On the Reliability of Code Comprehension Proxies

Erfan Arvan
New Jersey Institute of Technology
Newark, New Jersey, USA
ea442@njit.edu

Nadeeshan De Silva
William & Mary
Williamsburg, Virginia, USA
kgdesilva@wm.edu

Oscar Chaparro
William & Mary
Williamsburg, Virginia, USA
oscarch@wm.edu

Martin Kellogg
New Jersey Institute of Technology
Newark, New Jersey, USA
martin.kellogg@njit.edu

Abstract

Prior work on code comprehension uses different *comprehension proxies*—for example, Likert-scale ratings or answers to input-output questions about program snippets, usually collected from students, to approximate whether code is comprehensible to software engineers, but the relative reliability of these proxies is not known.

This paper investigates the relative reliability of a collection of proxies common in the extant literature with a pair of human studies. First, we conducted an expert-consensus study with a panel of five professional software engineers to establish a ground-truth comprehensibility ranking of eight code snippets by adapting the Delphi expert-consensus protocol. The Delphi protocol is widely used for expert consensus under conditions of uncertainty in other domains such as medicine and national-security forecasting, but to our knowledge, this is its first application to code comprehension research. Second, we conducted a study with 44 student participants who completed comprehension tasks, allowing us to measure 14 comprehension proxies derived from the literature on the same set of eight code snippets. Finally, we conducted a correlation analysis on the results, concluding that proxies 1) derived from input-output questions and 2) that measure response time rather than accuracy are especially reliable. We also found that proxies derived from questions about program syntax (rather than semantics) are especially unreliable, regardless of measurement strategy, which draws into question the reliability of parts of the existing comprehensibility literature.

CCS Concepts

• **Human-centered computing** → **Empirical studies in HCI**; • **General and reference** → *Empirical studies*; • **Software and its engineering** → Maintaining software.

Keywords

code comprehension, comprehensibility proxies, Delphi method, controlled human study

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834425

ACM Reference Format:
Erfan Arvan, Nadeeshan De Silva, Oscar Chaparro, and Martin Kellogg. 2026. On the Reliability of Code Comprehension Proxies. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3834425

1 Introduction

Code comprehension—the process of understanding source code—is a fundamental software engineering activity, yet there is little agreement on how to measure it. Most maintenance tasks, including refactoring, debugging, and extending functionality, depend on a developer's ability to build an accurate mental model of the code. Prior studies estimate that developers spend between 58% and 70% of their time on code comprehension [69, 89].

Comprehension is an internal cognitive process that cannot be directly observed, so most prior empirical studies rely on *comprehension proxies*: observable *measurements* collected from humans performing specific comprehension *tasks*. Examples of such tasks include summarizing code [17, 20, 37, 38, 51, 54, 63, 68, 93] and fixing bugs [37, 67, 95, 108]. Common measurements include subjective ratings of code understandability [17, 27, 51, 68, 83, 93], the time required to answer questions about the code [25, 38, 45, 48, 72, 77, 90, 101, 113], and the correctness of those answers [25, 38, 48, 64, 72, 77, 90, 101, 113]. These proxies are intended to capture the difficulty of comprehension (i.e., "code comprehensibility"). For example, longer response times are assumed to indicate that a code snippet is harder for humans to understand. Comprehension proxies have been used not only to investigate how different factors influence comprehensibility [17, 51, 76, 96, 100, 115] but also to develop predictive models that help developers measure and control comprehension difficulty during software evolution [19, 30, 31, 62, 90]. More recently, with advancements in large language models and automated AI-based tools, comprehension proxies have also been used to evaluate whether such tools generate human-understandable code (e.g., UTGen [32]).

While these proxies are widely used in code comprehensibility studies, their reliability remains largely unexamined.

This creates a fundamental problem: it is unclear which proxies, if any, accurately capture the underlying construct of code comprehension difficulty. Further complicating the problem, this underlying construct is not well-defined, