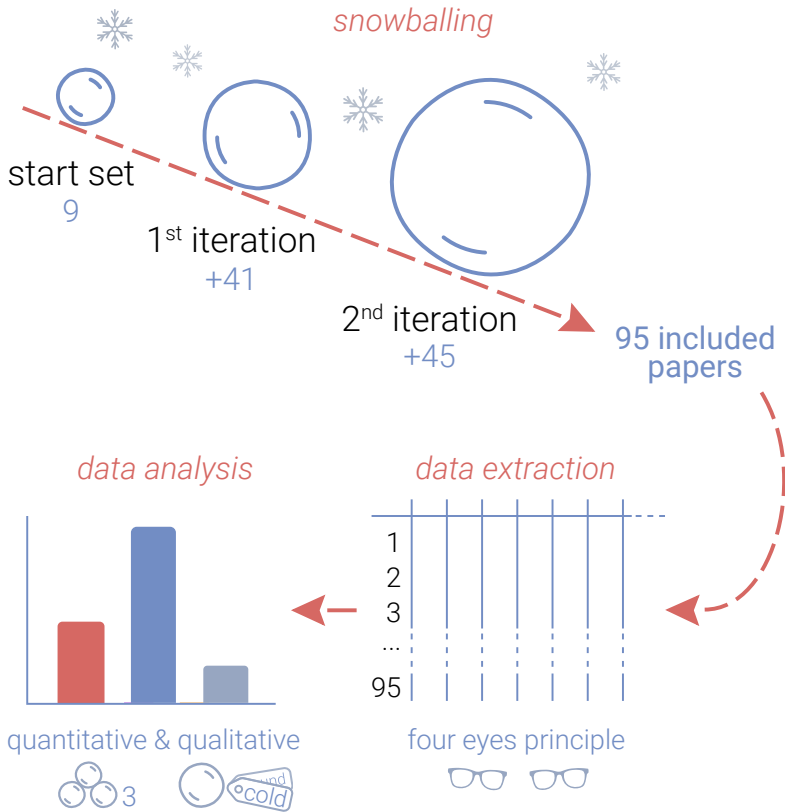


Figure 3.1.: Schematic representation of the research methodology

3.2.2.1. Search and Selection Process

We designed our search strategy as a pure snowballing approach, following Wohlin's guidelines [Woh14a]. Beginning with a start set of relevant papers known to us, the referenced as well as referencing literature was reviewed for a set of inclusion and exclusion criteria. This process is called backward and forward snowballing, and it is repeated on newly included literature. We stopped the literature search after two iterations of backward and forward

snowballing, as we approached saturation.

Snowballing as part of a systematic literature search is not uncommon. Often, however, a database search with predefined search strings is performed before snowballing to identify an initial set of relevant papers. We refrained from doing so, as a thematically broad literature search via databases leads to a high number of non-relevant search results and thus to considerable effort. Searching for all papers that measure bottom-up source code comprehension in some way would have to cover a wide range of research topics and would lead to very generic and extensive search terms. We consider snowballing with a reasonable start set to be more targeted and efficient. Furthermore, there is evidence that snowballing is similarly effective as and usually more efficient than database searches [BWP15; JW12; Woh14a].

Start Set. We build our tentative start set on the results of a recent literature search by Muñoz Barón et al. [MWW20], who searched for program comprehension data sets at the source code level. Similar to our systematic mapping study, their study searched for empirical studies that measured source code comprehension with human participants, but in a final step filtered for those that had published their dataset. They then continued working with these data sets to validate a code comprehensibility metric. Muñoz Barón et al. [MWW20] identified a total of 10 code comprehension datasets, whose associated papers we use as tentative start set for our SMS.

Nine of the ten papers meet our inclusion criteria (see below). The one excluded paper was a pilot study for another study in the start set and was therefore considered a duplicate. Wohlin stresses that “there is no silver bullet for identifying a good start set” [Woh14a]. However, he describes characteristics that a good start set should meet, and we will briefly discuss them.

Since Muñoz Barón et al. [MWW20] conducted a broad database search to find relevant literature, the start set fulfills the criterion of covering different communities and the criterion of containing keywords that are closely related to our research questions. The titles of the papers cover a variety of potential

clusters related to program comprehension: program comprehension in general, code and software readability, physiological measures related to program comprehension, code understandability, and code complexity.

A further criterion requires that the start set covers different publishers, publication years, and authors. The nine papers in the start set were published in IEEE TSE (5), ACM ESEC/FSE (2), and Springer EMSE (2), between 2003 and 2019. Three of the journal papers are extensions of previously published research papers at the International Conferences on Automated Software Engineering (ASE), Software Analysis, Evolution and Reengineering (SANER), and Program Comprehension (ICPC). The nine papers were authored by 39 distinct authors.

Finally, a good start set “should not be too small” [Woh14a]. While nine papers seems to be a small number, the papers in the start set have been cited about 400 times and refer to about 500 papers. Hence, already after the first snowballing iteration, we will have screened about 900 papers.

Inclusion and Exclusion Criteria. During the search process, a paper was included in the final dataset only if it met all the inclusion criteria (I) and none of the exclusion criteria (E):

- I1** Reports an empirical study with human participants.
- I2** Measures bottom-up source code comprehension.
- I3** Published before 2020.
- I4** Published in a peer-reviewed journal, conference, or workshop.
- E1** Is not available in English.
- E2** Is a meta-study.
- E3** Is a replication without substantial modification.
- E4** Is a duplicate or extension of an already included paper.

Inclusion criteria I1 and I2 ensured that a paper was within the scope of this work by describing empirical studies with human participants who needed to understand source code. Examples of code comprehension studies that do not meet I1 include those that only analyze existing data (see, e.g., [PHD11;

TCM+18]).

I2 was only fulfilled if the quality or performance of a participant in understanding code was measured (not, for example, *how* participants proceed in understanding code, as in, e.g., [AS96; LGHM07]). Of all the criteria, I2 represents the one that was the most challenging to evaluate because primary studies to date rarely define the construct under investigation, and we will later see that authors use different terms for the same construct. We also emphasize that in evaluating I2, we focused on the authors' intent and only rarely judged at this stage of our study whether the chosen task design of a primary study can actually measure understanding. If we could not find any explicit indication that the primary study was intended to measure code comprehension and at the same time the task design was not explicit in this respect, we did not include the primary study.

The rationale for restricting our search to papers published before 2020 (I3) was that we started our literature search in early 2020 and that this constraint would increase the reproducibility and extensibility of our approach. Including only peer-reviewed literature (I4) served the purpose of quality assurance. Thus, our dataset contains only primary studies whose methodology has been considered sound by at least a few members of the community.

A paper was excluded if it was inaccessible to us due to a language barrier (E1), if it was a meta-study (E2), or if the design of the study was similar to that of an already included one because it was a replication, duplicate, or extension (E3 and E4). We added E3 and E4 to avoid possible bias in the quantitative analysis of the data. Even with extensions, the original study designs are rarely modified, which means that inclusion of the original study and the extension would cause the specific design decisions to be considered twice in the analysis. Regarding the impact of E3, we identified only one replication that would have met our inclusion criteria, namely Fucci et al. [FGN+19].

The set of papers resulting from the search and selection process is characterized in detail in Section 3.2.3.1. In total, we included 95 papers. The 9 papers in the start set led us to 41 relevant papers. On these 41 papers, we

again performed a snowballing iteration, which yielded 45 more relevant papers. The ratio of newly included literature to literature screened in this second snowballing iteration made us confident to assume a saturated data set at this point. We therefore stopped searching for relevant literature after the second iteration.

3.2.2.2. Data Extraction and Analysis

Before data extraction, all data items to be extracted were defined with a description as precise as possible. In addition to DOI, APA citation, publication year, and venue, 17 additional columns were filled in a spreadsheet for each of the 95 included papers to characterize different aspects of the study design. The extracted data items are listed in [Table 3.1](#). The results are presented according to the same data item categories in [Section 3.2.3.1](#) under subsections of identical names.

Table 3.1.: Extracted data items

Category	Data Items
Included Papers	DOI, APA citation, publication year, venue
Study Themes	study category
Construct Naming	construct name
Study Designs	research method and experiment design
Participant Demographics	#participants, demographics
Setting and Materials	#snippets per participant, snippet pool size, code snippet source, snippet selection criteria, remote vs. onsite, paper vs. screen, programming languages
Comprehension Tasks and Measures	comprehension tasks, construct measures
Reported Limitations	limitations

Two researchers independently performed the extraction for all papers in the start set (9) and the first snowballing iteration (41). In several meetings, each extracted cell was then reviewed for agreement and, if necessary, a

consensus was reached through discussion. Over time, we thus established a strong understanding of the literature and difficulties in the extraction process. This four-eyes principle was highly time-consuming, but led to greater certainty that the papers examined were correctly understood and that no relevant details were overlooked.

In the second snowballing iteration, each of the first two authors extracted half of the 45 newly added papers. Items about which one was unsure were subsequently examined by the other. The same applied to papers where one of us came to the conclusion during extraction that the study should actually be excluded. Such a decision was never made alone. In the supplemental materials, we transparently indicate which papers were extracted by which author(s) [WBW22].

After extracting the design characteristics for all 95 papers, the data were analyzed both quantitatively and qualitatively. Some data items such as publication date, number of study participants, and used programming language could for the most part be analyzed without further data transformation. Other data items such as the study themes and comprehension tasks first had to be classified using thematic analysis [CD11], since the diversity of these data was unknown at the time of data extraction. Finally, there were items for which we extracted descriptive quotations that we first labeled and then analyzed by category building. Examples of such data items are the criteria for code snippet selection and threats to validity discussed by the authors of the primary studies.

3.2.2.3. Threats to Validity

Our methodological approach has a few limitations that are relevant for the interpretation of the results and that we would therefore like to point out in advance.

In the **search and selection process**, we restricted ourselves to papers published before 2020. The rationale was, as explained before, that we started the systematic literature search in 2020 and intended to ensure a replicability of the process. Our results will show a trend that a large

proportion of included papers were published in recent years. Accordingly, we should assume that some relevant papers were also published in 2021 and 2022 whose design characteristics are not all represented in our results. We consider it worthwhile to conduct a similar mapping study again as early as 2026 to capture the latest trends in designing code comprehension studies.

Inclusion criterion I2 required a subjective judgment about whether a given primary study intended to measure bottom-up source code comprehension. We prioritized avoiding false positive inclusion decisions over avoiding false negative ones. Our priority was to arrive at a dataset that included, with a high degree of certainty, only those studies that met our inclusion criteria. We may have incorrectly excluded a paper for this reason. On a related note, no search process, whether it consists of a database search, snowballing, or a hybrid approach, should claim completeness of the resulting dataset. Certainly, there are code comprehension studies that meet our inclusion criteria but were mistakenly excluded or not discovered during the search process. However, we are confident that the 95 papers found are a representative set of papers for our inclusion criteria.

Regarding the quality of individual primary studies, we relied solely on inclusion criterion I4, i.e., that the study was peer-reviewed. We did not perform any further quality assessment, as sometimes suggested in guidelines for systematic literature reviews [Kee+07]. This decision is based on our motivation to obtain the status quo of study designs accepted by the community. It is therefore unavoidable (and even *desirable* for us) that several of the included primary studies have flaws in their design or reporting. After all, it is these weaknesses that we aim to systematically identify and discuss.

As for the process of **data extraction and analysis**, we would like to point out that in the second snowballing iteration, data extractions were mostly performed by individual researchers. The same applies to the analysis of extracted data. The categories of extracted data items were generally analyzed and proposed by a single researcher. In both cases, however, results were discussed in detail, and the opinions of the other authors were regularly solicited. Each author also worked with data points extracted from the other authors in the data analysis, which in a sense verified extracted data points,

since one had to occasionally look at the papers again for their context.

Additionally, a few papers describe several different experiments. In case of differences between these experiment facets, we therefore either selected the dominant variant (e.g., for the experiment factor design) or a suitable aggregated value (e.g., median number of snippets per participant) during the synthesis. We make the entire process as transparent and traceable as possible by publishing the data and analyses [WBW22].

3.2.3. Results

We present the results grouped by the categories introduced in Table 3.1. A critical discussion of findings of particular interest follows separately in the subsequent discussion section 3.2.4.

3.2.3.1. Included Papers

We included a total of 95 primary studies published from 1979 to 2019. Figure 3.2 shows the number of published studies per year. A large proportion of the included studies were published between 2010 and 2019 (67.4%), and about half of all included studies were published from 2015 and onward (51.6%).

41 papers were published in a journal, 45 in conference proceedings, and 9 in workshop proceedings. The papers were published in 51 different venues, of which 39 appeared only once in the dataset. The top 4 venues by number of included papers are EMSE (10), IWPC/ICPC¹ (9), TSE (8), and ICSE (6).

¹The International Conference on Program Comprehension (ICPC) was a workshop (IWPC) until 2005. One included paper was published at IWPC.

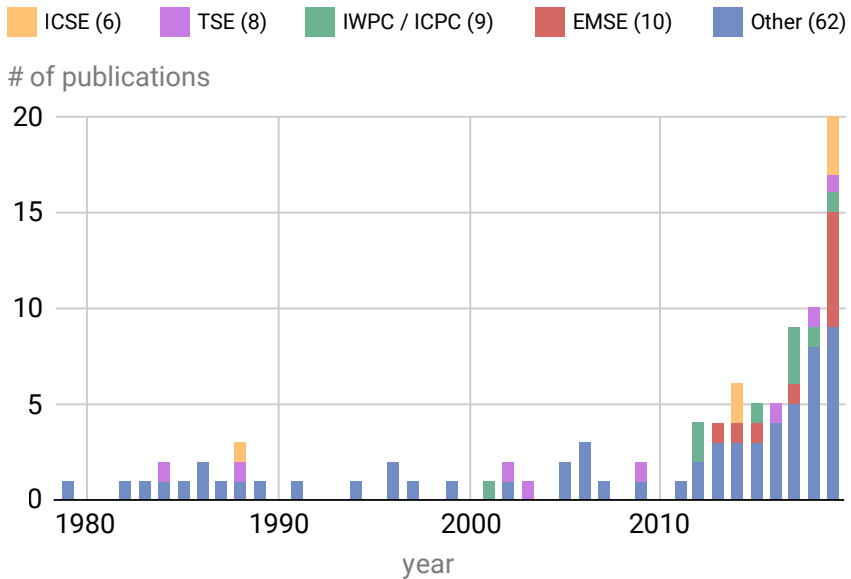


Figure 3.2.: Number of publications per year and venue

3.2.3.2. Study Themes

As a general starting point for study themes, we were interested in the study distribution according to the following three relationship types:

- **X ⇒ code comprehension:** the study analyzes how one or more other constructs (X) influence code comprehension, e.g., identifier naming length or developer experience.
- **X ⇔ code comprehension:** the study analyzes the correlation between code comprehension and one or more other constructs (X), e.g., source code metrics or eye tracking data.
- **code comprehension ⇒ X:** the study analyzes how code comprehension influences one or more other constructs (X), e.g., developer motivation or happiness.

A study could be mapped to several of such instances, depending on its scope and research questions. For example, Peitek et al. studied three instances in S1: *fMRI data* \Leftrightarrow *code comprehension* (RQ1), *code complexity* \Rightarrow *code comprehension* (RQ2), and *programming experience* \Rightarrow *code comprehension* (RQ3). Nonetheless, 78 of 95 studies in our sample were mapped to a single relationship instance (82%), while only 13 and 4 studies were mapped to 2 and 3 instances respectively. No study in our sample produced more than three instances. To analyze the distribution of relationship types, we assigned a primary type to the 17 studies with more than one type, e.g., S1 was categorized as $X \Leftrightarrow$ *code comprehension* because RQ1 was the primary focus of their study.

As a result, the majority of studies, namely 67 of 95 (71%), belong to the $X \Rightarrow$ *code comprehension* relationship type, i.e., analyzing which constructs influence comprehension in which way is the most popular type of research in our sample. The remaining 28 papers all belong to the relationship type $X \Leftrightarrow$ *code comprehension*, i.e., they analyzed correlations between constructs and code comprehension, mainly to evaluate suitable measurement proxies. Surprisingly, we did not find a single study for the type *code comprehension* \Rightarrow X . Potential explanations could be that such research is perceived as either not industry-relevant or not suitable until further progress is made in defining and measuring code comprehension as a construct.

In addition to these general relationship types, we also assigned one or more thematic labels to each paper based on the study purpose (between 1 and 10 labels, median of 1). Afterwards, these labels were grouped into higher-level categories (see Table 3.2). In total, we created 43 labels assigned to 11 categories. Categories consisted of between 1 and 10 labels (median of 3). The most popular categories in our sample were *semantic cues* (e.g., identifier naming or comments), *developer characteristics* (e.g., experience or age), *psycho-physiological measures* (e.g., eye tracking or EEG), and *code structure* (e.g., control structures or procedure usage). Less studied areas were *code improvement*, *test code*, and *mental models*. Concerning individual labels, 15 of the 43 themes were assigned to at least 4 papers (see Figure 3.3). Among the most popular themes were *identifier naming*

Table 3.2.: Study purpose categories (m. = mentions, l. = labels)

Category	#m.	#l.	Description ('One study goal is...')
semantic cues	30	3	to analyze how semantic information in source files (e.g., identifiers or comments) influence comprehension.
developer characteristics	30	5	to analyze characteristics of developers in the context of comprehension (e.g., experience or code familiarity).
psycho-physiological measures	29	7	to evaluate the feasibility of psycho-physiological measures collected from developers (e.g., fMRI or EEG) as proxies for comprehension.
code structure	28	10	to analyze how structural attributes of source files (e.g., control structures or code size) influence comprehension.
code evaluation	10	2	to analyze or provide ways to evaluate code understandability (e.g., metrics or self-assessment).
programming paradigms	9	4	to analyze and compare programming paradigms w.r.t. comprehension (e.g., object orientation or reactive programming).
comprehension support	9	4	to analyze or provide ways to better understand code without changing it (e.g., code browsing techniques or diagrams).
visual characteristics	8	3	to analyze the influence of visual code characteristics on comprehension (e.g., indentation or syntax highlighting).
code improvement	4	3	to analyze or provide ways to change existing code towards better understandability (e.g., methods or tools).
test code	3	1	to analyze or improve test code w.r.t. comprehension.
mental models	3	1	to analyze how developers mentally represent code and how this influences comprehension.

(16 studies), *experience* (15), *comments* (10), and *control structures* (9). Additionally, three labels from the *psycho-physiological measures* category are among the top 15: *eye tracking* (9), *EEG* (9), and *fMRI* (4).

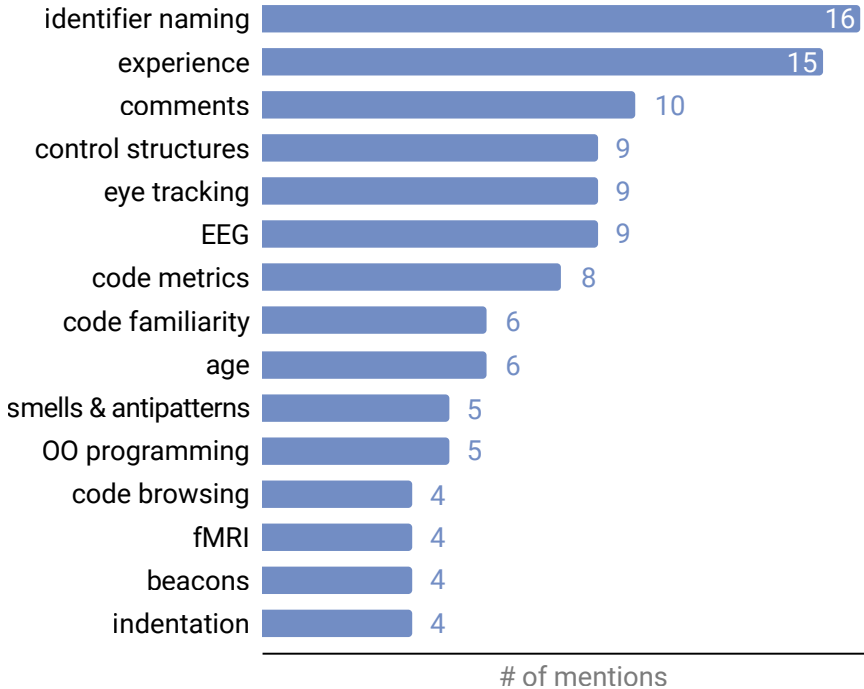


Figure 3.3.: Most popular study themes (assigned to at least 4 papers)

When analyzing the evolution of the four largest thematic categories over the years (see [Figure 3.4](#)), we see that studies on *semantic cues* like identifier naming and code comments were most popular early on but lost their relative market share in the last decade. The relative popularity of studies on *code structure* and *developer characteristics* increased, even though both are not comparable to the meteoric rise of *psycho-physiological measures*. Enabled by technological advances and the increased affordability of relevant devices, the last 10 years saw an abundance of studies evaluating data from, e.g.,

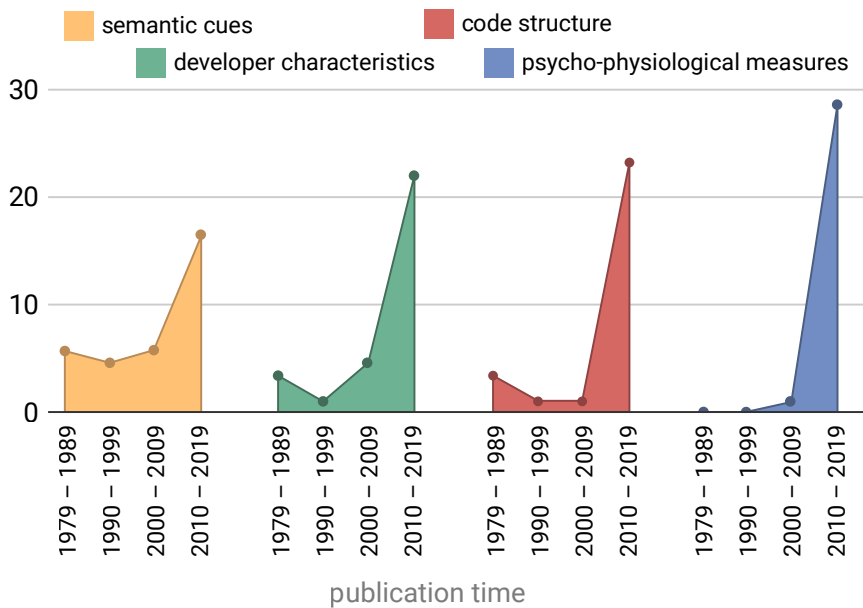


Figure 3.4.: Evolution of the four most popular study categories according to their number of occurrences

eye tracking, fMRI, EEG, NIRS, or HRV as proxies for code comprehension. This signifies a shifting focus of the research community towards a more developer-centric and individual perspective of code comprehension. As an example, Couceiro et al. even envision “biofeedback code highlighting techniques” in S51, i.e., reporting complex or potentially buggy lines of code in real-time during development, using data from non-intrusive techniques such as HRV, pupillography, and eye tracking.

3.2.3.3. Construct Naming

For each primary study, we extracted the name used for the central construct under investigation. In total, we identified 10 different construct names (see [Figure 3.5](#)), which optionally could be combined with “code” or “program”.

The most frequently used name was by far “comprehension”, with 72 of 95 studies (76%). Seven studies used “readability” as the construct name, even though they measured comprehension, i.e., participants’ level of understanding and not their ability for visual perception¹. Four studies focused on their concrete experiment tasks and explicitly used “task difficulty” as a construct. The remaining seven constructs were all used by at most two studies.

The majority of studies (73 of 95) also used constructs describing an *activity* like “comprehension” or “reading”. Only 22 studies used an *attribute* like “comprehensibility”, “complexity”, or “cognitive load” as the construct. Lastly, nearly half of the studies (47 of 95) combined their construct name with “program”, 24 studies combined it with “code”, with the remaining 24 studies refraining from using such a modifier. As an example, 45 of the 72 studies using “comprehension” as the construct phrase it as “program comprehension”, 18 as “code comprehension”, and 9 only as “comprehension”. With “program comprehension” being a superset of “code comprehension”, using the latter would be more precise, as many studies we excluded for not being concerned with bottom-up source code comprehension used the older term “program comprehension”.

3.2.3.4. Study Designs

Concerning the used research methods, 94 of 95 studies conducted some form of *experiment*. The one exception was S42, a field study performed by Sedano to test a proposed code improvement method. Since the used terminology on this may differ, we did not distinguish between different types of experimentation, such as *controlled experiments*, *experiments*, or *quasi-experiments* [WRH+12]. Moreover, such a distinction would have

¹Comprehensibility, readability, and legibility are different constructs, and the program comprehension community is starting to distinguish these more explicitly as can be seen, e.g., in the call for an 2022 EMSE special issue on ‘code legibility, readability, and understandability’ at <https://tinyurl.com/EMSECFP>. Code can be legible and readable due to being presented in a certain way and using familiar keywords. However, readable code does not necessarily have to be understood by the reader. In both code and text comprehension, there is evidence that comprehensibility and readability do not necessarily correlate [BP16; ST92]. In our mapping study, we restricted ourselves to studies in which code had to be understood.

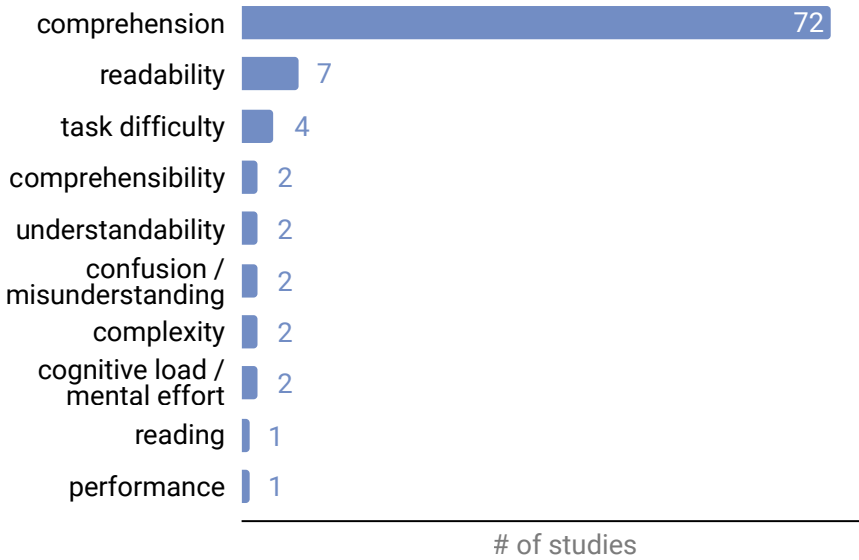


Figure 3.5.: Names for the central construct in our primary studies

provided little value for answering our RQs. We therefore classified all studies that manipulated at least one independent variable while controlling for other factors as an *experiment*. Nonetheless, the degree of manipulation and control differs between studies. For example, [Salvaneschi et al.](#) used two explicit treatments in S8, namely the object-oriented Observer design pattern vs. reactive programming constructs. In comparison, [Buse and Weimer](#) did not have such clear treatments in S6. Instead, their manipulation manifested in the varying complexity of the many code snippets shown to participants.

To be able to analyze trends and differences in experiment designs, we extracted the following properties that are loosely based on the methodological publications from Juristo and Moreno [[JM01](#)] and Wohlin et al. [[WRH+12](#)]:

- **Factor design:** we extracted whether the experiment used a *factorial* design, i.e., several factors were manipulated in parallel, or a *one-at-a-*

time design, i.e., only a single factor was manipulated in parallel.

- **Allocation design:** we extracted whether the experiment used a *within-subject* design (also called *repeated measures* design), i.e., each participant received each treatment at least once, or a *between-subject* design (also called *independent measures* design), i.e., each participant only received a single treatment.
- **Randomization or counterbalancing of tasks:** this is important for *within-subject* designs to combat familiarization and carryover effects, especially for the *within-subject* variant called *crossover* design [VAJ16]. We extracted whether this was applied (*yes* or *no*).
- **Experience- or skill-based balancing:** this is especially important for *between-subject* designs, as they do not account for individual differences between participants, e.g., regarding expertise or motivation [VAJ16]. Consciously balancing your groups regarding potential confounders can mitigate such effects. We extracted whether this was applied (*yes* or *no*).

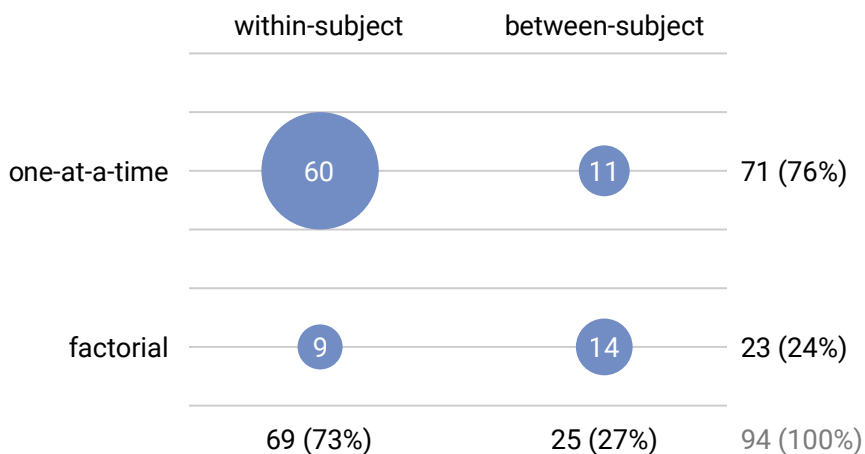


Figure 3.6.: Distribution of factor and allocation design

Concerning the factor design, 71 of the 94 experiments in our sample chose a *one-at-a-time* design. Only 23 studies used a *factorial* design, which is generally considered to be both more complex and powerful. Concerning allocation design, we see a similar imbalance, with 69 studies choosing a *within-subject* design, which is considered to be more robust against participant heterogeneity. Only 25 studies relied on a *between-subject* design, where skill differences between groups can be a threat to the validity of the results. When combining factor and allocation design (see [Figure 3.6](#)), *within-subject* is substantially more popular for *one-at-a-time* designs (85% of studies), while *between-subject* is a bit more frequently chosen for *factorial* designs (61%).

A combined analysis with the publication year also reveals a shift in experiment design over time (see [Figure 3.7](#)). Before 1990, most studies used *factorial between-subject* designs. A typical example is S67, a 2x2 factorial experiment published by [Tenny](#) in 1985. He analyzed the effect of internal procedures and code comments on comprehension by assigning each participant to one of four groups: no procedures and no comments, no procedures and comments, procedures and no comments, and procedures and comments. Over the years, *one-at-a-time within-subject* designs started to rise in popularity, until they became the dominant form of code comprehension experiments from 2010 and onwards. A typical example for this is S3, an experiment to analyze the impact of identifier naming styles on code comprehension published by [Hofmeister et al.](#) in 2019. In their study, every participant received tasks for each of the three levels of naming styles: single letters, abbreviations, and words.

Lastly, we analyzed the frequency of applying suitable precautions for the respective weakness of the two types of allocation design, i.e., if *within-subject* designs used randomization/counterbalancing and if *between-subject* designs used experience- or skill-based balancing during group assignment. Of the 69 *within-subject* designs in our sample, 45 applied randomization or counterbalancing of tasks (65%). This means that approximately one third did not use this proven technique to counteract potential familiarization effects. However, the numbers are much worse for *between-subject* designs.

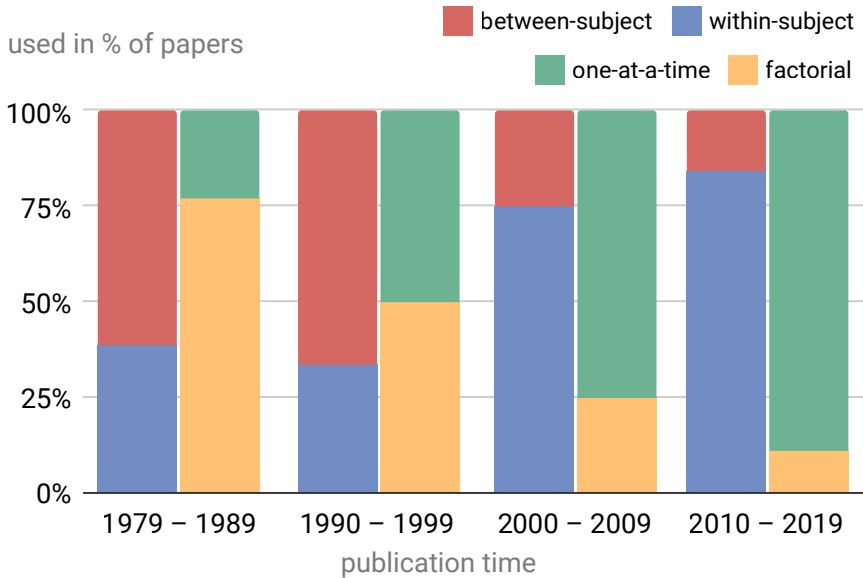


Figure 3.7.: Evolution of factor and allocation design

Of the 25 experiments with this allocation design, only 6 balanced their groups based on experience or skill (24%), i.e., 76% took insufficient or no precautions in this area.

3.2.3.5. Participant Demographics

All included studies have in common that they used a sample of human participants to answer their research question. The number of participants ranges from 5 to 277, with a median of 34. [Figure 3.8](#) shows the distribution of sample size per paper. For papers that reported different numbers of participants, for example, due to multiple reported sub-experiments, the mean of minimum and maximum reported number of participants was used for that paper (this has been the case for 13 papers).

of participants

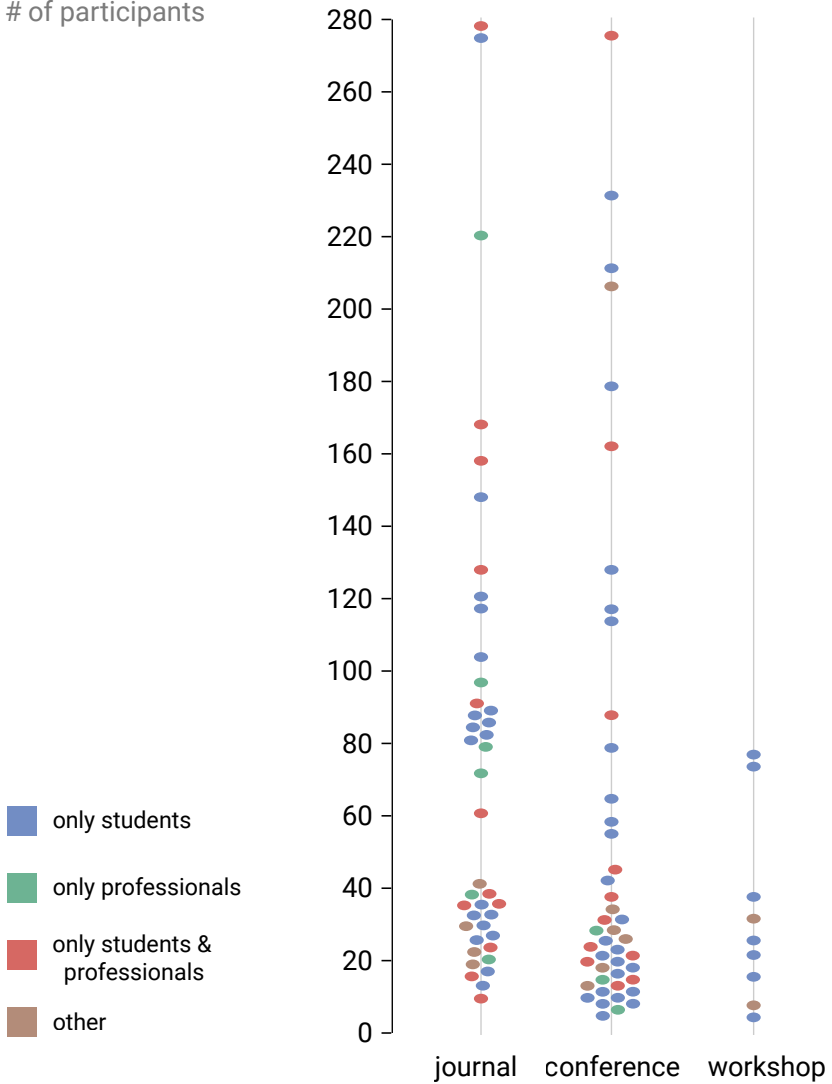


Figure 3.8.: Number of participants per study, grouped by venue type. Colors indicate the demographic composition of the sample.

About half of the 95 papers (53.7%) reported a sample that consisted entirely of students. For 9 papers (9.5%), only professionals were included in the sample. In 24.2% of the samples, the participants were composed of students and professionals. 7 papers (7.4%) recruited a sample of students and faculty members. For 4 papers (4.2%), we were either unable to find any information at all or only very vague information on participant characteristics.

Looking at the number of participants and their demographic characteristics by type of venue (journal/conference/workshop), substantial differences emerge. The number of participants for journal articles with a median of 61 (mean: 77.5) is higher than the median of 26 for conferences (mean: 58.2) and the median of 26 for workshops (mean: 33.1). Furthermore, a sample in a journal paper is least likely to be composed entirely of students (46.3% of papers compared to 55.6% for conferences and 77.8% for workshops). None of the nine included workshop papers sampled any professionals. Two of the workshop papers recruited faculty members in addition to students.

3.2.3.6. Setting and Materials

We also extracted and analyzed several aspects of the experiment settings and used materials, i.e., the code snippets. Regarding the *experiment location*, 72 of the 95 studies were conducted onsite (76%) and therefore had strong control of the study environment. In 14 cases, a remote experiment was chosen, thereby trading off some control for more and a wider variety of participants. Two studies adopted a mixed approach, with both onsite and remote participation. For seven studies, we could not reliably determine the study location from the paper. The earliest remote study (S24 by [Lawrie et al.](#)) was already conducted in 2007 based on a browser-based Java applet, but the median year was 2019. Due to the COVID-19 pandemic, we expect the percentage of remote experiments published in 2020 and later to be substantially larger.

As the *snippet medium*, 67 studies presented the code on a screen (71%), while 21 studies used code printed on paper. One study used both mediums.

In six cases, we could not reliably determine if a screen or paper was used. While we found studies published as recent as 2018 using paper (S34 by [Ribeiro and Travassos](#)), the median year was 1996, indicating that this becomes increasingly rare. Unsurprisingly, location and medium are also interrelated to some degree, i.e., all 14 remote experiments used a screen, while all 21 paper-based studies were conducted onsite.

Concerning the used *programming languages* for the snippets (see [Figure 3.9](#)), we identified a total of 21 unique languages (plus pseudocode). The majority of studies (84 of 95) only used a single language for their snippets, with eight studies using two languages, and a single study using three, four, and nine languages respectively. The latter outlier is S42, where [Sedano](#) let participants bring their own code snippets, resulting in a variety of languages. With 45 papers (47%), Java is by far the most used language in our sample. It is followed by C (14 studies), C++ (10), and Pascal (10). For seven papers, we could not reliably extract the used programming language from the paper (label n/a). A total of 13 languages were only used in a single study. Studies using older languages like Fortran, Cobol, Pascal, Algol, Basic, PL/I, or Modula-2 were all published before 2000, with a median year of 1987.

We also extracted and analyzed the *snippet pool size* and *number of snippets per participant* for each study. In four cases, we could not reliably determine these metrics from the paper. The remaining 91 studies had a median snippet pool size of 9, with a mean of 24.7 over the range from 1 to 389 snippets. 34 studies (37%) had 5 or fewer snippets, and 18 studies (20%) had between 6 and 10 snippets. Apart from that, we see a decent variety of snippet pool sizes, with some outliers in the lower hundreds, like S89 by [Asenov et al.](#) with 298 and S41 by [Sasaki et al.](#) with 389 snippets. In 45 cases, participants worked on the complete snippet pool (100% snippet percentage). The majority of these were *within-subject* designs, as treatments were most often embedded into the snippets. In the remaining studies, participants worked on very different percentages of the snippets, with peaks at 50%, 33%, and 25%. These correspond to frequently chosen numbers for groups or snippet versions, namely two, three, and four. With larger snippet pools (30 and

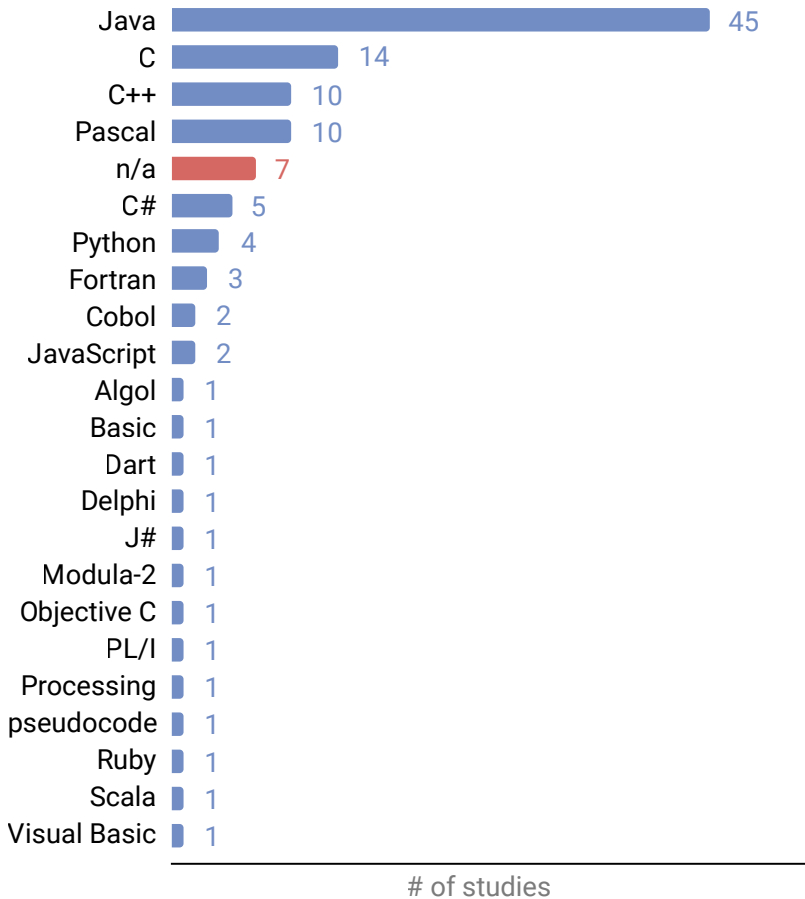


Figure 3.9.: Programming languages used in the experiments

more), the snippet percentage tended to be lower, even when no groups or snippet versions were used, most likely to not exhaust participants. However, there were also several exceptions, e.g., both [Buse and Weimer](#) (S6, 100 snippets) and [Sasaki et al.](#) (S41, 389 snippets) let each participant work on all snippets. In summary, the average study in our sample used 25 snippets

and let participants work on 70% of them (median: 9 snippets with 90% snippet participation).

We further analyzed where the used snippets originated from, i.e., the *snippet source* (see [Figure 3.10](#)). For 14 studies, we could not reliably determine the source (n/a). The majority of the remaining papers used a single source (68 studies), with 13 studies having two sources for their snippets. The most popular source was that researchers created snippets themselves (38 studies). In 22 studies, parts of open-source software were used, sometimes slightly adapted. 21 papers simply reused snippets from previous studies, and 8 papers relied on textbooks, e.g., from computer science education. Only three studies used closed-source code from industry. Lastly, two papers sourced from programming contests. Overall, the majority of snippets were self-made or from open-source software, with snippets from proprietary industry projects playing no major role.

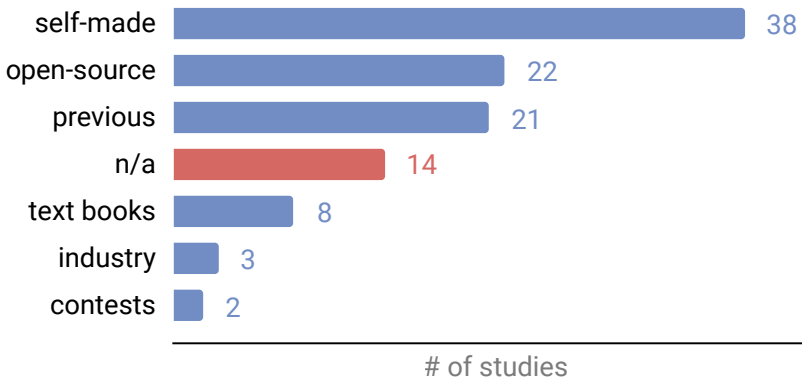


Figure 3.10.: Sources for code snippets

Finally, we extracted the *snippet criteria*, i.e., the requirements researchers had for the selection or creation of code snippets and how they motivated snippet usage. In total, we identified 17 criteria (see [Tables 3.3](#) and [3.4](#)).

Studies were labeled with between 0 and 8 criteria, with a median of 2 and a mean of 2.6. While 6 of the 95 studies did not specify any criterion

Table 3.3.: Criteria for the creation or selection of experiment code snippets (part 1/2, s. = studies)

Snippet Criterion	#s.	Description
balance between simplicity and complexity	42	simple/small enough to be understood in a reasonable time, but complex/large enough to require some effort and to be realistic
homogeneity concerning potential confounders	29	minimize the influence of unrelated factors such as naming styles, LoC, line length, formatting, or indentation
novice friendliness	25	common beginner programming tasks, typical text book problems or algorithms known to students such as shell sort or binary search
taken/adapted from existing studies	21	(partially) using/adapting snippets from existing studies against the results can be compared
real-world code	19	taken/adapted from open-source or industry code
self-contained functionality	19	no additional context or external functions necessary, e.g., a function, static method, main method, test class, etc. with a determinable purpose
heterogeneity concerning the targeted constructs	18	differences in code metrics, control structures, method chains, comments, etc. to increase variance and generalizability
requires bottom-up comprehension	15	obfuscate identifiers, remove beacons and comments, etc. to force line-by-line understanding
thematic heterogeneity	13	a different algorithm, domain, etc. per snippet to avoid learning effects or to increase generalizability

Table 3.4.: Criteria for the creation or selection of experiment code snippets (part 2/2, s. = studies)

Snippet Criterion	#s.	Description
no unnecessary cognitive load	12	small inputs, simple arithmetic, no recursion, no non-deterministic or non-portable code, low computational complexity, no advanced OO, etc.
no special domain knowledge necessary for understanding	10	avoid specific domains to reduce bias due to participant (in)experience, use familiar use cases like Pacman
written in a popular/familiar programming language	9	mainstream language or familiar to participants, e.g., Java, C, Python
n/a	6	not explicitly specified in the paper
small enough to avoid extensive scrolling	5	code fits completely on screen, especially important for, e.g., fMRI or eye-tracking studies
random sampling	5	snippets are randomly drafted from a pool
no extensive usage of (rare) APIs	4	avoid specific APIs to reduce bias due to extensive language experience
never seen before by participants	3	self-created/adapted to be new to participants
code written by participants	1	participants needed to be familiar with the code and should have written it themselves

(n/a), 22 studies provided 4 or more rationales. As examples, [Dolado et al.](#) did not provide any motivation why they used these snippets in S7, whereas [Siegmund et al.](#) used nearly a full page to describe their snippet rationales with admirable details in the dedicated subsection “Designing and Selecting Code Snippets” (S84, labeled with 8 criteria). Snippet criteria fell into one of three categories: a) related to *experiment design*, b) related to *generalizability*, and c) related to *participants*.

Experiment design: By far the most used criterion (42 of 95 studies)

was that snippets needed to fulfill a delicate *balance between simplicity and complexity*. They had to be simple and small enough so that participants could understand them within a reasonable timeframe of the experiment yet, simultaneously, they had to be large and complex enough to require a minimum of effort and to not lose all realism. What this meant in practice, however, could be very different depending on the studied constructs and the experiment design: Hansen et al. (S39) only used snippets between 3 and 24 LoC, Scalabrino et al. (S5) aimed for between 30 and 70 LoC, while Castelhana et al. (S10) argued that between 50 and 100 LoC would be ideal for their experiment. Another prominent dichotomy was that snippets should possess both *homogeneity concerning potential confounders* (29) and *heterogeneity concerning the targeted constructs* (18). In this sense, researchers normalized unrelated factors in their snippets, e.g., naming styles or formatting, but created differences within their studied constructs such as code metric ranges or control structure usage to increase variance or generalizability. A special case to avoid confounders was the conscious *elimination of unnecessary cognitive load* (12). Researchers removed or avoided, e.g., complex arithmetic, recursion, or high computational complexity to allow participants to focus on the comprehension of the essential parts of the study. In 19 studies, it was important for researchers that snippets represented *self-contained functionality* with a determinable purpose. These were, e.g., complete functions, static methods, or main methods. Additionally, 15 studies ensured that a snippet *requires bottom-up comprehension* by obfuscating identifiers and removing beacons and comments, thereby forcing line-by-line understanding. Lastly, five studies ensured that a snippet was *small enough to avoid extensive scrolling* because some fMRI or eye-tracking experiments required that snippets fit completely on screen.

Generalizability: The most prominent criterion in this category was using snippets *taken or adapted from existing studies* (21) to build on previous knowledge and to make results comparable. Similarly, several papers required *real-world code* (19), i.e., that snippets were taken or adapted from either open-source projects or proprietary industry projects to be realistic. A few studies combined this with *random sampling* (5) from a large corpus

of open-source projects. Furthermore, 13 studies prioritized the *thematic heterogeneity* of snippets and used, e.g., different algorithms or application domains per snippet, mostly to increase generalizability, but sometimes also to avoid learning effects.

Participants: Since a large portion of the studies relied on students as participants, a prominent snippet criterion was *novice friendliness* (25). Therefore, common programming tasks for beginners and typical problems or algorithms from text books were selected. Similarly, some researchers also ensured that *no special domain knowledge was necessary for understanding* (10) to not be limited to participants from certain backgrounds. In some cases, popular use cases or applications were chosen, e.g., Yu et al. used the game Pacman in S20. To increase their sampling pool, researchers also prioritized code *written in a popular programming language* (9) that was familiar to their participants (e.g., Java, Python, or C) and *avoided extensive usage of (rare) APIs* (4). Lastly, three studies ensured that their snippets were *never seen before by participants* and a single study (S42 by Sedano) required *code written by participants* so that they were very familiar with it.

3.2.3.7. Comprehension Tasks and Measures

The core of any code comprehension study, at least from the participant's point of view, is the actual code comprehension task. Such a task, sometimes several in a single study, provides data for the analysis of a participant's code comprehension performance. The precise nature of this analysis and the data collected for it are defined by comprehension measures. In this section, we first look at the variety of designed comprehension tasks, then at the variety of comprehension measures, and finally at how comprehension tasks and comprehension measures co-occur.

Comprehension Tasks. Table 3.5 shows the variety of comprehension tasks and how many studies implemented which type of task. The classification is a refinement of the division used by Oliveira et al. [OBMC20] to group comprehension and reading tasks in their study. All tasks could be assigned

to four overarching task categories (Tc): provide information about the code (Tc1), provide personal opinion (Tc2), debug code (Tc3), and maintain code (Tc4).

81% of all papers (77/95) use at least one task that requires participants to provide information about a code snippet to be understood (Tc1). Among these, 67.5% exclusively used tasks from Tc1. Three tasks in particular stand out as most frequently used: answering comprehension questions (33), determining the output of a program (26), and summarizing code verbally or textually (22).

27 (28.4%) of the papers were interested in a personal assessment of the participants (Tc2), of which 7 exclusively used tasks of this category. Preferably queried was a rating for the perceived code comprehensibility, in two cases for one's own code comprehension, four times for the confidence in one's own comprehension, in five papers the difficulty of the task was to be assessed, and in four papers code snippets were compared or ranked regarding the comprehensibility of other code snippets in these studies. In 7 studies, subjective evaluations were requested, which differ from all others and are therefore listed under "other subjective indications" (e.g., S46: provide opinion of identifier type importance, or S78: identify most helpful code lines).

For 11 studies, a debug task (Tc3) was used to derive statements about code comprehension. Five of them exclusively considered this task category. In seven papers, and thus most frequently in Tc3, participants had to find a bug. In three studies, a bug had to be fixed. One study required a judgment about whether the code was correct or not.

We identified eight papers that asked participants to complete a maintenance task (Tc4). In three cases, a Cloze test was used, a test in which gaps in the code must be filled. In three studies, code had to be extended, and in two studies, code had to be refactored. An example of the latter case is from [Bois et al.](#) (S82), who investigated the differences in the comprehension effectiveness between refactoring code and reading code.

While 66 of the 95 papers used exactly one task category, it was three for S4, S33 and S38. In two studies (S18 and S19), we could not find any

Table 3.5.: Comprehension tasks that participants had to work on

Comprehension Task	Studies
Tc1: Provide information about the code	
Answer comprehension questions	S5, S8, S31, S32, S33, S34, S36, S37, S43, S44, S45, S51, S52, S54, S56, S62, S63, S67, S68, S69, S71, S76, S77, S80, S81, S82, S83, S86, S89, S90, S91, S92, S93
Determine output of a program	S1, S2, S5, S7, S9, S11, S12, S13, S14, S15, S16, S17, S23, S25, S37, S39, S48, S50, S57, S59, S60, S68, S70, S77, S84, S88
Summarize code verbally or textually	S4, S11, S20, S24, S29, S35, S36, S37, S38, S42, S46, S47, S49, S58, S61, S72, S75, S77, S79, S87, S94, S95
Recall (parts of) the code	S29, S44, S61, S66, S72, S73, S74
Determine code trace	S35
Match with diagram or similar code	S77, S84, S85
Tc2: Provide personal opinion	
Rate code comprehensibility	S4, S6, S34, S36, S42, S49, S53, S69, S70, S75
Rate own understanding	S5, S31
Rate task difficulty	S12, S14, S23, S38, S51
Rate confidence in answer	S23, S24, S33, S94
Compare or rank snippets	S21, S22, S40, S78
Other subjective indications	S21, S22, S33, S41, S46, S78, S85
Tc3: Debug code	
Find a bug	S3, S10, S28, S38, S55, S64, S87
Fix a bug	S26, S27, S77
Determine if code is correct	S33
Tc4: Maintain code	
<i>Cloze test</i> on code	S4, S30, S74
Extend the code	S21, S26, S27
Refactor code	S65, S82

information about the comprehension tasks in the paper. Perhaps the most unusual comprehension task was reported by [Gilmore and Green \(S90\)](#), who presented their comprehension questions as crimes that a detective had to solve based on information from an informant. The peculiarity of this study was that the sample consisted of non-programmers, and therefore an interesting scenario had to be created.

Comprehension Measures. Almost all studies have measured how well participants perform in comprehension tasks. The three studies S16, S17 and S54 were the exception; participants had to understand code, but there was no attempt to quantify how well they understood it, since, for example, the focus was on the observation of psycho-physiological responses only. In all other cases, the comprehension measures used can be divided into six categories: correctness, time, subjective rating, physiological, aggregation of the former, and others.

[Figure 3.11](#) shows the percentage of comprehension measures to the number of papers in each period. In all decades, the share of papers that measured at least correctness was highest. Before 2000, correctness was even measured in all published papers, but the share decreased to 65.6% in 2010–2019. The proportion of papers that asked for subjective self-assessments or measured time increased over the years. Most notable, however, is the increase in the use of physiological measures, which were used in over a quarter of publications in 2010–2019.

measure used in % of papers

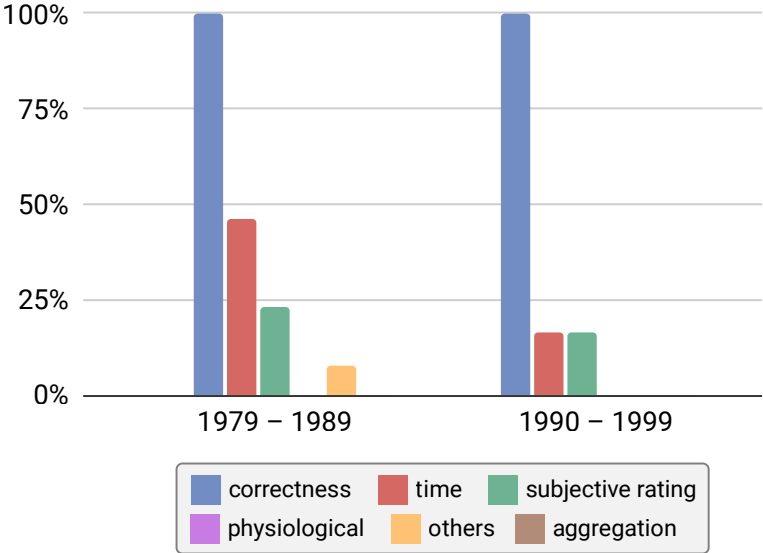


Figure 3.11.: Comprehension measures used relative to number of papers

Of the 95 papers, only 37 measure code comprehension in a single way. 36 papers rely on 2, 15 papers rely on 3, and 4 papers rely on 4 measures from different categories. The measurement of correctness and time was the most frequent combination of measures with 23 papers, if more than one measure was collected. In the period 2010–2019, we also identified for the first time four papers that not only measured multiple comprehension measures in their study, but even aggregated them. This involved combining correctness and time in three studies (S9, S26, S27). Scalabrino et al. (S5) aggregated correctness, time, and subjective ratings in different ways: by combining correctness and time, but also correctness and subjective ratings, and time and subjective ratings.

At this level of granularity, it is possible to summarize very well which trends there have been over the years and which way of measuring code comprehension seems to be the most popular. It is also closest to the terminology used in the primary studies themselves for the various measures if an abstraction of the specific calculation is used in the report (e.g., “The metrics used to investigate these questions are time and correctness [...]” [AWF18]). A closer look, however, reveals that almost every paper differs in the concrete implementation of the measures.

Consider *correctness* as an example. With 70 papers (73.7%), it is the most prominent comprehension measure in our dataset. Our primary studies often measured it via the relative or absolute number of correct answers to comprehension questions or bug finding tasks. However, we also found the usage of f-measures (e.g., S63, S65), number of correctly recalled identifiers or statements (e.g., S29, S73), number of fails until a correct answer was given (e.g., S28), number and severity of errors (e.g., S48), as binary variable (e.g., S61, S62), manually rated functional equivalence of recalled code (e.g., S73), automatically or manually rated code changes (e.g., S20, S26), and manually rated free-text or oral answers (e.g., S24, S29, S36, S39, S79, S94).

For the measurement of time and subjective ratings, there is a similarly large variance in the concrete implementation. For more detailed insights, we refer the interested reader to our dataset [WBW22]. At this point, we

would rather elaborate on the measurement of code comprehension via physiological measures, since these are still relatively novel in the field of code comprehension as a whole.

The category of physiological (or psycho-physiological) measures includes the measurement of code comprehension via responses of the human body or brain to performing code comprehension tasks. In our dataset, we encountered the measurement of fMRI data (brain activation regions, concentration levels, brain activation strength, cognitive load, blood oxygen level dependent), fNIRS data (regional brain activity), and eye-tracking data (visual/mental effort, e.g., on gaze behaviors such as number of fixations and their duration).

The primary studies that use such measures already provide initial findings on physiological responses in code comprehension, but are currently still largely to be understood as feasibility studies. For example, they investigate correlations with conventional comprehension measures (see, e.g., S1, S12–S19 and also [Section 3.2.3.2](#)). In the medium term, psycho-physiological measures could provide more objective information about the process and performance of an individual during code comprehension [[Fak18](#)] and, e.g., help to investigate “the role of specific cognitive subprocesses, such as attention, memory, or language processing” [[S17](#)].

Finally, we note that we could not assign measures to the categories described above in only five cases. These measures ended up in the “other” category. S21 and S59 used code metrics to measure code comprehension, S28 visual focus via a sliding window (not via physiological measures), S47 the number of animation runs of a code visualization tool, and S81 the number of times an algorithm was brought into view. However, none of the five corresponding papers exclusively used these measures, so they are also represented in at least one of the other measure categories.

Combinations of tasks and measures. Counterintuitively, comprehension tasks and comprehension measures were rarely mutually dependent in their selection. Almost all comprehension measures, or at least the categories pre-

sented, can be applied to all types of comprehension tasks that we identified in this mapping study. Nevertheless, some combinations occur significantly more often than others.

The combination of correctness and Tc1 (provide information about the code) appears most frequently; about two thirds (66.3%) of all papers use at least this combination of measure and task, and 16.8% of all papers use this combination exclusively. If debugging tasks (Tc4) are used, then at least correctness is measured in all cases. If, in contrast, tasks from Tc2 (provide personal opinion) are used, correctness only occurs in 55.6% of these cases in the paper. Tc2 tasks are mainly evaluated by subjective ratings (96.3% of all tasks in this task category appeared in papers at least with subjective rating measures).

Summary. In summary, there has been a change in the use of measures over time, both in terms of increasing variety and decreasing dominance of individual measures. The concrete design of comprehension tasks is in itself very diverse. Moreover, there are almost unlimited possibilities for combining tasks and measures, which makes almost every paper unique in its way of measuring how well a participant understood source code. We discuss the consequences of this observation in [Section 3.2.4](#).

3.2.3.8. Reported Limitations

We extracted study design limitations described by the authors of the primary studies. Already in the extraction of the limitations, we have only extracted those limitations that the authors themselves also considered as such and not aspects that the authors explicitly assume not to be threats because, for example, they have been mitigated. The extracted threats were then analyzed qualitatively and quantitatively in several steps, which we describe below.

Each limitation was labeled, and the resulting label was assigned to a class of validity threats. The classification is one that is usually used for reporting empirical experiments in software engineering [[JCP08](#); [WRH+12](#)]: internal,

external, construct, conclusion, and other validity threats. If the authors themselves have made use of this classification, we have followed their distinctions (which was the case for 30 papers). Otherwise, we have classified the labels ourselves by following the descriptions provided by Jedlitschka et al. [JCP08]:

- “Internal validity refers to the extent to which the treatment or independent variable(s) were actually responsible for the effects seen to the dependent variable.”
- “External validity refers to the degree to which the findings of the study can be generalized to other participant populations or settings.”
- “Construct validity refers to the degree to which the operationalization of the measures in a study actually represents the constructs in the real world.”
- “Conclusion validity refers to whether the conclusions reached in a study are correct.”
- “Other threats than those listed above may also need to be discussed, such as personal vested interests or ethical issues regarding the selection of participants (in particular, experimenter-subject dependencies).”

The individual labels are similar in part because different studies report similar limitations. We therefore categorized the labels to better grasp the diversity of limitations. The work by Siegmund and Schumann [SS15], who compiled a list of confounding parameters in program comprehension studies, builds the basis for these categories.

Since [Siegmund and Schumann](#) focused on confounders, their categories and descriptions are mainly focused on the consequences for internal (and construct) validity. We nevertheless tried to stay close to this list to allow for comparability of results. To accomplish this, we have slightly renamed a few categories and added missing categories as needed. Existing category descriptions were rewritten so that they answered the question of how the

authors of a primary study think that an aspect of validity of their study could be limited or threatened. All category descriptions are part of our supplemental materials [WBW22].

The specific aim of this review of reported limitations is to assist researchers in designing new studies by providing them with a list of threats that other researchers have previously reflected on for their design. We therefore considered it useful to group the categories according to different design aspects. For example, when selecting code snippets, one can specifically look at the list of threats found under “code snippet selection”, and when designing a comprehension task, consider the threats under “task design”. In such a phase of the study design, all potential threats have to be assessed anyway, regardless of how their consequences can be classified. Therefore, threats to internal, external, construct, and conclusion validity can be equally represented behind the respective categories.

Results. Of the 95 studies, 79 reported threats to validity. Interestingly, for the other 16 (16.8%), no correlation with publication year or venue can be found. Nine papers were published up to 2006, so they tend to be older. However, the remaining seven papers without threats to validity were published since 2017, i.e., in the past five years. Moreover, 13 of them are either journal or conference articles. Only three are workshop papers, where one can assume that space limitations are the reason for not reporting limitations.

The analysis resulted in 376 labels, which is equivalent to 376 individual, non-unique threats to validity that have been reported. As can be seen in Figure 3.12, most of the labels are equally distributed between internal (39.4%) and external (40.7%), followed by construct validity threats (15.4%) and rare cases of conclusion validity threats (4.3%). One can assume that at least one internal or one external threat is reported in a paper if threats to validity are reported. Nevertheless, at least one construct validity threat is reported in 46.8% and at least one conclusion validity threat in 17.7% of the papers that report any threats to validity.

The only label we assigned to the *other* class was a threat from study S17, which can be described as a high data drop-out due to the chosen comprehension measures (fMRI and eye tracking) in a proposed methodology. Since this class of *other* threats did not occur significantly often, we will only discuss internal, external, construct, and conclusion threats in further reporting for simplicity.

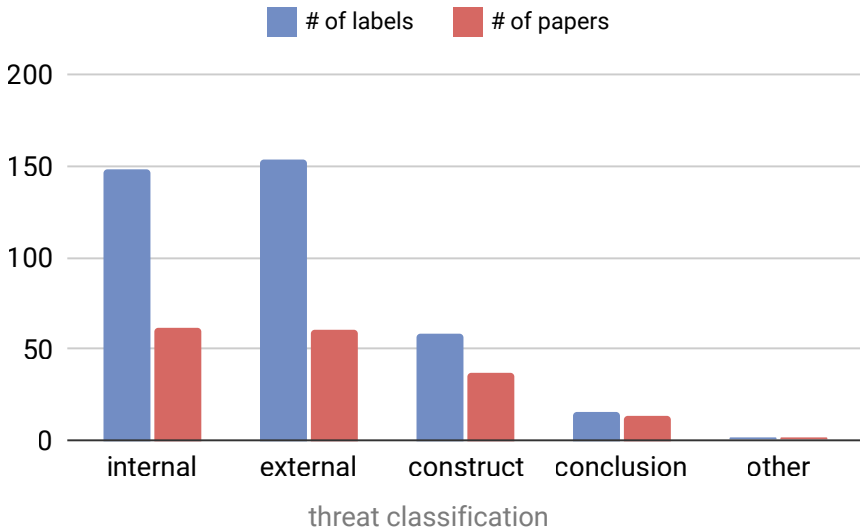


Figure 3.12.: Number of labels and papers within the threat classification

Tables 3.6 and 3.7 constitute the main contribution of this section. It lists 52 threat categories, 33 of which come from the study by Siegmund and Schumann [SS15]. For each category, the number of assigned labels and the distribution of the labels among the threat classifications are provided. In the following, we highlight some interesting findings.

Table 3.6.: Frequency of reported threat categories. For each category, a stacked bar chart indicates the share of threats assigned to one of the four classes (**internal**, **external**, **construct**, **conclusion**)






















Threats Related To	#Labels	
Participants		
Individual background		
Color blindness	0	
Culture	2	
Gender	3	
Intelligence	1	
Background (generic)	5	
Individual knowledge		
Ability	3	
Domain Knowledge	1	
Education	4	
Familiarity with study object & context	7	
Familiarity with tools	3	
Programming experience	23	
Reading time	0	
Individual circumstances		
Fatigue	5	
Motivation	5	
Treatment preference	0	
Stress	1	
Affective state	1	
Selection		
Sampling strategy	43	
Data analysis		
Data consistency	4	
Mortality and Study-object coverage	2	
Subjectivity in evaluating data	16	
ML data and model	2	
Data quality	12	
Preliminary of exploratory results	2	

Table 3.7.: Frequency of reported threat categories (continued). Bar chart colors: **internal**, **external**, **construct**, **conclusion**

Threats Related To	#Labels	
Materials		
Code snippet selection		
Content of study object	35	
Programming language	18	
Layout of study object	2	
Size of study object	22	
Number of study objects	7	
Snippet sampling strategy	7	
Code environment		
Code editor	7	
Snippet context	2	
Code environment (generic)	1	
Supporting materials and resources	1	
Task design		
Difficulty	8	
Time limit	2	
Task procedure	14	
Task description / study introduction	5	
Operationalization and measures		
Instrumentation	39	
Mono-method bias	1	
Mono-operation bias	4	
Construct operationalization	9	
Research design, procedure and conduct		
Learning effects	12	
Ordering	1	
Experimenter-subject interaction	1	
Evaluation apprehension	3	
Hawthorne effect	3	
Process conformance	5	
Technical issues	3	
Causal model	15	
Concentration impairment	2	
Diffusion or imitation of treatments	2	

First, we did not assign labels to three categories that we adopted from Siegmund and Schumann [SS15]: *Color blindness*, *Reading time*, and *Treatment preference*. This may be explained by our restriction to bottom-up source code comprehension studies, which means that we only consider a subset of the program comprehension papers that Siegmund and Schumann [SS15] analyzed and that these threats happened to be discussed only in the other papers. Threats in these three categories also occurred comparatively rarely in the analysis by Siegmund and Schumann [SS15]. We nevertheless left these categories in the table for ease of reference.

Second, many discussed threats are reported quite generically in our primary studies, i.e., without a clear description of what the actual threat is and without an explanation, why a certain design property is a limitation. For example, with 43 label occurrences, the rather generic category *Sampling strategy* is the largest threat category in our dataset. It contains, e.g., descriptive statements of the resulting participant sample of a primary study, such as all participants being students, and that it is unclear whether such a sample can be generalized to industrial professionals. Labels in this category do not provide a rationale for why such a threat would exist, although we identified 17 other participant-related categories that would allow for more nuanced assumptions. Speaking of assumptions, it is furthermore striking that hardly any studies explicitly cite any evidence at all for the suspected threats to validity. We address the issue of potential evidence gaps and the need for more intensive reflection on the threats to one's own primary study in our discussion.

Third, in some cases, threat classes dominate individual aspects of a study design. For example, the selection and design of the *materials* predominantly has a presumed influence on the external validity of a study. Researchers often discuss that their code snippets are relatively short (*size of study object*) and therefore the generalizability to longer snippets is unclear. By contrast, when it comes to *research design, procedure and conduct*, consequences for internal validity are discussed almost exclusively: *learning effects* and general threats to ensure causal relationships (*causal model*) are the categories with the most labels here.

At the same time, these are examples of threat categories that mutually affect each other. More extensive code snippets would lead to more variance in the snippet features, and thus to more potentially uncontrolled influences that make causal inferences difficult. The researcher must now balance between strengthening internal validity and strengthening external validity.

Fourth and finally, measuring how well someone has understood code is apparently difficult. It is not only the case that researchers in code comprehension studies have to deal with potential threats to validity from over 50 categories. In the end, 39 labels were assigned to the *instrumentation* category and another 9 to the *construct operationalization* category, two categories that both directly address the fundamental question of how the performance or process of code comprehension is operationalized and measured. In the respective papers (e.g., S4, S9, S63, S69), it is sometimes discussed quite openly that there is uncertainty about whether the chosen way to operationalize the construct of understanding, e.g., via time and correctness, is adequate at all, and that the used measurement instruments have validity issues. Such insights point to the need for the research community to achieve more certainty in the design of code comprehension studies through discourse and research in the future, and strengthen our motivation for this work to take a step in this direction with a synthesis of the state-of-the-art.

3.2.4. Discussion

One of the motivations for our systematic mapping study was to provide researchers with confidence in their study designs by highlighting what is considered acceptable by the research community. This should not only serve as a helpful overview for newcomers to the field, but may also enrich the perspective of seasoned researchers on this topic, e.g., by providing insights into trends and identified shortcomings. To summarize which design decisions have dominated the research field in the past, we therefore start this section with a description of the typical code comprehension experiment.

On the one hand, there *are* certain design options that are taken more frequently than others. For example, the majority of studies examine the

impact of a construct on code comprehension, using one-at-a-time within-subject designs. There are usually fewer than 50 participants sampled, of which at least some are students. The used code snippets are often specifically created for the study, Java is clearly the programming language of choice, and participants are shown the snippets on a screen. When it comes to understanding the snippets, participants are required to provide information about the code, e.g., by answering comprehension questions or determining the output for a given input. The typical evaluation assumes that the faster participants provide the correct answer, the better their understanding. Last but not least, researchers are aware that every study design comes with limitations, which is why almost all papers discuss threats to validity.

On the other hand, it also became clear that ‘the typical code comprehension experiment’ does not exist. Each study in our dataset is unique. For example, there is an incredible variance in the design of the concrete comprehension tasks and measures. Almost all studies come up with their own individual task design, even though all of them ‘only’ try to measure how well a participant understood a certain code snippet. Why does every study develop its own code comprehension tasks? Are existing study designs not convincing, or does every research question actually require an individual approach?

We suspect that part of the variance is due to uncertainty. The most important driver in the design of code comprehension studies is currently the intuition of the researchers behind the studies. There is a lack of theoretical foundations on which all code comprehension studies could be built. Such foundations should offer (1) a common definition of code comprehension, (2) a mental model describing how code is represented in a developer’s mind and (3) a cognitive model that explains the process of code comprehension. A strong theory with these three parts would greatly help with anchoring new research designs and integrating empirical findings. Without such foundations, you almost cannot help but rethink every design decision to answer a research question, as it is simply not apparent from existing primary studies whether assumptions made and views held about an underlying code comprehension model are consistent with your own. In other words, so far,

everyone has been doing it in a way they think is reasonable — after all, no one *really* knows what is reasonable at the moment.

In the 1980s and 1990s, several cognitive models have been proposed for code comprehension (see [Section 2.1](#)). We expected that these models, in particular the integrated model [VV95], would be the basis to inform hypothesis development and experiment design for code comprehension studies. Yet, in most experiments, this seems not to be the case. Regarding the *hypotheses* and *study purposes* of the primary studies, we partly found concepts of comprehension theories, although often without explicitly stating this connection. Semantic cues and mental models are important concepts in the integrated model. Likewise, code structure and visual characteristics are relevant for the theory of program model structures. Yet again, these concepts were most often not directly related to the respective theory, and hence, do not act as rationales for the chosen design. The major exception is S84, a study by [Siegmund et al.](#), who explicitly used the concepts *bottom-up comprehension*, *semantic comprehension*, and *programming plans* to motivate their experiment design.

The *study themes* we identified in the primary studies are also only partly related to comprehension theories. Examples of theoretical concepts are *beacons*, while *identifier naming* or *control structures* are at least related to theoretical concepts. Similarly, the often used *experience* is related to program-domain knowledge in the integrated model.

Concerning the *code snippets* used in the studies, grounding their properties in theoretical concepts seems especially important for an empirical investigation. For example, the integrated model assumes rules of discourse about how code is usually written, and that deviating from these rules will make comprehension more difficult. Hence, we would expect an evaluation of this characteristic for the used snippets. However, in our sample, we rather saw discussions whether open-source code or artificially generated code is used. This is related, but not the same concept. As a positive example, there was usually a clear reference to theory when researchers obfuscated identifiers or removed beacons to force bottom-up comprehension.

In our opinion, the most pressing issue in this area is that the selection

of *tasks* and *measures* is not based on the integrated (or any other) theory. Measures like the correctness of answers to comprehension questions are somewhat related to the mental model of the participant. Yet, in the integrated model, we have top-down structures, program model structures, and situation model structures. Are the questions covering all three of them, or only one? How does this impact the results of the study? A study could, e.g., explicitly aim for understanding the impact of certain semantic cues in the code only on the situation model. We found only [Shn77] evaluating explicitly the developer behavior theory proposed in [SM79] by using recall measures. All other studies seem not to rely on a theoretical foundation for their tasks and measures.

Again, we consider diversity in study designs to be important for addressing research questions from multiple perspectives. However, each study design in itself should be based on logically sound design decisions. This is not currently possible, or at least difficult to evaluate, as basic research on code comprehension has diminished significantly since the 2000s. The earlier drivers of such research, e.g., on the distinction between bottom-up and top-down comprehension, have not progressed to the point where we could already support entire study designs with their models. We will come back to this when we conclude this section with specific calls to action to the research community. Before we do so, however, we will discuss a few other peculiarities that we noticed when reviewing code comprehension experiments of the past forty years. Note that the following points all draw on our empirical findings, but we deliberately refrain from citing specific negative examples because we do not consider this necessary. Instead, we highlight positive examples where available.

Lack of studies investigating the impact of code comprehension on other constructs. In our sample, not a single study investigated the effects of understanding code on something else. During study selection, we also did not find such studies that we had to exclude, e.g., due to a lack of human participants or a focus on top-down comprehension. Most may agree that

high code comprehensibility is a desirable goal in itself, but, for others, it may become even more tangible and important if consequences of good or bad code comprehensibility were known for developers and management. Examples are how code comprehension impacts the motivation to work on the code, general job satisfaction, or accuracy in the effort estimation for code modifications. This type of experiment seems extraordinarily rare, even though some studies exist that analyze this for higher-level constructs that (partly) include code comprehension, e.g., the impact of technical debt on developer morale [BGMB20].

No definitions for the studied construct(s). For the majority of our primary studies, the authors do not provide a clear definition or description for the central construct(s) that they want to measure. Rare shining examples are S1, where Peitek et al. clearly describe the differences between bottom-up and top-down comprehension, and S5, where Scalabrino et al. do the same for understandability and readability. As a consequence, some authors call their construct ‘readability’, despite clearly measuring comprehension. Likewise, several studies use the older and more general term ‘program comprehension’, even though they measure bottom-up source code comprehension. This state of inconsistency makes it very hard for primary studies to compare their findings with those of others, and severely increases the required effort for secondary studies in this field, e.g., we spent countless hours figuring out whether studies were actually measuring bottom-up comprehension.

No clear operationalization for the construct. In several studies, the researchers did not explicitly state how they collected a measure like ‘correctness’. Sometimes, they also collected multiple measurements, but did not explicitly state which of those measurements are used to operationalize the construct during the analysis. For a study that measures, e.g., the time to read code and the time to answer comprehension questions, it must become clear if only one of the measures or a combination of both was used in the analysis. A good example is again provided in S5, where Scalabrino

et al. clearly describe their aggregations of correctness, time, and subjective ratings to evaluate their construct.

Unsuitable comprehension tasks. While there is no consensus (yet) on optimal tasks for code comprehension experiments, there are definitely some tasks that seem less suitable than others. Several studies used a recall task, where participants had to recite the (exact) line-by-line code snippet to judge their comprehension. Comprehending means to form a mental structure that represents the meaning of code. From research on text comprehension, we know that “whatever mental structure is incidentally generated in the process of comprehension also serves as a retrieval structure” [Kin98]. Shneiderman similarly hypothesizes that developers who perform better on a recall task have a better understanding of the program [Shn77]. However, this is not a sufficient criterion for assessing comprehension. An evaluation with recall tasks must be designed very carefully because being able to recite something flawlessly from memory is not the same as truly comprehending its meaning or purpose. Likewise, participants unable to remember the exact lines of code may still have understood what the snippet accomplishes. In this sense, we share Feitelson’s sentiment that “failure in recalling code verbatim from memory may identify totally wrong code or code that does not abide by conventions, not necessarily hard to understand code” and that such a task is far removed from what developers do in their daily work [Fei21].

In addition, we also found multiple studies that used tasks that do not measure comprehension in isolation but include much more, e.g., fixing a bug, refactoring code, or extending code. This type of problem-solving differs from mere comprehension in the way that problem-solving is a “controlled, resource-demanding process involving the construction of problem spaces and specialized search strategies”, whereas comprehension can be simplistically summarized as an automatic process [Kin98]. In the extreme case of S73, Sheppard et al. even let their participants write functionally equivalent code for a snippet. While these tasks certainly require a certain degree of comprehension, they also introduce many more confounders and

potential biases, and would be more suitable if the studied constructs were *maintenance* or *refactoring*.

Incomplete reporting of experiment design characteristics. Several studies omit important details of their experiments in the reporting, sometimes information so basic or essential that its absence was very surprising to us. For example, we could several times not reliably infer whether snippets were self-made or taken from somewhere else (14 papers), which programming language was used (7), whether the study was conducted remote or onsite (7), whether snippets were shown on screen or paper (6), why the respective snippets had been chosen (6), what type of participants were used (4), or how many snippets were used in total and per participant (4). Many of these details are important to judge the soundness of the design or to interpret the results, e.g., concerning generalizability.

Lack of closed-source code snippets from industry. Only 3 of the 95 primary studies used code snippets from proprietary industry projects. While open-source or constructed snippets may provide sufficient realism or industry relevance for many bottom-up code comprehension results, it is still noteworthy that basically no findings in this field are validated with closed-source industry code and that generalization is assumed. Today, many companies are involved in some projects in open-source repositories. Still, given the known differences between proprietary industry and open-source projects [PSE04; RF10], it would be highly unlikely that *all* findings are fully transferrable. Even if it seems plausible for the majority of scenarios, we simply do not know the extent. We are aware that it is difficult to obtain closed-source industry code that can also be used with minimal modifications in experiments. Additionally, this might also make sharing the code snippets with the publication impossible. For now, though, it would at least be desirable to find out more about the influence of the differences between proprietary industry and open-source snippets on code comprehension and thus contribute to a more informed discussion about generalizability.

Inconsistent usage of categories for threats to validity. While 79 of 95 studies included a description of threats to validity and many authors classified them into the typical categories, several instances of incorrect classifications made our aggregation more difficult. For example, [Bavota et al.](#) assigned potential confounding factors to construct validity instead of internal validity (S63), and [Saddler et al.](#) associated the lacking representativeness of their code snippets with internal validity instead of external validity (S68). This may be due to different definitions of the validity categories, but since most authors did not provide appropriate explanations or a reference for the used classification, we cannot say for sure. At least, the inconsistent and partly incorrect categorization of threats to validity does not seem to be an issue exclusive to code comprehension studies (see, e.g., [[ABA+19](#)]).

Implicit or shallow reporting of threats to validity. Many studies report both threats that were consciously mitigated and threats that remain. This observation is in line with that of another literature study by [Sjøberg and Bergersen](#) [[SB22](#)] on reporting construct validity threats in software engineering. Sometimes, however, it was difficult to distinguish between these two types because the authors did not explicitly mark the threats accordingly. Therefore, interpreting the limitations of such studies becomes difficult. [Hofmeister et al.](#) present a good solution to avoid this issue in S3: they report how they consciously addressed common threats in their study design section, while the remaining threats are discussed separately at the end. Moreover, a few papers reported very generic threats that give the impression of simply pre-empting criticism from readers, or initially from reviewers of the manuscript. For example, if the sample consists exclusively of students, it seems almost obligatory to mention this as a limitation (see [Section 3.2.3.8](#)). However, it is rarely explained in the specific study context which student characteristics could actually threaten validity regarding the research questions. Authors of primary studies that predominantly sampled students often cast doubt on the generalizability of their findings to the population of professional software developers, but do not explain why. That

such generalizability should not be questioned per se is shown at least by the existence of controversial discussions concerning student samples [FZB+18]. How it could be done instead is shown by the authors of S26 and S82, who discuss, e.g., that the students in their sample may be influenced by the teaching assistant's coding style requirements or may not be representative because they signed up for the particular course due to their particular interest in it.

To counteract some of these identified shortcomings, we propose several calls to action for the research community in the final section below.

3.3. Conclusion

We conducted a systematic mapping study to structure the research from 95 primary studies on source code comprehension published between 1979 and 2019. Through this review of the past 40 years, we arrived at some interesting findings that should motivate changes for the future. Below, we have focused on what we believe are the five most important action items that we should address as a code comprehension research community moving forward.

1. **Work on a definition for code comprehension and provide such a definition in every primary study.** Code comprehension is a complex construct, making it all the more important to define what you intend to measure. While there are a few program comprehension models and a few vague definitions of what it means to understand a program, the landscape of contemporary definitions for code comprehension is unfortunately quite limited (see [Chapter 2](#) and in particular [Section 2.5](#)). A definition of code comprehension must not only be clear in describing the construct itself. It should also clearly distinguish code comprehension from other related constructs. Without such a reusable definition, we burden each primary study with the task of defining and separating code comprehension from other constructs, which ultimately leads to a situation in which hardly any primary study

defines what it actually intends to measure. Consequently, speculating based on the properties of the comprehension tasks is often the only way to assess whether two experiments intended to study the same construct.

2. **Research the cognitive processes of code comprehension.** In short, we need more theory. Basic research on the cognitive processes involved in code comprehension, e.g., to discover different comprehension strategies and involved cognitive processes in the brain, is necessary to theoretically motivate a study design. Almost none of our primary studies linked their design to a comprehension theory that would justify, e.g., the used tasks and measures. Basic research on program comprehension is currently gaining momentum; the use of neurophysiological measures to “develop a neuroscientific foundation of program comprehension”¹ is particularly promising.
3. **Conduct research on the comparability of different design characteristics.** We have seen that each of our 95 primary studies used a unique design to answer its research questions. We lack evidence on the consequences of particular design decisions, such as the choice of particular comprehension measures or experimental materials. We can only meaningfully compare primary studies and synthesize results in meta-studies if we know whether and how different design decisions affect study outcomes differently.
4. **Gather evidence of commonly assumed consequences of discussed threats to validity.** Related to the previous point, we also lack evidence in discussing limitations of study designs. The catalog of potential threats to validity is too long to discuss in a primary study. Accordingly, the focus should be on those threats to validity that actually affect the results for a specific study design. However, to assess for which threats this is the case, we need more research on the influence of assumed confounding factors and generalization issues. Existing lit-

¹<https://www.se.cs.uni-saarland.de/projects/BrainsOnCode/>

erature supporting the influence of specific design decisions on study results should then be referenced appropriately in the discussion of primary studies on code comprehension.

5. **Agree on a set of design characteristics that must not be omitted when reporting a primary study.** Finally, we have a suggestion for improving the future reporting of code comprehension studies. As much as we desire to grant everyone their creative freedom in the organization of their papers, we nevertheless note that it was very cumbersome to extract the relevant design characteristics from some primary studies. Although we read all the papers from start to end, the relevant information was scattered throughout the report, sometimes ambiguous, not always where one would expect it to be, and sometimes missing completely. We would like to highlight S3 and S27 as positive examples that present a tabular overview of the “main factors” of the conducted study. This form of concise presentation is not only interesting for those conducting meta-studies [Woh14b], but in our opinion also helps every reader to better understand the study design. As a community, we can discuss which specific design characteristics *need* to be included in such a table. For a start, the primary studies mentioned above as well as most categories of our [Table 3.1](#) provide a good starting point.

The research field is currently flourishing in terms of the number of new publications and number of researchers involved. We are personally pleased to see this because we consider it an important field of research. This makes it all the more important for us to emphasize at this point that, in our opinion, much of the design and reporting of code comprehension experiments is already of good quality. The emphasis on the previous pages on what we could improve in the future is not intended to slow down the growth in this field. On the contrary, the five concrete action items are meant to encourage the community to use the momentum and engage even more deeply with code comprehension studies at the meta-level.

In this thesis, we focus on action items 3 and in particular 4, which aim

to provide evidence for more clarity in the design and evaluation of study designs. We begin with the following chapter, in which the presumed influence of intelligence and personality on code comprehension is empirically investigated for the first time.

HOW INDIVIDUAL CHARACTERISTICS INFLUENCE CODE COMPREHENSION

The influence of individual characteristics of software developers on their behavior and performance in various software engineering activities has been demonstrated in several primary studies. However, the influence of individual characteristics on *code comprehension* has been limited to a few characteristics in the past, most notably developer experience (see [Section 2.2](#)). Yet, this did not prevent researchers from speculating what other individual characteristics might be potential confounding factors in their studies (see [Section 3.2.3.8](#)). In this chapter, we examine the influence of two constructs previously hypothesized to influence code comprehension: intelligence and personality.

This chapter contributes to answering **RQ2**, which investigates how individual characteristics influence developers in code comprehension. It extends our journal publication [[WW22](#)]. The presented study should be exemplary

for evidence providing studies on individual characteristics, of which we expect more in the future. Such studies help, for example, to discuss confounding factors as threats to validity in an evidence-based manner or even to control them in the study design and conduct.

4.1. Context and Goals

Program comprehension is a cognitive psychological process in which, in addition to the characteristics of the code to be understood, the capacities of the person who wants to understand the code play a role. Accordingly, researchers suspect that intelligence, for example, has an impact on program comprehension performance [SS15]. Since measuring intelligence with validated questionnaires would often exceed the intended time frame of studies, researchers discuss this potential confounding parameter in code comprehension studies as a threat to validity [SS15]. However, these discussions have so far been conducted without a solid understanding of whether intelligence is actually a significant influencing factor on program understanding. A study on this question is missing so far.

The situation is similar with the personality of developers. Although personality has not been discussed as a potentially confounding parameter in code comprehension studies yet [SS15], there are several related studies that have shown that different personality traits affect developers' performance in software engineering activities [CSC15; DG07; KMG13; SNA09; WGW19].

When designing studies, insights into the influence of intelligence and personality on program comprehension can provide a useful basis for design decisions, which can ultimately lead to greater confidence in the validity of the results and can partially counteract existing uncertainty about what constitutes a good empirical study [Sie16; SSA15]. For this reason, we conducted the first large-scale empirical evaluation to answer the following research questions:

RQ2.1 Is there a relationship between code comprehension performance and intelligence?

RQ2.2 Is there a relationship between code comprehension performance and specific personality traits?

4.2. A Study of Intelligence and Personality

4.2.1. Background and Related Work

A confounding variable or *confounder* “is an extraneous variable whose presence affects the variables being studied so that the results do not reflect the actual relationship between the variables under study” [PBV12]. For example, in code comprehension experiments the most frequently discussed confounders are the programming experience and familiarity with the study object [SS15].¹ Not identifying and controlling such factors poses a threat to internal validity due to potential false positive errors [PBV12].

The literature survey by Siegmund and Schumann of program comprehension papers published between 2001 and 2010 shows that 11 studies mentioned intelligence as a confounding parameter, but its influence was analyzed only once [SS15]. Several other individual background variables were identified, both in their study and in our systematic mapping study (Chapter 3), but personality is so far not one of them. The present study aims to provide specific evidence whether the list should be extended to include the variable personality and whether intelligence actually has the presumed impact on performance in code understanding.

We investigate the influence of intelligence and personality in a joint study because both are individual characteristics that share similar traits, such as remaining relatively stable over a lifetime [Cas00; DWL+00], yet represent distinct constructs. In addition, their joint measurement integrates well into our planned study design, ultimately saving resources for us and potential participants by not conducting multiple separate studies.

¹Usually, confounders are distinguished from covariates, which are also related to the outcome, but are otherwise not related to the treatment variable. Since the terms are often used interchangeably and the distinction is not relevant for the motivation of this work, we follow [SS15] and use confounder as an umbrella term for extraneous variables that are relevant for the interpretation of an observed relationship between two other variables.

4.2.1.1. Intelligence in Software Engineering

In the context of our study, intelligence as a construct is defined by the test used to measure it [KLH13], and we elaborate on the test further in sections 4.2.2.5 and 4.2.2.6. At this point, it is sufficient to understand that our assumptions (and those of the test) are based on John Carroll's Three-Stratum Theory of intelligence [Car+93; Car05]. A series of subtests administered to participants in our study operationalizes first-order factors and allows the measurement of four second-order factors, i.e., crystallized intelligence, fluid intelligence, visual perception, and cognitive speed. All test performances combined serve to estimate general intelligence, also called g factor, as a third-order factor. Figure 4.1 visualizes the used intelligence test in the structure of the three-stratum theory. Three dots in the middle indicate the presence of other second-order factors that determine general intelligence but are not measured by the LPS-2, i.e., memory and learning, auditory perception, retrieval ability, and processing speed.

When researchers refer to intelligence as a potential confounding parameter in discussions of their code comprehension studies [SS15] and do not further specify intelligence as a construct, we suspect that this is deliberately left open. Whether the g factor, certain dimensions of intelligence, or a combination of both could be responsible for some developers understanding code more easily than others is difficult to assume as long as we still know too little about the concrete cognitive processes in the brain of a developer or even concrete studies on the research question, such as the present one, exist.

Fortunately, in recent years there has been a trend towards more neuroscience and physiological studies in the field of program comprehension [Fak18; PSP+18; SPB+20], for example, using fMRI scanners to investigate the active brain regions and cognitive processes of developers during source code comprehension. Peitek et al. [PSA+18] found in an fMRI study with 17 participants and subsequent replication with 11 participants that bottom-up program comprehension involves activation of five brain regions that are related to working memory, attention, and language processing. Ac-

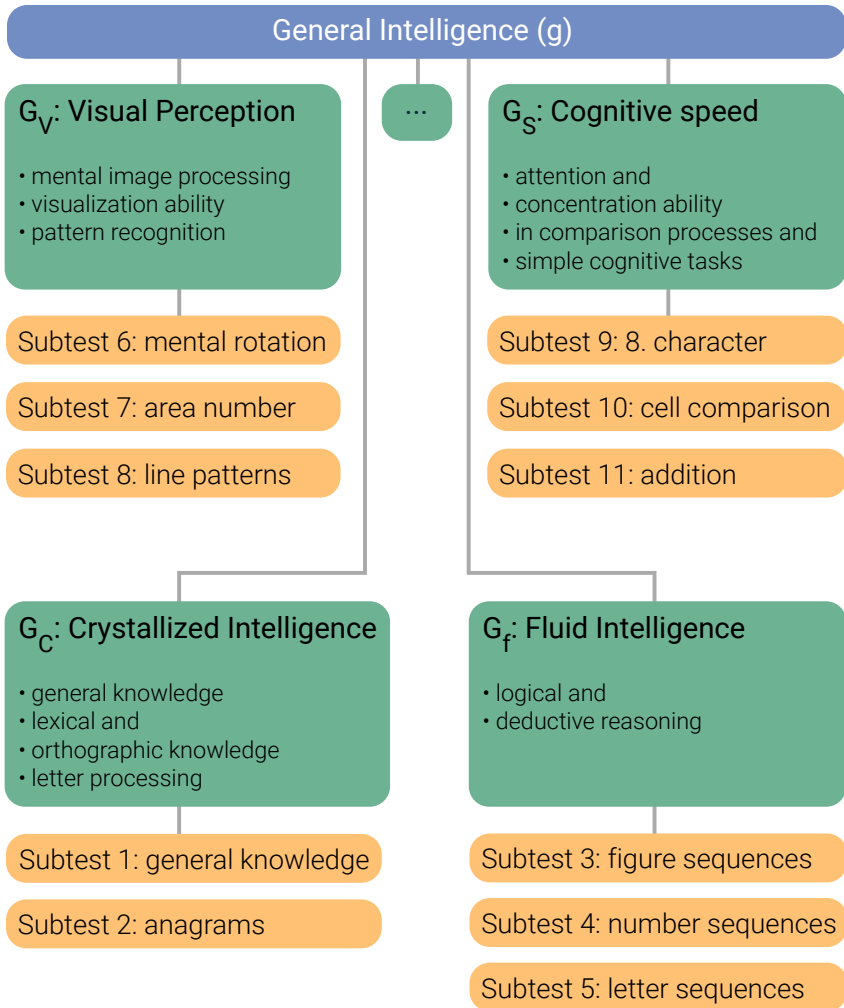


Figure 4.1.: Simplified representation of *LPS-2* for measuring intelligence, depicted in the structure of the three-stratum model by Carroll [Car+93; Car05]. Figure based on [KLH13].

cordingly, we suspect that at least the intelligence factors cognitive speed and crystallized intelligence should also correlate positively with performance in code comprehension in our study.

While such studies with new ideas for measurement methods already provide us with interesting early research findings, their implementation currently still represents an evaluation of these very measurement methods. To the best of our knowledge, there is no large-scale study that is directly aimed at investigating the relationship between program comprehension performance and intelligence. However, there are a few studies in software engineering which measured intelligence as part of their design.

Ko and Utzl [KU03] conducted an exploratory experiment with 75 undergraduates and measured, among other individual characteristics, verbal intelligence with the *Vocab27* test as well as problem-solving ability with a *problem-solving* test consisting of items from various intelligence tests. These individual differences did not appear to correlate with the success in debugging a program in an unfamiliar programming system.

Mindermann and Wagner [MW20] found in an experiment with 76 undergraduates that the successful usage of cryptographic libraries in terms of effectiveness, efficiency and satisfaction with the help of examples is not related to the participants' fluid intelligence. Fluid intelligence can be summarized by the ability for logical and deductive reasoning, which is why the non-existent influence on effectiveness and efficiency of task processing were surprising for the authors of the study.

4.2.1.2. Personality in Software Engineering

Since there are many definitions of the term *personality*, we are guided by what Ryckman describes as a consensus among investigators, that is, “the dynamic and organized set of characteristics possessed by a person that uniquely influences his or her cognitions, motivations, and behaviors in various situations” [Ryc12].

This definition was also the basis for a systematic mapping study by Cruz et al. [CSC15]. Their study provides us with valuable insights into forty

years of research on personality in software engineering (1970–2010). Several studies found that personality traits correlate with performance in various SE tasks [CSC15; DG07; KMG13; SNA09; WGW19], others found no significant relationship between personality and programming performance [BHH+09, e.g.]. However, of the nine papers found in the mapping study that could be classified under the research topic of individual performance, none explicitly examined the influence of personality on code comprehension performance [CSC15].

Karimi et al. [KBGW16] investigated how personality affects programming styles, including the approach to code understanding. Programmers with high conscientiousness tended to use depth-first style, and those high in openness to experience tended to use a breadth-first style. They further found that programmers who tended to use a depth-first approach often showed better programming performance.

The closest to our research questions is a study by Arockiam et al. [ABUL05] on the influence of personality traits on the correctness of comprehension questions on C++ programs. Unfortunately, the paper lacks a comprehensive description of the design that would enhance our confidence in the validity of the results. Furthermore, the used personality model and test seem not to be validated.

In summary, neither the influence of personality nor that of intelligence on code comprehension performance has been sufficiently studied to date. Since such insights would be useful for researchers in the field of code comprehension in designing studies and ensuring validity, we are taking a step in this direction and are beginning to fill the identified research gap.

4.2.2. Methodology

We follow the guidelines of Jedlitschka et al. [JCP08] on reporting experiments in software engineering.

4.2.2.1. Goals

The goal of the study is to analyze the impact of different facets of intelligence as well as personality traits on the performance in understanding source code. To this end, we formulated the following two research questions:

RQ2.1 Is there a relationship between code comprehension performance and intelligence?

RQ2.2 Is there a relationship between code comprehension performance and specific personality traits?

4.2.2.2. Research Design

We conducted a correlational study: We did not manipulate intelligence or personality, but measured them as given subject characteristics. This is a common study design in any study where the independent variable cannot be manipulated for practical or ethical reasons [Cro57; Roh18]. With this in mind, in [Section 4.2.2.7](#) we describe a causal model that provides the basis for a discussion of causal inferences from our collected data [Roh18].

We provided participants with a consent form and informed them in writing about the aim of the study. The study took place simultaneously and on-site for all participants at two given time slots on two days. Participants had the opportunity to ask questions at the beginning of each session.

In the first session, the participants received a short questionnaire on their programming experience as well as two code snippets for which they had to answer comprehension questions under time limits. In the second session, participants took an intelligence test and a personality test. We linked the data of both sessions by a panel code¹ that preserved the anonymity of the participants.

A schematic representation of the research design is provided in [Figure 4.2](#).

¹Participants answered 10 questions, with each answer consisting of one to a maximum of two characters. The concatenation of the characters results in the panel code. For example, one question is: *Your birthplace, 1st and 2nd letter*. We only considered data for which there is a matching panel code in both sessions in the analysis.

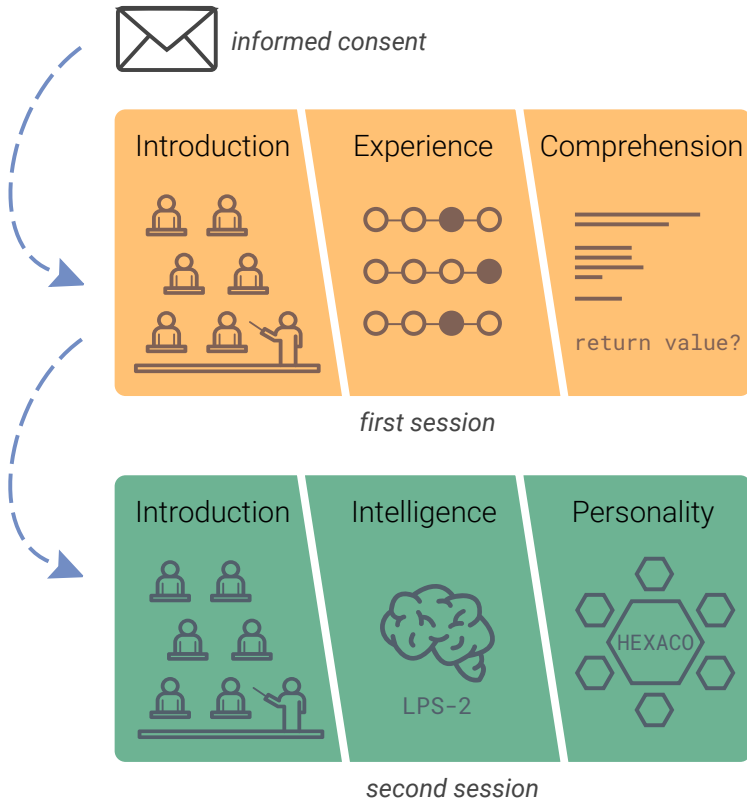


Figure 4.2.: Schematic representation of the research design. Icons [Ico21]

4.2.2.3. Participants

We consider everyone with at least one year of Java programming experience suitable for the experiment. Therefore, we invited a convenience sample of students of one of our computer science programs (Computer Science, Software Engineering, Media Informatics, Data Science) in their second year to participate in our study. All had mandatory Java courses in their first year. We see the sample properties of interest, namely enough experience to comprehend medium to understand Java code, ensured by our sampling

strategy so that the findings can be partially transferred to a population of experienced Java software engineers. We will discuss limitations of our sampling strategy and their implications in [Section 4.2.4.1](#).

We recruited students enrolled in the same course to keep their academic experience level and programming language experience similar. As part of their study duties, students had to participate in any study offered by the faculty. Students had the right to register for a study and then withdraw their participation at any time without consequences.

We informed the participants in writing before the first session about the study design, health risks, privacy and ethical issues and our contact details. Furthermore, all goals were open, and the study contains no deceptions of the participants. The panel code allowed the organizers to provide the individual's results for their personality and intelligence tests. This aspect was intended to motivate participation in the study.

To determine the needed sample size for our analyses, we conducted an a-priori power analysis. By convention, we used $\alpha = .05$ and $\beta = .2$. We wanted to be able to detect small effect sizes because they could still be interesting as confounding factors in a comprehension study. Therefore, we chose what Cohen [[Coh92](#)] considers a small effect: 0.2. Using the *pwr.t.test* function of the *pwr* R package, we calculated an optimal sample size of 198.

4.2.2.4. Tasks

Participants were shown two independent Java methods, one after the other. For each code snippet, five input values were given, for which the participants were asked to specify the return values according to the Javadoc and to determine the actual return value. Participants knew that the answers would be rated on correctness.

Since we told our participants that there might be bugs in the code, they could not rely on the Javadoc comment and had to understand what the code actually does. We consider the deviation of the documentation from the code and the inspection based on concrete values for the input parameters to be a realistic scenario. Furthermore, the task is in line with the conceptual

model that a developer in a maintenance scenario iteratively constructs and tests hypotheses about the functioning of the code during program comprehension [VV95].

There was a time limit of 12 minutes for processing each of the two code snippets. In between, there was a short break of two minutes.

4.2.2.5. Experimental Materials

The experiment took place in both sessions in a large lecture hall. Neither the participants nor the experimenters used electronic devices. All materials were presented to the participants on paper. We make all experimental materials publicly available (see [Appendix B](#)) except for the personality and intelligence tests, which we cannot republish for legal reasons.

Code Snippets. We used a total of two Java code snippets to conduct the study. Each code snippet consisted of exactly one method and its Javadoc documentation. The code was highlighted as in an Eclipse IDE with default settings.

The first code snippet to be understood was a solution to a coding challenge [WGW19], such as those given in programming contests or technical interviews. The method comprises 15 SLOC, has two parameters and contains, among other things, two nested for-loops.

The second code snippet was a method from the Apache Commons library for converting an integer to a boolean object. The method spans 18 SLOC, has four parameters, and is characterized by several if-else branches.

We selected the snippets in a way that no uncommon prior knowledge on, e.g., frameworks, would be required to understand them. As a result, the code contained mostly primitive data types and the features of newer Java versions were avoided. Cognitive complexity, a validated metric for assessing the comprehensibility of methods [MWW20], was 9 for the first and 8 for the second method, corresponding to moderately difficult comprehensibility.

In our study, we considered two code snippets to be a sufficient number to maintain a balance between appropriate time commitment from participants

and generalizability of results. First, understanding two code snippets means that participants already have to concentrate for 24 minutes, and second, from an ethical point of view, no more time is taken up by participants than is probably necessary to gain insightful knowledge. Since we have limited ourselves in the selection of code snippets as previously described, we view the two selected snippets as representative of code with the previously described characteristics. We acknowledge that this is a personal view of the authors, and different studies have handled the selection and number of code snippets very differently so far.

Comprehension Questions. For each of the two snippets, we provided the participants with a paper-based form which included five rows of a three-column table that had to be filled in. The cells of the first column each contained a method call, for example `toBooleanObject("1,1,0,null")`. The other two columns had to be filled with the actual return value of the method and the expected return value according to the Javadoc.

Questionnaires. To measure programming experience, we followed the recommendations of Siegmund et al. [SKL+14] to use self-assessment questions on general programming experience, programming experience compared to fellow students and with the object-oriented paradigm.

For the measurement of intelligence, we searched for an intelligence test that is established in psychology with a correspondingly good validation. Furthermore, the test should be detailed enough to distinguish between different intelligence factors, but also short enough to be conducted together with the personality test in a lecture slot of 90 minutes. The latter was important to be able to have the participants on-site and in an available lecture hall. Furthermore, it should be a paper test, as not all participants might carry a suitable device for an electronic test. Ideally, the tests should be freely available to support the easy replication of our study.

We found the LPS-2 questionnaire [KLH13] to fulfill most of our criteria, but, unfortunately, it needs to be bought from the publisher. LPS-2 mea-

sures four different factors of intelligence: crystallized intelligence (general knowledge, lexical and orthographic knowledge), fluid intelligence (logical and deductive reasoning), visual perception (visualization capability, pattern recognition), and cognitive speed (e.g., ability to concentrate in simple cognitive tasks). These four factors are operationalized by eleven subtests whose net processing time is 39 minutes. In total, about one hour should be scheduled to conduct the test. [Figure 4.1](#) shows how the 11 subtests are related to the factors of intelligence. The test was validated with 2,583 participants [[KLH13](#)], showing an internal consistency in all four subfactors between .86 and .94 with .96 for the general intelligence score in the form we used. Construct validity was shown using confirmatory factor analysis. Regarding criterion validity, its results corresponded well with other intelligence tests.

The scores of the subtests are summed up regarding the four factors and mapped to age-adjusted IQ values within a 95% CI. Scoring the intelligence test is done using a set of templates and takes a few minutes per participant. The total score for all subtests represents an estimate of overall intellectual capacity, commonly referred to as general intelligence (*g*).

Similarly, as for intelligence, we looked for a validated personality test accepted in psychology that can be done on paper in the same time slot as the intelligence test. “There is little doubt that the Five-Factor Model (FFM) of personality traits (the ‘Big Five’) is currently the dominant paradigm in personality research, and one of the most influential models in all of psychology.” [[McC20](#)] We chose the validated German version of the 100 item HEXACO Personality Inventory-Revise (PI-R) [[LA18](#)] which is a variation that adds a sixth dimension. Lee and Ashton have empirical support for this sixth factor from principal component analysis. The questionnaire contains 100 statements on which a participant must self-assess on a scale from fully agree to fully disagree. HEXACO assesses six major dimensions of personality: *Honesty-Humility*, *Emotionality*, *Extraversion*, *Agreeableness*, *Conscientiousness* and *Openness to Experience* [[AL01](#)]. In a large validation study [[LA18](#)], the facets in the self-assessment had a mean reliability (internal consistency) of about $\alpha = .70$. Furthermore, principal compo-

ment analyses supported the chosen facets that have low inter-correlations. Furthermore, the test is freely available [AL01].

4.2.2.6. Hypotheses, Parameters, and Variables

The variables relevant to RQ2.1 are code comprehension performance and the four factors of intelligence mentioned in the previous section, i.e., crystallized intelligence (G_c), fluid intelligence (G_f), visual perception (G_v) and cognitive speed (G_s). The average IQ values of university students for the four factors are each in the range of 100 to 107 with a standard deviation between 13 and 15 [KLH13]. We expect all factors to have a positive impact on code comprehension performance. For G_f we assume the highest correlation, since code comprehension intuitively has a lot to do with logical thinking. Similarly, we argue for the positive impact of G_s , a measure of attention and concentration ability in simple cognitive tasks. Visual perception (G_v) ability should also show a positive correlation in the data for code that represents a visually processed form of knowledge. We assume the weakest positive impact for G_c , on which, for example, knowledge of the Java programming language can be mapped, which is in principle relevant for code comprehension, but also only up to a certain degree, determined and limited by the requirements of the tasks.

We will address the statistical relationship to general intelligence exploratively in the results and discussion, and will not formulate a hypothesis. The construct of general intelligence is less accessible than the four factors of intelligence mentioned above, which are, first, well-defined and psychometrically validated [KLH13] and, second, allow for a finer-grained analysis of specific influences on code comprehension. It is believed that general intelligence has its own impact on performance and is positively correlated with the four factors [BKW95; KLH13]. To also simplify the causal model in relation to mediating variables [Roh18], we therefore omit general intelligence and do not consider it for the hypothesis analysis.

Code comprehension is measured by the correctness of answers to comprehension questions on two independent code snippets. Correctness is one

of the most commonly used measures to assess how well a participant understood source code (see [Chapter 3](#)), and depending on the response format, the measurement is reliable even in experiments conducted synchronously with many participants. For each of the two code snippets, participants could give 10 answers, each of which was scored with one point for a correct answer. In the analysis, we noticed that for the second snippet a question regarding the return value according to Javadoc could not be answered unambiguously, and therefore we decided not to score the answers to this subtask. Accordingly, code comprehension performance ranges from 0 to 19. We deliberately refrained from using individual measurements of time for the comprehension tasks as this would not have been practical in the lecture hall setting, and instead introduced a general time limit.

H_1 : Crystallized intelligence (G_c) is positively correlated with code comprehension performance.

H_2 : Fluid intelligence (G_f) is positively correlated with code comprehension performance.

H_3 : Visual perception (G_v) is positively correlated with code comprehension performance.

H_4 : Cognitive speed (G_s) is positively correlated with code comprehension performance.

To answer RQ2.2, personality was operationalized by the six dimensions of the HEXACO personality model. In hypothesizing, we limited ourselves to two traits for which we could find the most evidence of possible impact on performance in scientific literature, that is, conscientiousness (range = [1, 5]) and openness to experience ($r = [1, 5]$).

Both personality traits were found in previous studies to be associated with performance in various software engineering activities. We suspect a positive correlation for both with code comprehension performance, since persons with high values for these personality traits have argumentatively good prerequisites to also successfully work on the comprehension tasks. People with high values for conscientiousness “organize their time and their

physical surroundings, work in a disciplined way toward their goals, strive for accuracy and perfection in their tasks, and deliberate carefully when making decisions” [AL21]. Persons with low values for openness to experience tend to “feel little intellectual curiosity, avoid creative pursuits” [AL21].

H_5 : Conscientiousness is positively correlated with code comprehension performance.

H_6 : Openness to Experience is positively correlated with code comprehension performance.

Programming experience was rated following the recommendation in [SKL+14] on three 10-point scales from *very inexperienced* to *very experienced*. Experience is known as a potential covariate and was therefore measured to control for it by subsequent analysis of its influence on code comprehension performance [SS15].

4.2.2.7. Causal Inferences

We have already mentioned that this is a correlational study, and we could elaborate at this point on why appropriate caution is needed in interpreting potential correlations, that they are only cause-effect relationships with some probability, not guaranteed. One issue is that ‘carefully crafted language will not prevent readers—let alone the public—from jumping to causal conclusions’ [Roh18]. Moreover, we strive to close the gap between observational data and causal conclusions as good as we can, to counteract the issue of uncertainty in the design of empirical studies motivated at the beginning with the greatest possible certainty about the concrete nature of influence of our independent variables.

One way to improve causal inferences based on observational data are directed acyclic graphs (DAGs), which visually represent causal assumptions [Roh18]. Figure 4.3 shows the causal assumptions of the relevant variables in our study. For example, we conjecture that a change in the variable crystallized intelligence causally contributes to a change in code

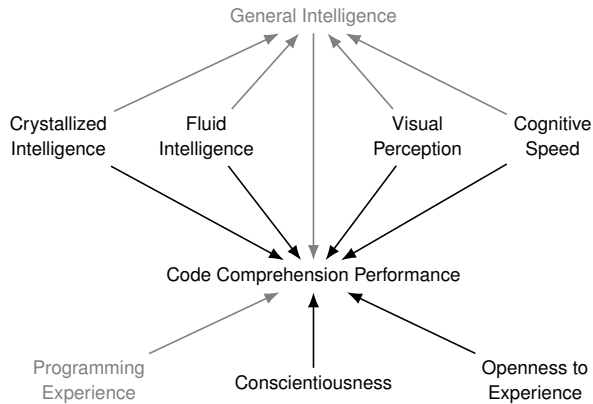


Figure 4.3.: Causal diagram of the experiment variables

comprehension performance.

We refer the interested reader to the work of Rohrer [Roh18], who explains in detail how such a graph can be used during study design to identify and eliminate spurious paths, for example by statistical control. The entire approach is based on the assumption that the DAG captures the true underlying causal web, which we recognize is a very strong assumption.

We chose this method to argue with greater certainty for a causal relationship among the variables in the six hypotheses. All authors of this paper were involved in the creation of the causal diagram during the study design. The influence of assumed confounding parameters in program comprehension studies [SS15] was rigorously discussed and literature on known influences on intelligence and personality was consulted.

The only notable variable that we could not control via our study design was the motivation of the participants. It may have an influence on the results of the intelligence test as well as on the results of the code comprehension tasks, and thus represents a potential confounder. To obtain a proxy for estimating participants' motivation to complete the test seriously, we used the number of requests for an individual participant's test results (about 40%) as well as examined outliers in the data set. The analysis provided

grounds to assume that a considerable proportion of participants answered the tests seriously. The descriptive statistics at the beginning of [Section 4.2.3](#) show that the sample performed very well on the code comprehension tasks, and the intelligence test scores are similar to those of the population.

Unless we have not controlled for any other factor that significantly influences both treatment and outcome, it is reasonable to assume that measured associations in hypothesis testing can be attributed to cause-effect relationships. We nevertheless agree with Rohrer's concluding remark that "the most convincing causal conclusions will always be supported by multiple designs" [[Roh18](#)] and call for further code comprehension studies on the given research questions to confirm or refute our model.

4.2.2.8. Analysis Procedure

We first used descriptive statistics to describe the sample. We employed the median (Mdn) and inter-quartile range (IQR) for ordinal data (self-assessed experience) and mean and standard deviation (SD) for interval data (code comprehension score). Also, intelligence and personality are considered interval data in psychology. Therefore, we did the same in our analysis.

Second, to control for the experience of the participants, we built linear models for each hypothesis to account for the factor affected in the hypothesis and the experience item with the strongest correlation to the score. We calculated standardized regression coefficients β together with their 95% confidence intervals (CI). We interpreted these intervals before using t-tests to calculate t -statistics and p -value as basis for statistical significance. To check the assumption of normality for the t-test, we conducted Shapiro-Wilk tests on the individual variables. For all but score and fluid intelligence, the test supported a normal distribution of the data. The QQ plots as well as the large sample size, however, make us confident that a t-test is still justified.

Because we have multiple tests, we adjusted the p -values using the Holm-Bonferroni method [[Hol79](#)]. It is a standard method to adjust for multiple testing but is more powerful than the often used Bonferroni method. We compared the final p -values with the significance level $\alpha = .05$.

Third, for exploratory analysis and to get a better understanding of the relationships of the various factors, we built further linear regression models. In particular, we analyzed a ‘complete’ linear model that contains all intelligence facets, the two personality traits also investigated in the hypotheses and all three experience items. This shows us the influence of all factors measured in the experiment. Furthermore, we included all measured factors (adding the other personality traits and the general intelligence to the ‘complete’ model) and conducted step-wise regression on it to get to the best model for explaining the data. This gives us a smaller model with only the most important factors. For both models, we calculated the adjusted R^2 as a measure of how much of the variance in the data can be explained by the model, as well as standardized regression coefficients with their 95% confidence intervals.

4.2.3. Results

4.2.3.1. Descriptive Statistics

We removed four participants from the data set because they provided data for the intelligence test and/or the personality test, but not the comprehension test or the other way around. We have complete data for 117 participants, further 17 participants without an intelligence test and one participant without a personality test.

The code comprehension performance as measured by the score concentrates strongly between 15 and 19, $M = 17.3$, $\text{Min.} = 8$, $\text{Max.} = 19$, $SD = 1.9$. The code comprehension score is depicted in relation to the three experience measures in [Figure 4.4](#). The scatter plots show no apparent strong relationships between the experience measures and the code comprehension scores. The strongest relationship seems to be when the participants rated their experience in comparison to others.

The three facets of experience, i.e., programming experience (PE), experience in comparison (EC) and experience in the object-oriented paradigm (EOO), have widely distributed results over the whole spectrum with a cen-

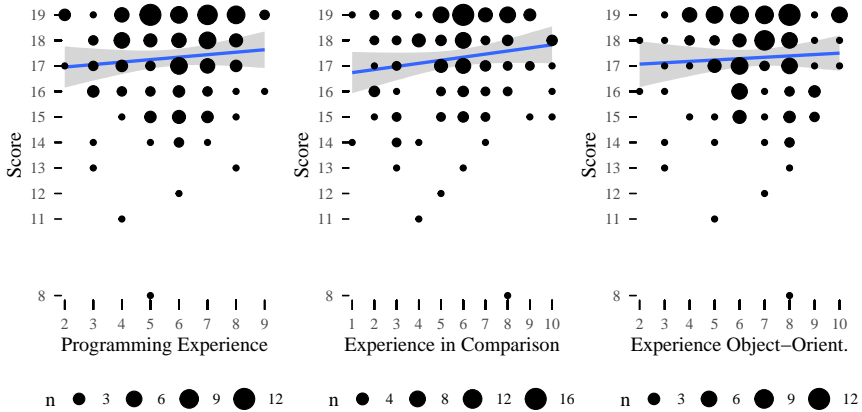


Figure 4.4.: Overview of experience in relation to code comprehension scores

tral tendency to slightly above the middle of the scale ($Mdn_{PE} = 6, IQR_{PE} = 2, Mdn_{EC} = 6, IQR_{EC} = 2, Mdn_{EOO} = 7, IQR_{EOO} = 2$). Hence, we seem to have a well-balanced sample in terms of experience.

In Figure 4.5, we see the relationships between the four different intelligence facets with the scores. As expected with intelligence tests, the results are distributed roughly around 100, but we also see rather low and rather high values in all of them ($M_{G_C} = 103.6, SD_{G_C} = 18.4, M_{G_F} = 115.6, SD_{G_F} = 13.9, M_{G_V} = 105.0, SD_{G_V} = 14.6, M_{G_S} = 94.5, SD_{G_S} = 16.7$). There appears to be a positive association between all the intelligence facets and the code comprehension score.

Finally, the data for the two personality traits conscientiousness and openness to experience is depicted in Figure 4.6. Both personality traits are strongly distributed over almost their whole spectrum. For conscientiousness, we have a mean result of 3.5 ($SD = 0.6$). The mean for openness to experience is only slightly lower at 3.2 ($SD = 0.6$). There appears to be no clear association of both traits with the code comprehension scores. Conscientiousness tends even to be slightly negatively related to the score.

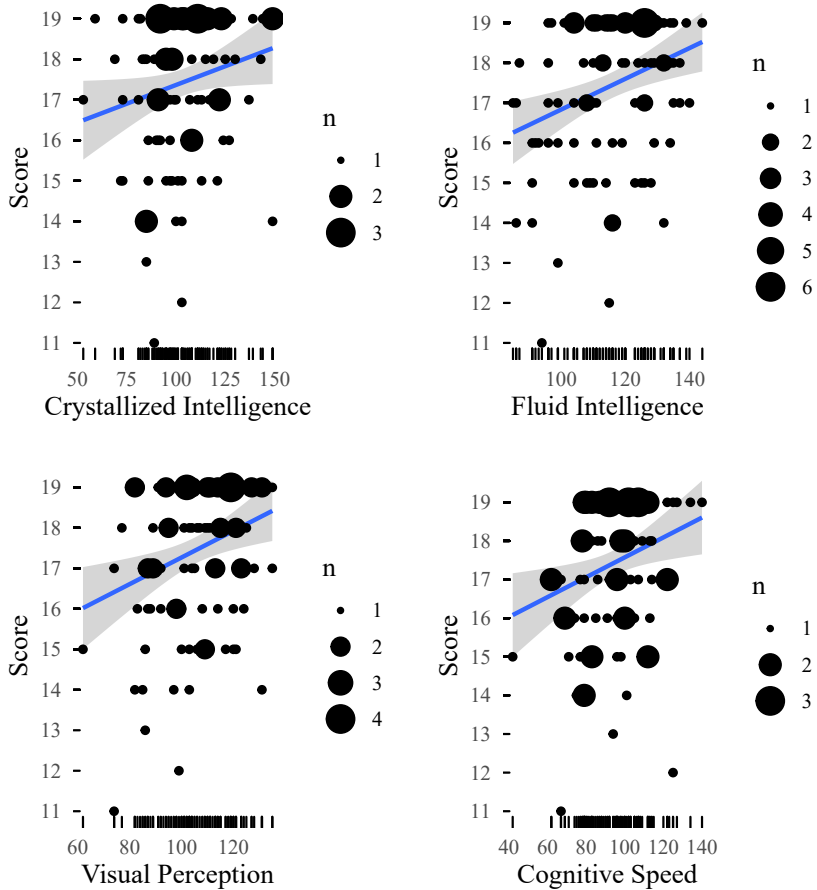


Figure 4.5.: Overview of intelligence in relation to code comprehension scores

4.2.3.2. Hypothesis Test

The results of the hypothesis tests of all six hypotheses are summarized in [Table 4.1](#). We see clear positive standardized regression coefficients between each intelligence facet and the code comprehension scores when controlling

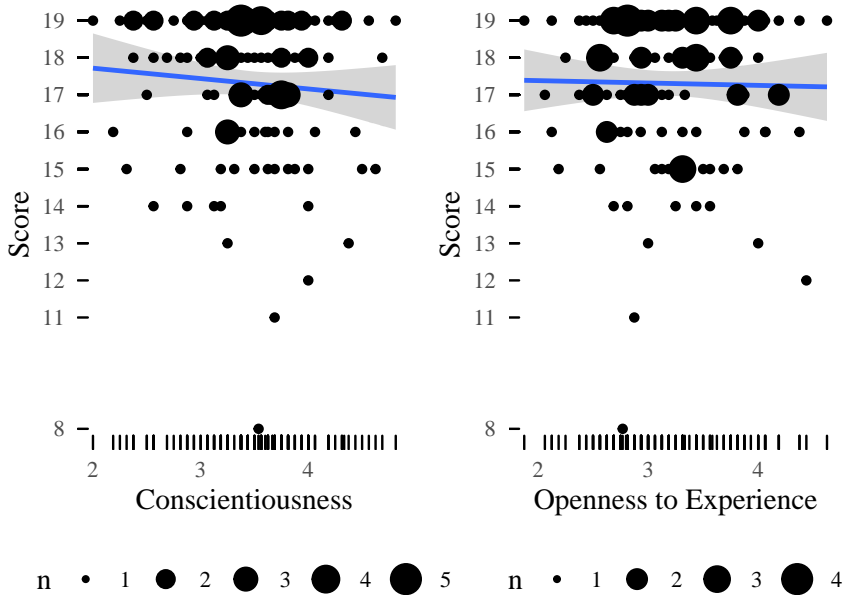


Figure 4.6.: Overview of personality traits in relation to code comprehension scores

for experience. They also constitute the effect size, which according to Cohen [Coh92], we can consider as small associations. The confidence intervals are all rather narrow so that the true coefficients will likely be a small positive association.

For both personality tests, there are very small negative standardized regression coefficients. The confidence intervals, however, are very large and span from a medium negative to a medium positive association in both cases. Hence, from our data set, it is not clear if there is an effect at all and in which direction an effect would go.

To test the statistical significance, we use t-tests for which we report the test statistic t as well as the corresponding p -value. Here again we see p -values for all intelligence facets lower than 5% while both personality traits

Table 4.1.: Results of individual linear regression models

Factor	β	95% CI	t	p	Adj. p
Crystal. Int.	0.188	[0.171, 0.206]	2.02	.0460	.1380
Fluid Int.	0.299	[0.276, 0.322]	3.30	.0013	.0078
Visual Percep.	0.261	[0.238, 0.283]	2.84	.0054	.0270
Cogn. Speed	0.247	[0.228, 0.267]	2.68	.0084	.0336
Conscientiousn.	-0.096	[-0.693, 0.501]	-1.11	.2710	.5420
Open. to Exp.	-0.040	[-0.633, 0.552]	-0.46	.6440	.6440

show p -values far greater than 5%. To account for multiple testing, however, we adjusted the p -values using the Holm-Bonferroni method [Hol79]. The adjusted p -values give us almost the same picture. Only crystallized intelligence has now a p -value larger than 5%.

Hence, we have to reject the corresponding null hypotheses and have support for H_2 , H_3 and H_4 : There is a statistically significant, positive relationship between fluid intelligence, visual perception and cognitive speed and code comprehension performance. Crystallized intelligence is positively related with comprehension performance, but this relationship is not statistically significant. There is no clear relationship between conscientiousness and openness to experience with comprehension performance.

4.2.3.3. Exploratory Analysis

Table 4.2 shows the results for the complete linear model. Overall, the model explains only almost 10% of the variance in the data (Adjusted $R^2 = .096$). The standardized regression coefficients show to some degree a different picture than the individual regression models for the hypothesis tests. Fluid intelligence and cognitive speed have slightly lower but similar regression coefficients. The coefficient for visual perception is close to zero, and the crystallized intelligence even has fully moved to a (very small) negative coefficient. The confidence intervals for these coefficients also support this, and are rather narrow. For the experience measures, surprisingly, only expe-

rience in comparison has a positive regression coefficient. Yet, for the other two measures, the confidence intervals show that the true values could be positive or negative. In any case, this also supports the usage of experience in comparison in the linear models for the hypotheses tests. For conscientiousness and openness to experience, the coefficients and confidence intervals are almost the same as for individual linear models.

Table 4.2.: Complete linear model

Predictor	β	95% CI	b
Intercept	0.000	[-4.020, 4.020]	13.722
Crystallized Int.	-0.045	[-0.070, -0.020]	-0.004
Fluid Int.	0.220	[0.185, 0.254]	0.028
Visual Perception	0.057	[0.023, 0.091]	0.007
Cogn. Speed	0.154	[0.129, 0.178]	0.016
Exp. in Comparison	0.316	[0.079, 0.553]	0.278
Experience OO	-0.061	[-0.308, 0.185]	-0.062
Program. Experience	-0.147	[-0.454, 0.162]	-0.163
Conscientiousness	-0.103	[-0.697, 0.491]	-0.327
Openness to Exp.	-0.051	[-0.642, 0.540]	-0.160

When looking at the non-standardized parameters, we see that the model has a large non-standardized intercept because most of the code comprehension scores are in the area above 13 points. All the intelligence facets have very small regression coefficients (b). Both personality traits have small to medium strength negative regression coefficients. Only experience in comparison has a considerable positive regression coefficient. So for the direct prediction of the outcome of the results of our study, mostly experience and the personality traits would be important.

As that model only explains 10% of the variance, we wanted to further explore if we can find a better model. For that, we employed backward stepwise regression starting from a model that includes all measured variables from the experiment. It reduced the variables to the model shown in [Table 4.3](#). This model consists of only three factors and explains more than 12% of the variance (adjusted $R^2 = 0.122$).

Table 4.3.: Linear model based on stepwise regression and all measured factors

Predictor	β	95% CI	<i>b</i>
Intercept	0	[-3.07, 3.07]	14.8820
General Intelligence	0.305	[0.285, 0.325]	0.033
Exp. in Comparison	0.174	[0.013, 0.335]	0.152
Conscientiousness	-0.157	[-0.754, 0.439]	-0.511

General intelligence has the strongest standardized regression coefficient with a narrow confidence interval in this model. The other two factors have roughly half the standardized regression coefficient, experience in comparison with a positive influence and conscientiousness with a negative influence. Both have wider confidence intervals. Especially, the confidence interval of conscientiousness goes from a large negative to a medium positive value.

In summary, the exploratory analysis supports the results that fluid intelligence, visual perception and cognitive speed are positively related to code comprehension performance. It also supports the use of experience in comparison as the best subjective measure for experience. Yet, it also shows that crystallized intelligence might actually be negatively related to code comprehension performance when taking all factors into account. Furthermore, there is indication that general intelligence might actually have the strongest relationship, and personality traits could also play a role when considering interactions between different factors.

4.2.4. Discussion

Literature in the field of code comprehension suggested that intelligence and personality might have an impact on code comprehension. We were able to demonstrate such a relationship of correct answers to questions on code comprehension tasks to intelligence and personality traits. The nature of these relationships must be considered separately for each factor, since some,

such as fluid intelligence, appear to represent an independent influence on code comprehension performance, and others, such as conscientiousness, appear to explain some of the variance only with other factors.

Fluid intelligence ($\beta = 0.299$), visual perception ($\beta = 0.261$), and cognitive speed ($\beta = 0.247$) showed significant association with code comprehension performance and can be preliminarily ranked in that order in their influence. Cognitive speed could be of greater importance in scenarios where there is greater time pressure. Crystallized intelligence showed the weakest standardized coefficient ($\beta = 0.188$) and might be of lesser importance in code comprehension.

Our study is underpowered with 135 participants at a calculated optimal sample size of 198, which could mean that in a higher powered experiment, more effects could be detected by reducing the type 2 error. Crystallized intelligence might be statistically significant because it has already had an only positive CI in our experiment. Yet, the exploratory analysis pointed in the opposite direction: a negative association with comprehension performance. Hence, this relationship remains unclear from our experiment. Similarly, a larger sample size might help to narrow the confidence intervals for the personality traits and make their influence clearer.

An exploratory analysis of age-adjusted general intelligence shows a moderate positive standardized coefficient of $\beta = 0.312$ and at the same time general intelligence is the best predictor in the linear model based on stepwise regression (see [Table 4.3](#)) with a very small confidence interval. This suggests that a combination of several high values for different intelligence facets is useful for the successful performance in code comprehension tasks.

We further noticed during the analysis that adding general intelligence to the linear model in [Table 4.2](#) increases R^2 from 9.7% to 12.4%, thus explaining more of the variance in the data and supporting the common assumption that general intelligence is an independent factor that has its own impact on performance.

RQ2.1: Main Findings

There is a positive relationship between individual intelligence factors and general intelligence with code comprehension performance. The combination of high values for different intelligence facets leads to high general intelligence, which in turn has the greatest predictive power for successful comprehension of code snippets.

The two personality traits examined for the hypothesis tests, i.e., conscientiousness and openness to experience, showed standardized coefficients close to 0 when controlling only for experience. A further exploratory analysis shows similar results for agreeableness ($\beta = 0.045$) and honesty/humility ($\beta = -0.061$). For emotionality ($\beta = 0.161$) and extraversion ($\beta = -0.152$) we see at least small individual coefficients.

RQ2.2: Main Findings

At first glance, personality does not appear to be a particularly interesting avenue for further research in the context of code comprehension performance. On their own, at least, individual personality traits do not have a significant impact on performance. However, we see that conscientiousness does play a predictive role in interaction with other factors. The linear models suggest that there may be other factors that interact with conscientiousness and may determine whether this personality trait has a positive or negative effect on code comprehension.

For the design of our study, we used a causal diagram (see [Figure 4.3](#)) that helped us identify spurious paths and covariates, for example. One such covariate is programming experience, and it was confirmed in our data to have a significant impact on performance in code understanding. Of the three experience measures proposed in [\[SKL+14\]](#), experience in comparison is best suited to predict performance, although the CI is wide and thus the actual impact unclear. At least the CI is all positive, and in a homogeneous

sample it might already be sufficient to query this one item as a proxy for the experience of a participant.

Siegmund et al.'s study [SKL+14] is comparable in terms of participant and task characteristics. They had a homogeneous student sample of 128 participants whose task consisted largely of having to mentally simulate code to determine the output of that code. Moreover, code comprehension was measured via correctness in these tasks. Their final recommendation that in a student population, experience in comparison might be sufficient to reliably measure programming experience is supported by our data. Our data differ in that the experience measures 'programming experience' and 'experience with object-oriented languages' are not positive factors influencing performance. We agree with their view that additional experiments are needed to construct a valid experience measure.

What is apparent from our results is that our causal diagram may not be complete or individual connections assumed to be causal relationships may not be causal in nature (or at least cannot be considered isolated from other factors). Our optimized linear model provided in Table 4.3 explains only a small portion of the variance in the code comprehension scores, i.e., intelligence, experience and conscientiousness do not seem to be sufficient to predict code comprehension performance as we have operationalized it. We see two possible explanations for this.

First, it may be that the specific way of measuring code comprehension affects the strength of the predictors studied. For example, the selection of our tasks and how we scored them led to low variance in code comprehension scores, which in turn tends to lead to lower regression coefficients in general. We elaborate on the relevance of task design and construct measure in the following subsection.

Second, code comprehension could be a construct that includes skills that are not captured by intelligence tests. One might even assume a missing individual characteristic of a developer in the causal diagram, which serves as a predictor for code comprehension proficiency. This, in turn, argues for measuring code comprehension via experimental procedures and tasks tailored to code comprehension, as has been done to date, rather than

being replaced by existing psychometric tests designed to measure related constructs such as intelligence.

4.2.4.1. Limitations

The results of this study should be seen in the light of some limitations.

First, we invited a convenience sample of students in their second year, so the results should only be generalized to more experienced developers with caution. While we have a well-balanced sample in terms of self-reported programming experience, it is clear that second-year students on average have less experience than students in higher semesters or even graduates, and such increased experience may lead to different results in the code comprehension tasks. In terms of the distribution of intelligence, our sample appears to be representative of university students, with slightly higher mean values for fluid intelligence and slightly lower for cognitive speed in our sample compared to the normalization sample [KLH13]. However, we lack data to assess representativeness for all developers, regardless of their educational background.

Given the expected sample characteristics, we limited our selection of tasks and code snippets to those that were of easy to moderate difficulty and thus tend to be in line with comparable code comprehension studies (see [Section 3.2.3.6](#)). Unfortunately, in our study, this resulted in the majority of the scores achieved in the code comprehension tasks being concentrated in a range close to the maximum score. Thus, while we were able to distinguish between the average participant and low performers, we were likely not able to identify nuances in the performance of individual participants sufficiently well. This in turn explains the low regression coefficients in the linear model. We recommend repeating the experiment with a wider range of code snippets of varying difficulty or changing the operationalization of code comprehension performance.

Related to this, it is in the nature of our study that we consider code comprehension as an isolated process that our participants had to go through on their own. While such a controlled setting is good for identifying individual

influences on performance with satisfying internal validity, we are aware that in practice, for example, code comprehension is accompanied by other activities or even that developers sometimes read code together with their peers. We are certain that studies in more realistic settings would provide many additional valuable insights into the effects of conscientiousness and individual intelligence factors on code comprehension performance and behavior. We consider our findings to be a valuable starting point that provides evidence for the influence of two constructs on the isolated cognitive process of code comprehension. Additional studies, including those of a qualitative nature, are needed to shed light on our research questions from other interesting angles.

Code comprehension as a construct has been measured solely by the correctness of answers to comprehension questions. While there is no validated measure of code comprehension so far, we suspect that correctness alone does not capture all facets of code comprehension. It would therefore be interesting to see which additional information we would gain if, for example, comprehension efficiency (see, e.g., [SBV+19; WPGW21]) or cognitive load [Fak18] were instead used as proxies for code comprehension. Due to our study design with a large number of participants, in presence and synchronous execution, we were limited in this respect, but aim for complementary studies in the future. Apart from that, we found that conducting our study synchronously with a three-digit number of participants worked smoothly. This type of code comprehension measurement scales very well, provided that large rooms are available to the study leaders.

Compared to the measurement of code comprehension, research on the measurement of personality and intelligence is more advanced. We were able to use validated questionnaires for their measurement, but like code comprehension, they are latent variables that only approximate what currently corresponds to (parts of) our definition of them. Accordingly, we would like to note that our conclusions on the influences of personality and intelligence in this work are to be considered under the assumptions and definitions that the respective tests establish for these constructs. In this specific case, we do not consider this to be a significant limitation, since the used instruments

are well-established questionnaires whose results can usually be compared with those of other studies on personality and intelligence.

4.2.4.2. Implications

Since several intelligence factors correlate significantly with developers' code comprehension performance, researchers have so far probably been correct in their assumption that not controlling intelligence in their study design is a potential threat to validity [SS15]. For example, if one of two experimental groups has a significantly higher mean intelligence value, this will have a positive effect on the code comprehension performance of that same group and, as is common for confounding variables, may lead to false conclusions about the influence of the independent variable that is actually being measured. This situation remains even if our assumptions about the causal relationship are refuted in the future. The measured correlations remain valid and explain some of the variance in code comprehension performance.

However, these findings do not mean that previous research on code comprehension performance is invalidated, but only that intelligence may have had an impact on the measured code comprehension data in some cases. Moreover, for now, the limitations of the transferability of our results to studies that also have a sample of university students apply.

As for the influence of personality traits, we see great potential for more in-depth studies examining the specific nature of the relationship to code comprehension and the interplay with other factors. Conscientiousness seems to play a role in predicting code comprehension performance, but apparently only with other factors and probably with some we did not measure in our study.

Consequently, what should be considered in the design of future code comprehension studies? Code comprehension studies in which intelligence and personality might play a role and which do not control for these constructs by design risk bias in their results. If intelligence and personality traits are not explicitly controlled by, e.g., matching or post hoc analysis because it would not be feasible to have every participant take an intelli-

gence and personality test, then other control techniques should be used. One mentioned by Siegmund and Schumann [SS15] is randomization and the notable advantage of this technique is that one may also control for additional, possibly even unknown extraneous variables.

We see alternatives like using validated, reduced IQ or personality tests, which take only a fraction of the time of full-scale tests, but at the same time are a useful approximation. Further, we suggest a more solid discussion of threats to validity based on data about the influence of potential confounders. For example, in a quantitative study of the influence of syntax highlighting on code comprehension in which intelligence was not controlled, if the effect size is large enough, it can at least be argued that the measured influence of syntax highlighting is larger than that expected in groups with significantly different intelligence distributions.

Independent of controlling for a potential confounding factor, we consider it to be necessary and at the same time equally interesting for a study to measure intelligence and personality to examine their individual influences in the context of a specific research question and thus enrich our more abstract results. While we consider intelligence and personality as potential confounding variables in the context of this study, they are in the end also given individual characteristics of each developer, and if we could better understand their influence on the work of those same developers, we can support developers outside controlled experiments through, for example, appropriately customizable tools and consideration of individual capabilities. Examples include targeted support for learning programming languages, meaningful partnering in pair programming, and assignment of specific tasks based on individual characteristics. However, these are potential implications for SE practice that are at best not derived based on the results of a single study. Accordingly, future work can build on our findings to explore their concrete consequences in practice and, eventually, to develop measures, if necessary, to counteract potentially negative consequences of inequalities in given individual characteristics.

Finally, the present study has shown that a potential influence of intelligence discussed in the literature not only actually has an impact on perfor-

mance, but also provides insight into the direction and strength of individual facets. We appreciate that it has now become the norm to discuss threats to validity in code comprehension studies. Backing these discussions up with evidence in the future should be the next step in the maturation of the research field, and for this it needs further studies that identify new confounders or confirm or refute assumed ones.

4.3. Conclusion

We have seen in [Chapter 3](#) and in the work of Siegmund and Schumann [SS15] that the list of suspected confounding parameters on program comprehension is long. Taking them all into account in the design of valid studies remains a challenge. It is therefore essential that we obtain certainty about the extent and the nature of the relationship of each parameter to code comprehension through empirical studies, for example, to better evaluate alternative study designs.

We investigated the influence of intelligence and personality on code comprehension performance in a study with 135 university students. While personality traits showed no association with performance on their own, we found significant small to moderate positive association between code comprehension performance and the intelligence factors fluid intelligence, visual perception and cognitive speed. We found a weak relationship of performance to crystallized intelligence, which was not statistically significant. An exploratory investigation further showed a moderate positive relationship of performance with general intelligence and that it is the variable with the greatest predictive power in our linear model.

Given our sample size, the measured regression coefficients are large enough for intelligence to be a noteworthy potential confounding variable, especially from a researcher's perspective. The results indicate that the control of intelligence in code comprehension experiments is necessary for valid conclusions from obtained study data. We draw a similar conclusion for the personality trait conscientiousness, although the specific nature of

the relationship to code comprehension requires further research. At the same time, the results should be considered in light of the selected tasks, sample characteristics, and code comprehension measures, which is why we encourage further studies on the relationships between intelligence and personality with code comprehension performance to enhance our understanding of these relationships. The more in-depth investigation of potential confounding parameters, be it intelligence, personality or any other variable, will eventually lead to more confidence in the validity of code comprehension studies.

HOW CONTEXTUAL FACTORS INFLUENCE CODE COMPREHENSION

In the previous chapter, we showed that the human factor is essential in understanding code comprehension. We keep the focus on the human, but shift a bit of our perspective to contextual conditions that can affect the developer in understanding code (see [Section 2.3](#) for background).

In this chapter, we address **RQ3**, which seeks to understand how contextual factors influence developers in code comprehension. We have published two primary studies on the topic. In the first, we found that developers can be easily biased in their subjective code comprehension by a made-up displayed code comprehensibility metric [[WPGW21](#)]. We sought to overcome some of the limitations of that study, so we subsequently designed a reproduction study that confirmed the results with a slightly modified study design [[WMG22a](#)].

This chapter contributes to the goal of the thesis in two ways: first, the

studies presented provide evidence that can help in the design of code comprehension experiments. Second, we demonstrate the importance of different study designs to answer the same research question. The second study provides valuable additional insights and contributes to a more complete overall picture.

5.1. Context and Goals

Development teams strive to make their code as understandable as possible and base their activities on the results of static code analysis tools to identify areas of code that are still difficult to understand. Most of the metrics reported by such tools are either not validated [NAG19], or have been empirically shown not to measure what they claim to measure [SBV+19]. The latter issue seems especially prevalent in the field of code comprehensibility [MWW20; SBV+19]. In other words: several metrics do not reflect what they are supposed to measure. Yet, they are considered when making decisions and change course of what developers think of their source code.

“ *The mind is a powerful place / And what you
feed it can affect you in a powerful way* ”
— NF, *THE SEARCH* (SONG). 2019.

Feeding the mind with a belief influences it and causes changes that might go beyond the mind itself. Crum and Langer [CL07] divided a sample of room attendants at different hotels into two groups. To the first one only, the researchers presented the supposed positive effects of work-related physical activities on their health. After four weeks, the informed group felt that they received significantly more exercise than the second group. Not just that: Weight, blood pressure, and body fat of the informed group significantly decreased compared to the non-informed one—without any detected change in workload, outside work physical activity, or eating habits. If the consequence of a treatment is not attributed to the treatment itself, but to pure beliefs and expectations of its effectiveness, we call it the *placebo*

effect [Sha68]. This effect occurs in many ways. Placebos are administered in clinical trials in the form of sham drugs to distinguish the pharmaceutical effect of a drug from the placebo effect. The placebo effect can go beyond the subjective perception of an affected person and provide measurable effects. The room attendants were specifically manipulated by the researchers and a desirable effect was achieved.

Various contextual factors can also influence our reasoning and decision-making, some of which make us deviate from beneficial results. Many of these factors are cognitive in nature [Hil12]. Software engineering is no exception. In a recent systematic mapping study on cognitive biases in software engineering, 65 articles were identified that provide evidence for the presence of cognitive biases of at least eight different categories [MST+18]. Negative consequences of such biases are, for example, overly optimistic effort estimates or insufficient software modifications.

A cognitive bias which is related to estimates and adjustments is the *anchoring effect*, introduced by Tversky and Kahneman [TK74]. The anchoring effect means that an initial value is insufficiently adjusted so that “different starting points yield different estimates, which are biased toward the initial values” [TK74]. The anchoring effect is one of the most robust cognitive biases [FB11] and, in the context of software engineering, the most studied [MST+18].

In the following, we present two studies. The first one is a randomized, double-blind experiment in which we divided 45 software engineering students into two groups. Participants in both groups were asked to work on a source code comprehension task on code snippets. We showed the two groups a metric value that represents the understandability of the code snippets. One group saw a value that indicates an easy understandability of the source code. The other group saw one that indicates a hard understandability of the source code. Unbeknownst to the participants, tasks and snippets were the very same for both groups. Also, the metric is not real and placed there to anchor them. We formulated three research questions, which are further framed later in [Section 5.2.1](#) and in [Section 5.2.2.5](#):

- RQ3.1** Does the value of a shown code comprehensibility metric influence subjective ratings of code comprehensibility?
- RQ3.2** Does the value of a shown code comprehensibility metric influence the actual code understanding?
- RQ3.3** To which extent do selected individual characteristics correlate with the deviation of the subjective rating from the shown metric value?

The second study is again a controlled experiment and its design is based on that of the other experiment on RQs 3.1 – 3.3. It modifies and improves upon the first experiment in several ways, which we discuss in more detail in [Section 5.3.1](#). The most considerable differences are that we now have a much more heterogeneous and larger sample with 206 students and 50 professionals, and that the displayed metric value for anchoring participants is shown *before* the code comprehension task (not during it). The second experiment addresses the following research question:

- RQ3.4** Does specific information available in advance about a code snippet influence developers in their subjective assessment of the code's comprehensibility?

In the context of this thesis, understanding the consequences of displaying a metric value on a developer's code comprehensibility ratings brings valuable insights. If the presence of a single metric value significantly influences a developer's rating, this would be a strong call to meticulously control code comprehension experiments for potential confounding variables that could bias a developer's judgement or to advise against this measurement method altogether.

5.2. A Study of Displaying a Metric to Affect Code Comprehension

5.2.1. Background and Related Work

In this section, we define two central constructs of this work, namely the placebo effect and the anchoring effect, and place the work in the context of related code comprehension literature.

5.2.1.1. Placebo Effect

One of the most quoted definitions of *placebo* comes from Shapiro [Sha68], who studied the etymology and semantics of the word to provide a basis for an appropriate definition and to address the diversity of opinion about the meaning of the term. The proposed definition is as follows:

“ A placebo is defined as any therapy (or that component of any therapy) that is deliberately used for its nonspecific psychologic or psychophysiologic effect, or that is used for its presumed specific effect on a patient, symptom, or illness, but which unknown to therapist and patient is without specific activity for the condition being treated.

— A.K. SHAPIRO [SHA68]

The *placebo effect* is defined as “the nonspecific psychologic or psychophysiologic effect produced by a placebo” [Sha68]. The introduction of the term in medical literature was accompanied by “the widespread introduction of controlled methodology in the evaluation of treatment” [Sha68] and it became standard to control for the placebo effect in clinical trials. In this paper, we follow Shapiro’s definition. We would like to point out that the placebo effect, however, is not limited to the medical context and can be applied to everyday aspects, as numerous studies have shown.

In a study on *placebo sleep* [DE14], participants had to report their previous night's sleep quality. One group was then told after a supposedly reliable measurement that their sleep quality was above average, and the other group was informed that their sleep quality was below average. The assigned sleep quality, but not the self-reported sleep quality, significantly predicted, among others, the auditory information processing speed of the participants. The authors conclude that mindset can influence cognitive performance both positively and negatively [DE14].

Other studies show, for example, that smelling a supposedly creativity-enhancing odorant actually results in a creativity-enhancing effect [RMI+17], that non-invasive sham brain stimulation improves learning performance [TBG+18], and that different forms of placebos have an effect on the performance of athletes [BKSB11].

Investigations of the placebo effect in software engineering research have rarely been conducted so far. One recent study deals with the influence of a three-minute breathing exercise on the perceived effectiveness of stand-up meetings in agile project teams [HKS17]. A placebo group was added to compare the effect with a non-meditative form of relaxation, i.e. listening to classical music. They conclude that the breathing exercise has an immediate positive impact on meetings in agile teams. Another study [SAY+18] investigates how the subjective evaluation of an automatically generated solution is positively influenced by involving the decision maker in the process but not considering their decisions at all. They conducted a placebo-controlled study with 12 software engineering practitioners and found an increase of 68% in the subjective evaluation of an automatically generated but supposedly decision influenced solution is due to a placebo effect. We are not aware of any study investigating a potential placebo effect on performance in code understanding activities.

5.2.1.2. Anchoring Effect

In our behaviors, we act within a specific context. Such context provides us with cues, verbal suggestions, and social information that influence our

expectations, appraisals and memories, which in turn influence our behavior and reported experiences [WA15]. Consequently, parts of the placebo effect on subjective assessments are attributed to various forms of decision bias [WA15].

A systematic mapping study on cognitive biases in software engineering highlights that the everyday life of a software engineer is also full of situations in which their decisions are subconsciously manipulated [MST+18]. Mohanani et al. [MST+18] identified 65 articles in the context of software engineering that investigated 37 cognitive biases of at least eight different categories. In the worst case, such bias leads to systematic deviations from optimal reasoning, such as overly optimistic effort estimates or insufficient software modifications [MST+18].

The specific cognitive bias that we investigate is called *anchoring effect*, which we defined in the introduction to this chapter. According to the aforementioned mapping study, it is the most frequently investigated cognitive bias in the context of software engineering [MST+18]. For example, one study used SQL queries as an anchor for query formulation tasks. They found that while subjects complete the tasks more quickly when modifying a query instead of writing it from scratch, accuracy decreases and overconfidence in the results increases [APO6]. Another example where anchoring plays a role is planning poker. In planning poker, it is considered as positive that all effort estimates remain initially hidden from view, so that no one is anchored in their initial estimate by the estimates of their colleagues [Hau06].

The anchor would not even have to be relevant for the estimate [FB11] and could, for example, result from the previous turning of a wheel of fortune with numbers between 0 and 100 [TK74]. Since we are interested in the transfer of the anchoring effect to a realistic software engineering scenario, we have decided to display a code comprehensibility metric value.

Limited literature is available that has shown that individual characteristics of the participant may influence the strength of the anchoring effect. Furnham and Boo [FB11] argue in their literature review on the anchoring effect that previous research “neglected individual differences variables because people tend to look for a universal rule that would predict reactions or

behaviour”. Nevertheless, they could identify a total of 17 studies that considered the influence of experience, personality, mood, motivation and cognitive abilities. We aim to contribute to this investigation in the context of RQ3.3 and explored the influence of experience, personality, happiness and dispositional optimism and pessimism on the anchoring effect. All these constructs are also associated in literature with the placebo effect [GWF+10; MWEJ09; PAL+13; WA15]. In 5.2.2.4 and 5.2.2.5 we describe the used questionnaires and how we have operationalized and measured the constructs.

5.2.1.3. Code Comprehensibility Metrics and Subjective Ratings

Many factors impact the comprehensibility of source code. For example, one study has shown that shorter identifiers take longer to comprehend [HSH17], and another that a number of certain code patterns lead to an increased rate of misunderstanding [GIY+17b].

Static code analysis tools attempt to measure code understandability automatically to efficiently point out sections of the code that are difficult to comprehend and should therefore be refactored. Not only are most of the metrics in static code analysis tools not validated [NAG19], but in addition, there seems to be only one metric that is validated and positively correlates with measures of source code comprehensibility [MWW20; SBV+19].

Scalabrino et al. [SBV+19], for example, investigated 121 metrics that would measure source code understandability. Code snippets were evaluated with 63 developers and various proxy variables that are related to the time and correctness required to complete comprehension tasks. According to their results, none of the investigated metrics showed a significant correlation with the measured source code comprehensibility.

In one of our own studies [MWW20] we empirically evaluated the *Cognitive Complexity*, a newly introduced metric that claims to measure source code understandability [Cam18]. The metric evaluates code syntactically and assigns each method a calculated value on a ratio scale. For Java methods, the authors of the metric suggest a threshold value of 15 above which a snippet should be refactored. The authors of the evaluation study conclude

that the metric is a reliable predictor of the required understanding time and the subjective comprehensibility rating of developers [MWW20]. Both aspects encouraged us to consider the metric when selecting code snippets for the present study (see 5.2.2.4; they all had a value of 19).

Displaying non-validated metrics can lead to confusion and unnecessary effort due to improper prioritization of development efforts. It becomes especially problematic if the displayed metric value actually has an influence on developers, even if only in their subjective perception of the code because this would mean that they are subconsciously manipulated and are very unlikely to resist this circumstance since the anchoring effect is one of the most robust cognitive processes [FB11].

Finally, in experiments like ours, we do not intend to measure how understandable source code is, but the degree to which a participant has understood the given source code. It is in the nature of our study to investigate selected influences on code comprehension, and not to compare variants of source code for their comprehensibility. As we know from the mapping study presented in Chapter 3, the most common measures for this purpose are time and correctness in processing comprehension questions, subjective ratings and physiological measurements such as eye tracking. We also learned there and via Siegmund and Schumann's study that numerous potential threats to validity and confounding variables are suspected to influence code comprehension [SS15]. A handful of these are related to cognitive biases. One example is the Hawthorne effect [MWI+07; RD03], which describes that participants in experiments would behave differently because they were observed. With the present study, we close a research gap and investigate whether we should add the anchoring effect as an entry to the catalog of confounding variables on code comprehension, particularly if code comprehension is operationalized through subjective self-assessments.

5.2.2. Methodology

We follow the guidelines of Jedlitschka et al. [JCP08] on reporting experiments in software engineering.

5.2.2.1. Goals

The goal of our study is to analyze the effect of showing a specific code comprehensibility metric on measures of a software engineer's code understanding to identify a potential cognitive bias and placebo effect. To this end, we formulated the three research questions 3.1–3.3 given in the introduction to this chapter ([Section 5.1](#)).

5.2.2.2. Participants

We invited a convenience sample of students of a software engineering M.Sc. study program in Germany to participate in the study. Convenience sampling is controversial because it threatens generalizability [[BR22](#)], but at the same time appropriate for our goal of studying universal phenomena such as the anchoring effect. Additionally, we limited participants to those with good knowledge of Java and German, as the study was conducted in German; in both aspects we relied on the self-assessment of the participants. We see the sample properties of interest, namely enough experience to comprehend medium to hard to understand Java code, ensured by our sampling strategy so that the findings can be transferred to a population of experienced Java software engineers. Limitations of our sampling strategy and their implications are discussed in [Section 5.2.4.1](#).

As part of their study duties, students had to participate in any study offered by the faculty. Students had the right to register for a study and then withdraw their participation at any time (including before the start) without consequences, with course organizers being unaware of their withdrawal. We reminded them about this during the informed consent phase, which included a partial design disclosure, health risks, privacy and ethical issues, and our contact details. Consent was obtained in written form.

Participants knew that we aimed to investigate factors that influence the understanding of source code, and that they would have to work on short methods written in Java and calculate the results for given input values. They were not aware of the metric manipulation.

5.2.2.3. Tasks

Participants were shown three independent Java methods, one after the other. For each code snippet, five input values were given, for which the participants were asked to specify the return values according to the Javadoc and to determine the actual return value. Since we told our participants that there might be bugs in the code, they could not rely on the Javadoc comment and had to understand what the code actually does. We consider the deviation of the documentation from the code and the inspection based on concrete values for the input parameters to be a realistic scenario. Furthermore, the task is in line with the conceptual model that a developer in a maintenance scenario iteratively constructs and tests hypotheses about the functioning of the code during program understanding [VV95].

Right after determining the return values, the participants were asked to rate the comprehensibility of the method on a scale of 0 (very easy) to 10 (very hard) and fill out questionnaires on their individual characteristics (details in section 5.2.2.7).

5.2.2.4. Experimental Materials

Environment. The tasks were all solved on a laptop provided by us. Code snippets were presented in a web environment specially developed for this study. The look and feel of the web environment is based on the Eclipse IDE default look. Tooltips for variables and functions were displayed as typically expected in IDEs when hovering them. Syntax highlighting and line numbers were available. Selecting a variable highlights all occurrences of that variable. Next to the source code, the comprehensibility value of the method was displayed. A screenshot of the environment is shown in Fig. 5.1.

Code Snippets. We used a total of five Java code snippets to conduct the study, two of which were used to introduce the study and explain the task, and the remaining three had to be understood by the participants. All participants received the same Java code snippets, regardless of treatment.

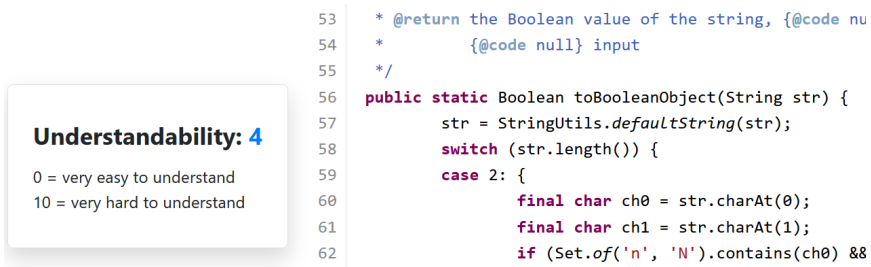


Figure 5.1.: Look and feel of the development environment that all participants used for the code comprehension tasks.

Each code snippet consisted of exactly one class with exactly one method. The method was documented via Javadoc.

The three task-related snippets were taken from either the Apache Commons Lang or Apache Commons Collection project. We selected the snippets in a way that no uncommon prior knowledge on, e.g., frameworks, would be required to understand them. As a result, the code contained mostly primitive data types and the features of newer Java versions were avoided. The snippets were slightly modified, either to introduce a bug or to make sure that all task snippets have the same cognitive complexity, an indicator for the comprehensibility of the method, which is particularly reliable regarding the subjective rating of developers [Cam18; MWW20]. This allowed us to weigh the answers to the three tasks equally and limited potential confusion or loss of trust in the displayed metric if the same metric value was displayed (by design) but very different difficulties were perceived. The three task snippets had a cognitive complexity score of 19, which is considered moderate to difficult to understand for Java methods.

Comprehension Questions. For each of the three tasks a participant was provided with a paper-based form which included five rows of a three-column table that had to be filled in. The cells of the first column each contained a method call, for example `toBooleanObject("of0")`. The other two columns had to be filled with the actual return value of the method and the

expected return value according to the Javadoc.

Questionnaires. Participants had to fill out several questionnaires. Related constructs are detailed in the next section.

To assess happiness, we use the Scale of Positive and Negative Experiences (SPANE) [DWT+10] which quantifies the frequency of positive (SPANE-P) and negative (SPANE-N) affective experiences, and the happiness overall of our participants (SPANE-B). The questionnaire was successfully used (and fully described) in other studies of behavioral software engineering, e.g., [GFWA17; GWA14]. To appraise personality traits, we use the Big Five Inventory [Dig90; MJ92]. To measure dispositional optimism and pessimism, we use the Life Orientation Test (LOT-R) [SCB94].

All measurement instruments have been psychometrically validated in several large-scale studies and show good psychometric properties [BJ98; CMB16; Jov15; LBW13; SC13; SCB94; Sum14], including consistency across full-time workers and students [SC13]. For all questionnaires we used a further psychometrically validated German version, i.e. [RHS17] for SPANE, [LLA01] for the Big Five Inventory and [GHKH08] for LOT-R.

5.2.2.5. Hypotheses, Parameters, and Variables

The independent variable relevant to RQ3.1 and RQ3.2 is the **displayed metric value (DMV)**. We developed the DMV to express the understandability of the source code. The DMV ranges from 0 (very easy to understand) to 10 (very hard to understand). The choice was driven by how natural it is for human beings to rate a concept from 0 to 10. The individual participant only saw three values for the metric. For the two introductory examples to explain the study tasks, we have chosen the values 1 (a very easy task) and 9 (a very hard task) to show possible extremes for coding snippet understandability. For all three experiment tasks, a participant either saw a value of 4 (moderately easy) or 8 (moderately hard), to cause the anchoring effect into two opposed directions (easy and hard).

Regarding RQ3.1, the relevant dependent variable is **perceived understandability (PU)**. Perceived understandability is defined as the sum of a participant's ratings for the three code snippets. The rating of each code snippet ranges, identical to the DMV, from 0 (very easy to understand) to 10 (very hard to understand). Accordingly, the value for PU is in the range of 0 to 30.

$H1_0$: There is no significant difference in perceived understandability (PU) between the two anchoring directions of a displayed metric value (DMV).

$H1_A$: There is a significant difference in perceived understandability (PU) between the two anchoring directions of a displayed metric value (DMV).

Regarding RQ3.2, we consider two common measures for code comprehension, which are time needed to complete all three tasks and correctness of the answers to the comprehension questions. The time was recorded for each task and summed up at the end. Correctness is the sum of correct answers to the comprehension questions of all three tasks, including both correct answers to actual return values and correct answers to return values according to the documentation. Therefore, the value for correctness is in the range of 0 to 30. To answer the research question, we combine time and correctness, as Scalabrino et al. [SBV+19] did, for example, to score participants higher that are both fast and correct. This results in **timed actual understanding (TAU)**, a participant's performance score obtained by combining correctness and time. Equation (5.1) provides the calculation for TAU, which ranges from 0 (the worst possible) to 1 (the best possible) and in which t_{max} is the time of the participant who took the longest.

$$TAU(correctness, time) = \frac{correctness}{30} * \left(1 - \frac{time}{t_{max}}\right) \quad (5.1)$$

$H2_0$: There is no significant difference in timed actual understanding (TAU) between the two anchoring directions of a displayed metric value (DMV).

$H2_A$: There is a significant difference in timed actual understanding (TAU)

between the two anchoring directions of a displayed metric value (*DMV*).

The investigation of RQ3.3, the extent to which individual characteristics influence the deviation from the displayed metric value, is exploratory research. Therefore, no hypotheses were formulated for this research question. The **metric deviation** is defined as mean absolute deviation of a participant's rating from the displayed metric value and calculated as shown in (5.2), where PU_i is the perceived understandability for task *i*.

$$\frac{|PU_1 - DMV| + |PU_2 - DMV| + |PU_3 - DMV|}{3} \quad (5.2)$$

The individual characteristics of interest in this study are experience with Java, personality, happiness and dispositional optimism and pessimism. Java experience was measured in years. Personality was operationalized by the five dimensions of the Five Factor model, i.e., extraversion $range = [0, 32]$, agreeableness $r = [0, 36]$, conscientiousness $r = [0, 36]$, neuroticism $r = [0, 32]$ and openness to experience $r = [0, 40]$. The higher the value, the more pronounced is the respective personality facet of a participant. The range of SPANE-P (positive affect) and SPANE-N (negative affect) is $r = [6, 30]$, from low frequency to high frequency of positive and negative experiences, respectively. SPANE-B, or happiness, has a range $r = [-24, 24]$, the negative pole refers to unhappiness and the positive one to happiness. Dispositional optimism and pessimism are independent constructs rather than a bipolar trait. Both are in the range $r = [0, 12]$, from low to high degree of optimism and pessimism, respectively.

5.2.2.6. Experiment Design

The experiment was a between-subject design with two treatment groups. Assignment to a treatment was double-blind and random.¹ None of the authors knew which participant, even as an anonymous data point, was in which treatment group until the data was evaluated.

One group saw a *DMV* of 4 next to all code snippets. We call this group the *easy* group from this point on. The other group saw a *DMV* of 8. We refer to this group as the *hard* group from this point on. There were no further differences in the treatment of the two groups.

The reader might notice an absence of a control group, which does not see any *DMV*. This is intentional and suggested in literature on the anchoring effect, which we discuss in more detail in [Section 5.2.4.1](#).

Following [Wohlin et al.](#)'s guidelines [[WRH+12](#)] for conducting experiments in software engineering, we had the study design reviewed by two peers in two iterations and conducted a pilot test. Furthermore, we have identified and implemented a number of measures that we believe contribute to mitigating threats to validity. Limitations of our final study design, that is, what affects the interpretation of our results, are discussed in [Section 5.2.4.1](#).

Measures to Address Construct Validity. We used psychometrically validated questionnaires for assessing individual characteristics. Regarding the operationalization of code understanding, we have oriented ourselves on how the construct was measured by our peers in peer-reviewed research [[MWW20](#); [SBV+19](#)]. With time and correctness, we measured two argumentatively important and objective aspects of code understanding and combined them with equal weight, similar to what Scalabrino et al. [[SBV+19](#)] did.

We designed a plausible scenario to justify the display of the metric value and developed an experiment description for participants which did not reveal the essence of the experiment to prevent *hypothesis guessing* [[WRH+12](#)].

¹We agree, to some extent, with Baltes and Ralph [[BR22](#)] that random should be used sparingly, so we will add here that participants were assigned to a condition based on the time slot they signed up for. When assigning treatment conditions to a time slot, it was ensured that the conditions were distributed equally over different times of day.

The description did not present false information, but hid only the objective of the anchoring effect. Closely related, we prevented the threat of *experimenter expectations* [WRH+12] by double-blind assignment of participants to treatment groups.

To prevent *evaluation apprehension* [WRH+12] and unnecessary stress we always tried to have exactly two participants simultaneously in the room, the experimental supervisor could not look over the shoulder of the participants, and it was emphasized several times that the answers were anonymous. Due to their intimate nature, the Big Five and LOT-R questionnaires were completed only after the code comprehension tasks had been completed. Both participants in the same time slot were also in the same treatment group to avoid *diffusion or imitation of treatments* [WRH+12] by one of the two participants making a comment on the displayed metric value.

Measures to Address Internal Validity. We discussed every variable listed in Siegmund and Schumann [SS15]’s mapping study on confounding variables in code comprehension experiments. Of the 37 variables listed, only two remained even after thorough planning of the experiment, which we see as potential threats to validity: the Hawthorne effect and selection (the generalizability of student participants). We discuss both in 5.2.4.1.

Of the potential confounders that we have explicitly controlled, we highlight the following two. First, we selected several code snippets that a participant had to understand to reduce the influence of individual data structures and program semantics. Snippets were also neither too difficult nor too easy according to a validated code comprehensibility metric [MWW20], and moreover of comparable comprehensibility. Second, we implemented a tool to view and interact with the code in the tasks. This gave us full control over the environment and allowed us to reduce the displayed elements to a minimum to increase internal validity. In addition, no participant had an advantage, since no one was more familiar with the environment than everyone else.

Measures to Address External Validity. We used code snippets taken from real-world, actively developed famous open-source projects and avoided removing comments or obscuring method or variable names to force code understanding. Instead, we have created a realistic software maintenance scenario in which the developer needs to understand code that does not necessarily fit the documentation and that may contain a bug, requiring the developer to check what the actual output values are for certain input values.

Measures to Address Conclusion Validity. Regarding *reliability of treatment implementation* [WRH+12], we used a strict protocol and double-blind condition assignment to ensure that each participant received the same information and was treated equally. The same investigator conducted the experiment with all 45 participants. The DMV of either 4 or 8 was displayed the same for all participants in a treatment group, as it was an automatic display in an environment controlled by us.

To prevent *random irrelevancies in experimental setting* [WRH+12] we reserved the room two weeks in advance and for the entire day on each day the study was conducted. We were prepared to document irregularities, but did not have to do so.

5.2.2.7. Procedure

The following steps took place once a participant arrived in the room reserved for the experiment at the agreed time. Participants were provided with a consent form and were informed verbally about the aim of the study. They were shown the laptop and the files on the laptop. Then they had to fill out two questionnaires, one on demographic data and on their happiness (SPANE). The instructor made the participants save and close the questionnaires on their own after completing them so that they did not feel observed by the instructor.

The instructor explained the task, the scenario and the development environment to the participants. Thereby, the participants were shown a

filled out task sheet for one method as an example and the solution was exemplified for the first two input values. The visualization of the metric and its display for supposedly informative reasons was explained and a second introductory example, which was significantly harder to understand than the first one, demonstrated that the metric works well. When asked, the instructor told that the metric was developed by experts, similar to what Draganich and Erdal [DE14] did in their placebo sleep experiment.

The instructor summarized the task and mentioned that the employer in the task scenario wants them to work efficiently but also correctly. Participants then had the opportunity to ask questions before starting with the tasks. The time recording was done per task and the recorded value was stored in a spreadsheet without the participants noticing it. After all three tasks had been completed, the participants finally had to fill out the questionnaires on individual characteristics.

5.2.3. Results

5.2.3.1. Descriptive statistics and dataset preparation

45 students participated in the study¹. Two of them did not submit complete data for RQ3.3 (e.g., missing an item for the SPANE questionnaire). We decided to exclude them from the dataset to enhance our confidence in how serious all participants were in completing all tasks. We thus had an overall sample size of $n = 43$ participants (41 male, 2 female). Mean age was 24.47 ($SD = 2.84$), average declared experience with the Java programming language was 5.83 years ($SD = 2.37$).

Participants were randomly allocated to the *easy* group, $n = 20$, or the *hard* group, $n = 23$. Declared experience with the Java programming language was comparable for both groups after random assignment ($M = 5.25$, $SD = 2.60$ for the easy group, $M = 6.33$, $SD = 2.06$ for the hard group).

The easy group was shown a *DMV* of 4 for the three tasks, or 12 combined, and provided a *PU* of 15.40 ($SD = 4.17$, *median* = 14.50). The hard group

¹We cannot offer a precise acceptance rate because course attendance is not mandatory and cannot be recorded.

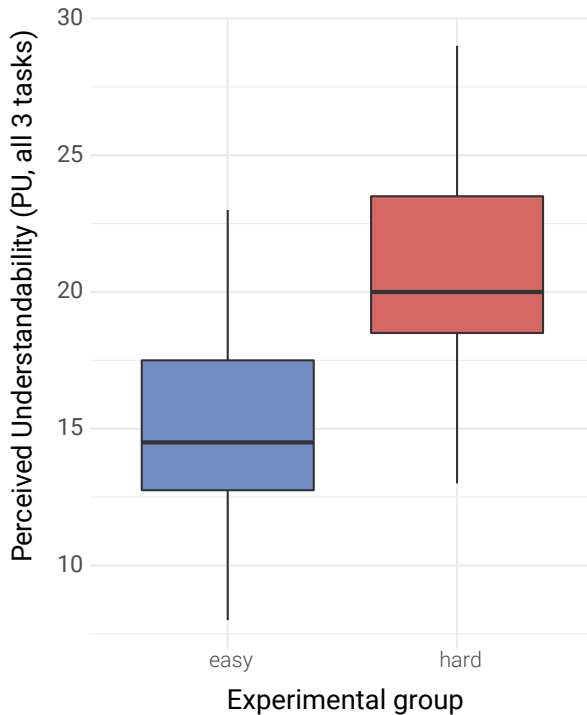


Figure 5.2.: Perceived understandability (PU) of the tasks for the easy group and the hard group (three tasks, range 0 (easiest) to 10 (hardest), combined range 0 (easiest) to 30 (hardest)).

was shown a *DMV* of 8 for the three tasks, or 24 combined, and provided *PU* of 20.83 ($SD = 4.23$, *median* = 20.00). A graphical comparison is offered in the boxplot of [Figure 5.2](#).

The easy group performed with an average *TAU* of $M = .37$ ($SD = .17$, *median* = .41). The hard group performed with an average *TAU* of $M = .37$ ($SD = .11$, *median* = .36). A further boxplot comparison (included in the supplemental material) did not suggest significant difference.

5.2.3.2. Hypothesis testing

There was a significant difference between PU of the two groups¹², $t(40.318) = -4.227$, $p = .000132$, 95% CI $[-8.02, -2.83]$, with a large effect size, $d = -1.29$, 95% CI $[-1.97, -0.61]$.

We thus reject $H1_0$ in favor of $H1_A$: **There is evidence for a difference in perceived understandability between the two anchoring directions of a displayed metric value.**

There was no significant difference between the timed actual understanding (TAU) of the two groups³, $W = 256$, $p = .5385$, with a negligible effect size, $d = -.006$, 95% CI $[-0.62, 0.61]$.

We do not reject $H2_0$. **There is no evidence for a difference in timed actual understanding between the two anchoring directions of a displayed metric value.**

5.2.3.3. Exploratory analysis

We provide the computed correlation coefficients of the metric deviation (calculated as in formula 5.2) with affect-related metrics and personality-related metrics in Table 5.1. The first two rows provide the correlation coefficients for the two experimental groups (between), while the third row combines all participants (within).

For brevity's sake, we will call the "metric deviation" simply "deviation" the rest of this section. As a reminder, the wider the deviation, the bigger the gap between the subjective rating and the shown metric value, or the manipulation, on the easy direction or on the hard direction.

Given the exploratory nature of RQ3.3, no estimation of significance was conducted for the correlation coefficients. As a cutoff for potentially interest-

¹Welch Two Sample t-test given evidence for normality (Shapiro-Wilk test, $p > .26$ for both groups) and no further assumption on the population variance.

²We are aware of the open debate on whether Likert items are ordinal data or continuous data [Mur13]. We believe, in line with psychometric theory, to have Likert items capture discrete points over a continuous scale. All scales that we use for individual characteristics are psychometrically validated Likert items.

³Wilcoxon rank sum exact test given evidence for non-normality (Shapiro-Wilk test, $p = .03$ for the easy group).

Table 5.1.: Spearman’s ρ correlation coefficient of individual characteristics with the deviation of the subjective rating. Abbrev.: Exp=Java experience in years, Consc=Conscientiousness.

Group	Exp.	SPANE-P	SPANE-N	SPANE-B	Optimism	Pessimism
easy	-0.28	0.11	-0.23	0.18	0.32	0.18
hard	0.24	-0.15	0.13	-0.23	0	-0.04
combined	-0.03	-0.12	0.06	-0.16	0.14	0.07

Group	Extraversion	Agreeableness	Consc.	Neuroticism	Openness
easy	-0.1	0.08	0.46	-0.12	-0.01
hard	-0.09	0	-0.09	-0.06	0.05
combined	-0.13	0.08	0.14	-0.05	0.01

ing individual characteristics, we will only consider correlation coefficients $|\rho| > 0.1$.

When exploring the data between the two experimental groups, we notice that an increase in programming language experience is associated with a decreased deviation when the manipulation suggests an easy task and an increased deviation when the manipulation suggests a hard task. The opposite happens with happiness (SPANE-B). An increase in happiness is associated with an increased deviation for the easy group and a decreased deviation for the hard group. The two major components of happiness, SPANE-P and SPANE-N, show coherence with the aggregated happiness score.

An increase for *both* optimism and pessimism is associated with a wider deviation for the easy group, while no correlation is observed for the hard group. Of all personality traits, an increase in conscientiousness seems to be strongly correlated with an increased deviation for the easy group, followed by neuroticism but with an inverse relationship. No personality trait shows a $|\rho| > 0.1$ for the hard group.

When combining the two groups, in a within subject analysis, with the assumption that the two groups are from the same population, an overall negative relationship between happiness and the deviation is observed (happier participants deviated less). An increase in optimism is associated with a wider deviation. Of the personality traits, an increase in optimism and

conscientiousness were correlated with a bigger deviation, but an increase with extraversion was correlated with a smaller deviation. Programming experience seems not to play a role in the deviation.

5.2.4. Discussion

Metrics provide developers with quantitative insights into the quality of their source code, but most of the metrics used in practice are not sufficiently validated. We have investigated the extent to which developers are actually subconsciously influenced by the value of a displayed made-up metric to raise an awareness of responsibility in code quality reporting.

We found a significant and strong anchoring effect, which means that developers are strongly influenced by a displayed metric value in their rating of source code comprehensibility. This finding is consistent with over 40 years of research on the anchoring effect [FB11], yet investigation in a specific software engineering context is nevertheless a valuable contribution to build the foundation for future studies, for example to investigate consequences of the demonstrated effect.

One such potential consequence could be an improved or worsened understanding of the code. However, in our study we could not provide evidence that the suggestion of simple or difficult code provokes a placebo effect, in the sense that the beliefs concerning the comprehensibility of the source code influences the speed and correctness with which a software engineer answers comprehension questions. Since cognitive performance and creativity, arguably both characteristics necessary for code understanding, can be influenced by a placebo [RMI+17; TBG+18], we would have expected to observe such an effect. Assuming that a placebo effect for code comprehension can be observed theoretically, we see two possible reasons why we could not in our experiment.

First, it could be that a displayed metric value is simply not a strong enough influence to cause performance changes. Compared to other placebo studies [CL07; DE14; RMI+17], we did not manipulate the participants to claim that our treatment had specific beneficial or performance-enhancing

effects. Instead, we displayed a metric value that indicated how easy or difficult code is to understand. We left it up to the participants to interpret what it means and what consequences it might have if source code is easy or hard to understand. Accordingly, code comprehension may have actually improved or worsened as a result of the manipulation, just not in the way we measured it.

This brings us to the second possibility that time and correctness are not all-inclusive proxies for code comprehension. We speculate that physiological measurements may have led to a different result. For example, cognitive load and stress levels at the end of task processing might have been lower in the easy group, as they may have been more comfortable with the task. While the question of ideal measures of code comprehension is beyond the scope of this work, and we have not measured these variables, we argue that they nevertheless reflect relevant dimensions of code comprehension. With the increase in physiological measurements in the field of code comprehension [Fak18; PSP+18; SPB+20], we see much potential for future studies to replicate our experiment with alternative measurement methods to shed light on the matter.

Regarding individual characteristics that influence the strength of the observed anchoring effect, our results, while exploratory and based on correlations, are only partially consistent with the few relevant studies conducted to date [FB11]. We observed that participants with low extraversion are less subjected to the anchoring effect, which is consistent with existing literature [EC10]. However, our results indicate that participants with high conscientiousness are less susceptible to the anchoring effect, which is contrary to the findings of Eroglu and Croxto [EC10]. We could not find a positive correlation between anchoring and the personality trait openness as McElroy and Dowd [MD07] did. Further, our results suggest that happier people might be more subjected to the anchoring effect, which also does not coincide with previous studies [BGL00; ES09].

We found that optimism positively correlates with the metric deviation, which corresponds to a weaker anchoring effect. Programming experience, on the other hand, might not play a role in anchoring, but this should be

taken with caution, since we had a homogeneous group of experienced developers and results could be different for inexperienced developers.

If the code comprehensibility metric shows a low value for a rather difficult task, personality factors could play a bigger role. In particular, conscientiousness might be the strongest predictor of a deviation when metrics are too conservative in how difficult a task might be (easy group), followed by optimism. In conclusion, our results suggest that the anchoring effect might not be a universal rule that applies equally to all participants. The partial deviation of our results from previous findings is an indication that more studies are needed in this regard. The exploratory settings of RQ3.3 set basic building blocks upon which we call our peers to conduct future research.

5.2.4.1. Limitations

A detailed description of design decisions made in advance to mitigate validity risks can be found in [Section 5.2.2.6](#). What follows are limitations that affect the final study design and that should be considered when interpreting our results.

As for potential confounding variables, we could reduce a lengthy list of known variables [SS15] to two that might have affected the results: the Hawthorne effect and the selection of students as participants.

The Hawthorne effect [MWI+07; RD03] describes that participants in experiments would behave differently because they were observed. While we addressed *hypothesis guessing* [WRH+12] with a plausible scenario that does not reveal the essence of the experiment, and we addressed the threat of *evaluation apprehension* [WRH+12] through privacy and data anonymization, it cannot be ruled out that participants followed the proposed metric value more closely than they would have done outside an experimental setting. According to [FB11] the current dominant view of the anchoring paradigm focuses on confirmatory hypothesis testing in the sense that information is activated that is consistent with the anchor presented. We assume that this also applies to our experiment and that the observed anchoring effect, if at all, is only to a small extent due to an experimentally provoked good will of

the participants for the metric. A future field study could provide clarity in this respect.

As argued earlier in the description of participants (Section 5.2.2.2), we invited a convenience sample of students of a software engineering M.Sc. study program. In our sampling strategy, we prioritized internal validity, which was enhanced by a homogeneous level of experience with the programming language, paradigm, and task type. We also believe that our results can be generalized to a population of professional software engineers. In the discussion about representativeness of software engineering students, opposing opinions have existed for the last 20 years [KPP+02; SAA+02; Tic00]. Recent studies have shown that comparable results can be achieved with both groups of students and professionals—as long as the scope of the investigation is carefully considered (see, e.g., [SMJ15]). We have confidence in the robustness and soundness of our research design. Recent commentaries [FZB+18] have, once again, highlighted how diverse the views are on the topic. We side with the view summarized by Runeson [FZB+18] as well as Baltes and Ralph [BR22] that a convenience sample of students is justified in the investigation of central behavioral and cognitive processes, as was the case in our study. There is evidence, for example, that the anchoring effect is not restricted to laymen and that more experienced people are influenced by it as well [FB11; TK74].

Then, since we did not have a control group in our experiment that was *not* shown a metric value, we cannot say how a control group would have rated the snippets. For the demonstration of the anchoring effect this is not a limitation, and it is consistent with the body of research on the anchoring effect to not have a no-anchor control group [CG08; ME05]. However, regarding the placebo effect, we would like to stress that our design would not be able to decide strictly speaking whether any observed effect is due to the placebo, but with the design, it is still possible to demonstrate the extent to which a displayed metric value influences code understanding in a positive or negative direction. Our study is based on a comprehensive body of research that has provided evidence for the placebo effect, so we assumed that the effect would also exist in our scenario and opted for the

study design described above.

Finally, we are aware that the way in which the metric value was displayed is very prominent. We argue that developers are used to static code analysis tools reporting metric values in similar ways, and IDEs increasingly offer the possibility to display code quality metrics directly in the source code. Even if this situation is not as common, we refer to Critcher and Gilovich [CG08] and related works showing that for the anchoring effect to work, much more inconspicuous anchors, which do not even have to be highlighted, are usually sufficient. Again, a field study with realistic IDE plugins and existing metrics could be an option to repeat the investigation of the effects in an industrial environment.

5.2.4.2. Implications

Since developers are influenced by a shown metric value in their subjective evaluation of a code snippet's comprehensibility, we highlight the following implications.

First, those responsible for reporting code quality metrics should be aware of their responsibility that non-validated metrics lead to unwarranted manipulation of developers, the consequences of which we do not know yet. Future studies can build on our results and investigate possible consequences.

Second, since it is a common practice in code understanding studies to ask developers for a subjective rating, such studies should ensure that individual participants are not anchored by context factors such as displayed metric values. It may already be sufficient that the instructor or the task description hint at something about the complexity of a code snippet to be examined. Also, for example, different amounts of time available for processing different code snippets could lead to an anchoring of the participants in their subjective assessment. If the study cannot be controlled with certainty in this regard, the measure of subjective ratings should not be used.

We echo the call of Mohanani et al. [MST+18] and propose a debiasing technique for the anchoring effect. The debiasing here is about developing validated metrics (or validate existing ones) before showing their values to

software developers.

We do not consider it an issue when developers are anchored in their subjective judgement by a validated metric that may be able to consider more factors and evaluate code more objectively than a developer could. We are aware that, for example, static code analysis tools do not intentionally lie, and that many of the metrics seem to make intuitive sense. It becomes problematic, however, when developers interpret quality aspects into metrics that were not intended to be measured by the metric, or when the metric is no more than an implemented, albeit well-thought-out, idea. Therefore, tools should describe very precisely what the metric intends to measure and support this measurement with systematic research.

A number of previous studies already called for more effort to be put into disseminating research findings among practitioners so that they can rely on evidence rather than forming biased and error-prone conclusions based on personal impressions [DZB16; KDJ04]. Especially regarding the clearly shown anchoring effect, validated metrics should have an easy time anchoring developers where they should be anchored evidence-wise and overcome the circumstance that developers sometimes tend to prefer their own opinions over empirical evidence [RHB03].

We consider the negative results on RQ3.2 to be good results under the circumstances described above. Since the situation is unlikely to change in the near future, it is at least good to know that a few random numbers may not have a negative impact on a developer's understanding time and correctness during maintenance. Whether other aspects of code understanding can be influenced by a placebo will need to be investigated in future studies.

5.3. A Study of Anchoring Developers Through Task Descriptions

We learned from the previously described study that developers can be anchored and therefore influenced in their code comprehensibility judgments if they are shown a metric value during the code comprehension task. To

remain in the usual linguistic style and not to cause too much confusion by constantly referring to a first and a present study, in this section we refer to the present study as ‘ours’ and reference the publication of the first study [WPGW21] in third person as if it were from a different author team.¹ Our goal now is to replicate the findings in a more realistic setting and overcome some of the limitations of the previous study. In particular, we now attempt to manipulate study participants by displaying a code comprehensibility cue to them only at the beginning of the study.

Thinking about what you tell participants at the beginning of your study matters — at least if your concern is to produce a study design with high validity and reduce the probability of biased results. Consider the following scenario. Caroline is a developer who wants to take part in an advertised study to research the influence of code comments on source code comprehensibility. On site, the study leader explains the study process to her. She would have to look at a source code snippet and rate its comprehensibility. Caroline is a little nervous, but the study leader instinctively reassures her that the code will not be too difficult to understand. The study leader has good intentions here, yet there might be unattended consequences for this action.

The reader might start seeing, at this point, the internal validity of the fictitious study threatened, since the assessment of code comprehensibility was most likely influenced at that moment by the words of the study leader. The introduction to a study should strictly follow a predefined script. The Standards for Educational and Psychological Testing enlist several recommendations to assemble and present instructions to administer a test, including the instructions presented to test takers so that “it is possible for others to replicate the administration conditions under which the data on reliability, validity [...] were obtained” [APA14][p. 90]. Additionally, by following a predefined script, experimenter expectancies can be avoided [WRH+12]. We share this sentiment, and yet, from the researchers’ perspective, it does

¹In fact, the author teams are not identical, but they overlap. The reader will notice when looking at the publication on the present study that we also reported there in third person about the first one, which was primarily due to the double-blind review process.

not always turn out to be that simple to control for any external influences on subjective assessments.

Subjective code comprehension assessments are a common measure in code comprehension studies because of their simplicity. At the same time, we know from more than forty years of research on the anchoring effect [FB11; TK74] that even inconspicuous environmental factors are sufficient to influence people in their estimation [CG08].

If we could confirm the findings of the previously described study in Section 5.2, the presence of an anchoring effect would imply that subjective ratings should only be used when contextual factors can be controlled for with a high degree of certainty for all participants, thus minimizing the risk of anchoring individual participants and biasing the experiment results. In any case, additional insights on the influence of scenario descriptions on subjective code comprehension ratings provide a useful basis for design decisions, and can partially counteract existing uncertainty about what constitutes a good empirical study [Sie16; SSA15].

5.3.1. Background and Related Work

We covered the anchoring effect and the studies on it in Section 5.2.1. Here, we will focus on how the present study builds on that of Wyrich et al. [WPGW21].

They showed in their experiment that displaying different values of a made-up code comprehensibility metric significantly anchored study participants in their subjective ratings of source code comprehensibility. Participants were assigned to one of two groups and saw three code snippets one after the other, which they had to understand and evaluate in terms of their understandability. All participants were shown a supposedly validated metric next to the code snippets, which, however, displayed a different value depending on the treatment group (either 4 or 8) and was intended to anchor the participants in this way. The observed anchoring effect was significant ($p < .01$) with a large effect size ($d = -1.29$).

The experiment by Wyrich et al. [WPGW21] is closest to ours, but differs

decidedly in some respects, which is why we do not refer to ours as a *replication* study. Wyrich et al. [WPGW21] anchored the participants by displaying the anchor, i.e., the numeric value for the metric, during the code comprehension task, next to the code snippets, thus much more prominently than we do. We study the anchoring of experiment participants at the beginning of the study using a description of the expected code snippet, hence, displayed before showing the coding snippet. Our choice is based on the limitation discussed by Wyrich et al. [WPGW21] that showing the anchoring element during the code comprehension task might be unusual. We sought to verify that the results still hold true when we were closer to realistic scenarios, some of which we discuss in Section 5.3.4.

Other differences in design characteristics between the two studies are outlined in Table 5.2. Nevertheless, similar to a *reproduction* study¹, we build on the work by Wyrich et al. [WPGW21] to make both studies as comparable as possible and thus contribute to a common overall picture.

Table 5.2.: Comparison of experiment characteristics

	Wyrich et al. [WPGW21]	Our study
Scenario presentation	next to code snippet, metric only	prior to code snippet, metric and snippet details
Metric values (scenarios)	4, 8	3, 8, none
Comp. task	determine output and rate understandability	rate understandability
Participants	43 students	206 students, 50 professionals
Setting	onsite, code on screen	remote, code on screen

5.3.2. Methodology

The goal of our study is to investigate the effect of the presence and content of information about a code snippet at the beginning of a scientific study on

¹<https://www.acm.org/publications/policies/artifact-review-badging>

the participant's rating of the comprehensibility of that code snippet. To this end, we formulated research question 3.4 given in the introduction to this chapter ([Section 5.1](#)).

5.3.2.1. Research Design

We conducted a controlled and randomized between-subjects experiment with 3x2 factorial design. Each participant was assigned to one of three scenario groups and had to understand one of two code snippets. A schematic representation of the research design is provided in [Figure 5.3](#).

The experiment took place online via a self-hosted instance of LimeSurvey, and participation was entirely anonymous. Participants first confirmed that their consent to participate was informed and agreed to the data and privacy policy. On the next page, participants saw a description of the task that awaited them in the next step. Depending on the randomly assigned group, this description included information about the code comprehensibility rating of a fictitious expert system. Then, each participant had to understand one randomly selected code snippet and provide their code comprehensibility rating. Finally, a demographic data questionnaire concluded the study.

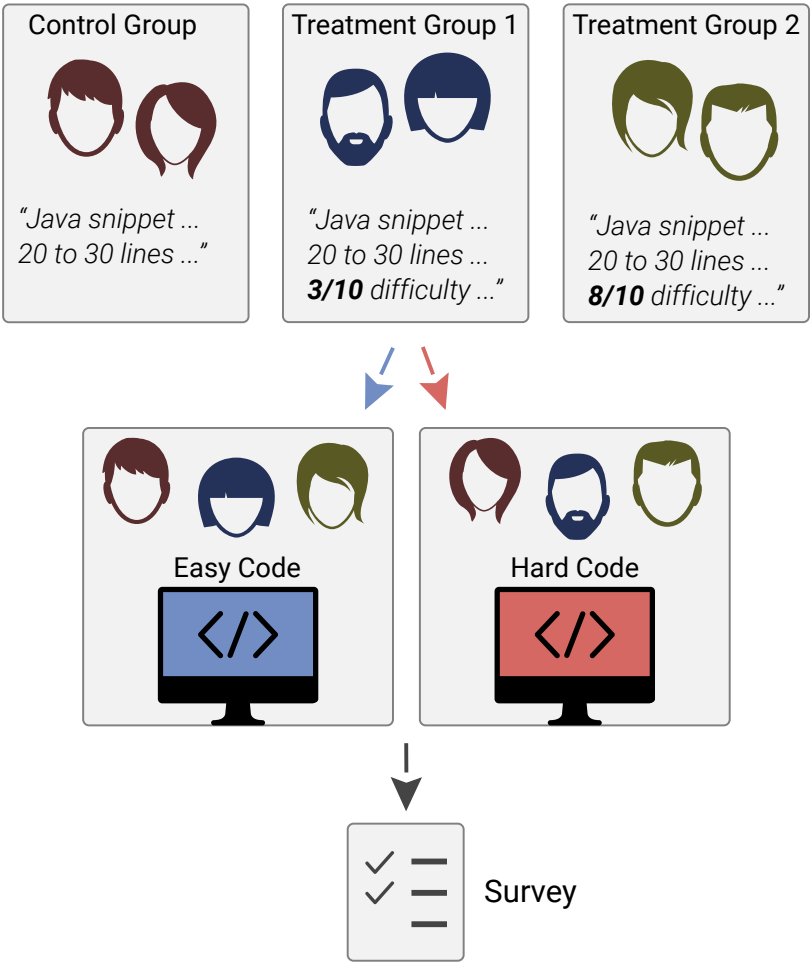


Figure 5.3.: Schematic representation of the research design. Participants are randomly assigned to one of three groups that provide different information about the code snippet to be understood next. Each participant then randomly sees exactly one from a pool of two snippets, either an easy one or a difficult one. A survey that is the same for all concludes the experiment.

5.3.2.2. Experimental Materials

Scenarios. All participants saw a textual scenario description of the expected code comprehension task. The wording of this description was the same for all three groups, but one of three paragraphs was not shown to the control group, and there was a hint to Treatment Groups 1 and 2 for a value of either 3 (easy) or an 8 (hard) as a comprehensibility score by the expert system. The description was as follows:

You will look at a Java code snippet in a bit. The only information that we provide you about the snippet is the following: The code snippet is between 20 and 30 lines long and tests a String, a sequence of characters, for a criterion.

<<Begin Treatment Group text snippet>>

We developed an expert system to rate the understandability of code based on multiple metrics. This system rated the code snippet you will look at in a few moments as

<<Treatment Group 1>> **a 3 out of 10.**

<<Treatment Group 2>> **an 8 out of 10.**

The system uses a scale from 1 to 10, where 1 is very easy and 10 is very hard to understand.

<< End Treatment Group text snippet>>

Your only task is to judge its understandability on a scale from 1 to 10, where 1 is very easy to understand and 10 is very hard to understand. The code is fully functioning and bug-free. You will have unlimited time to look at the code snippet. Feel free to rate the code snippet whenever you think you have an adequate impression to judge its understandability.

Code Snippets. Since the influence of a scenario description on anchoring in source code comprehensibility ratings might depend on the actual difficulty

of the code snippet, we were interested in studying both an easy and a hard snippet in the study.

Every developer has a slightly different idea of how understandable certain code is. Therefore, we pre-selected five Java code snippets and invited eight software developers to assess their comprehensibility. The pre-selected snippets all met the criteria that no domain knowledge is necessary for understanding and that they are neither too long to be displayed in full on a screen nor too trivial to be understood after just a few seconds. In pre-selecting functions that are rather complex, we, like Wyrich et al. [WPGW21], used the cognitive complexity metric [Cam18], which has been shown to correlate in particular with subjective evaluations by developers [MWW20].

Through this preliminary evaluation, we were able to identify one easy and one difficult code snippet that we subsequently used in our study. The easy code snippet is a method from the apache commons-lang `StringUtils` class and checks if a given character sequence contains both uppercase and lowercase characters. The difficult code snippet is the solution to a coding challenge [WGW19] to find the longest palindromic substring within a string. Both code snippets are included in the supplemental materials [WMG22b].

Questionnaire. The experiment was completed by a short demographic questionnaire, which asked about the current main occupation of the participant. The choices were *Student*, *IT professional*, *Researcher* and *Other* (with the possibility to specify the occupation). We then clarified the specific intent of our study and again listed ways to contact us with potential questions or comments.

5.3.2.3. Participants

We invited a convenience sample of software professionals and computer science university students to take part in our study. To disseminate the invitation, we used social media and asked personal contacts to draw attention to the study in their software companies. Computer science university students (software engineering curriculum, B.Sc. and M.Sc.) were asked

to participate in the study, for example to fulfill course requirements to participate in scientific studies. We ensured the participants of anonymity, and they could withdraw from the study at any time or not participate at all, and still fulfill their requirements. The only requirement for participation was a basic understanding of the Java programming language.

We encouraged participation with the low amount of time commitment of 10 to 15 minutes for the entire study. Furthermore, we pledged to donate €5 to a good cause for every participant among the software professionals that completed the study. Subjects could choose between three charity projects on different topics, or split the donation evenly among the three projects.

Following the goal-setting theory of motivation [LL02], we also invited participants with the clear goal of assessing the comprehensibility of a particular code snippet. Setting such a specific and challenging goal may lead to increased effort and persistence, which is desirable for completing the study. Our purpose in doing so was to pique the interest of developers who want to demonstrate their ability to understand code.

5.3.2.4. Conceptual Model

We build on the study by Wyrich et al. [WPGW21] by developing the conceptual model of Figure 5.4, which summarizes all variables and their hypothesized relationships graphically. We investigate whether the anchoring effect is confirmed in the form that a *Scenario* description, which controls for information presented at the beginning of the study about the code snippet to be understood, influences *Perceived Code Comprehensibility* (H1). In our model, we introduce two other factors that we theorize to have an effect on *Perceived Code Comprehensibility*. The *Code Snippet Difficulty* will influence the *Perceived Code Comprehensibility* (H2). Wyrich et al. [WPGW21] selected the code snippets for their study to be of comparable complexity to control for the potential influence of snippet complexity in this way. We now want to understand the extent to which an easy or a hard task affects the *Perceived Code Comprehensibility*, and we argue that this influence will be stronger than the one provided by the *Scenario*.

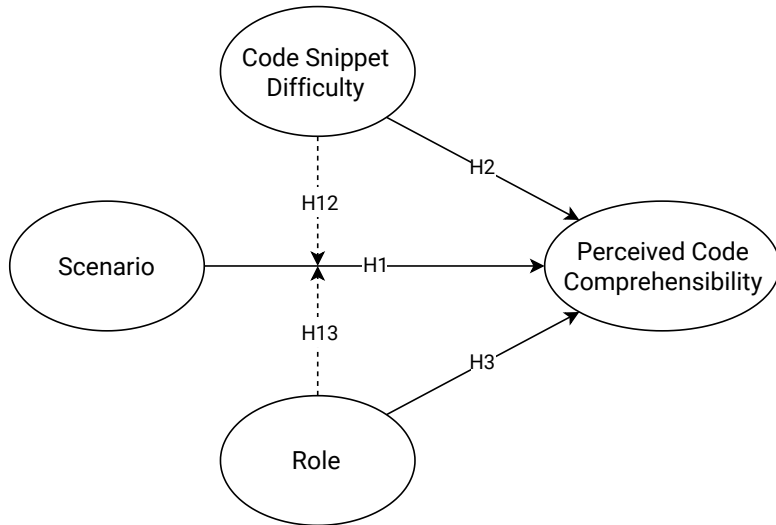


Figure 5.4.: Conceptual model for the study (dashed arrows = moderators)

Table 5.3.: Assignable values for the variables of the model

Variable	Values
Scenario	{baseline, easy, hard}
Code Snippet Difficulty	{easy, hard}
Role	{student, professional}
Perceived Code Comprehensibility	[1.. 10]

Furthermore, Wyrich et al. [WPGW21] had only students as participants in their study. Much discussion has happened in the literature on differences between students and professionals [FZB+18] in terms of productivity, performance, and software quality, with some claiming or finding that there are little to none [SMJ15, e.g.], others that there are [SAA+02, e.g.]. Wyrich et al. [WPGW21] themselves discuss this circumstance as a potential limitation of their study, although literature suggest that the anchoring effect is not restricted to inexperienced people [FB11; TK74]. We avoid any debate and

investigate whether a *Role* influences the Perceived Code Comprehensibility (H3).

Finally, given the absence of prior literature, we want to test for the influence that these factors have with each other. The interaction between Scenario and Code Snippet Difficulty (H12) as well as the interaction between Scenario and Role (H13) are further modeled as *moderators*¹ on the influence that Scenario has on Perceived Code Comprehensibility. Investigating the moderators will enable us to better characterize a potential anchoring effect of Scenario on Perceived Code Comprehensibility.

Table 5.3 provides a summary of the values that can be assigned to each of the four variables in the conceptual model. Due to the small number of six self-identified researchers, we decided to combine them with the 44 software professionals into one group for the analysis. We still consider that a meaningful distinction can be made between students and professionals. Nevertheless, we will discuss the potential consequences of this design decision in Section 5.3.4.2.

5.3.2.5. Analysis Procedure

The data violated at least one assumption of most of the commonly used modelling techniques², and it also presented evidence for non-normality on the dependent variable (Shapiro-Wilk Test, $W = 0.97$, $p < .00001$).

We thus opt for a Partial Least Squares Structural Equation Model (PLS-SEM). PLS-SEM was recently introduced to the discipline by Russo and Stol [RS21] as part of the SEM statistical technique family for causal-predictive approaches in the behavioral sciences. We direct readers to Russo and Stol's work [RS21] for an introduction but, in short, PLS-SEM is suited for testing a theoretical framework from a prediction perspective, when the structural model is complex, and when distribution issues are a concern [HRSR19].

¹A variable w is called a moderator when the relationship between other two variables A and B is influenced by w , and it is modelled as interaction effect [CCWA13].

²Including linear regression models, methods from the various ANOVA families with applied data transformation techniques, and ordinal logistic regression methods. The latter could not be applied because of proportional odds violation.

We model the factors of [Figure 5.4](#), as defined in [Section 5.3.2.4](#), as a PLS-SEM in *R* 4.1.2 [[R C21](#)] and *SEMinR* 2.2.1 [[RDC21](#)]. The output will provide us with statistics on the predictive power and significance of the relationship between the factors¹.

5.3.3. Results

We recruited 256 participants, of which 206 were students and 50 were professional software developers (see [Section 5.3.2.3](#)).

[Table 5.4](#) provides descriptive statistics for the Perceived Code Comprehensibility (PCC) grouped by factorial assignment condition and role. Students and professionals were overall very close or identical in their median PCC for the same scenario and code snippet. The code snippet we considered easy was actually perceived by participants as easier to understand than the snippet predicted to be perceived harder to understand, regardless of scenario and role. Participants provided the highest median PCC ratings for the combination of hard scenario and hard code snippet (a median value of 7 and 7.5 from students and professionals). For the easy code snippet, the scenario seems to have had a smaller overall impact on the PCC ratings.

In [Table 5.5](#) we show the results of the statistical analysis for each hypothesis. We find a significant path from Scenario to Perceived Code Comprehensibility (path coefficient² $\beta = 0.165, p < .001$) confirming the presence of the anchoring effect (H1). Code snippet difficulty had the expected strong influence on PCC (H2; $\beta = 0.418, p < .001$), but, furthermore, seems not to be a moderator of the relationship between Scenario and PCC (H12; $\beta = 0.076, p > .10$). The paths from Role to PCC (H3) and the moderating effect of Role on the path from Scenario to PCC were insignificant (H13). Thus, our results support H1 and H2; they do not support H12, H3 and H13.

¹Readers familiar with SEM will notice that we are applying PLS-SEM as an analysis technique to estimate single-item indicators. That is, we rely on PLS-SEM robustness to perform a ‘regression job’ instead of using the technique for its intended psychometric purposes. We elaborate on these issues in the limitations section.

²Path coefficients are expressed as standardized regression coefficients in terms of standard deviations; see, e.g., [[HRS11](#)].

Table 5.4.: Descriptive statistics and group assignment. CSD = Code Snippet Difficulty, n = group size, PCC = Perceived Code Comprehensibility, M = mean, SD = standard deviation, Mdn = median.

CSD	Role	n	PCC M	PCC SD	PCC Mdn
Scenario: baseline					
easy	student	35	4.0	3.33	2
	professional	10	3.3	2.79	2
hard	student	32	5.25	2.05	5
	professional	10	5.6	2.88	6
Scenario: easy					
easy	student	25	2.56	1.94	2
	professional	9	2.89	2.67	2
hard	student	33	4.67	1.49	4
	professional	5	6.0	2.65	7
Scenario: hard					
easy	student	32	3.25	2.90	2
	professional	2	1.0	0.0	1
hard	student	49	6.22	2.05	7
	professional	14	6.86	2.35	7.5

The model explains $R^2 = .223$ of the variance in Perceived Code Comprehensibility. Bootstrapped heterotrait-monotrait ratio of correlations (HTMT) are all below zero, bootstrapped loadings and weights are all approximately 1.00, all variance inflation factors (VIF) are 1.00, and all reliability coefficients are 1.00. All this is expected with single item constructs that are assumed to be fully independent.

Table 5.5.: Estimated and bootstrapped path coefficients with 95% CI, model explanatory power expressed as R^2 and adjusted R^2 . PCC = Perceived Code Comprehensibility. SD = standard deviation. * = $p \leq .10$, ** = $p \leq .01$, *** = $p \leq .001$.

	PCC	Bootstrap Mean (SD)	T Stat.	95% CI
H1	0.165***	0.169 (0.049)	3.357	[0.071, 0.265]
H2	0.418***	0.418 (0.063)	6.673	[0.293, 0.541]
H3	0.033	0.031 (0.058)	0.564	[-0.08, 0.148]
H12	0.076	0.074 (0.055)	1.384	[-0.034, 0.181]
H13	-0.013	-0.017 (0.058)	-0.228	[-0.124, 0.096]
R^2	0.223			
Adj R^2	0.208			

5.3.4. Discussion

We can answer the research question whether specific information available in advance about a code snippet influences developers in their subjective assessment of code’s comprehensibility as follows: The anchoring effect is, once again, significant. Information about the expected complexity of a code snippet to be understood influences developers in their supposedly independent code evaluations. Students and professionals are equally affected, which is in line with the investigation of Wyrich et al. [WPGW21] who measured programming experiences instead of role, and found programming experience to likely not play a role in anchoring. The finding is further in line with the body of literature on the anchoring effect, which states that the anchoring effect is not limited to laymen or those inexperienced in an activity [FB11; TK74].

The choice of code snippet also has a significant impact on perceived code comprehensibility. While this is not too surprising, nor is it the central point of this work, it is still good to have data points that show that the choice of code snippets for code comprehension studies should not be underestimated. Studies seeking to ensure that code snippets of different tasks are of comparable difficulty should invest effort in, e.g., a pilot study to evaluate

appropriate snippets.

What is interesting for the characterization of the observed anchoring effect, however, is that the complexity of a code snippet did not have an influence on the strength of the anchoring effect (H12). Based on our results, we suspect that a snippet must be complex enough so that PCC ratings do not concentrate too much on the lower end of the scale. Apart from that, the actual complexity of a snippet does not seem too important for anchoring.

5.3.4.1. Implications

We know, regarding the anchoring effect, that the specific anchor can take many forms and does not necessarily always have anything to do with the situation being assessed [FB11; TK74]. When designing and conducting code comprehension studies, researchers have a lot of freedom and just as much potential to inadvertently set anchors. In the introduction to [Section 5.3](#), we described a scenario in which the instructor mentioned, to encourage the participant, that the code snippets to be assessed were not too difficult to understand. Similarly, however, even in an online experiment without a human instructor, many examples of potential anchors can be found.

For example, explicitly mentioning in a study description that it is a study for novice programmers could lead to code snippets being rated as easier to understand. Communicated time limits for the code comprehension tasks can convey to participants how complex the tasks are supposed to be. Presenting a code comprehensibility metric to developers will anchor them in their own judgments when asking them whether they agree with it or how they would assess the code instead (i.e., a conceivable study design for validating a metric). For the latter example, we and the study of Wyrich et al. [WPGW21] have demonstrated the anchoring effect empirically.

One can easily find further examples in which study participants would be anchored. We see great research potential to empirically investigate these scenarios and to find solutions for affected studies, which for example cannot disregard subjective evaluations because they are relevant for their research intentions. Yet, for all study designs that allow to dispense with subjective

assessments, we recommend more reliable and objective measurements to draw conclusions about how well a developer has understood code. Corresponding tasks and measures are available for this purpose (see [Chapter 3](#) and, e.g., [[Fak18](#); [Fei21](#); [OBMC20](#)]).

5.3.4.2. Limitations

With our design, we were able to overcome a number of limitations discussed in the work by Wyrich et al. [[WPGW21](#)]. Our experiment had a much larger and more heterogeneous sample of developers, a less prominent presentation of the anchor, and we had a control group that allowed us to measure the perceived code comprehensibility for participants who were not explicitly anchored. Still, the results of our study should be seen in the light of some limitations.

A confounding factor for the assessment of the code snippets' understandability might be diverse understandings of what constitutes comprehensible code between the participants. We discussed, while designing the study, whether to provide a definition of understandability at the beginning of the survey. However, since there exists neither an agreement in the literature nor does it seem realistic that developers all share the same view on understandability, we decided against it. On the one hand, it might be possible that participants in one group share similar views on what constitutes understandable code, while participants in other groups differ. On the other hand, the randomized assignment of participants to scenario and code snippet should have mitigated this threat.

Regarding the generalizability of our results, we would like to emphasize that the scope clearly lies on the evaluation of individual code snippets and participants applied a comprehension process commonly referred to as bottom-up code comprehension (see [Chapter 2](#)). We see much value in reproductions of our experiment with larger software systems to be assessed.

We conducted our experiment remotely to make the study accessible to as many developers as possible and to minimize contacts due to the pandemic. The context in which the participants took part in the study could therefore

not be controlled, which could have caused individual participants to be distracted during the conduct of the study or not to complete the study conscientiously. We do not know what influence a potential distraction can have on a code evaluation, but at least we think that the activity can be resumed after an interruption. Participants who spent less than 20 seconds viewing and rating the code snippet's understandability were excluded from the analysis (average time for this task was around four minutes; a total of 10 participants were excluded based on this criterion).

Apart from this, running the experiment remotely worked well, and we are pleased to have recruited 50 software professionals as participants in addition to students. We have previously disclosed that this experiment group included both software professionals and six participants who identified themselves as researchers. This could be seen as a threat to construct validity, since software development experience might be less pronounced among software engineering researchers than among full-time software engineering professionals.

We assume that researchers in computer science, who hold either a related M.Sc. or a PhD, have comparable experience compared to software professionals to be able to deal with our tasks. Yet, we repeated the statistical analysis without the six researchers, and there was no change in the significance of the results. The path coefficients for the significant hypotheses would be $\beta_{H_1} = 0.161$ and $\beta_{H_2} = 0.408$ (change ≤ 0.10).

Further, when designing the experiment, we took care to minimize the time required for participation to achieve a correspondingly high response rate. This is the practical reason why we did not measure actual code comprehension with additional tasks. Furthermore, Wyrich et al. [WPGW21] found evidence for a strong anchoring effect in perceived code comprehensibility ratings, but also that actual code comprehension was not affected by the anchoring effect. We assume that by our random assignment of a sample six times larger than that of Wyrich et al. [WPGW21] the absence of effect anchoring/actual understanding still holds.

As anticipated in [Section 5.3.2.5](#), we make use of PLS-SEM in ways that reduce it to a simpler regression modelling tool with single-item constructs.

That is, we use PLS-SEM to understand the relationship between variables rather than typical reflective and/or formative constructs in structural and measurement models. As disclosed in the same section, we were driven to this choice because we preferred not to rely on robustness against assumption violations of other analysis and regression tools. PLS-SEM has just a few assumptions that our data did not violate, and it suited our needs.

Furthermore, we tested our results with an alternative model specification and report coherent results. We fit an equivalent multilevel mixed effects model with *lme4 1.1-27.1* [BMBW15]. Relying on its robustness against non-randomness of residuals, which is the case of our data, we find that 1. the same path coefficients that are found as significant in our PLS-SEM are also significant in the multilevel mixed effects model, 2. the model provides comparable explanatory power ($R^2 = .22$ and adjusted $R^2 = .22$). We are thus confident in our chosen analysis tools and its results. For elaborating on the limitations, we opted for multilevel mixed effects models based on Clark's report [Cla16] on their positive comparability with SEM. We still opted not to use multilevel mixed effects models for our main analysis because of cautiousness: the residuals are not randomly distributed.

The other related issue lies in using single item scales to represent our underlying constructs. PLS-SEM allows for single item constructs, and it is also used this way in information systems research [Pet13]. This use is, however, encouraged only when no other alternatives are available since psychometric properties of single item constructs are subject of debate of providing low psychometric validity and low reliability [Pet13; RSS12]. We argue that we fall within the recommendation of employing PLS-SEM for single item constructs. First, our independent variables are experimental groups represented by either two or three ordinal or categorical options. Furthermore, they provide highly observable validity and reliability (e.g., `role = student` indeed represents students, and `scenario = easy` really represents the easy scenario). Only the dependent variable, Perceived Code Comprehensibility, is operationalized by a single item, and the results might indeed suffer from low psychometric validity and reliability.

Finally, the Perceived Code Comprehensibility is a variable whose scale

we adopted by reproducing [Wyrich et al.](#)'s study [WPGW21]. This was a deliberate choice to be able to compare results and build on their study. A psychometrically validated multi-item scale to assess the Perceived Code Comprehensibility would allow for a richer and better explanation for the variance therein and offer (more) valid and reliable interpretation of its values. Such a scale, alas, does not exist, to the best of our knowledge. We report on this issue that is shared by most, if not all, studies that investigate Perceived Code Comprehensibility and call for future research on more robust ways to assess it.

5.4. Conclusion

In summary, the anchoring effect is present in software engineering and its investigation follows not only current efforts to debias software engineering practice eventually [CNA+20; MST+18; Ral11], but also to identify confounding factors in the context of scientific studies and thus enable the design of valid studies.

We have shown in two studies how subjective code comprehension assessments can be influenced by contextual factors. Because such contextual factors can take many forms and are difficult to control, we conclude that code comprehension assessments based on subjective self-assessments are less reliable than alternative methods for measuring code comprehension (see [Chapter 3](#)).

In general, each of the two presented studies provides valuable insights, or evidence, on a specific question, which can help in the design of future code comprehension experiments. The decision on how to measure code comprehension in one's own study can thus be made evidence-informed by these and comparable studies. Related to this point, we consider two thoughts to be relevant for the following chapter of the thesis.

First, we have made the observation that a second study, even one that differs only minimally in design from an initial study on a topic, can provide many new insights and additional certainty. At first glance, this observation

seems quite trivial; after all, the way science works is to conduct as many primary studies as possible on the same research question, therefore to test the validity of previous studies, and then derive recommendations for action based on meta-studies. In reality, however, it is easy to find papers whose introductory motivation brags that it is about *the first study* on a particular topic, as if that makes the study more valuable. Also, the introduction of replication tracks at major software engineering conferences is only a recent development, and one that shows that there was some urge for change. The International Conference on Program Comprehension offered such a track for the first time in 2019 [ICP19]. A change in thinking is slowly taking place, at least if one looks optimistic at the development of our and related research fields, but it will take time and active education on the importance of replication and reproduction studies to counteract their lower status [BGNT20]. We may even eventually move beyond the current phase of offering separate tracks for papers that confirm or refute existing knowledge, but consider evidence to be valuable regardless of whether someone has already studied something similar on the same topic.

Second, a third study might provide even more new insights. Many more studies should be conducted on the influence of the anchoring effect on code comprehension, for example, to find out whether there are scenarios in which the anchoring effect is not present or is less present, or, for example, to find out by what means we can mitigate or control for the effect. Any further studies on similar research questions will complement our overall picture. Furthermore, any variation in a study design and any diversity of author teams counteracts the inherent bias in the design of a study and the interpretation of its results (see [Section 1.3.2](#) for a philosophical discussion of study design diversity). In the best case, however, this also means that we will eventually reach the point where there are many studies on the same research question. Staying with the anchoring example and the context of the thesis, these could all be studies that investigate whether it is better to avoid using subjective comprehension measures when designing one's own code comprehension experiment. This will lead to many pieces of evidence, some of which may be contradictory. Even with the best efforts to make

evidence-based study design decisions, it can then become difficult to keep track of the literature, especially considering that there are numerous other decisions to be made in addition to choosing a comprehension measure.

The following chapter will address this particular issue of synthesizing evidence on a research question to support researchers in the design of their primary studies.

CHAPTER
6

EVIDENCE PROFILES FOR VALIDITY THREATS IN CODE COMPREHENSION EXPERIMENTS

Up to this point, we have come a long way toward our goal of evidence-based study design decisions. This includes presenting an overview of findings from the code comprehension literature in [Chapter 2](#), discussing the various options for design decisions in code comprehension experiments and the challenges of making reasonable choices in [Chapter 3](#), as well as providing our own evidence for evaluating design decisions and potential factors influencing code comprehension in [Chapters 4 and 5](#).

This chapter will contribute most to answering our main research question, how available evidence can be used pragmatically to evaluate the validity of code comprehension experiments. We will take an exemplary look at the

three most frequently discussed threats to validity in code comprehension experiments, or more precisely, what the evidence of their influence actually looks like and thus provide an answer to **RQ4**. Our approach for answering RQ4 is at the same time a proposal for a concrete methodology to synthesize evidence in secondary studies to facilitate the work of authors of primary studies. The contents of this chapter extend our publication [[MWGW23](#)].

6.1. Context and Goals

Searching for clues, gathering evidence, and reviewing case files are all techniques used by criminal investigators to draw sound conclusions and avoid wrongful convictions. Similarly, medicine has a long tradition of evidence-based practice, in which administering a treatment without evidence of its efficacy is considered malpractice. In software engineering (SE), study designs that are based on evidence enable sound methodologies, including the mitigation of validity threats. The SE body of knowledge is, however, missing out on evidence of validity threats.

In 2013, Wohlin [[Woh13](#)] published a paper on evidence profiles for software engineering research and practice. Wohlin motivates the work with the increasing importance of evidence-based software engineering (EBSE). For example, critical decisions, such as the introduction of a new tool that could affect software quality or developer productivity, should be based on scientific evidence. Wohlin proposes a model by which evidence from different studies could be evaluated in a manner similar to criminal law investigations. Practical conclusions could be drawn by putting the individual pieces together in an evidence profile.

Researchers have long been in the dark about the consequences of design decisions in code comprehension experiments. In the corresponding papers, researchers regularly discuss and speculate about the threats to validity of their experiments. Two meta-studies have categorized these threats, coming up with more than 50 different threat categories [[SS15](#); [WBW23](#)]. Moreover, papers that do not discuss threats to validity in code comprehension exper-

iments are rather the exception today [SKSL17a; WBW23]. At the same time, hardly anyone is sure about the actual extent of the discussed threats, and almost no paper cites evidence on the assumed threats [WBW23]. This makes it difficult to evaluate study designs in an evidence-based manner. Researchers have to decide which of the more than 50 potential threat categories do, in fact, threaten the validity of a study design, execution, and interpretation, and to which extent they should be disclosed and elaborated on.

In the following section, we apply Wohlin’s methodology of evidence profiles [Woh13]. We echo our own call to provide more evidence in the design of empirical program comprehension studies (see Section 3.3, action item 4). To that end, we examined the threats to validity in code comprehension experiments to collect evidence of their existence, to understand the context and nature in which they occur, and to ultimately assist researchers in designing controlled experiments with high validity. Specifically, we extracted the threats to validity reported in 95 code comprehension experiments through thematic synthesis. Then, focusing on the three most frequently mentioned threat categories, we collected evidence that contradicted or supported the influence of the threat, using systematic literature searches and snowballing. Finally, we individually scored the evidence that passed our filtering criteria to create an evidence profile, serving as an overview of the evidence for each of the three threat categories.

6.2. A Study of Evidence Profiles for Validity Threats

6.2.1. Background and Related Work

Medicine has a long tradition of evidence-based practice, in which administering a treatment without evidence of its efficacy is considered malpractice. Sackett et al. [SRG+96] describe evidence-based medicine as “the conscientious, explicit, and judicious use of current best evidence in making decisions about the care of individual patients”.

In software engineering, attempts have been made to establish similar

approaches to help practitioners make informed decisions in their daily work [DKJ05]. Proponents of evidence-based methods emphasize the importance of using evidence when adopting new technologies and when understanding and identifying problems in existing development processes. Making uninformed decisions may lead practitioners to favor ineffective solutions over better alternatives, resulting in financial losses or even harm to humans.

Since the first calls for evidence-based software engineering in the early 2000s [DKJ05; KDJ04], investigative methods from EBSE found widespread usage. There has been a significant increase in the number of secondary software engineering studies [BB22; KPB+10] and educators are actively incorporating EBSE in their university curricula [JDK05; OC09; RB08]. However, recent discussions highlight difficulties in the application of evidence gained from primary and secondary research [HCKH14; SS13]. In response, more tools and structured approaches are being introduced in an attempt to address the slow transfer of knowledge from research to practice (e.g., evidence profiles [Woh13], evidence briefings [CPVS16], and rapid reviews [CPS20]).

While evidence-based approaches may assist practitioners in making decisions, Kitchenham et al. [KDJ04] also describe how these approaches place additional requirements on researchers when developing experimental protocols. Ideally, researchers can maximize the range of application of a study's results while minimizing potential threats to validity.

We consider *evidence* to be “the available body of empirical knowledge indicating whether a belief or proposition is true or valid” [Ste10]. Further, we consider *validity* to be the degree to which we can trust the results of an empirical study [KBB15]. In our study, a threat to validity refers to deliberate design decisions and uncontrolled extraneous factors that may impair the validity of experimental results. These two definitions of the terms have guided us in this specific study. A philosophical discussion of the concepts of evidence and validity can be found in [Section 1.3](#).

When readers are explicitly informed about validity threats of a study, they can, e.g., better assess the context in which the study results may be

applied. Consequently, they are empowered to understand what difficulties may arise when they attempt to replicate the study design or plan a similar study of their own.

There have been several meta-studies to summarize and categorize common threats to validity in software engineering research. Petersen and Gencel [PG13] compared validity threats with different worldviews. They assumed that researchers have a subconscious tendency to choose methods based on their worldview. Likewise, Devanbu et al. [DZB16] found in a case study that programmers tend to hold strong beliefs based on personal experience rather than empirical evidence. The main implication of these studies is that researchers and practitioners make biased decisions based on their intuition, which is at odds with the main goals of evidence-based methods. Rather, they should address the discrepancy between the evidence and their perceptions and reconsider their decisions accordingly. Petersen and Gencel [PG13] suggest that the literature needs to be further analyzed regarding the threats and mitigation techniques mentioned therein and that inquiries need to be made into what worldviews dominate in the various sub-disciplines of software engineering.

Biffel et al. [BKE+14] followed this suggestion and created a knowledge base of threats to validity in software engineering experiments to assist researchers in planning their studies. They found that only a small fraction of validity threats are reported in most studies and that, instead, the vast majority of threats are too specific to be generalized outside the particular research area they occur in. These findings highlight the complexity of managing threats to validity, as even switching between different sub-disciplines of software engineering introduces a whole new set of potential threats that must be considered. They conclude that there is a need for an overview of threats to validity as they are reported in specific areas of software engineering research.

Managing threats to validity is particularly difficult in code comprehension experiments, where researchers seek to uncover the underlying processes of how developers understand code and evaluate ways to support that comprehension process scientifically [RC97]. This complexity is reflected in

the wide variety of different methodologies researchers use to address the potential validity threats in their experiments [WBW23].

For example, Siegmund and Schumann [SS15] surveyed the literature to obtain information on confounding parameters in studies of program comprehension. The main insights they gained were that each paper reported only a small subset of all confounding factors, and that researchers used different methods to mitigate the same factors. They recommend that other researchers include the identification of confounding parameters in their experimental design and explicitly report the relevant parameters and how their influence is controlled.

We previously built on the findings of Siegmund and Schumann by analyzing the threats to validity of 95 source code comprehension experiments and how they were reported (see Chapter 3). There we noted that, currently, researchers tend to rely on intuition when designing their experiments because of how difficult it is to reliably measure a person's understanding of code. Researchers have to deal with potential threats to validity from over 50 categories, and sometimes question whether their measure of understanding is adequate at all.

These meta-studies illustrate the complexity of designing valid code comprehension experiments, considering potential confounding factors and other threats to validity [SS15; WBW23]. Rather than choosing methods based on intuition, researchers should make informed decisions based on empirical evidence. Previous research has outlined a need for common knowledge bases of validity threats and collections of evidence backing up both the existence of threats and the effectiveness of their mitigation techniques. The first step in solving these issues is to identify where evidence is lacking.

Surveying the evidence landscape on any particular issue can be quite daunting. For this purpose, Wohlin [Woh13] proposes the creation of evidence profiles to gain an overview of both the amount and the direction of evidence. By scoring each individual piece of evidence in the context of an explicit research question, researchers are assisted in identifying sufficient or missing evidence. This approach shares similarities with other meta-studies, such as meta-analyses, in that it aims to summarize and combine the results

of multiple studies. Evidence profiles differ in their method of synthesis, however, as meta-analyses take a quantitative approach in composing effect sizes, whereas evidence profiles use multiple reviewers to qualitatively evaluate the studies by scoring them.

6.2.2. Methodology

We first investigate which threats to validity are most frequently discussed in the program comprehension literature (Section 6.2.2.1). The frequency of discussion, however, should only be considered as an indicator of what researchers are most often concerned about, not of the threats' evidence. Therefore, in a second step, we will examine the scientific evidence for the most frequently discussed threats (Sections 6.2.2.2 and 6.2.2.3).

Figure 6.1 provides a schematic overview of the research methodology. The research questions that guided us in our endeavor are as follows:

- RQ4.1** Which validity threat categories are most often discussed in primary studies of code comprehension?
- RQ4.2** What are the evidence profiles for the three most commonly reported threats to validity in code comprehension studies?



Figure 6.1.: Schematic representation of the research methodology.

6.2.2.1. Scoping the Relevant Threats

To answer **RQ4.1**, we surveyed existing studies of code comprehension and identified which threats were most frequently mentioned by researchers. In particular, we extracted the threats to validity reported in 95 code com-

prehension experiments found in the systematic mapping study that we reported in [Chapter 3](#) (due to different author teams, we reference the study in this chapter as Wyrich et al. [[WBW23](#)]). The search protocol included empirical studies of bottom-up code comprehension with human participants, published in a peer-reviewed journal, conference, or workshop before 2020. While Wyrich et al. [[WBW23](#)] reviewed the threats to validity to some extent, we opted to repeat the coding and categorization activities to enable a more fine-grained classification of threats with the explicit goal of gaining insights into the reporting of validity threats as a whole. Moreover, we performed this bottom-up coding activity to stay closer to the data, building a foundation for the subsequent evidence searches.

We examined the list of primary papers, categorizing and summarizing each threat to validity reported in the full text. Specifically, we adopted a thematic synthesis [[CD11](#)] approach to identify the individual threats to validity and, if given, the mitigation techniques mentioned by the study. For this, we extracted relevant text areas using inductive coding [[CS08](#)] to find and describe the passage. In inductive coding, codes are formulated during the review process as the corresponding concepts become evident. Throughout the coding process, these codes are refined and reapplied, improving their quality through iteration. Upon the completion of code assignment, we categorized threat codes into high-level categories and themes. Further, in some cases, papers already contained evidence of a threat in the form of references to other studies and consequently, those were also documented. Finally, we count the number of papers that mention a threat for each threat category, so we can see which threat categories are most often discussed.

In summary, the review process consisted of the following individual steps:

1. Extraction of relevant text passages on threats to validity, mitigation techniques, and evidence.
2. Inductive coding, in which each threat is assigned a descriptive code.
3. Categorization of threat codes and composition into higher-order themes.
4. Counting the number of papers mentioning each threat category.

6.2.2.2. Evidence Collection

At this point, we had a list of threat categories, how frequently they were reported, and in some cases, a pool of starting evidence. Due to the large number of different threat categories, we could not collect evidence for each individual one. Instead, we focused on the three categories that were mentioned most frequently. These threat categories were the level of programming experience of participants, the length of the programs used in the experiment, and the measures that were used as a proxy for the concept of comprehension.

To answer **RQ4.2**, we conducted systematic searches for each of the three threat categories, collecting evidence on their influence. The steps described in this section and in [Section 6.2.2.3](#) were repeated for each individual category.

Search Protocol. Our search protocol utilized four different sources to search for potentially relevant papers. We describe each of these sources and list the filtering criteria applied to the literature found within them. Furthermore, we describe how we used snowballing [[Woh14a](#)] as a technique to further extend the search. Backward snowballing in this context means including the reference list of a paper, while forward snowballing means including papers that cite the paper in question.

a) Primary Papers: The 95 papers with primary research on program comprehension may already examine the threat in question as part of their study. Consequently, we evaluated them as potential evidence. Forward snowballing was less likely to yield relevant results, as investigating the threat in question was not the main focus of these papers. Unless the threat was the main subject of a primary study, other studies examining the threat are unlikely to cite the primary study in the context of discussing the threat. Backward snowballing for this set of papers was not required because the evidence cited in the primary papers was already a separate source, as we describe in the next paragraph.

b) Evidence Cited in the Primary Papers: As part of the analysis described

in [Section 6.2.2.1](#), we identified and documented all references in the primary papers that were cited to support assertions made about a threat to validity. We analyzed and filtered this list of references in the same manner as the primary papers. The evidence from this source did not necessarily focus on the threat as their primary object of research. Forward snowballing was therefore not used, as it was less likely to yield relevant results. However, we performed backward snowballing, as the papers may refer to similar evidence when comparing their results with other works.

c) Evidence Found Through the Title Search: To further enrich the dataset with research from sources independent of the primary papers, we also conducted additional systematic searches. While the literature search by Wyrich et al. [[WBW23](#)] already captured code comprehension experiments up to 2020, we were able to extend the range of our search to additionally include literature published between 2020 and 2022. Furthermore, we focused this search on studies that mention the particular threat to validity in their title using the following search strings in Google Scholar:

- **Programming Experience** - allintitle: (experience OR novice OR expert) (code OR software OR program) (understandability OR comprehension OR comprehensibility OR readability OR analyzability OR ‘cognitive load’)
- **Program Length** - allintitle: (size OR length OR short OR long OR LOC OR "lines of code") (code OR software OR program) (understandability OR comprehension OR comprehensibility OR readability OR analyzability OR ‘cognitive load’)
- **Comprehension Measures** - allintitle: (measure OR measures OR measurement) (code OR software OR program) (understandability OR comprehension OR comprehensibility OR readability OR analyzability OR ‘cognitive load’)

Each search string was composed by combining terms describing a threat (e.g. experience, novice, expert) with terms related to program comprehension experiments (e.g. code, understandability, comprehension). Both backward and forward snowballing was considered valuable here, as the studies found

in the title search were likely to have the threat in question as the main subject of their research.

d) Evidence Found Through Snowballing Snowballing is helpful to further emphasize relevant work by including their reference list as an additional search source. By exploiting clusters of related research, snowballing can identify relevant literature with high precision, but tends to miss some papers [BWP15]. Therefore, we use a hybrid approach to complement disadvantages of snowballing with a database search and vice versa. Backward snowballing was applied to the evidence cited in the primary papers, while backward and forward snowballing were used on the evidence from the title search. In all three cases, we did one iteration of snowballing.

All papers found in the sources a) to d) were filtered according to the following four inclusion criteria and only included if all of them were fulfilled:

- I1** The paper reports a primary study measuring program comprehension.
- I2** The paper reports an analysis of the threat in question.
- I3** The paper is published in a peer-reviewed journal or conference proceeding.
- I4** The paper's full text is available in English.

6.2.2.3. Evidence Profiles

After collecting all available evidence of a particular threat, we evaluated the evidence itself. To this end, we employed the evidence profile proposed by Wohlin [Woh13]. This profile is a model for evaluating evidence based on criminal law. Each piece of evidence is judged individually and rated with a corresponding level of strength, from 1 (lowest) to 5 (highest). This way, evidence strength is represented by an ordinal scale with no zero. In addition, the profile distinguishes between positive and negative evidence, with positive evidence supporting the theory in question and negative evidence contradicting it. It is important to stress that evidence of low quality is not synonymous with negative evidence. Negative evidence can be of high quality, but it opposes the notion that a threat has an influence on the

validity of a primary study. For example, negative evidence with a score of -5 has the same evidence strength as positive evidence with a score of 5. Thus, a score close to zero indicates the strength of a piece of evidence is low, while a negative or positive score indicates the outcome of the study.

Table 6.1 describes the different levels of strength for evidence. Due to the flexible nature of the evidence profile, the descriptions do not match the ones provided by Wohlin word-for-word. Wohlin emphasizes that the evidence profile should be adapted to the context in which it is used. For example, quality aspects such as *vested interest* and the *aging of evidence* do not play a major role when collecting evidence on threats to validity, unless the threat pertains to a specific technology or approach that may influence experiment results. By contrast, the methodological rigor captured in the *quality of evidence* as well as the *relevance of the evidence* is of utmost importance and strongly influenced the descriptions of the different levels of evidence strength.

Table 6.1.: Types of evidence according to the evidence profile. Higher values mean stronger evidence. **Positive values** support the threat, **negative values** contradict it.

Score	Level of Strength	Description
+5/-5	Strong evidence	Studies that focused on the threat in question as the main subject of their investigation or conducted an in-depth analysis of the threat as part of their overall approach and show significant results. Or: systematic reviews that examined the threat in question and provided a conclusion.
+4/-4	Evidence	Studies that did not have the threat as their main focus but still included it in their analysis. These studies may have more uncontrolled confounding factors, which is why they should be considered separate from strong evidence.
+3/-3	Circumstantial evidence	Similar to evidence, studies that are considered circumstantial evidence did not have the threat as the main focus of their study. Furthermore, they showed additional methodological shortcomings that reduced the reliability of the study and decreased its strength as evidence in our evaluation.
+2/-2	Third-party claim	Studies that made claims about the threat in question but only provided a slim level of empirical backing for said claim. They may have deferred to other sources of information or given general impressions about the influence of the threat in their study, but provided no dedicated statistical analysis of their own to support their claims.
+1/-1	First- or second-party claim	Studies that made a claim about the threat in question but did not provide any empirical backing. This could have been, for example, references to “common knowledge” or speculation.

The placement of a study on a particular level is based on its adherence to the factors described in the level description, as well as on the previously mentioned quality aspects. Therefore, a study may be placed in a lower category if it has significant shortcomings regarding any of the quality aspects. The concrete definitions of what each level of strength means were established before the evaluation process began. Because the evaluation of evidence is a largely subjective process, we involved two researchers in this step. Once both researchers had scored a piece of evidence, they compared their scores and discussed possible disagreements. In these discussions, each researcher explained their reasoning for their score, and, together, they decided on a single final score. Where the initial scores were identical, the final score was set to the same value without discussion. We documented the score of each study and the motivation behind its placement, which can be used to paint an overall picture of the evidence landscape for each threat. We documented scoring and agreement in our replication package [MWGW22]. Based on each evidence profile, we provide conclusive recommendations to researchers.

6.2.3. Results

We first present the results of extracting and categorizing validity threats from 95 primary studies. This provides us with the answer to RQ4.1, which validity threats are most frequently discussed in primary studies on code comprehension. We then answer RQ4.2 by presenting the evidence profiles for the three most frequently reported threats to validity in code comprehension experiments.

6.2.3.1. Validity Threats in Code Comprehension Experiments

Among the 95 papers, 81 (85 %) mentioned at least one threat to validity, with 45 (47 %) reporting them in a dedicated section and 33 (35 %) differentiating between different types of validity, such as internal or external validity. The largest concentration of studies was published in the period

from 2012 to 2019, with a total of 63 of the 95 studies (66 %), while only covering 7 years (18 %) of the entire 40-year span. The first study to include a dedicated ‘threats to validity’ section was published in 1997.

In total, we found 409 individual threat mentions. Out of the 409 threat mentions, 198 (48 %) included an explanation of a possible mitigation technique, but only 31 (8 %) were reported with supporting evidence. The 31 references to supporting evidence were found in 20 out of 95 studies (21 %). The 409 threat mentions were then assigned 215 unique threat codes, which captured the different nuances of how a threat was mentioned in a study. Multiple threat mentions could receive the same code, which is why the number of unique codes is lower than the number of total threat mentions. To better analyze related threat codes, we additionally categorized them, which resulted in 81 unique threat categories. For example, the threat category *Programming Experience* included threat codes such as “Missing diversity in participants’ programming experience leads to limited generalizability” and the opposite code “Diversity in participants’ programming experience confounds treatment effects.” The two individual threat codes differed in nuance, but they both emphasized the importance of programming experience as a threat to validity. When we counted the number of threat mentions in the prioritization, each occurrence of either threat code would increase the category’s rank.

Table 6.2 highlights the threat categories with more than five reported threat mentions and shows the themes to which the categories were assigned. Both categories and themes are presented with the total number of threat mentions. All mentions from categories with less than five mentions are counted in ‘Other Categories’ under their respective themes. ‘Theme: Other’ contains mentions of threat codes that did not fit into any given theme. The three most common threat categories (i.e., *Programming Experience*, *Program Length*, and *Comprehension Measures*) are highlighted in blue. Overall, most threats were related to the characteristics of the code snippets, the individual factors of the participants, or general threats in experimentation.

Table 6.2.: Number of threat mentions per theme and category for categories with more than five threat mentions

Theme and Category	Count
Theme: Code Snippets	112
Program Length	26
Complexity	16
Code Selection	13
Programming Language	9
Synthetic Samples	9
Familiarity	6
Other Categories	33
Theme: Participant Factors	101
Programming Experience	44
Number of Participants	16
Programming Skills	10
Other Categories	31
Theme: Experimentation	89
Learning Effect	18
Lab Experiment	11
Fatigue	9
Code Presentation	7
Cheating	6
Other Categories	38
Theme: Measurement	67
Comprehension Measures	22
Eye-Tracking	20
Instrumentation	9
Other Categories	16
Theme: Comprehension Tasks	21
Type of Comprehension Task	7
Task Difficulty	6
Other Categories	8
Theme: Data Analysis	14
Statistics	10
Other Categories	4
Theme: Other	5
Total	409

RQ4.1: Main Findings

- 85 % of code comprehension experiments report at least one threat to validity.
- Only 8 % of threat mentions are supported with referenced evidence.
- The three most commonly reported threat categories are **programming experience**, **program length**, and **comprehension measures**.

6.2.3.2. Evidence Profiles

We selected the three threat categories that were most frequently reported and collected evidence on their influence on study validity. These categories are programming experience, program length, and comprehension measures. We provide the complete evidence lists, including rationales for the placement of studies in the different evidence categories of an evidence profile, in our replication package [MWGW22].

Programming Experience. In this analysis, we examine the effect a participant's programming experience has on their code understanding in comprehension experiments. We collected evidence in the form of studies that measured the programming experience of participants and determined whether it had a significant influence on program comprehension. Table 6.3 shows how many papers we found in each step of the evidence collection and how many we excluded because they did not meet the inclusion criteria. Overall, most evidence was found in the primary papers, followed by snowballing. We excluded 12 of 13 (92 %) documents cited in the references of the primary papers, as they did not meet the filtering criteria. The reasons for exclusion varied and are described in more detail in Section 6.2.4.

After filtering, 60 papers remained and were evaluated as potential pieces of evidence to be used in the evidence profile. In this evaluation, 11 additional papers were discarded as they did not meet the criteria to be considered

Table 6.3.: Overview of the evidence analyzed and filtered in each step of the evidence collection for programming experience

Source	Analyzed	Excluded	Final
a) Primary Papers	95	-52	43
b) Cited in Primary Papers	13	-12	1
c) Title Search	54	-52	2
d) Snowballing	276	-262	14
Evidence Profile	60	-11	49
Total	438	-389	49

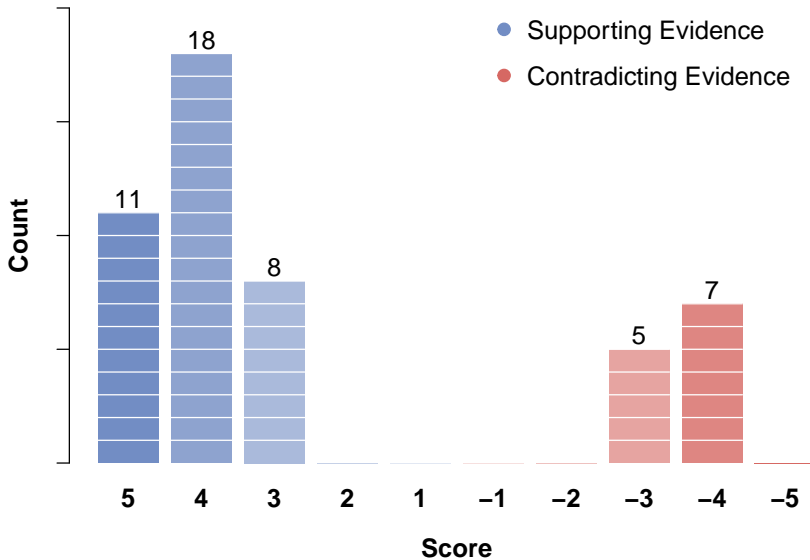


Figure 6.2.: Evidence Profile 1: Programming experience influences code comprehension

evidence. For example, one paper was excluded because it was not peer-reviewed and multiple papers were excluded because they either did not measure programming experience or they did not use the gathered experience data in their analysis. [Figure 6.2](#) presents the final evidence profile. The result largely indicates that programming experience influences code comprehension. In total, 37 (76 %) pieces of evidence were rated as positive evidence and 12 (24 %) were rated as negative evidence. Furthermore, there were 11 pieces of strong positive evidence and no strong negative evidence.

Program Length. In this analysis, we examine how the length of a program affects program comprehension. We gather evidence in the form of studies that measure the length of a program and examine its impact on program comprehension. [Table 6.4](#) shows how many papers we found in each step of the evidence collection and how many we excluded because they did not meet the inclusion criteria. The only evidence was found in the list of primary papers. All documents cited in the references of the primary papers were excluded as they did not meet the filtering criteria. Moreover, as we found no relevant papers in b) and c), no snowballing was performed.

After filtering, 17 papers remained and were evaluated as potential pieces of evidence using the evidence profile. In this evaluation, 4 more papers were discarded as they did not meet the criteria to be considered evidence. The final evidence profile is shown in [Figure 6.3](#). We found conflicting results regarding the influence of program length on program comprehension. In total, 6 (46 %) pieces of evidence were rated as positive evidence and 7 (54 %) were rated as negative evidence. Furthermore, there were 2 pieces of strong positive evidence and no strong negative evidence.

Comprehension Measures. In this analysis, we investigate whether common comprehension measures are associated with distinct aspects of comprehension. We gather evidence in the form of comparative studies that analyze correlations between commonly used comprehension measures. Positive evidence describes that if different comprehension measures were used in a

Table 6.4.: Overview of the evidence analyzed and filtered in each step of the evidence collection for program length

Source	Analyzed	Excluded	Final
a) Primary Papers	95	-78	17
b) Cited in Primary Papers	3	-3	0
c) Title Search	29	-29	0
d) Snowballing	0	0	0
Evidence Profile	17	-4	13
Total	127	-114	13

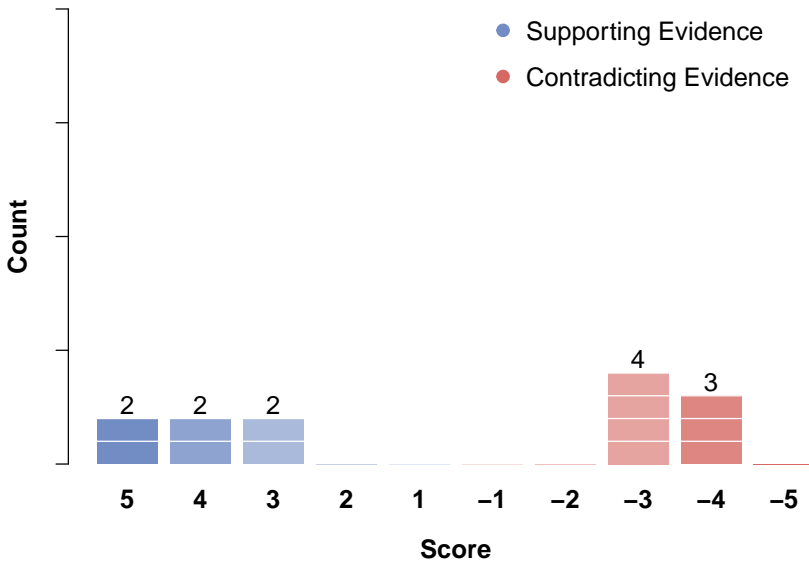


Figure 6.3.: Evidence Profile 2: Program length influences code comprehension

primary study, one would arrive at a different conclusion. Table 6.5 shows how many papers were found in each step of the evidence collection and how many were excluded because they did not meet the inclusion criteria. Overall, we found most evidence in the primary papers, with slightly less evidence found in both the title search and through snowballing. All documents cited in the references of the primary papers were excluded as they did not meet the filtering criteria.

After filtering, 12 papers remained and were evaluated as potential pieces of evidence using the evidence profile. In this evaluation, 5 more papers were discarded as they did not meet the criteria to be considered evidence. The final evidence profile is shown in Figure 6.4. Most evidence supported that the commonly used comprehension measures do measure distinct aspects of program comprehension and are not correlated. But overall, only a small amount of evidence was found. In total, 5 (71 %) pieces of evidence were rated as positive evidence and 2 (29 %) were rated as negative evidence. Furthermore, there were 3 pieces of strong positive evidence and no strong negative evidence.

Table 6.5.: Overview of the evidence analyzed and filtered in each step of the evidence collection for comprehension measures

Source	Analyzed	Excluded	Final
a) Primary Papers	95	-88	7
b) Cited in Primary Papers	3	-3	0
c) Title Search	53	-51	2
d) Snowballing	134	-131	3
Evidence Profile	12	-5	7
Total	285	-278	7

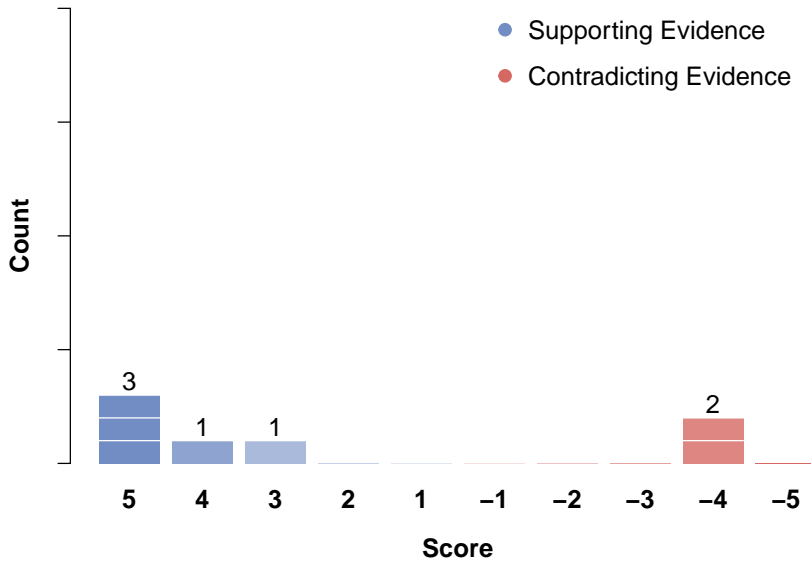


Figure 6.4.: Evidence Profile 3: Different comprehension measures do not correlate with each other

RQ4.2: Main Findings

- **Programming experience:**
37 positive and 12 negative evidence
- **Program length:**
6 positive and 7 negative evidence
- **Comprehension measures:**
5 positive and 2 negative evidence
- 94 % of cited evidence in the primary papers did not meet our criteria for evidence and was excluded.
- No evidence was categorized as strong negative evidence (-5).

6.2.4. Discussion

We will first consider the results of the evidence profiles on the three most frequently discussed threats to validity.

Overall, the **first evidence profile** confirms that, often, programming experience does influence the comprehension performance of programmers. In multiple cases, experienced programmers showed different comprehension behavior when compared to novices [Ade84; YTA19] and this difference could be measured in their performance [MLA+19; OB17; Wie85]. In contrast, however, we also found credible evidence contradicting those assertions. Twelve studies reached the conclusion that programming experience does not influence code comprehension. One explanation for this contradiction could be the specific contextual factors of each study. Siegmund et al. [SKL+14] found that depending on how programming experience is measured and operationalized, its predictive power varies. Moreover, some negative evidence still found correlations with very specific types of experience measures such as self-estimated Java knowledge [PSA+18] or correlations for only specific comprehension measures such as the number of eye fixations [JWAL21]. In other cases, the range of programming experience was quite limited, for example due to only including students as participants [HCM02; PSA+18]. These results imply that mentioning programming experience as potential threat alone is insufficient when publishing experiment results. Researchers should instead discuss the threat within the context of their study and discuss how and why they adapted procedures, measures, and artifacts to mitigate it.

The **second evidence profile** indicates that an influence of program length on code comprehension behavior or performance should not be assumed blindly in every context. We found conflicting evidence for this potential threat to validity. Ribeiro and Travassos [RT18b] discuss similarly conflicting evidence on program length in their study. Even after conducting a follow-up study, they were unable to reach a clear conclusion. Novices and experts had differing opinions regarding program length and its influence on comprehensibility and readability. They also mention that differences in the procedures

and tools used to measure lines of code may affect the comparability of results from different studies. While the overall number of samples is small, we can identify some patterns when comparing positive and negative evidence. All studies rated as positive evidence included students as participants, one of them included both students and experts. Every study that only included experts was rated negative. This pattern suggests that while program length may influence the comprehension performance of novices, experts appear to perform consistently, regardless of program length.

The results of the **third evidence profile** mostly support the notion that the commonly reported comprehension measures are not correlated. The three supporting pieces of evidence found no correlations between the time and accuracy of comprehension task performances, suggesting that they measure different effects, aspects, or dimensions of comprehension task difficulty [AWF18; BLMM09; Ise88]. However, both pieces of negative evidence found correlations between the time and accuracy of comprehension tasks [GG84; HZ86]. Furthermore, the two studies that compared physiological measures with other measures found no correlation with subjective ratings [YYZD21] and time and accuracy of comprehension tasks [FRM+20; YYZD21]. The significance of this finding becomes more apparent when one considers that of the 95 primary papers, more than one third (37) used only a single measure to assess the comprehension performance of participants. Depending on the study context, this may cause them to miss aspects of code understanding that would have been uncovered if they had analyzed more than one measure. If different proxy measures are associated with different aspects of program comprehension, this poses further difficulties for meta-studies of comprehension experiments. For example, two studies might obtain different results because they measured comprehension performance in different ways and not due to other intrinsic factors. Using the results from this work, we were unable to identify concrete patterns in the context factors which would suggest why comprehension measures correlate in some instances and not in others. A dedicated follow-up study, solely focused on comparing different comprehension measures, may shed more light on this question. For now, combining studies with different comprehension mea-

asures should be avoided until we better understand the way they measure different aspects of comprehension.

In comparison, we found much less evidence overall for the influence of program length and the comparability of comprehension measures than for the influence of programming experience. Even though program length and comprehension measures are the second and third most discussed threats to validity in code comprehension experiments, there are few studies examining their influence. With the little evidence we found, we nevertheless arrived at similar results to the first evidence search: The extent of all three validity threats is context-dependent. The influence of a validity threat varies for each individual study and must, as such, be interpreted in the light of the contextual factors surrounding it. When seeking patterns to explain the contradiction of evidence, we found that the way a threat impacts the result of a study can depend on how a confounding factor is measured, what the sample population is, and how the experiment is designed in general.

While this conclusion might sound obvious at first, we often find generic statements about a potential threat in primary studies. For example, a study might solely mention that their experiment sample consisted of students without further elaboration. Discussing a study characteristic this way, however, does not clarify why that design decision might constitute a threat to the validity of that specific study. This issue is problematic regardless of whether references are used to back up assumptions, as generic threats are used in place of more nuanced discussions that take the study-specific context factors into account. Evidence should be cited as an additional layer to further support the explanations. For example, Jbara and Feitelson [JF17] show how evidence from past studies can be used when discussing the threats to validity of their study.

Minor Observations

Besides this main finding, we also made some further observations. First, the studies used in the primary papers to support claims about threats to validity were, with one exception, almost entirely dismissed as evidence.

Studies were mostly excluded because they either did not relate to program comprehension, solely cited another mention of the threat, or they were not relevant at all. This pattern was consistent over all three evidence searches.

Second, the results of our review of how threats to validity are reported match the broader investigation by Wyrich et al. [WBW23]. In general, researchers in program comprehension tend to focus on threats relating to the code snippets and the study participants the most. The factors we investigated in our evidence search, programming experience, program length, and comprehension measures are among the most frequently mentioned threats in both studies. The terms used to describe these threats differ between the works. For example, the category named ‘comprehension measures’ can be found in the category ‘instrumentation’ in Wyrich et al. and ‘program length’ is named ‘program size’. Looking at the bigger picture, the results for program comprehension studies are in line with reporting in SE in general. Siegmund et al. [SSA15] found that in SE, 51 % (47 % in our study) of papers discussed threats to validity in a dedicated section and 23 % (35 % in our study) differentiated between different types of validity.

Third, we found that, overall, there was less evidence for less common threats to validity. This is in line with what would be expected intuitively, if fewer researchers deem that a threat poses a danger to a study’s validity then corollary, fewer will investigate whether that assumption holds. Meta-studies such as this one can highlight which threats are less frequently examined and provide directions for further experimentation.

Fourth, we observed that throughout the entire evidence collection process, not a single study was classified as a ‘-5’, which would represent strong negative evidence. While we do not have a conclusive reason for this, it is common for researchers to experience difficulties publishing negative results. Obtaining negative results can be disheartening, but publishing them is nonetheless crucial and provides value to the scientific community [Bor18; PCE17; Wei16].

6.2.4.1. Implications

The results presented in this study lead us to concrete suggestions for the larger research community. First, threats to validity are dependent on individual context factors of a study. Researchers should therefore explicitly discuss the influence of each threat within the context of their study. Merely listing off potential threats alone is not sufficient. Second, context discussions for validity threats should be supported by existing evidence, rather than relying on intuition or speculation. Meta-studies provide the appropriate evidence base for this endeavor, as they analyze the influence of a threat in multiple different study contexts.

While both aspects could be ensured at an appropriate point in the peer review process, we believe that the authors of a manuscript should themselves have an intrinsic interest in making an evidence-based case for their study design. In this way, a differing intuitive view of the reviewer may be prevented and the validity of a study design is assessed more strongly based on evidence rather than personal views. Peer review decisions will thus hopefully be less biased. In addition, replication studies that seek to overcome limitations of the original study can prioritize such limitations that are backed up with evidence.

6.2.4.2. Limitations

Evidence profiles are created by qualitative evidence evaluation from human reviewers, and thus may contain bias. To mitigate this threat, two researchers independently rated the evidence and then compared their results to reach an agreement. The threat extraction, categorization, and subsequent evidence collection was done by only one researcher, which again might incur a bias and threaten the internal validity of our study. However, when comparing the threat categories and number of occurrences with a previous systematic review on confounding factors in program comprehension studies by Siegmund and Schumann [SS15], we find mostly similar results regarding threat codes and their frequencies.

When developing the evidence profiles, we decided to focus on threat categories instead of individual codes. For example, rather than searching for evidence of programming experience as threat to internal or external validity, we combined the two into the category of programming experience. A finer distinction was not necessary for our purpose, as evidence for the influence of programming experience on code comprehension is equally relevant for all validity types. Further, a finer distinction would have been difficult to make. As previously noted, validity threats in primary studies are currently not discussed in a way that makes clear what type of validity the authors believe their study characteristics would affect.

We focused on threats to validity from code comprehension *experiments*. Evidence profiles for validity threats in code comprehension studies using other research methodologies can, provided equal or similar epistemological stances, help similarly well in making informed design decisions. Whether using evidence to design studies other than experiments makes sense depends on the epistemological stance one takes and how one understands the concept of evidence. We leave this discussion to future work.

Another limitation in the external validity of our study is that due to the amount of effort required to systematically collect evidence, we were able to analyze only three of dozens of different validity threats. We focused on the most frequently mentioned threats, as we expected them to provide the most value for the widest range of program comprehension studies. It is possible that repeating the same methods for the remaining threats will uncover new idiosyncrasies of program comprehension. As such, generalizing the results obtained in this work to other threats should be done with care.

Finally, when analyzing our data, we found a rather strong bias in the number of studies with quantitative data from experiments over qualitative data. Even though we did not explicitly exclude any papers on the grounds of them being a qualitative study, the results still heavily favored quantitative experiments. In numbers, 68 out of 69 (99 %) pieces of evidence in the evidence profiles were from some form of experiment. However, we suspect that potentially undetected evidence from qualitative studies would have only minimally changed the emerging evidence profiles in our study. To

avoid such bias in future studies, we recommend formulating descriptions of evidence profiles in a more method-neutral terminology (e.g., avoiding terms such as confounding factors, which are primarily indicative of experiments).

6.3. Conclusion

In this chapter, we investigated threats to validity in code comprehension experiments. First, we analyzed the state of the art by reviewing the reported validity threats in 95 papers of primary research. We found that while most studies mentioned threats to validity, few supported them with corresponding evidence. Furthermore, only in one case did the cited evidence support the validity of a threat and meet our quality criteria to be included in an evidence profile. Next, we searched for evidence regarding the three most frequently mentioned threat categories. Our evidence collection yielded both positive and negative evidence for each threat category.

After looking closer into the collected evidence and comparing the characteristics of the different studies, we concluded that validity threats are highly context-dependent. Even the threats that are intuitively expected to affect program comprehension, such as programming experience, depend on how they are measured, the sample population, comprehension tasks, and other context factors. Therefore, we must consider all individual characteristics of a study when assessing potential validity threats to develop methodologies that use evidence as a basis for implementing appropriate mitigation techniques. Furthermore, discussions of validity threats in papers need to explicitly address context factors and researchers should use evidence to support these discussions.

We encourage the usage of threat catalogs and recommend the adherence to reporting guidelines for threats to validity to improve the reproducibility and comparability of study results. Structured guidelines for reporting threats to validity need to be established further to inform researchers on how they can incorporate evidence into their validity assessments. We need more knowledge documentation on which threats exist, the evidence supporting

them, the context in which they occur, and which mitigation techniques can be used to address them. Previous works laid the groundwork in this endeavor by documenting threats in software engineering studies in a knowledge base and providing guidelines for controlling the influence of confounding factors in program comprehension experiments, respectively [BKE+14; SS15]. These works can be further extended by incorporating evidence and documenting threats to validity in a common database.

We envision a future in which scientists regularly apply our methodology to collect and summarize the evidence for additional threats to validity and for experiments far beyond code comprehension. Even when looking at just three of the most common threats, we found less evidence for less popular threats, which might indicate even bigger gaps in evidence for those threats that have yet to be analyzed. Moreover, our approach may be applied to other domains to investigate how threats to validity affect experiments there.

To summarize, we have made progress toward our goal of helping researchers design and evaluate their code comprehension study designs through two contributions in this chapter. First, we have collected evidence for the three most frequently discussed threats to validity and found that, depending on the actual study parameters, these threats might actually not always threaten the validity of a study. Second, we have piloted the compilation of evidence using a pragmatic research methodology that we expect will help us to synthesize even more evidence for consequences of design decisions in the future. Yet, the methodology of evidence profiles still bears some open challenges and limitations, which we will now address in [Chapter 7](#).



CHAPTER
7

DISCUSSION

In this chapter, we first summarize our contributions by answering the research questions, then discuss open challenges and limitations, and finally highlight promising future research directions.

7.1. Answers to the Research Questions

We briefly answer the research questions introduced in [Section 1.2](#). Within these answers, we refer to the relevant sections and chapters that addressed the research questions in more detail.

RQ1: What are differences and similarities in the design characteristics of code comprehension experiments?

In [Chapter 3](#), we looked at the design characteristics of 95 code comprehension experiments published between 1979 and 2019. The studies are comparable in some respects. For example, the majority of them investigate the influence of a treatment on the code comprehension performance of the study participants. In particular, the influences of semantic cues, code

structures, and developer characteristics on code comprehension are frequently the focus of such research. Furthermore, most of the experiments use a one-at-a-time within-subject study design and usually sample less than 50 participants for their study.

At the same time, we have found that each of the study designs we examined is unique. This is partly due to the amount of design decisions that have to be made, but also because there are plenty of options to choose from for each design decision. Of particular importance are the comprehension tasks and measures used. Almost each code comprehension experiment used its own individual task design and a suitable way to quantify the performance of the participants. While the authors of the experiments had the same task, i.e., to measure how well their participants understood a certain code snippet, they sometimes followed widely differing ways to achieve this goal.

In response to this finding, we discussed whether the diversity of study designs is a consequence of uncertainty. What can be said for sure is that each research team puts ample thought into their study design. This can be seen, for example, in the diversity of discussed threats to validity: We found that authors of code comprehension experiments discuss threats to validity drawn from no fewer than about 50 threat categories. However, it is striking that for hardly any discussed threat to validity, evidence in the form of empirical studies is cited. The discussions of the validity of experimental designs therefore currently seem rather speculative. This motivated us to contribute evidence on the actual influences of suspected confounding variables (RQ2 and RQ3) and to examine existing evidence more closely (RQ4).

RQ2: How do individual characteristics of a developer influence code comprehension?

In [Section 2.2](#), we have described the background to this question and exemplified the influence individual characteristics can have on the behavior and performance of developers in various activities. In [Chapter 4](#), we then took on the investigation of two concrete constructs that are supposed to have an influence on code comprehension. These two constructs were a

developer's personality and their intelligence.

One hundred thirty-five students took part in our correlational study of the relationship of personality and intelligence facets with code comprehension. What we primarily looked at was the correctness of the study participants in answering comprehension questions on code snippets. The participants who scored higher in the intelligence facets fluid intelligence, visual perception, and cognitive speed were the ones who performed better in the code comprehension tasks. The personality facet conscientiousness, in interaction with other factors, also explains some of the variance in code comprehension performance. Other personality factors, such as Honesty-Humility, Emotionality, Extraversion, Agreeableness, and Openness to Experience, however, do not seem to be promising predictor variables.

The answer to the question of how individual characteristics of a developer influence code comprehension can be answered as so often with: it depends. We have shown that it depends both on the concrete individual characteristics themselves and on their interaction. That the influence of individual characteristics on code comprehension may also depend on the study context itself is something we learn when answering RQ4. Accordingly, we recommend being cautious about making general statements about the influence of presumed individual characteristics on code comprehension.

On a related note, we understand that assumptions about certain influences seem intuitively reasonable. That more intelligent developers understand code more easily hardly sounds controversial. Evidence from different studies nevertheless helps us in any case to assess more accurately the actual influence in a particular study context. For example, if intelligence cannot be controlled as a confounding variable in a specific study design, it is helpful to have studies that provide evidence on the effect size of intelligence on code comprehension. This allows, for example, an informed discussion about the influence of the interfering variable on the conclusion validity of obtained study results.

RQ3: How do contextual factors influence code comprehension?

Following on from research on the influence of individual characteristics, we then focused on the influence of contextual factors on code comprehension. Developers will always comprehend code under certain conditions that arise from their current work context and setup. The question here is which of these conditions pose constraints and which may be beneficial for code comprehension. In [Section 2.3](#), we laid the foundation for this question by reviewing studies of code comprehension in virtual reality as an example. Then, in [Chapter 5](#), we conducted two primary studies ourselves on the influence of contextual factors.

The contextual factor we were interested in was the presence of cues about the difficulty of a code snippet that participants in our studies needed to understand. In experiments with human participants, all potential environmental factors can never be known, let alone controlled. We wanted to draw attention to this problem because some ways of measuring code comprehension are more robust to such influences than others. Specifically, we were able to show that subjective evaluations of the difficulty of source code can be easily biased by, for example, subtle cues in the introduction to the study. We found a very strong and significant anchoring effect. At the same time, such cues had no effect on the actual code comprehension performance of the study participants, as measured by correctness and time in task completion.

The answer to the question of how contextual factors influence code comprehension can therefore be answered as follows: depending on the concrete factor, it can strongly influence a developer's performance and approach to code comprehension. At the same time, however, at least performance depends on the specific way code comprehension is measured. According to our studies, subjective self-assessments of the participants are more easily influenced by cognitive biases than correctness and speed in answering comprehension questions.

RQ4: What evidence can be found for frequently discussed threats to validity in code comprehension experiments?

In the previous [Chapter 6](#), we addressed the question of available evidence by looking again at the code comprehension literature. The influence of programming experience, program length, and the selected comprehension measures are the three most commonly discussed threats to validity in code comprehension experiments. To analyze the evidence for these influences on code comprehension, we mapped the findings from existing primary studies into evidence profiles, which provide us with a visual summary of whether most of the studies confirm or refute the corresponding hypotheses.

For programming experience, we hypothesized that experience influences a developer's code comprehension. 49 pieces of evidence were assessed for their strength and included in an evidence profile. The majority of 37 studies provided supporting evidence for the influence of programming experience on code comprehension, while 12 studies found no evidence for the presumed influence within their study context. One reason for the divergent results could be that programming experience is measured in different ways, and thus the predictive power varies [[SKL+14](#)]. How exactly one should measure experience should be discussed more intensively within the research community. In any case, we consider it is sensible to discuss the programming experience of your own study participants as a threat to the validity of your experiment in such a way that it becomes clear what exactly is meant by 'programming experience' and how the influence is potentially noticeable in the concrete study context (if at all).

Then, we found considerably less evidence for the hypothesis that code snippet length influences code comprehension. Only six papers provided supporting evidence for this hypothesis, while seven could not show such an influence in their data. The case here seems less clear than for the influence of programming experience. Therefore, we looked more closely at the individual pieces of evidence and found at least the pattern that in the studies in which only experts participated, program length did not play a role in their comprehension. Thus, program length primarily seems to be a

factor to be controlled for when novices participate in the study.

Finally, we looked for evidence regarding the hypothesis that different comprehension measures do not correlate with each other. Five studies provided evidence to support this hypothesis, while two studies contradicted the hypothesis. Thus, assuming that not all common comprehension measures correlate with each other, it would be at least advisable to use multiple comprehension measures within a study to capture multiple dimensions of comprehension. Currently, this is often not the case: of the 95 primary studies we looked at in [Chapter 3](#), more than a third (37) used only a single measure to measure their participants' code comprehension performance. Depending on the study context, this may cause them to miss aspects of code understanding that would have been uncovered if they had analyzed more than one measure.

Using Evidence to Evaluate the Validity of Code Comprehension Experiments

Overall, we can say that for the most commonly discussed threats to validity, some evidence already exists that argues for or against their actual impact. What struck us, however, is that almost no study cites this evidence in their discussion of validity threats, and in 94% of the rare cases where they do, the cited evidence provides little to no support for the claim. Unquestionably, it is an additional effort for the authors of a primary study to first identify among the more than 50 potential threats to validity those that could have an influence and then to search the literature for evidence on them. Existing evidence profiles can greatly reduce this search effort for authors of primary studies, as they already compile and assess the available evidence.

The central research question of this thesis asks how evidence can be used to assess the validity of code comprehension experiments. Our answer to this question is that a few researchers need to create evidence profiles that would then benefit the majority of researchers in the discussion of the validity of their code comprehension experiments. We have demonstrated while answering RQ4 that the creation of such evidence profiles for frequently

discussed validity threats in code comprehension experiments is feasible. Yet, there are a couple of open challenges and points for consideration that we will discuss in the next section.

7.2. Open Challenges and Limitations

The idea of creating and using evidence profiles for an evidence-based discussion of the validity of code comprehension experiments works under certain assumptions and limitations. We will now reflect critically on these to better assess the full potential of the idea and perhaps overcome some of the limitations in the future.

On potential bias due to implicit vote counting

The idea of synthesizing research findings from multiple studies is not new. Secondary research to compile and synthesize research findings from multiple primary studies has proven useful in the past to save decision makers time and effort in reviewing relevant research findings. One can also derive practical implications from a secondary study with more confidence than one could from a single primary study. This is due, first, to different methodological and ideological perspectives inherent in individual primary studies, which can be balanced in a secondary study. Second, individual primary studies are often subject to limitations such as a small sample size or specific contextual conditions that can be overcome in secondary studies by combining the data and research findings.

What is new is the use of Wohlin's evidence profiles [Woh13] for such a synthesis. In a recent position paper, Ralph and Baltes [RB22] argue that authors of secondary studies should identify the type of their systematic review to move toward more mature secondary studies. They list six types of systematic reviews: meta-analysis, meta-synthesis, case survey, critical reviews, scoping reviews, and rapid reviews. Wohlin [Woh13] himself remains rather vague in differentiating his methodology of evidence profiles. He intends to consider qualitative evidence as well as quantitative evidence,

while avoiding the difficulties and the great effort of meta-analysis and meta-synthesis. At the same time, he distinguishes his methodology from simple *vote counting*, in that the available pieces of evidence are classified according to their meaningfulness on an ordinal scale, dependent on the context.

Why are these reflections important? Vote counting refers to drawing conclusions by simply counting studies whose results support a hypothesis or research question and counting studies whose results oppose it. The majority wins. Since neither study quality nor effect size nor study heterogeneity within the two camps are considered, classical vote counting is critical from a scientific point of view [Hun97; Sta01].

In our view, evidence profiles provoke implicit vote counting, in the sense that it is visually depicted for individual research questions on which side the evidence predominates. Evidence profiles, however, have a decisive advantage over classical vote counting: study quality, effect size, and other properties of individual studies can become part of the evidence strength that is qualitatively assessed by the researchers and can thus be included in the visual presentation. Few highly meaningful studies on one side and many less meaningful studies on the other side of an evidence profile are thus made visible. Since a certain bias in the conclusion of the reader of an evidence profile due to quantitative superiority of evidence on one of the two sides cannot be excluded, we call this a limitation of evidence profiles.

Why are evidence profiles nevertheless currently advantaged over the more common meta-analysis and meta-synthesis techniques? In the code comprehension field, there is currently no completeness in the reporting of statistics that would be necessary, for example, for a quantitative meta-analysis. The problem of missing statistics is amplified when the evidence provided was not the primary focus of a particular study. For example, this was often the case in our investigation (Chapter 6) when we looked for evidence on the influence of programming experience on code comprehension: many studies provide evidence for or against this influence in a casual analysis, for example to estimate the influence as a confounding variable. In the reporting, there are sometimes no statistics at all, but only statements about the observed influence. Alternatively, using the raw data for analysis

in a secondary study is hardly practicable at present, since still very few primary studies publish their raw data. While it would be possible to increase the focus on reporting guidelines, the chances are that they would not be followed across the board. Moreover, such guidelines would not have a retroactive effect on the research of the past fifty years.

Evidence profiles allow the integration of different types of evidence, almost independently of the reporting of individual studies, and can thereby rank those studies higher that already meet certain quality characteristics. We would like to stress again that we do not oppose alternative research methods in secondary studies, and are also aware of their advantages¹. However, we would argue that the methodology of evidence profiles is one of the most pragmatic methods that is currently directly applicable in the code comprehension research field.

The dependence on evidence for evidence

If we intend to rely more on evidence when discussing the validity of experimental designs in the future, we need to ask ourselves how *good* the available evidence is. In this thesis, we have primarily looked for evidence of factors influencing code comprehension. Such evidence is naturally found in the results of code comprehension studies. However, we can only estimate intuitively how valid these studies are, since they could not build on existing evidence. Evidence for the consequences of certain design decisions thus inevitably comes from studies whose study design we actually seek to evaluate in an evidence-based manner.

Breaking out of this circular dependence is theoretically possible with some philosophy. If there are enough different study designs by enough different authors on a research question, a less biased and overall more complete picture of research designs on the contribution of evidence emerges (see [Section 1.3.2](#)). When this diversity is present, we can identify which

¹Ironically, much of the motivation for this dissertation arose from a situation in which we conducted a quantitative meta-analysis [MWW20], but this endeavor was quite challenging due to the diversity of study designs and lack of raw data in code comprehension research.

methodological features are responsible for potential variations in the results of individual primary studies. We can thus make retrospective evidence-based assessments of the validity of individual primary studies.

In [Chapter 4](#), for example, we presented one of our experiments on the influence of intelligence on code comprehension performance. We based the experiment design as best we could on existing theory and evidence, but it is clear that there will always be consequences of certain design decisions that we do not yet know. Putting our experiment and other future studies of the influence of intelligence into an evidence profile, such an evidence profile can itself provide evidence that certain different design decisions affect study results.

The success of this approach (and, for that matter, the success of almost all types of synthesizing secondary studies) depends on there being multiple studies on the same research question. We have previously argued for the recognition of the importance of replication and reproduction studies in [Section 5.4](#). At the same time, the reality that novelty is a weighty evaluation criterion of a study in the software engineering research field, remains a challenge for contributing additional evidence to an already studied research question.

When and how to update evidence profiles?

The three evidence profiles we created in [Chapter 6](#) are snapshots of the evidence landscape on three research questions. We explained in [Section 1.3.1](#) that evidence is seldom timeless and that we may need to completely reconsider existing evidence when paradigm shifts occur. Furthermore, it is in the nature of science that over time, new evidence emerges. Thus, an open question, possibly another challenge, is when and in what form evidence profiles should be updated. We have two comments on this.

First, we think it is a great opportunity if evidence profiles are not created from scratch every time, but only updated. The more teams of authors work together on a single evidence profile over time and argue about individual pieces of evidence, the more one can help reduce bias and the more likely it

is that quality guidelines will come from a consensus. Contributing can be straightforward, for example via a public database.

Second, however, we also believe that minimal peer review is necessary both in the initial creation and potential update of evidence profiles. This should ensure that the process of finding, selecting, and classifying evidence from primary studies is at least transparently documented. Evidence profiles have the potential to serve as decision support in relevant controversy. Accordingly, there is a risk that individuals may attempt to manipulate the evidence profiles in their own interests. This is comparable to the deliberate incorporation of misinformation into Wikipedia; here, too, it is advantageous that updates to articles are reviewed by the community. When we created the evidence profiles mentioned above, we therefore documented the evaluation of each individual primary study contributing to an evidence profile transparently via the supplemental materials. We consider it necessary to make such a practice a standard when creating and updating evidence profiles.

On the generalizability of our approach

We have deliberately limited ourselves in two respects in this thesis.

For one thing, we have focused on the design of experiments. This is because validity, for example, has a different meaning in qualitative research methodologies and is therefore only partly comparable to validity of experiments (see [Section 1.2](#) and [Section 2.5](#)). However, if one adopts comparable epistemological views as we have assumed for experiment designers in the context of this thesis, the rough idea of using evidence in evaluating study designs can be transferred. For example, in designing an interview study, one can draw on primary studies that provide evidence on various interviewer biases, that is, the bias in findings due to a particular view of the observer. Following the evidence to prevent bias would be in the spirit of this thesis. However, it is also clear that this way of thinking only makes sense for a researcher who does not already assume that our knowledge about the world is entirely constructed by the observer. Such a view seems more common among qualitative researchers than among researchers who

primarily conduct experimental studies. Experimental research is usually subject to the assumption that independent researchers will obtain the same results when replicating the experiment. Hence, there is a stronger view here that there exists evidence independent of the observer, and this evidence should then be followed.

For another, we have been working in the context of code comprehension research throughout this thesis. The code comprehension field is in its maturity and the views of the community certainly comparable to other sub-research fields of software engineering, in which empirical research is conducted. Points of contact can be found, for example, where there is methodological proximity and where common guidelines are used, such as the general book chapters on experimentation by Wohlin et al. [WRH+12] or the experiment reporting guidelines by Jedlitschka et al. [JCP08]. The discussion of validity threats is taking place throughout SE research, and probably in all sub-fields with similar uncertainty, for example, about how to deal with the trade-off between different validity dimensions [SSA15]. What existing guidelines do not yet provide is an answer to how evidence-driven validity evaluation should be. The corresponding chapters on validity evaluation [JCP08; WRH+12, e.g.] so far only advocate that the validity of one's own study should be assessed. Lists of potential threats to the validity of experiments in general are provided [WRH+12]. As we know, however, there is no lack of potential validity threats to discuss (see, e.g., Section 3.2.3.8), but rather a pragmatic method to prioritize the variety of threats to report only those that are actually relevant. Since concrete approaches to this issue are rarely discussed, we think that both the problem and our proposed approach can be applied to and be useful to other SE research fields.

Finally, regarding the internal and external validity of the studies we conducted (Chapters 3 to 5), we would say that we have placed greater emphasis on internal validity than on external validity. We have discussed what implications this has for each study in the discussion sections of each chapter. We do not see any consequences for the bigger picture here, as we consider each of our primary studies to be building blocks (with high

internal validity) that can only ever be combined with other building blocks to form an overall (externally valid) picture.

7.3. Future Research Directions

Now that we have dealt extensively with the past and present situation, let us conclude with an outlook on the future. The code comprehension research field is flourishing. We are happy about that and if we may point researchers to possible directions for future work, it would be the following.

Explicit Clarity: The Terminology, Theory, and Perspectives in Code Comprehension Research

A research field can flourish to the extent that more and more researchers take up the topic and conduct primary studies on the same subject. However, an indicator that the field is doing well not only quantitatively but also qualitatively would be, in our view, a notably high number of secondary studies. The existence of secondary studies would suggest that studies build on each other, that a scientific discourse takes place, and that individual studies are comparable with each other. There is a need to improve in this respect.

We have previously formulated two fundamental issues as action items in [Section 3.3](#): there is a need for a contemporary definition of the code comprehension construct and an underlying theory of people's cognitive processes and behavior in understanding source code. Code comprehension experiments currently use different terms for the (presumably) same construct, and different tasks and measures to operationalize code comprehension. Whether researchers share a common underlying view of what they mean by code comprehension is often not clear, as explicit descriptions of their own views are missing from the papers. It would ease the work of all researchers if we had more theoretical knowledge about code comprehension and could synthesize it into one or more theories. Authors of primary studies could cite the underlying theory of their research as a reference that is consistent

with their own view and use it to justify their design decisions. Authors of secondary studies could more easily compare primary studies with each other that explicitly build on the same theoretical foundation. If we do not pick up in theory building where we left off after the 1990s, the code comprehension research field will remain in a state where an unmanageable conglomeration of primary studies are difficult to compare with each other and will likely have limited impact on their own.

Primary Research on Contextual Factors Influencing Code Comprehension

The working environment of a software developer is already very diverse today. Different tools support the work, different processes ensure optimized workflows and different colleagues work together on the code. These are all contextual factors that influence how and how well a developer understands source code. And it is precisely these factors that offer vast potential for research. Consider, for example, two developers pair programming, i.e., working on code together synchronously in the roles of thinker and writer. How, for example, does a developer's code comprehension strategy change when he or she has to understand code in pair programming rather than alone? How does the mental model of the code of the two developers involved in pair programming differ? How will the code comprehension strategy change in the future when virtual coding assistants become pair programming peers? Answers to these questions can, for example, help to design tools and processes in such a way that high cognitive load is prevented and logic errors in the code are detected more reliably.

Further Advancement of the Evidence Profile Methodology

Evidence profiles for validity threats should be further developed in a way that they can at least indicate why certain evidence was negative or positive in a specific study setup. This allows authors of primary studies to better assess whether a threat is actually a threat in their study, rather than only

whether there is conflicting evidence for the influence of a particular threat among existing studies. We have previously explained that the methodology has its advantages and disadvantages compared to other types of secondary studies. Therefore, in a next step, in addition to the creation of further evidence profiles, the methodology itself should be improved. In addition to the strength and direction of the evidence, an evidence profile could be extended to include a third dimension that provides direct information about relevant study parameters. Possible patterns in the overall evidence could thus be identified and related to one's own study context.

A second important point for the further development of the methodology has already been addressed in the limitations, i.e. the question of when and how evidence profiles should be updated. At a minimum, we need to evaluate whether we can arrive at a procedure to efficiently keep evidence profiles up to date over time without having to develop them from scratch each time new evidence emerges. Eventually, this question will have to be addressed because by then our evidence profiles presented in this thesis will be a few years old and new evidence will have been published. Quantitative meta-analysis has the advantage that when updating the status of a research question, new data sets can be added to the old ones and analyzed together with the existing analysis script. In qualitative synthesis procedures, one could take a similar approach, analyzing only the newly added evidence, and plugging the results into an evidence profile with the previous ones. However, the consistency in the qualitative assessment of new and old evidence could then be lower with different author teams than if one author team or the community classified all the existing evidence via a consistent procedure.

Exploration of the Research Community's Views on Evidence

One may discuss the nature and role of the threats to validity section in scientific literature itself. While we examined and reported on various descriptive aspects of the threats reported in existing papers, we did not inquire into why researchers chose to report and discuss specific threats in a certain way. Furthermore, our research does not provide explicit guidance

on how a paper author ought to write a *threats to validity* section. While the position presented in this work suggests that reported threats should be supported with evidence whenever possible, this is not necessarily a sentiment shared by all members of the scientific community. A section on validity threats may also be a place where researchers should be able to speculate without concrete evidence and point out potential shortcomings as directions for future research. This debate should be conducted within the respective research communities, and agreements should then be recorded and incorporated into existing guidelines for the reporting of validity threats.

Siegmund et al. [SSA15] once surveyed 79 program committee and editorial board members about their views on the importance and trade-off between internal and external validity. The participants answered from a reviewer's perspective, and different personal views emerged that potentially affect the acceptance of a manuscript. These views included even those that would reject papers in principle if they attempted to maximize internal validity [SSA15]. Here one could proceed and similarly investigate, for example, why researchers would (not) use evidence for the evaluation of empirical studies, and what constitutes good evidence for them. We speculate that views will vary as much as they did in Siegmund et al.'s survey.

Consider the following scenario. A hypothetical experimental design that considers all available evidence is judged by one peer-reviewer to be the best design currently possible. A second reviewer will disagree because they have a different view of how to handle available evidence and that potentially all available evidence is based on false paradigms. Thus, for the second reviewer, the quality of a study design cannot be judged by whether it conforms to existing evidence.

These two views admittedly serve the extremes on a spectrum of willingness to evaluate study designs based on evidence. However, should these two views be present in our community in this form, we can potentially gain insights into those causes of inconsistency in peer review decisions that have not been explored before. And then we need to consider how to deal as a research community with the possibility that individual philosophical views determine the publication of research findings.

CONCLUSION

Walter Tichy, one of the early advocates of empirical research in computer science [TLPH95], recently told his personal story of how he experienced the maturing of SE research: “It took about 25 years for research in software engineering to become evidence based” [Tic22]. The acceptance of data-based evidence had to be championed in the early days of the field because rationalism, i.e., deductive reasoning for the superiority of a contribution, would have been the rule. “Today,” Tichy notes, “we have a new generation of computer scientists for which empirical studies are the norm” [Tic22].

Evidence has taken on a prominent role in this thesis. While we could stand on the shoulders of those who have advocated evidence-based software engineering in the past, we now sought to bring this effort to a whole new level. That is, by using evidence not only to make informed decisions in practice, but also in designing the studies that provide the evidence. We hope that it will not take another 25 years before it becomes the norm for intuition to be complemented by evidence when evaluating a study design.

For our vision to become reality, however, there was and still is work to be done. We made a start in the context of experimental code comprehension research. Specifically, we mapped the landscape of design decisions and

threats to validity, contributed our own experimental studies to provide evidence for commonly discussed threats to validity, and piloted the evidence profile methodology to synthesize available evidence.

Once created, such an evidence profile serves each researcher as an overview of the evidence landscape for a specific question, such as the influence of a suspected confounding variable. Researchers can then prioritize potential threats to validity based on empirical evidence for specific study design parameters when designing and reporting their research. For example, we have found that the frequently discussed influence of programming experience on code comprehension should not be assumed across the board. The actual influence varies depending on factors such as how experience is operationalized and the characteristics of the participant sample.

Overall, we consider the evidence profile methodology to be a pragmatic tool that can already today support code comprehension researchers by reducing uncertainty in experiment design and mitigating some of the bias in peer review. Furthermore, the profiles show for which assumptions there is sufficient evidence at all. However, it is also worth noting that not all researchers may share the view that evidence-based study designs are superior. We have elaborated on our view in the introduction to this thesis, and suspect that it aligns in essence with the views of those who prefer to use experimental research methodologies to obtain new knowledge. Thus, the key message of this thesis will certainly resonate with a few researchers. Nevertheless, we should now consider the opinions within the research community regarding whether discussions about the validity of a study design should be based on evidence. Such an investigation will potentially strike a chord and bring to light different perspectives on what constitutes evidence and how much the research community may want to rely on its own past contributions.

BIBLIOGRAPHY

- [AAV+19] V. Averbukh, N. Averbukh, P. Vasev, I. Gvozdarev, G. Levchuk, L. Melkozerov, I. Mikhaylov. ‘Metaphors for Software Visualization Systems Based on Virtual Reality’. In: *Augmented Reality, Virtual Reality, and Computer Graphics*. Cham: Springer International Publishing, 2019, pp. 60–70 (cit. on p. 58).
- [ABA+19] A. Ampatzoglou, S. Bibi, P. Avgeriou, M. Verbeek, A. Chatzigeorgiou. ‘Identifying, categorizing and mitigating threats to validity in software engineering secondary studies’. In: *Information and Software Technology* 106 (2019), pp. 201–230 (cit. on p. 124).
- [ABUL05] L. Arockiam, T. L. A. Beena, K. Uma, H. Leena. ‘Object-oriented program comprehension and personality traits’. In: *Proceedings of SMEF* (2005) (cit. on p. 135).
- [Ade81] B. Adelson. ‘Problem solving and the development of abstract categories in programming languages’. In: *Memory & cognition* 9.4 (1981), pp. 422–433 (cit. on p. 40).
- [Ade84] B. Adelson. ‘When novices surpass experts: The difficulty of a task may increase with expertise’. In: *Journal of Experimental Psychology: Learning, Memory, and Cognition* 10.3 (1984), pp. 483–495 (cit. on p. 234).
- [AL01] M. C. Ashton, K. Lee. ‘A theoretical basis for the major dimensions of personality’. In: *European Journal of Personality* 15.5 (2001), pp. 327–353 (cit. on pp. 141, 142).

- [AL16] K. Albusays, S. Ludi. ‘Eliciting Programming Challenges Faced by Developers with Visual Impairments: Exploratory Study’. In: *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*. CHASE ’16. Austin, Texas: Association for Computing Machinery, 2016, pp. 82–85 (cit. on pp. 56, 57).
- [AL21] M. C. Ashton, K. Lee. *The HEXACO Personality Inventory - Revised*. 2021. URL: <https://hexaco.org> (cit. on p. 144).
- [ALH17] K. Albusays, S. Ludi, M. Huenerfauth. ‘Interviews and Observation of Blind Software Developers at Work to Understand Code Navigation Challenges’. In: *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*. ASSETS ’17. Baltimore, Maryland, USA: Association for Computing Machinery, 2017, pp. 91–100 (cit. on pp. 56, 57).
- [AP06] G. Allen, B. J. Parsons. ‘A little help can be a bad thing: Anchoring and adjustment in adaptive query reuse’. In: *ICIS 2006 Proceedings* (2006), p. 45 (cit. on p. 169).
- [APA14] American Educational Research Association, American Psychological Association, National Council on Measurement in Education, Joint Committee on Standards for Educational and Psychological Testing (U.S.) *Standards for educational and psychological testing*. Washington, DC: American Educational Research Association, 2014 (cit. on p. 191).
- [ARM18] A. Armaly, P. Rodeghero, C. McMillan. ‘A Comparison of Program Comprehension Strategies by Blind and Sighted Programmers’. In: *IEEE Transactions on Software Engineering* 44.08 (2018), pp. 712–724 (cit. on p. 56).
- [AS96] V. Arunachalam, W. Sasso. ‘Cognitive processes in program comprehension: An empirical analysis in the context of software reengineering’. In: *Journal of Systems and Software* 34.3 (1996), pp. 177–189 (cit. on p. 80).
- [ATFJ22] C. Ayala, B. Turhan, X. Franch, N. Juristo. ‘Use and Misuse of the Term “Experiment” in Mining Software Repositories Research’. In: *IEEE Transactions on Software Engineering* 48.11 (2022), pp. 4229–4248 (cit. on p. 66).

- [AWF18] S. Ajami, Y. Woodbridge, D. G. Feitelson. ‘Syntax, predicates, idioms - what really affects code complexity?’ In: *Empirical Software Engineering* 24.1 (2018), pp. 287–328 (cit. on pp. 61, 75, 108, 235).
- [BAV20] L. Bidlake, E. Aubanel, D. Voyer. ‘Systematic literature review of empirical studies on mental representations of programs’. In: *Journal of Systems and Software* 165 (2020), p. 110565 (cit. on p. 68).
- [BB22] D. Budgen, P. Brereton. ‘Evolution of secondary studies in software engineering’. In: *Information and Software Technology* 145 (2022), p. 106840 (cit. on p. 214).
- [BBB + 15] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, S. Tamm. ‘Eye movements in code reading: Relaxing the linear order’. In: *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE. 2015, pp. 255–265 (cit. on p. 53).
- [BBL76] B. W. Boehm, J. R. Brown, M. Lipow. ‘Quantitative Evaluation of Software Quality’. In: *Proceedings of the 2nd International Conference on Software Engineering*. ICSE ’76. San Francisco, California, USA: IEEE Computer Society Press, 1976, pp. 592–605 (cit. on p. 36).
- [BDA14] J. Belmonte, P. Dugerdil, A. Agrawal. ‘A three-layer model of source code comprehension’. In: *Proceedings of the 7th India Software Engineering Conference*. 2014, pp. 1–10 (cit. on p. 49).
- [BGL00] G. V. Bodenhausen, S. Gabriel, M. Lineberger. ‘Sadness and susceptibility to judgmental bias: The case of anchoring’. In: *Psychological Science* 11.4 (2000), pp. 320–323 (cit. on p. 186).
- [BGMB20] T. Besker, H. Ghanbari, A. Martini, J. Bosch. ‘The influence of Technical Debt on software developer morale’. In: *Journal of Systems and Software* 167 (2020), p. 110586 (cit. on p. 121).
- [BGNT20] A. B. Brendel, R. S. Greulich, F. Niederman, S. T.-N. Trang. ‘Towards a greater diversity of replication studies’. In: *AIS Transactions on Replication Research* 6.1 (2020), p. 20 (cit. on p. 209).

- [BHH+09] D. Bell, T. Hall, J. E. Hannay, D. Pfahl, S. T. Acuna. ‘Software engineering group work: personality, patterns and performance’. In: *Software engineering group work: personality, patterns and performance*. Vol. SIGMIS CPR’10 Proceedings of the 2010 ACM SIGMIS Computer Personnel Research Conference. ACM, 2009, pp. 43–47 (cit. on p. 135).
- [BJ98] V. Benet-Martínez, O. P. John. ‘Los Cinco Grandes across cultures and ethnic groups: Multitrait-multimethod analyses of the Big Five in Spanish and English.’ In: *Journal of personality and social psychology* 75.3 (1998), p. 729 (cit. on p. 175).
- [BKE+14] S. Biffl, M. Kalinowski, F. Ekaputra, A. A. Neto, T. Conte, D. Winkler. ‘Towards a semantic knowledge base on threats to validity and control actions in controlled experiments’. In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Ed. by M. Morisio. ACM Digital Library. New York, NY: ACM, 2014, pp. 1–4 (cit. on pp. 66, 215, 241).
- [BKSB11] M. Bérdis, F. Köteles, A. Szabó, G. Bárdos. ‘Placebo effects in sport and exercise: a meta-analysis’. In: *European Journal of Mental Health* 6.2 (2011), pp. 196–212 (cit. on p. 168).
- [BKW95] P. G. Bickley, T. Z. Keith, L. M. Wolfle. ‘The three-stratum theory of cognitive abilities: Test of the structure of intelligence across the life span’. In: *Intelligence* 20.3 (1995), pp. 309–328 (cit. on p. 142).
- [BLMM09] D. Binkley, D. Lawrie, S. Maex, C. Morrell. ‘Identifier length and limited programmer memory’. In: *Science of Computer Programming* 74.7 (2009), pp. 430–445 (cit. on pp. 61, 235).
- [BM82] V. Basili, H. Mills. ‘Understanding and Documenting Programs’. In: *IEEE Transactions on Software Engineering* SE-8.3 (1982), pp. 270–283 (cit. on p. 40).
- [BMBW15] D. Bates, M. Mächler, B. Bolker, S. Walker. ‘Fitting Linear Mixed-Effects Models Using lme4’. In: *Journal of Statistical Software* 67.1 (2015), pp. 1–48 (cit. on p. 207).
- [Bor18] A. Borji. ‘Negative results in computer vision: A perspective’. In: *Image and Vision Computing* 69 (2018), pp. 1–8 (cit. on p. 237).

- [BP16] J. Börstler, B. Paech. ‘The role of method chains and comments in software readability and comprehension—An experiment’. In: *IEEE Transactions on Software Engineering* 42.9 (2016), pp. 886–898 (cit. on pp. 75, 90).
- [BR22] S. Baltes, P. Ralph. ‘Sampling in software engineering research: A critical review and guidelines’. In: *Empirical Software Engineering* 27.4 (2022), pp. 1–31 (cit. on pp. 172, 178, 188).
- [Bro78] R. Brooks. ‘Using a behavioral theory of program comprehension in software engineering’. In: *Proceedings of the 3rd international conference on Software engineering*. 1978, pp. 196–201 (cit. on pp. 39, 54).
- [Bro83] R. Brooks. ‘Towards a theory of the comprehension of computer programs’. In: *International journal of man-machine studies* 18.6 (1983), pp. 543–554 (cit. on pp. 19, 39, 40, 43, 48).
- [BWP15] D. Badampudi, C. Wohlin, K. Petersen. ‘Experiences from using snowballing and database searches in systematic literature studies’. In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1–10 (cit. on pp. 78, 222).
- [Cam18] G. A. Campbell. ‘Cognitive Complexity: An Overview and Evaluation’. In: *Proceedings of the 2018 International Conference on Technical Debt*. TechDebt ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 57–58 (cit. on pp. 24, 170, 174, 197).
- [Car+93] J. B. Carroll et al. *Human cognitive abilities: A survey of factor-analytic studies*. 1. Cambridge University Press, 1993 (cit. on pp. 132, 133).
- [Car05] J. B. Carroll. ‘The three-stratum theory of cognitive abilities.’ In: *Contemporary intellectual assessment: theories, tests, and issues* (2005) (cit. on pp. 132, 133).
- [Cas00] A. Caspi. ‘The child is father of the man: personality continuities from childhood to adulthood.’ In: *Journal of personality and social psychology* 78.1 (2000), p. 158 (cit. on p. 131).

- [CBF84] C. Cook, W. Bregar, D. Foote. ‘A preliminary investigation of the use of the cloze procedure as a measure of program understanding’. In: *Information Processing & Management* 20.1-2 (1984), pp. 199–208 (cit. on pp. 61, 75).
- [CC79] T. D. Cook, D. T. Campbell. *Quasi-experimentation: Design & analysis issues for field settings*. Vol. 351. Houghton Mifflin Boston, 1979 (cit. on p. 28).
- [CCWA13] J. Cohen, P. Cohen, S. G. West, L. S. Aiken. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. 3rd ed. New York, NY, USA: Routledge, 2013, p. 536 (cit. on p. 200).
- [CD11] D. S. Cruzes, T. Dyba. ‘Recommended Steps for Thematic Synthesis in Software Engineering’. In: *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2011, pp. 275–284 (cit. on pp. 82, 219).
- [CERS17] N. Capece, U. Erra, S. Romano, G. Scanniello. ‘Visualising a software system as a city through virtual reality’. In: *International Conference on Augmented Reality, Virtual Reality and Computer Graphics*. Springer, 2017, pp. 319–327 (cit. on p. 58).
- [CG08] C. R. Critcher, T. Gilovich. ‘Incidental environmental anchors’. In: *Journal of Behavioral Decision Making* 21.3 (2008), pp. 241–251 (cit. on pp. 188, 189, 192).
- [CL07] A. J. Crum, E. J. Langer. ‘Mind-Set Matters: Exercise and the Placebo Effect’. In: *Psychological Science* 18.2 (2007), pp. 165–171 (cit. on pp. 164, 185).
- [Cla16] M. Clark. *SEM vs. Mixed*. Tech. rep. University of Michigan, 2016, p. 1. URL: https://m-clark.github.io/docs/mixedModels/growth_vs_mixed_old.html (cit. on p. 207).
- [CLB20] R. Castelo-Branco, A. Leitão, C. Brás. ‘Program Comprehension for Live Algorithmic Design in Virtual Reality’. In: *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*. Programming ’20. Porto, Portugal: Association for Computing Machinery, 2020, pp. 69–76 (cit. on pp. 58, 59).

- [CMB16] G. Corno, G. Molinari, R. M. Baños. ‘Assessing positive and negative experiences: validation of a new measure of well-being in an Italian population’. In: *Rivista di psichiatria* 51.3 (2016), pp. 110–115 (cit. on p. 175).
- [CNA+20] S. Chattopadhyay, N. Nelson, A. Au, N. Morales, C. Sanchez, R. Pandita, A. Sarma. ‘A tale from the trenches: cognitive biases and software development’. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 654–665 (cit. on p. 208).
- [Coh92] J. Cohen. ‘A power primer’. In: *Psychological bulletin* 112.1 (1992), p. 155 (cit. on pp. 138, 150).
- [CPS20] B. Cartaxo, G. Pinto, S. Soares. ‘Rapid reviews in software engineering’. In: *Contemporary Empirical Methods in Software Engineering*. Springer, 2020, pp. 357–384 (cit. on p. 214).
- [CPVS16] B. Cartaxo, G. Pinto, E. Vieira, S. Soares. ‘Evidence Briefings: Towards a Medium to Transfer Knowledge from Systematic Reviews to Practitioners’. In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’16. Ciudad Real, Spain: Association for Computing Machinery, 2016 (cit. on p. 214).
- [Cro57] L. J. Cronbach. ‘The two disciplines of scientific psychology.’ In: *American psychologist* 12.11 (1957), p. 671 (cit. on p. 136).
- [CS08] J. Corbin, A. Strauss. *Basics of Qualitative Research (3rd ed.): Techniques and Procedures for Developing Grounded Theory*. 2455 Teller Road, Thousand Oaks California 91320 United States: SAGE Publications, Inc, 2008 (cit. on p. 219).
- [CSC15] S. Cruz, F. Q. da Silva, L. F. Capretz. ‘Forty years of research on personality in software engineering: A mapping study’. In: *Computers in Human Behavior* 46 (2015), pp. 94–113 (cit. on pp. 57, 130, 134, 135).
- [CSW02] M. E. Crosby, J. Scholtz, S. Wiedenbeck. ‘The Roles Beacons Play in Comprehension for Novice and Expert Programmers.’ In: *PPIG*. 2002, p. 5 (cit. on pp. 47, 48, 51).

- [DE14] C. Draganich, K. Erdal. ‘Placebo sleep affects cognitive functioning’. In: *Journal of Experimental Psychology: Learning, Memory, and Cognition* 40.3 (2014), pp. 857–864 (cit. on pp. 168, 181, 185).
- [DFKR03] A. van Deursen, J.-M. Favre, R. Koschke, J. Rilling. ‘Experiences in teaching software evolution and program comprehension’. In: *11th IEEE International Workshop on Program Comprehension*. 2003, pp. 283–284 (cit. on p. 49).
- [DG07] A. D. Da Cunha, D. Greathead. ‘Does personality matter? An analysis of code-review ability’. In: *Communications of the ACM* 50.5 (2007), pp. 109–112 (cit. on pp. 130, 135).
- [Dig90] J. M. Digman. ‘Personality structure: Emergence of the five-factor model’. In: *Annual review of psychology* 41.1 (1990), pp. 417–440 (cit. on p. 175).
- [DKJ05] T. Dyba, B. A. Kitchenham, M. Jorgensen. ‘Evidence-based software engineering for practitioners’. In: *IEEE software* 22.1 (2005), pp. 58–65 (cit. on p. 214).
- [DSK07] M. Di Penta, R. K. Stirewalt, E. Kraemer. ‘Designing your next empirical study on program comprehension’. In: *15th IEEE International Conference on Program Comprehension (ICPC’07)*. IEEE. IEEE, 2007, pp. 281–285 (cit. on p. 73).
- [DTH+20] J. Dominic, B. Tubre, J. Houser, C. Ritter, D. Kunkel, P. Rodeghero. ‘Program comprehension in virtual reality’. In: *Proceedings of the 28th International Conference on Program Comprehension*. 2020, pp. 391–395 (cit. on p. 58).
- [DTR+20] J. Dominic, B. Tubre, C. Ritter, J. Houser, C. Smith, P. Rodeghero. ‘Remote Pair Programming in Virtual Reality’. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 406–417 (cit. on pp. 58, 59).
- [DWL+00] I. J. Deary, L. J. Whalley, H. Lemmon, J. Crawford, J. M. Starr. ‘The stability of individual differences in mental ability from childhood to old age: Follow-up of the 1932 Scottish Mental Survey’. In: *Intelligence* 28.1 (2000), pp. 49–55 (cit. on p. 131).

- [DWT+10] E. Diener, D. Wirtz, W. Tov, C. Kim-Prieto, D.-w. Choi, S. Oishi, R. Biswas-Diener. ‘New well-being measures: Short scales to assess flourishing and positive and negative feelings’. In: *Social Indicators Research* 97.2 (2010), pp. 143–156 (cit. on p. 175).
- [DZB16] P. Devanbu, T. Zimmermann, C. Bird. ‘Belief & evidence in empirical software engineering’. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 108–119 (cit. on pp. 190, 215).
- [EC10] C. Eroglu, K. L. Croxton. ‘Biases in judgmental adjustments of statistical forecasts: The role of individual differences’. In: *International Journal of Forecasting* 26.1 (2010), pp. 116–133 (cit. on p. 186).
- [ES09] B. Englich, K. Soder. ‘Moody experts—How mood and expertise influence judgmental anchoring’. In: *Judgment and Decision making* 4.1 (2009), p. 41 (cit. on p. 186).
- [Ext02] C. Exton. ‘Constructivism and program comprehension strategies’. In: *Proceedings 10th International Workshop on Program Comprehension*. IEEE, 2002, pp. 281–284 (cit. on p. 49).
- [Fak18] S. Fakhoury. ‘Moving towards Objective Measures of Program Comprehension’. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 936–939 (cit. on pp. 30, 51, 75, 109, 132, 158, 186, 205).
- [FB11] A. Furnham, H. C. Boo. ‘A literature review of the anchoring effect’. In: *The journal of socio-economics* 40.1 (2011), pp. 35–42 (cit. on pp. 165, 169, 171, 185–188, 192, 199, 203, 204).
- [Fei21] D. G. Feitelson. ‘Considerations and Pitfalls in Controlled Experiments on Code Comprehension’. In: *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 106–117 (cit. on pp. 72, 73, 122, 205).
- [Fey93] P. Feyerabend. *Against Method*. Third edition. Verso, 1993 (cit. on p. 27).

- [FGN+19] D. Fucci, D. Girardi, N. Novielli, L. Quaranta, F. Lanubile. ‘A replication study on code comprehension and expertise using lightweight biometric sensors’. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. IEEE, 2019, pp. 311–322 (cit. on p. 80).
- [FKH17] F. Fittkau, A. Krause, W. Hasselbring. ‘Software landscape and application visualization for system comprehension with ExplorViz’. In: *Information and Software Technology* 87 (2017), pp. 259–277 (cit. on pp. 58, 59).
- [FLS63] R. P. Feynman, R. B. Leighton, M. Sands. *The Feynman Lectures on Physics*. 1963 (cit. on p. 20).
- [FRM+20] S. Fakhoury, D. Roy, Y. Ma, V. Arnaoudova, O. Adesope. ‘Measuring the impact of lexical and structural inconsistencies on developers’ cognitive load during bug localization’. In: *Empirical Software Engineering* 25.3 (2020), pp. 2140–2178 (cit. on pp. 61, 235).
- [FSW17] B. Floyd, T. Santander, W. Weimer. ‘Decoding the representation of code in the brain: An fMRI study of code review and expertise’. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 175–186 (cit. on p. 52).
- [FWS93] V. Fix, S. Wiedenbeck, J. Scholtz. ‘Mental representations of programs by novices and experts’. In: *Proceedings of the INTERACT’93 and CHI’93 conference on Human factors in computing systems*. New York, NY, USA: Association for Computing Machinery, 1993, pp. 74–79 (cit. on p. 75).
- [FZB+18] R. Feldt, T. Zimmermann, G. R. Bergersen, D. Falessi, A. Jedlitschka, N. Juristo, J. Münch, M. Oivo, P. Runeson, M. Shepperd, et al. ‘Four commentaries on the use of students and professionals in empirical software engineering experiments’. In: *Empirical Software Engineering* 23.6 (2018), pp. 3801–3820 (cit. on pp. 16, 125, 188, 199).
- [GFWA17] D. Graziotin, F. Fagerholm, X. Wang, P. Abrahamsson. ‘On the unhappiness of software developers’. In: *Proceedings of the 21st international conference on evaluation and assessment in software engineering*. 2017, pp. 324–333 (cit. on p. 175).

- [GG84] D. J. Gilmore, T. Green. ‘Comprehension and recall of miniature programs’. In: *International Journal of Man-Machine Studies* 21.1 (1984), pp. 31–48 (cit. on p. 235).
- [GHKH08] H. Glaesmer, J. Hoyer, J. Klotsche, P. Y. Herzberg. ‘Die deutsche version des Life-Orientation-Tests (LOT-R) zum dispositionellen Optimismus und Pessimismus’. In: *Zeitschrift für Gesundheitspsychologie* 16.1 (2008), pp. 26–31 (cit. on p. 175).
- [Gil91] D. J. Gilmore. ‘Models of debugging’. In: *Acta Psychologica* 78.1 (1991), pp. 151–172 (cit. on pp. 37, 46, 47, 64).
- [GIY+17a] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K.-C. Yeh, J. Cappos. ‘Understanding Misunderstandings in Source Code’. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2017*. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 129–139 (cit. on p. 24).
- [GIY+17b] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K.-C. Yeh, J. Cappos. ‘Understanding misunderstandings in source code’. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. New York, New York, USA: ACM Press, 2017, pp. 129–139 (cit. on p. 170).
- [GLFW21] D. Graziotin, P. Lenberg, R. Feldt, S. Wagner. ‘Psychometrics in behavioral software engineering: A methodological introduction with guidelines’. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.1 (2021), pp. 1–36 (cit. on p. 20).
- [God21] P. Godfrey-Smith. *Theory and Reality: An Introduction to the Philosophy of Science, Second Edition*. University of Chicago Press, 2021 (cit. on pp. 21–24, 26).
- [Gué09] Y.-G. Guéhéneuc. ‘A theory of program comprehension: Joining vision science and program comprehension’. In: *International Journal of Software Science and Computational Intelligence (IJSSCI)* 1.2 (2009), pp. 54–72 (cit. on p. 48).
- [GWA14] D. Graziotin, X. Wang, P. Abrahamsson. ‘Happy software developers solve problems better: psychological measurements in empirical software engineering’. In: *PeerJ* 2 (2014), e289 (cit. on p. 175).

- [GWF+10] A. L. Geers, J. A. Wellman, S. L. Fowler, S. G. Helfer, C. R. France. ‘Dispositional optimism predicts placebo analgesia’. In: *The Journal of Pain* 11.11 (2010), pp. 1165–1171 (cit. on p. 170).
- [GZFC18] D. Gopstein, H. H. Zhou, P. Frankl, J. Cappos. ‘Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild’. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 281–291 (cit. on p. 24).
- [Hau06] N. C. Haugen. ‘An empirical study of using planning poker for user story estimation’. In: *AGILE 2006 (AGILE’06)*. IEEE. 2006, 9–pp (cit. on p. 169).
- [HCKH14] E. Hassler, J. C. Carver, N. A. Kraft, D. Hale. ‘Outcomes of a Community Workshop to Identify and Rank Barriers to the Systematic Literature Review Process’. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’14. London, England, United Kingdom: Association for Computing Machinery, 2014 (cit. on p. 214).
- [HCM02] D. Hendrix, J. H. Cross, S. Maghsoodloo. ‘The effectiveness of control structure diagrams in source code comprehension activities’. In: *IEEE Transactions on Software Engineering* 28.5 (2002), pp. 463–477 (cit. on p. 234).
- [HD17] E. Harth, P. Dugerdil. ‘Program Understanding Models: An Historical Overview and a Classification.’ In: *ICSOFT*. 2017, pp. 402–413 (cit. on p. 63).
- [Hil12] M. Hilbert. ‘Toward a synthesis of cognitive biases: how noisy information processing can bias human decision making.’ In: *Psychological bulletin* 138.2 (2012), p. 211 (cit. on p. 165).
- [HKS17] P. den Heijer, W. Koole, C. J. Stettina. ‘Don’t Forget to Breathe: A Controlled Trial of Mindfulness Practices in Agile Project Teams’. In: *Agile Processes in Software Engineering and Extreme Programming*. Cham: Springer International Publishing, 2017, pp. 103–118 (cit. on p. 168).

- [HLHF22] A. Heinonen, B. Lehtelä, A. Hellas, F. Fagerholm. ‘Synthesizing Research on Programmers’ Mental Models of Programs, Tasks and Concepts—a Systematic Literature Review’. In: *arXiv preprint arXiv:2212.07763* (2022) (cit. on p. 68).
- [Hol79] S. Holm. ‘A Simple Sequentially Rejective Multiple Test Procedure’. In: *Scandinavian Journal of Statistics* 6.2 (1979) (cit. on pp. 146, 151).
- [HRS11] J. F. Hair, C. M. Ringle, M. Sarstedt. ‘PLS-SEM: Indeed a Silver Bullet’. In: *Journal of Marketing Theory and Practice* 19.2 (Apr. 2011), pp. 139–152 (cit. on p. 201).
- [HRSR19] J. F. Hair, J. J. Risher, M. Sarstedt, C. M. Ringle. ‘When to use and how to report the results of PLS-SEM’. In: *European Business Review* 31.1 (2019), pp. 2–24 (cit. on p. 200).
- [HSH17] J. Hofmeister, J. Siegmund, D. V. Holt. ‘Shorter identifier names take longer to comprehend’. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 217–227 (cit. on p. 170).
- [Hun97] M. Hunt. *How science takes stock: The story of meta-analysis*. Russell Sage Foundation, 1997 (cit. on p. 250).
- [HZ86] W. E. Hall, S. H. Zweben. ‘The cloze procedure and software comprehensibility measurement’. In: *IEEE Transactions on Software Engineering* SE-12.5 (1986), pp. 608–623 (cit. on pp. 61, 75, 235).
- [Ico21] Icons from the Noun Project: ‘Brain’ and ‘classroom’ by Dairy Free Design. 2021. URL: <https://thenounproject.com/> (cit. on p. 137).
- [ICP19] ICPC Organization Committee. *ICPC 2019 - Replications*. 2019. URL: <https://web.archive.org/web/20221210145046/https://conf.researchr.org/track/icpc-2019/icpc-2019-replications> (cit. on p. 209).
- [Ise88] E. R. Iselin. ‘Conditional statements, looping constructs, and program comprehension: an experimental study’. In: *International Journal of Man-Machine Studies* 28.1 (1988), pp. 45–66 (cit. on pp. 61, 235).
- [ISO11] ISO/IEC. *ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. Geneva, CH, 2011 (cit. on p. 36).

- [Jaw11] W. Jaworski. *Philosophy of mind: A comprehensive introduction*. John Wiley & Sons, 2011 (cit. on pp. 30, 68).
- [JCP08] A. Jedlitschka, M. Ciolkowski, D. Pfahl. ‘Reporting experiments in software engineering’. In: *Guide to advanced empirical software engineering*. Springer, 2008, pp. 201–228 (cit. on pp. 28, 110, 111, 135, 171, 254).
- [JDK05] M. Jorgensen, T. Dyba, B. Kitchenham. ‘Teaching evidence-based software engineering to university students’. In: *11th IEEE International Software Metrics Symposium (METRICS’05)*. IEEE, 2005, 8–pp (cit. on p. 214).
- [JF17] A. Jbara, D. G. Feitelson. ‘How programmers read regular code: a controlled experiment using eye tracking’. In: *Empirical Software Engineering* 22.3 (2017), pp. 1440–1477 (cit. on p. 236).
- [JM01] N. Juristo, A. M. Moreno. *Basics of Software Engineering Experimentation*. Boston, MA: Springer US, 2001, pp. 12–26 (cit. on pp. 72, 91).
- [Jov15] V. Jovanović. ‘Beyond the PANAS: Incremental validity of the Scale of Positive and Negative Experience (SPANE) in relation to well-being’. In: *Personality and Individual Differences* 86 (2015), pp. 487–491 (cit. on p. 175).
- [JW12] S. Jalali, C. Wohlin. ‘Systematic literature studies: Database searches vs. backward snowballing’. In: *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2012, pp. 29–38 (cit. on p. 78).
- [JWAL21] S. Jessup, S. M. Willis, G. Alarcon, M. Lee. ‘Using Eye-Tracking Data to Compare Differences in Code Comprehension and Code Perceptions between Expert and Novice Programmers’. In: *Proceedings of the 54th Hawaii International Conference on System Sciences*. Ed. by T. Bui. Proceedings of the Annual Hawaii International Conference on System Sciences. Hawaii International Conference on System Sciences, 2021 (cit. on p. 234).

- [KBB15] B. A. Kitchenham, D. Budgen, P. Brereton. *Evidence-Based Software Engineering and Systematic Reviews*. Vol. 4. Chapman & Hall / CRC Innovations in Software Engineering and Software Development Series. Boca Raton: CRC Press, 2015 (cit. on pp. 16, 214).
- [KBGW16] Z. Karimi, A. Baraani-Dastjerdi, N. Ghasem-Aghaee, S. Wagner. ‘Links between the personalities, styles and performance in computer programming’. In: *Journal of Systems and Software* 111 (2016), pp. 228–241 (cit. on p. 135).
- [KC07] B. Kitchenham, S. Charters. *Guidelines for performing Systematic Literature reviews in Software Engineering*. Tech. rep. Keele, UK: School of Computer Science and Mathematics, Keele University, 2007, p. 65 (cit. on p. 75).
- [KDJ04] B. A. Kitchenham, T. Dyba, M. Jorgensen. ‘Evidence-based software engineering’. In: *Proceedings. 26th International Conference on Software Engineering*. IEEE. 2004, pp. 273–281 (cit. on pp. 190, 214).
- [Kee+07] S. Keele et al. *Guidelines for performing systematic literature reviews in software engineering*. Tech. rep. Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007 (cit. on p. 83).
- [Kin20] A. Kind. *Philosophy of Mind: The Basics*. Routledge, 2020 (cit. on pp. 30, 68).
- [Kin98] W. Kintsch. *Comprehension: A paradigm for cognition*. Cambridge University Press, 1998 (cit. on pp. 36, 122).
- [KLH13] L. Kreuzpointner, H. Lukesch, W. Horn. *Leistungsprüfsystem 2. LPS-2. Manual*. Göttingen: Hogrefe, 2013 (cit. on pp. 132, 133, 140–142, 157).
- [KMB20] B. Kitchenham, L. Madeyski, P. Brereton. ‘Meta-Analysis for Families of Experiments in Software Engineering: A Systematic Review and Reproducibility and Validity Assessment’. In: *Empirical Software Engineering* 25.1 (Jan. 2020), pp. 353–401 (cit. on p. 72).
- [KMG13] T. Kanij, R. Merkel, J. Grundy. ‘An empirical study of the effects of personality on software testing’. In: *2013 26th International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE. 2013, pp. 239–248 (cit. on pp. 130, 135).

- [KPB+10] B. Kitchenham, R. Pretorius, D. Budgen, O. P. Brereton, M. Turner, M. Niazi, S. Linkman. ‘Systematic literature reviews in software engineering—a tertiary study’. In: *Information and software technology* 52.8 (2010), pp. 792–805 (cit. on p. 214).
- [KPP+02] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, J. Rosenberg. ‘Preliminary guidelines for empirical research in software engineering’. In: *IEEE Transactions on software engineering* 28.8 (2002), pp. 721–734 (cit. on p. 188).
- [KR91] J. Koenemann, S. P. Robertson. ‘Expert problem solving strategies for program comprehension’. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1991, pp. 125–130 (cit. on p. 42).
- [KU03] A. J. Ko, B. Uttl. ‘Individual differences in program comprehension strategies in unfamiliar programming systems’. In: *11th IEEE International Workshop on Program Comprehension, 2003*. IEEE. 2003, pp. 175–184 (cit. on p. 134).
- [Kuh62] T. S. Kuhn. *The structure of scientific revolutions*. University of Chicago Press, 1962 (cit. on pp. 24, 25).
- [LA18] K. Lee, M. C. Ashton. ‘Psychometric properties of the HEXACO-100’. In: *Assessment* 25.5 (2018), pp. 543–556 (cit. on p. 141).
- [Lak76] I. Lakatos. ‘Falsification and the Methodology of Scientific Research Programmes’. In: *Can Theories be Refuted? Essays on the Duhem-Quine Thesis*. Ed. by S. G. Harding. Dordrecht: Springer Netherlands, 1976, pp. 205–259 (cit. on p. 24).
- [Lau77] L. Laudan. *Progress and its problems: Towards a theory of scientific growth*. Vol. 282. University of California Press, 1977 (cit. on p. 24).
- [LBW13] F. Li, X. Bai, Y. Wang. ‘The Scale of Positive and Negative Experience (SPAN): psychometric properties and normative data in a large Chinese sample.’ In: *PloS one* 8.4 (Apr. 2013), pp. 1–9 (cit. on p. 175).
- [Let87] S. Letovsky. ‘Cognitive processes in program comprehension’. In: *Journal of Systems and software* 7.4 (1987), pp. 325–339 (cit. on pp. 41–43).

- [LFW15] P. Lenberg, R. Feldt, L. G. Wallgren. ‘Behavioral software engineering: A definition and systematic literature review’. In: *Journal of Systems and software* 107 (2015), pp. 15–37 (cit. on p. 20).
- [LGHM07] T. D. LaToza, D. Garlan, J. D. Herbsleb, B. A. Myers. ‘Program comprehension as fact finding’. In: *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 361–370 (cit. on p. 80).
- [LL02] E. A. Locke, G. P. Latham. ‘Building a practically useful theory of goal setting and task motivation: A 35-year odyssey.’ In: *American psychologist* 57.9 (2002), p. 705 (cit. on p. 198).
- [LLA01] F. R. Lang, O. Lüdtke, J. B. Asendorpf. ‘Testgüte und psychometrische Äquivalenz der deutschen Version des Big Five Inventory (BFI) bei jungen, mittelalten und alten Erwachsenen’. In: *Diagnostica* 47.3 (2001), pp. 111–121 (cit. on p. 175).
- [LMH+16] S. Lee, A. Matteson, D. Hooshyar, S. Kim, J. Jung, G. Nam, H. Lim. ‘Comparing programming language comprehension between novice and expert programmers using eeg analysis’. In: *2016 IEEE 16th international conference on bioinformatics and bioengineering (BIBE)*. IEEE. 2016, pp. 350–355 (cit. on p. 52).
- [LPLS87] D. C. Littman, J. Pinto, S. Letovsky, E. Soloway. ‘Mental models and software maintenance’. In: *Journal of Systems and Software* 7.4 (1987), pp. 341–355 (cit. on p. 42).
- [LS86] S. Letovsky, E. Soloway. ‘Delocalized Plans and Program Comprehension’. In: *IEEE Software* 3.3 (1986), pp. 41–49 (cit. on p. 37).
- [LW13] B. Latour, S. Woolgar. *Laboratory life: The construction of scientific facts*. Princeton University Press, 2013 (cit. on p. 26).
- [MB19] I. McChesney, R. Bond. ‘Eye tracking analysis of computer program comprehension in programmers with dyslexia’. In: *Empirical Software Engineering* 24.3 (2019), pp. 1109–1154 (cit. on p. 53).

- [McC20] R. R. McCrae. ‘The Five-Factor Model of personality traits: consensus and controversy’. In: *The Cambridge Handbook of Personality Psychology*. 2nd. Cambridge University Press, 2020 (cit. on p. 141).
- [MD07] T. McElroy, K. Dowd. ‘Susceptibility to anchoring effects: How openness-to-experience influences responses to anchoring cues’. In: *Judgment and Decision making* 2.1 (2007), pp. 48–53 (cit. on p. 186).
- [ME05] T. Mussweiler, B. Englich. ‘Subliminal anchoring: Judgmental consequences and underlying mechanisms’. In: *Organizational Behavior and Human Decision Processes* 98.2 (2005), pp. 133–143 (cit. on p. 188).
- [Mera] Merriam-Webster. ‘Experiment’. In: *Merriam-Webster.com dictionary*. URL: <https://www.merriam-webster.com/dictionary/experimentation> (visited on 12/26/2022) (cit. on p. 66).
- [Merb] Merriam-Webster. ‘Validity’. In: *Merriam-Webster.com dictionary*. URL: <https://www.merriam-webster.com/dictionary/validity> (visited on 11/09/2022) (cit. on p. 28).
- [MGWS20] D. Mendez, D. Graziotin, S. Wagner, H. Seibold. ‘Open science in software engineering’. In: *Contemporary Empirical Methods in Software Engineering*. Cham: Springer International Publishing, 2020, pp. 477–501 (cit. on p. 313).
- [MJ92] R. R. McCrae, O. P. John. ‘An introduction to the five-factor model and its applications’. In: *Journal of personality* 60.2 (1992), pp. 175–215 (cit. on p. 175).
- [MLA+19] F. Medeiros, G. Lima, G. Amaral, S. Apel, C. Kästner, M. Ribeiro, R. Gheyi. ‘An investigation of misunderstanding code patterns in C open-source software projects’. In: *Empirical Software Engineering* 24.4 (2019), pp. 1693–1726 (cit. on p. 234).
- [MLMD01] J. Maletic, J. Leigh, A. Marcus, G. Dunlap. ‘Visualizing object-oriented software in virtual reality’. In: *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*. 2001, pp. 26–35 (cit. on p. 58).
- [MM12] S. Mealin, E. Murphy-Hill. ‘An exploratory study of blind software developers’. In: *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2012, pp. 71–74 (cit. on pp. 56, 57).

- [MML15] R. Minelli, A. Mocci, M. Lanza. ‘I know what you did last summer-an investigation of how developers spend their time’. In: *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE. IEEE, 2015, pp. 25–35 (cit. on pp. 15, 50, 71).
- [MMV+21] D. Moreno-Lumbreras, R. Minelli, A. Villaverde, J.M. González-Barahona, M. Lanza. ‘CodeCity: On-Screen or in Virtual Reality?’ In: *2021 Working Conference on Software Visualization (VISSOFT)*. 2021, pp. 12–22 (cit. on p. 59).
- [MrV22] C. MrValdez. *How can you program if you’re blind?* Mar. 29, 2022. URL: <https://stackoverflow.com/questions/118984/how-can-you-program-if-youre-blind> (visited on 01/16/2023) (cit. on p. 56).
- [MST+18] R. Mohanani, I. Salman, B. Turhan, P. Rodríguez, P. Ralph. ‘Cognitive biases in software engineering: a systematic mapping study’. In: *IEEE Transactions on Software Engineering* (2018) (cit. on pp. 60, 165, 169, 189, 208).
- [MTRK14] W. Maalej, R. Tiarks, T. Roehm, R. Koschke. ‘On the Comprehension of Program Comprehension’. In: 23.4 (2014) (cit. on pp. 45, 49, 50).
- [Mur13] J. Murray. ‘Likert data: what to use, parametric or non-parametric?’ In: *International Journal of Business and Social Science* 4.11 (2013) (cit. on p. 183).
- [MW20] K. Mindermann, S. Wagner. ‘Fluid intelligence doesn’t matter! effects of code examples on the usability of crypto APIs’. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 2020, pp. 306–307 (cit. on p. 134).
- [MWEJ09] D. L. Morton, A. Watson, W. El-Deredy, A. K. Jones. ‘Reproducibility of placebo analgesia: Effect of dispositional optimism’. In: *Pain* 146.1-2 (2009), pp. 194–198 (cit. on p. 170).
- [MWGW22] M. Muñoz Barón, M. Wyrich, D. Graziotin, S. Wagner. *Replication package: Evidence Profiles for Validity Threats in Program Comprehension Experiments*. Zenodo, Aug. 2022. URL: <https://doi.org/10.5281/zenodo.7038907> (cit. on pp. 225, 228, 314).

- [MWGW23] M. Muñoz Barón, M. Wyrich, D. Graziotin, S. Wagner. ‘Evidence Profiles for Validity Threats in Program Comprehension Experiments’. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. IEEE, 2023 (cit. on pp. [16](#), [17](#), [32](#), [33](#), [61](#), [212](#), [314](#)).
- [MWI+07] R. McCarney, J. Warner, S. Iliffe, R. Van Haselen, M. Griffin, P. Fisher. ‘The Hawthorne Effect: a randomised, controlled trial’. In: *BMC medical research methodology* 7.1 (2007), p. 30 (cit. on pp. [60](#), [171](#), [187](#)).
- [MWW20] M. Muñoz Barón, M. Wyrich, S. Wagner. ‘An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability’. In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM ’20. Bari, Italy: Association for Computing Machinery, 2020 (cit. on pp. [24](#), [78](#), [139](#), [164](#), [170](#), [171](#), [174](#), [178](#), [179](#), [197](#), [251](#)).
- [NAG19] M. Nilson, V. Antinyan, L. Gren. ‘Do Internal Software Quality Tools Measure Validated Metrics?’ In: *International Conference on Product-Focused Software Process Improvement*. Vol. 11915. Lecture Notes in Computer Science. Cham, Switzerland: Springer, 2019, pp. 637–648 (cit. on pp. [164](#), [170](#)).
- [NP15] M. Nosál, J. Porubán. ‘Program comprehension with four-layered mental model’. In: *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*. IEEE. 2015, pp. 1–4 (cit. on p. [49](#)).
- [OB17] P. A. Orlov, R. Bednarik. ‘The Role of Extrafoveal Vision in Source Code Comprehension’. In: *Perception* 46.5 (2017), pp. 541–565 (cit. on p. [234](#)).
- [OBMC20] D. Oliveira, R. Bruno, F. Madeiral, F. Castor. ‘Evaluating code readability and legibility: An examination of human-centric studies’. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. IEEE, 2020, pp. 348–359 (cit. on pp. [72](#), [74](#), [103](#), [205](#)).
- [OBS04] M. P. O’Brien, J. Buckley, T. M. Shaft. ‘Expectation-based, inference-based, and bottom-up software comprehension’. In: *Journal of Software Maintenance and Evolution: Research and Practice* 16.6 (2004), pp. 427–447 (cit. on pp. [19](#), [48](#)).

- [OC09] B. J. Oates, G. Capper. 'Using systematic reviews and evidence-based software engineering with masters students'. In: *Proceedings of the 13th international conference on Evaluation and Assessment in Software Engineering*. BCS Learning & Development Ltd., 2009, pp. 79–87 (cit. on p. 214).
- [OMP18] R. Oberhauser, A. Matic, C. Pogolski. 'HyDE: A Hyper-Display Environment in Mixed and Virtual Reality and its Application in a Software Development Case Study'. In: *International Journal on Advances in Software* 11.1 & 2 (2018), pp. 195–204 (cit. on pp. 58, 59).
- [PAL+13] M. Peciña, H. Azhar, T. M. Love, T. Lu, B. L. Fredrickson, C. S. Stohler, J.-K. Zubieta. 'Personality trait predictors of placebo analgesia and neurobiological correlates'. In: *Neuropsychopharmacology* 38.4 (2013), pp. 639–646 (cit. on p. 170).
- [PBV12] M. A. Pourhoseingholi, A. R. Baghestani, M. Vahedi. 'How to control confounding effects by statistical analysis'. In: *Gastroenterology and hepatology from bed to bench* 5.2 (2012), p. 79 (cit. on p. 131).
- [PC15] P. D. Pearson, G. N. Cervetti. 'Fifty years of reading comprehension theory and practice'. In: *Research-based practices for teaching Common Core literacy* (2015), pp. 1–24 (cit. on p. 53).
- [PCE17] R. F. Paige, J. Cabot, N. A. Ernst. 'Foreword to the special section on negative results in software engineering'. In: *Empirical Software Engineering* 22.5 (2017), pp. 2453–2456 (cit. on p. 237).
- [Pei18] N. Peitek. 'A neuro-cognitive perspective of program comprehension'. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. 2018, pp. 496–499 (cit. on p. 51).
- [Pen87] N. Pennington. 'Stimulus structures and mental representations in expert comprehension of computer programs'. In: *Cognitive Psychology* 19.3 (1987), pp. 295–341 (cit. on pp. 37, 41, 43, 53, 64).
- [Pet13] M. Petrescu. 'Marketing research using single-item indicators in structural equation models'. In: *Journal of Marketing Analytics* 1.2 (2013), pp. 99–117 (cit. on p. 207).

- [PFMM08] K. Petersen, R. Feldt, S. Mujtaba, M. Mattsson. ‘Systematic mapping studies in software engineering’. In: *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*. Swindon, GBR: BCS Learning & Development Ltd., 2008, pp. 1–10 (cit. on p. 75).
- [PG13] K. Petersen, C. Gencel. ‘Worldviews, Research Methods, and their Relationship to Validity in Empirical Software Engineering Research’. In: *2013 Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement*. 2013, pp. 81–89 (cit. on pp. 18, 28, 215).
- [PHD11] D. Posnett, A. Hindle, P. Devanbu. ‘A simpler model of software readability’. In: *Proceedings of the 8th Working Conference on Mining Software repositories*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 73–82 (cit. on p. 79).
- [Pop05] K. Popper. *The Logic of Scientific Discovery*. Routledge Classics. Taylor & Francis, 2005 (cit. on pp. 21, 22).
- [PSA+18] N. Peitek, J. Siegmund, S. Apel, C. Kästner, C. Parnin, A. Bethmann, T. Leich, G. Saake, A. Brechmann. ‘A Look into Programmers’ Heads’. In: *IEEE Transactions on Software Engineering* (2018), pp. 1–1 (cit. on pp. 132, 234).
- [PSA20] N. Peitek, J. Siegmund, S. Apel. ‘What Drives the Reading Order of Programmers? An Eye Tracking Study’. In: *Proceedings of the 28th International Conference on Program Comprehension*. ICPC ’20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 342–353 (cit. on p. 53).
- [PSE04] J. Paulson, G. Succi, A. Eberlein. ‘An empirical study of open-source and closed-source software products’. In: *IEEE Transactions on Software Engineering* 30.4 (2004), pp. 246–256 (cit. on p. 123).
- [PSP+18] N. Peitek, J. Siegmund, C. Parnin, S. Apel, J. C. Hofmeister, A. Brechmann. ‘Simultaneous measurement of program comprehension with fmri and eye tracking: A case study’. In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2018, pp. 1–10 (cit. on pp. 52, 132, 186).

- [PVI+18] V. Potluri, P. Vaithilingam, S. Iyengar, Y. Vidya, M. Swaminathan, G. Srinivasa. ‘CodeTalk: Improving Programming Environment Accessibility for Visually Impaired Developers’. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI ’18. Montreal QC, Canada: Association for Computing Machinery, 2018, pp. 1–11 (cit. on p. 58).
- [PVK15] K. Petersen, S. Vakkalanka, L. Kuzniarz. ‘Guidelines for conducting systematic mapping studies in software engineering: An update’. In: *Information and Software Technology* 64 (2015), pp. 1–18 (cit. on p. 75).
- [R C21] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2021. URL: <https://www.R-project.org/> (cit. on p. 201).
- [Ral11] P. Ralph. ‘Toward a theory of debiasing software development’. In: *EuroSymposium on Systems Analysis and Design*. Springer, 2011, pp. 92–105 (cit. on p. 208).
- [RB08] A. Rainer, S. Beecham. ‘A follow-up empirical evaluation of evidence based software engineering by undergraduate students’. In: *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*. 2008, pp. 1–10 (cit. on p. 214).
- [RB22] P. Ralph, S. Baltes. ‘Paving the Way for Mature Secondary Research: The Seven Types of Literature Review’. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 1632–1636 (cit. on p. 249).
- [RC97] V. Rajlich, G. S. Cowan. ‘Towards standard for experiments in program comprehension’. In: *Proceedings Fifth International Workshop on Program Comprehension. IWPC’97*. IEEE. IEEE, 1997, pp. 160–161 (cit. on pp. 61, 75, 215).
- [RCE+19] S. Romano, N. Capece, U. Erra, G. Scanniello, M. Lanza. ‘On the use of virtual reality in software visualization: The case of the city metaphor’. In: *Information and Software Technology* 114 (2019), pp. 92–106 (cit. on pp. 58, 59).

- [RCT16] B. A. Rogowsky, B. M. Calhoun, P. Tallal. ‘Does modality matter? The effects of reading, listening, and dual modality on comprehension’. In: *Sage Open* 6.3 (2016), p. 2158244016669550 (cit. on p. 48).
- [RD03] F. J. Roethlisberger, W. J. Dickson. *Management and the Worker*. Vol. 5. Psychology press, 2003 (cit. on pp. 60, 171, 187).
- [RDC21] S. Ray, N. P. Danks, A. Calero Valdez. *semnr: Building and Estimating Structural Equation Models*. R package version 2.2.1. 2021. URL: <https://CRAN.R-project.org/package=semnr> (cit. on p. 201).
- [RF10] B. Robinson, P. Francis. ‘Improving Industrial Adoption of Software Engineering Research: A Comparison of Open and Closed Source Software’. In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’10. Bolzano-Bozen, Italy: Association for Computing Machinery, 2010 (cit. on p. 123).
- [RHB03] A. Rainer, T. Hall, N. Baddoo. ‘Persuading developers to "buy into" software process improvement: a local opinion and empirical evidence’. In: *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings*. IEEE. 2003, pp. 326–335 (cit. on p. 190).
- [RHS17] T. Rahm, E. Heise, M. Schuldt. ‘Measuring the frequency of emotions—validation of the Scale of Positive and Negative Experience (SPAN) in Germany’. In: *PloS one* 12.2 (2017), e0171288 (cit. on p. 175).
- [Ris+86] R. S. Rist et al. ‘Plans in programming: definition, demonstration, and development’. In: *Empirical studies of programmers*. 1986, pp. 28–47 (cit. on pp. 41, 48).
- [RJ66] R. Rosenthal, L. Jacobson. ‘Teachers’ expectancies: Determinants of pupils’ IQ gains’. In: *Psychological reports* 19.1 (1966), pp. 115–118 (cit. on p. 60).
- [RMI+17] L. Rozenkrantz, A. E. Mayo, T. Ilan, Y. Hart, L. Noy, U. Alon. ‘Placebo can enhance creativity’. In: *PLOS ONE* 12.9 (Sept. 2017), pp. 1–15 (cit. on pp. 168, 185).

- [Roh18] J.M. Rohrer. ‘Thinking clearly about correlations and causation: Graphical causal models for observational data’. In: *Advances in Methods and Practices in Psychological Science* 1.1 (2018), pp. 27–42 (cit. on pp. 136, 142, 144–146).
- [RS21] D. Russo, K.-J. Stol. ‘PLS-SEM for Software Engineering Research: An Introduction and Survey’. In: *ACM Computing Surveys (CSUR)* 54.4 (2021), pp. 1–38 (cit. on p. 200).
- [RSP+07] N. Raz, E. Striem, G. Pundak, T. Orlov, E. Zohary. ‘Superior Serial Memory in the Blind: A Case of Cognitive Compensatory Adjustment’. In: *Current Biology* 17.13 (2007), pp. 1129–1133 (cit. on p. 57).
- [RSS12] Ringle, Sarstedt, Straub. ‘Editor’s Comments: A Critical Look at the Use of PLS-SEM in “MIS Quarterly”’. In: *MIS Quarterly* 36.1 (2012), p. iii (cit. on p. 207).
- [RT18a] P. Ralph, E. Tempero. ‘Construct Validity in Software Engineering Research and Software Metrics’. In: *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. EASE’18*. Christchurch, New Zealand: Association for Computing Machinery, 2018, pp. 13–23 (cit. on pp. 29, 61, 62).
- [RT18b] T. V. Ribeiro, G. H. Travassos. ‘Attributes Influencing the Reading and Comprehension of Source Code – Discussing Contradictory Evidence’. In: *CLEI Electronic Journal* 21.1 (2018) (cit. on p. 234).
- [Rud53] R. Rudner. ‘The Scientist Qua Scientist Makes Value Judgments’. In: *Philosophy of Science* 20.1 (1953), pp. 1–6 (cit. on p. 23).
- [RW02] V. Rajlich, N. Wilde. ‘The role of concepts in program comprehension’. In: *Proceedings 10th International Workshop on Program Comprehension*. IEEE. 2002, pp. 271–278 (cit. on p. 48).
- [Ryc12] R. M. Ryckman. *Theories of personality*. Cengage Learning, 2012 (cit. on p. 134).
- [SAA+02] D. I. Sjøberg, B. Anda, E. Arisholm, T. Dyba, M. Jorgensen, A. Karahasanovic, E. F. Koren, M. Vokác. ‘Conducting realistic experiments in software engineering’. In: *Proceedings international symposium on empirical software engineering*. IEEE. 2002, pp. 17–26 (cit. on pp. 188, 199).

- [SAA+15] A. Shargabi, S. A. Aljunid, M. Annamalai, S. Mohamed Shuhidan, A. Mohd Zin. 'Program comprehension levels of abstraction for novices'. In: *2015 International Conference on Computer, Communications, and Control Technology (I4CT)*. 2015, pp. 211–215 (cit. on p. 49).
- [SAY+18] J. Souza, A. A. Araújo, I. Yeltsin, R. Saraiva, P. Soares. 'On the Placebo Effect in Interactive SBSE: A Preliminary Study'. In: *Search-Based Software Engineering*. Cham: Springer International Publishing, 2018, pp. 370–376 (cit. on p. 168).
- [SB22] D. I. Sjøberg, G. R. Bergersen. 'Construct Validity in Software Engineering'. In: *IEEE Transactions on Software Engineering* 49.3 (2022), pp. 1374–1396 (cit. on pp. 29, 62, 124).
- [SBV+19] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshy-vanyk, R. Oliveto. 'Automatically Assessing Code Understandability'. In: *IEEE Transactions on Software Engineering* (2019) (cit. on pp. 158, 164, 170, 176, 178).
- [SC13] A. J. Silva, A. Caetano. 'Validation of the Flourishing Scale and Scale of Positive and Negative Experience in Portugal'. In: *Social Indicators Research* 110.2 (2013), pp. 469–478 (cit. on p. 175).
- [SCB94] M. F. Scheier, C. S. Carver, M. W. Bridges. 'Distinguishing optimism from neuroticism (and trait anxiety, self-mastery, and self-esteem): a reevaluation of the Life Orientation Test.' In: *Journal of personality and social psychology* 67.6 (1994), p. 1063 (cit. on p. 175).
- [Sch08] C. Schulte. 'Block Model: An Educational Model of Program Comprehension as a Tool for a Scholarly Approach to Teaching'. In: *Proceedings of the Fourth International Workshop on Computing Education Research. ICER '08*. Sydney, Australia: Association for Computing Machinery, 2008, pp. 149–160 (cit. on p. 49).
- [SCT+10] C. Schulte, T. Clear, A. Taherkhani, T. Busjahn, J. H. Paterson. 'An Introduction to Program Comprehension for Computer Science Educators'. In: *Proceedings of the 2010 ITiCSE Working Group Reports. ITiCSE-WGR '10*. Ankara, Turkey: Association for Computing Machinery, 2010, pp. 65–86 (cit. on p. 49).

- [SE84] E. Soloway, K. Ehrlich. ‘Empirical studies of programming knowledge’. In: *IEEE Transactions on software engineering* 5 (1984), pp. 595–609 (cit. on pp. 40, 41, 43, 48).
- [SF18] K.-J. Stol, B. Fitzgerald. ‘The ABC of Software Engineering Research’. In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (2018) (cit. on p. 66).
- [SFM00] A. C. Smith, J. M. Francioni, S. D. Matzek. ‘A Java Programming Tool for Students with Visual Disabilities’. In: Assets ’00. Arlington, Virginia, USA: Association for Computing Machinery, 2000, pp. 142–148 (cit. on p. 58).
- [SFM99] M.-A. Storey, F. Fracchia, H. Müller. ‘Cognitive design elements to support the construction of a mental model during software exploration’. In: *Journal of Systems and Software* 44.3 (1999), pp. 171–185 (cit. on pp. 42, 68, 69).
- [Sha68] A. K. Shapiro. ‘Semantics of the placebo’. In: *Psychiatric Quarterly* 42.4 (1968), pp. 653–695 (cit. on pp. 165, 167).
- [SHLW21] Z. Sharafi, Y. Huang, K. Leach, W. Weimer. ‘Toward an Objective Measure of Developers’ Cognitive Activities’. In: *ACM Trans. Softw. Eng. Methodol.* 30.3 (2021) (cit. on pp. 51, 53).
- [SHM+09] A. Stefik, A. Haywood, S. Mansoor, B. Dunda, D. Garcia. ‘SODBeans’. In: *2009 IEEE 17th International Conference on Program Comprehension*. 2009, pp. 293–294 (cit. on p. 58).
- [Shn77] B. Shneiderman. ‘Measuring computer program quality and comprehension’. In: *International Journal of Man-Machine Studies* 9.4 (1977), pp. 465–478 (cit. on pp. 36, 61, 75, 120, 122).
- [Sie16] J. Siegmund. ‘Program comprehension: Past, present, and future’. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. IEEE. IEEE, 2016, pp. 13–20 (cit. on pp. 54, 72, 73, 130, 192).
- [SKA+14] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, A. Brechmann. ‘Understanding understanding source code with functional magnetic resonance imaging’. In: *Proceedings of the 36th international conference on software engineering*. 2014, pp. 378–389 (cit. on p. 52).

- [SKL+14] J. Siegmund, C. Kästner, J. Liebig, S. Apel, S. Hanenberg. ‘Measuring and modeling programming experience’. In: *Empirical Software Engineering* 19.5 (2014), pp. 1299–1334 (cit. on pp. [140](#), [144](#), [155](#), [156](#), [234](#), [247](#)).
- [SKR19] M. Steinbeck, R. Koschke, M. O. Rudel. ‘Comparing the EvoStreets Visualization Technique in Two-and Three-Dimensional Environments A Controlled Experiment’. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 2019, pp. 231–242 (cit. on pp. [58](#), [59](#)).
- [SKSL17a] I. Schröter, J. Krüger, J. Siegmund, T. Leich. ‘Comprehending Studies on Program Comprehension’. In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 2017, pp. 308–311 (cit. on pp. [16](#), [213](#)).
- [SKSL17b] I. Schröter, J. Krüger, J. Siegmund, T. Leich. ‘Comprehending studies on program comprehension’. In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, IEEE, 2017, pp. 308–311 (cit. on p. [74](#)).
- [SM18] A. Schreiber, M. Misiak. ‘Visualizing software architectures in virtual reality with an island metaphor’. In: *International Conference on Virtual, Augmented and Mixed Reality*. Springer. 2018, pp. 168–182 (cit. on p. [58](#)).
- [SM79] B. Shneiderman, R. Mayer. ‘Syntactic/semantic interactions in programmer behavior: A model and experimental results’. In: *International Journal of Computer & Information Sciences* 8.3 (1979), pp. 219–238 (cit. on pp. [19](#), [39](#), [40](#), [43](#), [48](#), [120](#)).
- [SMHB21] V. S. Segura Castillo, L. Merino, G. Hecht, A. Bergel. ‘VR-Based User Interactions to Exploit Infinite Space in Programming Activities’. In: *2021 40th International Conference of the Chilean Computer Science Society (SCCC)*. 2021, pp. 1–5 (cit. on p. [58](#)).
- [SMJ15] I. Salman, A. T. Misirli, N. Juristo. ‘Are students representatives of professionals in software engineering experiments?’ In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 666–676 (cit. on pp. [188](#), [199](#)).

- [SNA09] L. Shoaib, A. Nadeem, A. Akbar. ‘An empirical evaluation of the influence of human personality on exploratory software testing’. In: *2009 IEEE 13th International Multitopic Conference*. IEEE, 2009, pp. 1–6 (cit. on pp. [130](#), [135](#)).
- [SPB+20] J. Siegmund, N. Peitek, A. Brechmann, C. Parnin, S. Apel. ‘Studying programming in the neuroage: just a crazy idea?’ In: *Communications of the ACM* 63.6 (2020), pp. 30–34 (cit. on pp. [51](#), [75](#), [132](#), [186](#)).
- [SPP+17] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, A. Brechmann. ‘Measuring Neural Efficiency of Program Comprehension’. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 140–150 (cit. on p. [52](#)).
- [SRG+96] D. L. Sackett, W. M. Rosenberg, J. M. Gray, R. B. Haynes, W. S. Richardson. *Evidence based medicine: what it is and what it isn't*. 1996 (cit. on p. [213](#)).
- [SS13] R. E. Santos, F. Q. d. Silva. ‘Motivation to Perform Systematic Reviews and their Impact on Software Engineering Practice’. In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 2013, pp. 292–295 (cit. on p. [214](#)).
- [SS15] J. Siegmund, J. Schumann. ‘Confounding parameters on program comprehension: a literature survey’. In: *Empirical Software Engineering* 20.4 (2015), pp. 1159–1192 (cit. on pp. [16](#), [54](#), [55](#), [59](#), [74](#), [111](#), [113](#), [116](#), [130–132](#), [144](#), [145](#), [159–161](#), [171](#), [179](#), [187](#), [212](#), [216](#), [238](#), [241](#)).
- [SSA15] J. Siegmund, N. Siegmund, S. Apel. ‘Views on internal and external validity in empirical software engineering’. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE, IEEE, 2015, pp. 9–19 (cit. on pp. [17](#), [72](#), [130](#), [192](#), [237](#), [254](#), [258](#)).
- [ST92] M. Smith, R. Taffler. ‘Readability and understandability: Different measures of the textual complexity of accounting narrative’. In: *Accounting, Auditing & Accountability Journal* 5.4 (1992), pp. 84–98 (cit. on p. [90](#)).

- [Sta01] T. D. Stanley. ‘Wheat from chaff: Meta-analysis as quantitative literature review’. In: *Journal of economic perspectives* 15.3 (2001), pp. 131–150 (cit. on p. 250).
- [Sta22] Stack Overflow. *Stack Overflow Developer Survey 2022*. June 22, 2022. URL: <https://survey.stackoverflow.co/2022/> (visited on 01/16/2023) (cit. on p. 56).
- [Ste10] A. Stevenson. *Oxford dictionary of English*. Oxford University Press, USA, 2010 (cit. on p. 214).
- [Sum14] K. Sumi. ‘Reliability and Validity of Japanese Versions of the Flourishing Scale and the Scale of Positive and Negative Experience’. In: *Social Indicators Research* 118.2 (2014), pp. 601–615 (cit. on p. 175).
- [SV95] T. M. Shaft, I. Vessey. ‘The relevance of application domain knowledge: The case of computer program comprehension’. In: *Information systems research* 6.3 (1995), pp. 286–299 (cit. on p. 19).
- [TBG+18] Z. Turi, E. Bjørkedal, L. Gunkel, A. Antal, W. Paulus, M. Mittner. ‘Evidence for cognitive placebo and nocebo effects in healthy individuals’. In: *Scientific reports* 8.1 (2018), pp. 1–14 (cit. on pp. 168, 185).
- [TCM+18] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, B. Vasilescu. “Automatically assessing code understandability” reanalyzed: combined metrics matter’. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, IEEE, 2018, pp. 314–318 (cit. on p. 80).
- [Tea94] B. E. Teasley. ‘The effects of naming style and expertise on program comprehension’. In: *International Journal of Human-Computer Studies* 40.5 (1994), pp. 757–770 (cit. on p. 75).
- [Tic00] W. F. Tichy. ‘Hints for reviewing empirical work in software engineering’. In: *Empirical Software Engineering* 5.4 (2000), pp. 309–312 (cit. on p. 188).
- [Tic22] W. F. Tichy. ‘Workings of Science: How Software Engineering Research Became Empirical’. In: *Ubiquity* 2022.July (Aug. 2022) (cit. on p. 259).
- [TK74] A. Tversky, D. Kahneman. ‘Judgment under uncertainty: Heuristics and biases’. In: *science* 185.4157 (1974), pp. 1124–1131 (cit. on pp. 165, 169, 188, 192, 199, 203, 204).

- [TLPH95] W. F. Tichy, P. Lukowicz, L. Prechelt, E. A. Heinz. ‘Experimental evaluation in computer science: A quantitative study’. In: *Journal of Systems and Software* 28.1 (1995), pp. 9–18 (cit. on p. 259).
- [VAJ16] S. Vegas, C. Apa, N. Juristo. ‘Crossover Designs in Software Engineering Experiments: Benefits and Perils’. In: *IEEE Transactions on Software Engineering* 42.2 (2016), pp. 120–135. (Visited on 11/15/2022) (cit. on p. 92).
- [VF21] R. Vieira, K. Farias. ‘On the Usage of Psychophysiological Data in Software Engineering: An Extended Systematic Mapping Study’. In: *arXiv preprint arXiv:2105.14059* (2021) (cit. on pp. 51, 52, 75).
- [VK83] T. A. Van Dijk, W. Kintsch. ‘Strategies of discourse comprehension’. In: (1983) (cit. on p. 41).
- [VV95] A. Von Mayrhauser, A. M. Vans. ‘Program comprehension during software maintenance and evolution’. In: *Computer* 28.8 (1995), pp. 44–55 (cit. on pp. 43–45, 68, 119, 139, 173).
- [WA15] T. D. Wager, L. Y. Atlas. ‘The neuroscience of placebo effects: connecting context, learning and health’. In: *Nature Reviews Neuroscience* 16.7 (2015), pp. 403–418 (cit. on pp. 169, 170).
- [WBW22] M. Wyrich, J. Bogner, S. Wagner. *Replication package: 40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study (v1.0)*. Zenodo, June 2022. URL: <https://doi.org/10.5281/zenodo.6657640> (cit. on pp. 82, 84, 108, 112, 314).
- [WBW23] M. Wyrich, J. Bogner, S. Wagner. ‘40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study’. In: *ACM Computing Surveys* (2023). To appear. Preprint available at <https://arxiv.org/abs/2206.11102> (cit. on pp. 16, 32, 33, 36, 61, 62, 71, 212, 213, 216, 219, 221, 237, 314).
- [Wei16] P. G. Weintraub. ‘The Importance of Publishing Negative Results’. In: *Journal of Insect Science* 16.1 (2016), p. 109 (cit. on p. 237).
- [WGW19] M. Wyrich, D. Graziotin, S. Wagner. ‘A theory on individual characteristics of successful coding challenge solvers’. In: *PeerJ Computer Science* 5 (2019), e173 (cit. on pp. 130, 135, 139, 197).

- [Wie85] S. Wiedenbeck. 'Novice/expert differences in programming skills'. In: *International Journal of Man-Machine Studies* 23.4 (1985), pp. 383–390 (cit. on p. 234).
- [Wie86] S. Wiedenbeck. 'Beacons in computer program comprehension'. In: *International Journal of Man-Machine Studies* 25.6 (1986), pp. 697–709 (cit. on pp. 40, 48).
- [WMG22a] M. Wyrich, L. Merz, D. Graziotin. 'Anchoring Code Understandability Evaluations through Task Descriptions'. In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. ICPC '22. Virtual Event: Association for Computing Machinery, 2022*, pp. 133–140 (cit. on pp. 32, 33, 75, 163, 314).
- [WMG22b] M. Wyrich, L. Merz, D. Graziotin. *Replication Package for 'Anchoring Code Understandability Evaluations Through Task Descriptions'*. Zenodo, Jan. 2022. URL: <https://doi.org/10.5281/zenodo.5877351> (cit. on pp. 197, 314).
- [Woh13] C. Wohlin. 'An Evidence Profile for Software Engineering Research and Practice'. In: *Perspectives on the Future of Software Engineering: Essays in Honor of Dieter Rombach*. Ed. by J. Münch, K. Schmid. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 145–157 (cit. on pp. 212–214, 216, 222, 249).
- [Woh14a] C. Wohlin. 'Guidelines for snowballing in systematic literature studies and a replication in software engineering'. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–10 (cit. on pp. 77–79, 220).
- [Woh14b] C. Wohlin. 'Writing for synthesis of evidence in empirical software engineering'. In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–4 (cit. on pp. 72, 127).
- [WPGW20] M. Wyrich, A. Preikschat, D. Graziotin, S. Wagner. *Replication package - The Mind Is a Powerful Place: How Showing Code Comprehensibility Metrics Influences Code Understanding*. Zenodo, Dec. 2020. URL: <https://doi.org/10.5281/zenodo.4498084> (cit. on p. 314).

- [WPGW21] M. Wyrich, A. Preikschat, D. Graziotin, S. Wagner. ‘The Mind Is a Powerful Place: How Showing Code Comprehensibility Metrics Influences Code Understanding’. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, IEEE, 2021, pp. 512–523 (cit. on pp. [32](#), [33](#), [75](#), [158](#), [163](#), [191–193](#), [197–199](#), [203–206](#), [208](#), [314](#)).
- [WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in Software Engineering*. Vol. 9783642290. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–236 (cit. on pp. [28](#), [72](#), [90](#), [91](#), [110](#), [178–180](#), [187](#), [191](#), [254](#)).
- [WW21] S. Wagner, M. Wyrich. *Replication package: Code Comprehension Confounders: A Study of Intelligence and Personality*. Zenodo, June 2021. URL: <https://doi.org/10.5281/zenodo.5031619> (cit. on p. [314](#)).
- [WW22] S. Wagner, M. Wyrich. ‘Code Comprehension Confounders: A Study of Intelligence and Personality’. In: *IEEE Transactions on Software Engineering* 48.12 (2022), pp. 4789–4801 (cit. on pp. [32](#), [33](#), [57](#), [75](#), [129](#), [314](#)).
- [XBL+18] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, S. Li. ‘Measuring Program Comprehension: A Large-Scale Field Study with Professionals’. In: *IEEE Transactions on Software Engineering* 44.10 (2018), pp. 951–976 (cit. on pp. [15](#), [50](#), [72](#)).
- [YTA19] C. S. Yu, C. Treude, M. Aniche. ‘Comprehending Test Code: An Empirical Study’. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 501–512 (cit. on p. [234](#)).
- [YYZD21] M. K.-C. Yeh, Y. Yan, Y. Zhuang, L. A. DeLong. ‘Identifying program confusion using electroencephalogram measurements’. In: *Behaviour & Information Technology* 0.0 (2021), pp. 1–18 (cit. on pp. [61](#), [75](#), [235](#)).

LIST OF FIGURES

2.1.	The “integrated metamodel” by Von Mayrhauser and Vans .	44
2.2.	The “schematic view of debugging” by Gilmore	47
2.3.	Conceptual model for the design of code comprehension experiments	67
3.1.	SMS: Schematic representation of the research methodology	77
3.2.	Number of publications on code comprehension experiments per year and venue	85
3.3.	Most popular study themes in code comprehension experiments	88
3.4.	Evolution of the four most popular study categories according to their number of occurrences	89
3.5.	Names for the central construct in code comprehension experiments	91
3.6.	Distribution of factor and allocation design in code comprehension experiments	92
3.7.	Evolution of factor and allocation design in code comprehension experiments	94
3.8.	Number of participants in code comprehension experiments	95

3.9.	Programming languages used in code comprehension experiments	98
3.10.	Sources for code snippets used in code comprehension experiments	99
3.11.	Comprehension measures used relative to number of papers	107
3.12.	Threat classification in code comprehension experiments . .	113
4.1.	Simplified representation of <i>LPS-2</i> for measuring intelligence	133
4.2.	Experiment on individual characteristics: Schematic representation of the research design.	137
4.3.	Experiment on individual characteristics: Causal diagram of the experiment variables	145
4.4.	Correlation of experience and code comprehension scores .	148
4.5.	Correlation of intelligence and code comprehension scores .	149
4.6.	Correlation of personality traits and code comprehension scores	150
5.1.	Anchoring Experiment 1: Development environment	174
5.2.	Anchoring Experiment 1: Results for perceived understandability	182
5.3.	Anchoring Experiment 2: Schematic representation of the research design	195
5.4.	Anchoring Experiment 2: Conceptual model for the study . .	199
6.1.	Evidence Profile Study: Schematic representation of the research design	218
6.2.	Evidence Profile 1: Programming experience influences code comprehension	229
6.3.	Evidence Profile 2: Program length influences code comprehension	231
6.4.	Evidence Profile 3: Different comprehension measures do not correlate with each other	233

LIST OF TABLES

3.1. SMS: Extracted data items	81
3.2. SMS: study purpose categories	87
3.3. SMS: Criteria for the creation or selection of experiment code snippets	100
3.4. SMS: Criteria for the creation or selection of experiment code snippets (continued)	101
3.5. SMS: Comprehension tasks that participants had to work on .	105
3.6. SMS: Frequency of reported threat categories	114
3.7. SMS: Frequency of reported threat categories (continued) . .	115
4.1. Experiment on individual characteristics: Results of individual linear regression models	151
4.2. Experiment on individual characteristics: Complete linear model	152
4.3. Experiment on individual characteristics: Linear model based on stepwise regression and all measured factors	153
5.1. Anchoring Experiment 1: Correlations of individual characteristics with deviation of subjective rating	184
5.2. Anchoring Experiment 2: Comparison of experiment characteristics	193

5.3. Anchoring Experiment 2: Assignable values for the variables of the model	199
5.4. Anchoring Experiment 2: Descriptive statistics and group assignment	202
5.5. Anchoring Experiment 2: Estimated and bootstrapped path coefficients	203
6.1. Evidence Types in the evidence profile	224
6.2. Number of threat mentions per theme and category	227
6.3. Number of papers analyzed in the first evidence collection	229
6.4. Number of papers analyzed in the second evidence collection	231
6.5. Number of papers analyzed in the third evidence collection	232
B.1. Replication package overview	314

LIST OF DEFINITIONS

2.1. Source Code Comprehension	64
2.2. Experiment	66



PRIMARY STUDIES OF THE SYSTEMATIC MAPPING STUDY

The 95 code comprehension experiments we studied in our systematic mapping study (see [Section 3.2](#)) are listed below.

Primary Studies

- [S1] N. Peitek, J. Siegmund, S. Apel, C. Kastner, C. Parnin, A. Bethmann, T. Leich, G. Saake, A. Brechmann. ‘A Look into Programmers’ Heads’. In: *IEEE Transactions on Software Engineering* 46.4 (2020), pp. 442–462 (cit. on pp. [86](#), [121](#)).
- [S2] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K.-C. Yeh, J. Cappos. ‘Understanding misunderstandings in source code’. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017.
- [S3] J. C. Hofmeister, J. Siegmund, D. V. Holt. ‘Shorter identifier names take longer to comprehend’. In: *Empirical Software Engineering* 24.1 (2018), pp. 417–443 (cit. on pp. [93](#), [124](#)).

- [S4] J. Borstler, B. Paech. ‘The Role of Method Chains and Comments in Software Readability and Comprehension-An Experiment’. In: *IEEE Transactions on Software Engineering* 42.9 (2016), pp. 886–898.
- [S5] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, R. Oliveto. ‘Automatically Assessing Code Understandability’. In: *IEEE Transactions on Software Engineering* 47.3 (2021), pp. 595–613 (cit. on pp. 102, 108, 121).
- [S6] R. P. L. Buse, W. R. Weimer. ‘Learning a Metric for Code Readability’. In: *IEEE Transactions on Software Engineering* 36.4 (2010), pp. 546–558 (cit. on pp. 91, 98).
- [S7] J. Dolado, M. Harman, M. Otero, L. Hu. ‘An empirical investigation of the influence of a type of side effects on program comprehension’. In: *IEEE Transactions on Software Engineering* 29.7 (2003), pp. 665–670 (cit. on p. 101).
- [S8] G. Salvaneschi, S. Amann, S. Proksch, M. Mezini. ‘An empirical study on program comprehension with reactive programming’. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014 (cit. on p. 91).
- [S9] S. Ajami, Y. Woodbridge, D. G. Feitelson. ‘Syntax, predicates, idioms - what really affects code complexity?’ In: *Empirical Software Engineering* 24.1 (2018), pp. 287–328.
- [S10] J. Castelhana, I. C. Duarte, C. Ferreira, J. Duraes, H. Madeira, M. 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 (2018), pp. 623–637 (cit. on p. 102).
- [S11] J. Feigenspan, C. Kastner, J. Liebig, S. Apel, S. Hanenberg. ‘Measuring programming experience’. In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2012.
- [S12] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, M. Züger. ‘Using psychophysiological measures to assess task difficulty in software development’. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.

- [S13] Y. Ikutani, H. Uwano. ‘Brain activity measurement during program comprehension with NIRS’. In: *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE, 2014.
- [S14] S. Lee, D. Hooshyar, H. Ji, K. Nam, H. Lim. ‘Mining biometric data to predict programmer expertise and task difficulty’. In: *Cluster Computing* 21.1 (2017), pp. 1097–1107.
- [S15] S. Lee, A. Matteson, D. Hooshyar, S. Kim, J. Jung, G. Nam, H. Lim. ‘Comparing Programming Language Comprehension between Novice and Expert Programmers Using EEG Analysis’. In: *2016 IEEE 16th International Conference on Bioinformatics and Bioengineering (BIBE)*. IEEE, 2016.
- [S16] T. Nakagawa, Y. Kamei, H. Uwano, A. Monden, K. Matsumoto, D. M. German. ‘Quantifying programmers’ mental workload during program comprehension based on cerebral blood flow measurement: a controlled experiment’. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.
- [S17] N. Peitek, J. Siegmund, C. Parnin, S. Apel, J. C. Hofmeister, A. Brechmann. ‘Simultaneous measurement of program comprehension with fMRI and eye tracking’. In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2018 (cit. on p. 109).
- [S18] R. Couceiro, G. Duarte, J. Duraes, J. Castelhana, C. Duarte, C. Teixeira, M. C. Branco, P. Carvalho, H. Madeira. ‘Biofeedback Augmented Software Engineering: Monitoring of Programmers’ Mental Effort’. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2019.
- [S19] J. Medeiros, C. Teixeira, R. Couceiro, J. Castelhana, M. C. Branco, G. Duarte, C. Duarte, J. Duraes, H. Madeira, P. Carvalho. ‘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.
- [S20] C. S. Yu, C. Treude, M. Aniche. ‘Comprehending Test Code: An Empirical Study’. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019 (cit. on p. 103).

- [S21] W. Lucas, R. Bonifácio, E. D. Canedo, D. Marcílio, F. Lima. ‘Does the Introduction of Lambda Expressions Improve the Comprehension of Java Programs?’ In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. ACM, 2019.
- [S22] F. Medeiros, G. Lima, G. Amaral, S. Apel, C. Kästner, M. Ribeiro, R. Gheyi. ‘An investigation of misunderstanding code patterns in C open-source software projects’. In: *Empirical Software Engineering* 24.4 (2018), pp. 1693–1726.
- [S23] M. K.-C. Yeh, D. Gopstein, Y. Yan, Y. Zhuang. ‘Detecting and comparing brain activity in short program comprehension using EEG’. In: *2017 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2017.
- [S24] D. Lawrie, C. Morrell, H. Feild, D. Binkley. ‘Effective identifier names for comprehension and memory’. In: *Innovations in Systems and Software Engineering* 3.4 (2007), pp. 303–318 (cit. on p. 96).
- [S25] J. Bauer, J. Siegmund, N. Peitek, J. C. Hofmeister, S. Apel. ‘Indentation: Simply a Matter of Style or Support for Program Comprehension?’ In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019.
- [S26] G. Beniamini, S. Gingichashvili, A. K. Orbach, D. G. Feitelson. ‘Meaningful Identifier Names: The Case of Single-Letter Variables’. In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017.
- [S27] S. Nielebock, D. Krolikowski, J. Krüger, T. Leich, F. Ortmeier. ‘Commenting source code: is it worth it for small programming tasks?’ In: *Empirical Software Engineering* 24.3 (2018), pp. 1418–1457.
- [S28] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, M. Beigl. ‘Descriptive compound identifier names improve source code comprehension’. In: *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018.
- [S29] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, B. Sharif. ‘The impact of identifier style on effort and comprehension’. In: *Empirical Software Engineering* 18.2 (2012), pp. 219–276.
- [S30] A. F. Norcio. ‘Indentation, documentation and programmer comprehension’. In: *Proceedings of the 1982 conference on Human factors in computing systems - CHI '82*. ACM Press, 1982.

- [S31] A. A. Takang, P. A. Grubb, R. D. Macredie. ‘The effects of comments and identifier names on program comprehensibility: an experimental investigation’. In: *J. Prog. Lang.* 4.3 (1996), pp. 143–167.
- [S32] T. Tenny. ‘Program readability: procedures versus comments’. In: *IEEE Transactions on Software Engineering* 14.9 (1988), pp. 1271–1279.
- [S33] J. Johnson, S. Lubo, N. Yedla, J. Aponte, B. Sharif. ‘An Empirical Study Assessing Source Code Readability in Comprehension’. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019.
- [S34] T. V. Ribeiro, G. H. Travassos. ‘Attributes Influencing the Reading and Comprehension of Source Code - Discussing Contradictory Evidence’. In: *CLEI Electronic Journal* 21.1 (2018) (cit. on p. 97).
- [S35] N. Kasto, J. Whalley. ‘Measuring the Difficulty of Code Comprehension Tasks Using Software Metrics’. In: *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*. ACE ’13. Adelaide, Australia: Australian Computer Society, Inc., 2013, pp. 59–65.
- [S36] R. J. Miara, J. A. Musselman, J. A. Navarro, B. Shneiderman. ‘Program indentation and comprehensibility’. In: *Communications of the ACM* 26.11 (1983), pp. 861–867.
- [S37] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, S. Tamm. ‘Eye Movements in Code Reading: Relaxing the Linear Order’. In: *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 2015.
- [S38] S. Fakhoury, D. Roy, Y. Ma, V. Arnaoudova, O. Adesope. ‘Measuring the impact of lexical and structural inconsistencies on developers’ cognitive load during bug localization’. In: *Empirical Software Engineering* 25.3 (2019), pp. 2140–2178.
- [S39] M. Hansen, A. Lumsdaine, R. Goldstone. ‘An experiment on the cognitive complexity of code’. In: *Proceedings of the Thirty-Fifth Annual Conference of the Cognitive Science Society, Berlin, Germany*. 2013 (cit. on p. 102).
- [S40] B. Katzmarski, R. Koschke. ‘Program complexity metrics and programmer opinions’. In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2012.

- [S41] Y. Sasaki, T. Ishihara, K. Hotta, H. Hata, Y. Higo, H. Igaki, S. Kusumoto. 'Preprocessing of Metrics Measurement Based on Simplifying Program Structures'. In: *2012 19th Asia-Pacific Software Engineering Conference*. IEEE, 2012 (cit. on pp. 97, 98).
- [S42] T. Sedano. 'Code Readability Testing, an Empirical Study'. In: *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. IEEE, 2016 (cit. on pp. 90, 97, 103).
- [S43] D. Hendrix, J. Cross, S. Maghsoodloo. 'The effectiveness of control structure diagrams in source code comprehension activities'. In: *IEEE Transactions on Software Engineering* 28.5 (2002), pp. 463–477.
- [S44] N. Pennington. 'Stimulus structures and mental representations in expert comprehension of computer programs'. In: *Cognitive Psychology* 19.3 (1987), pp. 295–341.
- [S45] V. Ramalingam, S. Wiedenbeck. 'An empirical study of novice program comprehension in the imperative and object-oriented styles'. In: *Papers presented at the seventh workshop on Empirical studies of programmers - ESP '97*. ACM Press, 1997.
- [S46] E. Avidan, D. G. Feitelson. 'Effects of Variable Names on Comprehension: An Empirical Study'. In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017.
- [S47] R. Bednarik, M. Tukiainen. 'An eye-tracking methodology for characterizing program comprehension processes'. In: *Proceedings of the 2006 symposium on Eye tracking research & applications - ETRA '06*. ACM Press, 2006.
- [S48] E. R. Iselin. 'Conditional statements, looping constructs, and program comprehension: an experimental study'. In: *International Journal of Man-Machine Studies* 28.1 (1988), pp. 45–66.
- [S49] A. Jbara, D. G. Feitelson. 'How programmers read regular code: a controlled experiment using eye tracking'. In: *Empirical Software Engineering* 22.3 (2016), pp. 1440–1477.
- [S50] E. S. Wiese, A. N. Rafferty, A. Fox. 'Linking Code Readability, Structure, and Comprehension Among Novices: It's Complicated'. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 2019.

- [S51] R. Couceiro, P. Carvalho, M. C. Branco, H. Madeira, R. Barbosa, J. Duraes, G. Duarte, J. Castelhamo, C. Duarte, C. Teixeira, N. Laranjeiro, J. Medeiros. 'Spotting Problematic Code Lines using Nonintrusive Programmers' Biofeedback'. In: *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019.
- [S52] S. Schulze, J. Liebig, J. Siegmund, S. Apel. 'Does the discipline of preprocessor annotations matter?' In: *Proceedings of the 12th international conference on Generative programming: concepts & experiences - GPCE '13*. ACM Press, 2013.
- [S53] M. V. Kosti, K. Georgiadis, D. A. Adamos, N. Laskaris, D. Spinellis, L. Angelis. 'Towards an affordable brain computer interface for the assessment of programmers' mental workload'. In: *International Journal of Human-Computer Studies* 115 (2018), pp. 52–66.
- [S54] A. Duraisingam, R. Palaniappan, S. Andrews. 'Cognitive task difficulty analysis using EEG and data mining'. In: *2017 Conference on Emerging Devices and Smart Systems (ICEDSS)*. IEEE, 2017.
- [S55] B. Sharif, M. Falcone, J. I. Maletic. 'An eye-tracking study on the role of scan time in finding source code defects'. In: *Proceedings of the Symposium on Eye Tracking Research and Applications - ETRA '12*. ACM Press, 2012.
- [S56] R. Bednarik, C. Schulte, L. Budde, B. Heinemann, H. Vrzakova. 'Eye-movement Modeling Examples in Source Code Comprehension'. In: *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. ACM, 2018.
- [S57] P. A. Orlov, R. Bednarik. 'The Role of Extrafoveal Vision in Source Code Comprehension'. In: *Perception* 46.5 (2016), pp. 541–565.
- [S58] N. J. Abid, B. Sharif, N. Dragan, H. Alrasheed, J. I. Maletic. 'Developer Reading Behavior While Summarizing Java Methods: Size and Context Matters'. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019.
- [S59] N. Peitek, J. Siegmund, C. Parnin, S. Apel, A. Brechmann. 'Beyond gaze'. In: *Proceedings of the Workshop on Eye Movements in Programming - EMIP '18*. ACM Press, 2018.
- [S60] I. Crk, T. Kluthe, A. Stefik. 'Understanding Programming Expertise'. In: *ACM Transactions on Computer-Human Interaction* 23.1 (2016), pp. 1–29.

- [S61] S. Wiedenbeck. 'The initial stage of program comprehension'. In: *International Journal of Man-Machine Studies* 35.4 (1991), pp. 517–540.
- [S62] R. Mosemann, S. Wiedenbeck. 'Navigation and comprehension of programs by novice programmers'. In: *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*. IEEE Comput. Soc.
- [S63] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, D. Binkley. 'Are test smells really harmful? An empirical study'. In: *Empirical Software Engineering* 20.4 (2014), pp. 1052–1094 (cit. on p. 124).
- [S64] J. Melo, F. B. Narcizo, D. W. Hansen, C. Brabrand, A. Wasowski. 'Variability through the Eyes of the Programmer'. In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017.
- [S65] L. Guerrouj, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol. 'An experimental investigation on the effects of context on source code identifiers splitting and expansion'. In: *Empirical Software Engineering* 19.6 (2013), pp. 1706–1753.
- [S66] D. Binkley, D. Lawrie, S. Maex, C. Morrell. 'Identifier length and limited programmer memory'. In: *Science of Computer Programming* 74.7 (2009), pp. 430–445.
- [S67] T. Tenny. 'Procedures and comments vs. the banker's algorithm'. In: *ACM SIGCSE Bulletin* 17.3 (1985), pp. 44–53 (cit. on p. 93).
- [S68] J. A. Saddler, C. S. Peterson, P. Peachock, B. Sharif. 'Reading Behavior and Comprehension of C++ Source Code - A Classroom Study'. In: *Augmented Cognition*. Springer International Publishing, 2019, pp. 597–616 (cit. on p. 124).
- [S69] E. Daka, J. Campos, G. Fraser, J. Dorn, W. Weimer. 'Modeling readability to improve unit tests'. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015.
- [S70] T. R. Beelders, J.-P. L. D. Plessis. 'Syntax highlighting as an influencing factor when reading and comprehending source code'. In: *Journal of Eye Movement Research* 9.1 (2015).
- [S71] G. K. Rambally. 'The influence of color on program readability and comprehensibility'. In: *Proceedings of the seventeenth SIGCSE technical symposium on Computer science education - SIGCSE '86*. ACM Press, 1986.

- [S72] S. Wiedenbeck. 'Beacons in computer program comprehension'. In: *International Journal of Man-Machine Studies* 25.6 (1986), pp. 697–709.
- [S73] Sheppard, Curtis, Milliman, Love. 'Modern Coding Practices and Programmer Performance'. In: *Computer* 12.12 (1979), pp. 41–49 (cit. on p. 122).
- [S74] E. Soloway, K. Ehrlich. 'Empirical Studies of Programming Knowledge'. In: *IEEE Transactions on Software Engineering* SE-10.5 (1984), pp. 595–609.
- [S75] A. Wulff-Jensen, K. Ruder, E. Triantafyllou, L. E. Bruni. 'Gaze Strategies Can Reveal the Impact of Source Code Features on the Cognitive Load of Novice Programmers'. In: *Advances in Neuroergonomics and Cognitive Engineering*. Springer International Publishing, 2018, pp. 91–100.
- [S76] B. E. Teasley. 'The effects of naming style and expertise on program comprehension'. In: *International Journal of Human-Computer Studies* 40.5 (1994), pp. 757–770.
- [S77] J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. Ajith Kumar, C. Prasad. 'An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies'. In: *Conferences in Research and Practice in Information Technology Series*. 2006.
- [S78] A. S. Nunez-Varela, H. G. Perez-Gonzalez, F. E. Martinez-Perez, D. Esqueda-Contreras. 'A Study And Experimental Assessment Of The Cognitive Weight, Base Of The Cognitive Metrics'. In: *2019 7th International Conference in Software Engineering Research and Innovation (CONISOFT)*. IEEE, 2019.
- [S79] A. C. Benander, B. A. Benander, H. Pu. 'Recursion vs. iteration: An empirical study of comprehension'. In: *Journal of Systems and Software* 32.1 (1996), pp. 73–82.
- [S80] A. Stefik, C. Hundhausen, R. Patterson. 'An empirical investigation into the design of auditory cues to enhance computer program comprehension'. In: *International Journal of Human-Computer Studies* 69.12 (2011), pp. 820–838.
- [S81] D. Scanlan. 'Structured flowcharts outperform pseudocode: an experimental comparison'. In: *IEEE Software* 6.5 (1989), pp. 28–36.

- [S82] B. D. Bois, S. Demeyer, J. Verelst. 'Does the "Refactor to Understand" Reverse Engineering Pattern Improve Program Comprehension?' In: *Ninth European Conference on Software Maintenance and Reengineering*. IEEE, 2005 (cit. on p. 104).
- [S83] R. Baecker. 'Enhancing program readability and comprehensibility with tools for program visualization'. In: *Proceedings. 11th International Conference on Software Engineering*. IEEE Comput. Soc. Press, 1989.
- [S84] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, A. Brechmann. 'Measuring neural efficiency of program comprehension'. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017 (cit. on pp. 101, 119).
- [S85] M. E. Crosby, J. Scholtz, S. Wiedenbeck. 'The Roles Beacons Play in Comprehension for Novice and Expert Programmers.' In: *PPIG*. 2002, p. 5.
- [S86] T. Ishida, H. Uwano. 'Synchronized Analysis of Eye Movement and EEG during Program Comprehension'. In: *2019 IEEE/ACM 6th International Workshop on Eye Movements in Programming (EMIP)*. IEEE, 2019.
- [S87] R. Turner, M. Falcone, B. Sharif, A. Lazar. 'An eye-tracking study assessing the comprehension of c++ and Python source code'. In: *Proceedings of the Symposium on Eye Tracking Research and Applications*. ACM, 2014.
- [S88] M. Konopka, A. Talian, J. Tvarozek, P. Navrat. 'Data flow metrics in program comprehension tasks'. In: *Proceedings of the Workshop on Eye Movements in Programming - EMIP '18*. ACM Press, 2018.
- [S89] D. Asenov, O. Hilliges, P. Müller. 'The Effect of Richer Visualizations on Code Comprehension'. In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 2016 (cit. on p. 97).
- [S90] D. Gilmore, T. Green. 'Comprehension and recall of miniature programs'. In: *International Journal of Man-Machine Studies* 21.1 (1984), pp. 31–48 (cit. on p. 106).
- [S91] S. Wiedenbeck, V. Ramalingam, S. Sarasamma, C. Corritore. 'A comparison of the comprehension of object-oriented and procedural programs by novice programmers'. In: *Interacting with Computers* 11.3 (1999), pp. 255–282.

- [S92] A. S. Alardawi, A. M. Agil. 'Novice comprehension of Object-Oriented OO programs: An empirical study'. In: *2015 World Congress on Information Technology and Computer Applications (WCITCA)*. IEEE, 2015.
- [S93] S. Blinman, A. Cockburn. 'Program Comprehension: Investigating the Effects of Naming Style and Documentation'. In: *Proceedings of the Sixth Australasian Conference on User Interface - Volume 40*. AUIC '05. Newcastle, Australia: Australian Computer Society, Inc., 2005, pp. 73–78.
- [S94] I. McChesney, R. Bond. 'Eye tracking analysis of computer program comprehension in programmers with dyslexia'. In: *Empirical Software Engineering* 24.3 (2018), pp. 1109–1154.
- [S95] R. Bednarik, N. Myller, E. Sutinen, M. Tukiainen. 'Program visualization: Comparing eye-tracking patterns with comprehension summaries and performance'. In: *Proceedings of the 18th Annual Psychology of Programming Workshop*. Citeseer. 2006, pp. 66–82.



REPLICATION PACKAGES

Following the principles of open science in software engineering [MGWS20], we publish our datasets and additional materials for all our studies presented in this thesis for reproducibility and transparency. [Table B.1](#) provides an overview of the replication packages and their contents.

Table B.1.: Replication package overview

Dataset	Description
[WBW22]	For the SMS [WBW23] described in Section 3.2 : The dataset comprises the 95 primary studies together with the extracted data items. The dataset details which studies were included in which step and which studies were extracted by which authors. In addition, we publish a large part of our data analyses, which can be used, e.g., to trace how characteristics of each individual study design were labeled and categorized in specific analyses.
[WW21]	For the experiment [WW22] described in Section 4.2 : We disclose code snippets, task sheets with comprehension questions, anonymized raw data, and the R script for the analysis openly.
[WPGW20]	For the experiment [WPGW21] described in Section 5.2 : We disclose code snippets, task sheets with comprehension questions, anonymized raw data, and the R script for the analysis openly.
[WMG22b]	For the experiment [WMG22a] described in Section 5.3 : We disclose code snippets, anonymized raw data, and the R script for the analysis openly.
[MWGW22]	For the meta-study [MWGW23] described in Section 6.2 : The dataset comprises data that emerged in intermediate steps of the validity threat analysis. This includes extracted text passages from the primary studies and complete lists of threat codes, threat categories and threat themes. Further, the dataset contains artifacts produced during the evidence search and the creation of the evidence profiles. We document comprehensibly, for example, how individual reviewers evaluated individual pieces of evidence and reached a final agreement. Finally, R scripts are included.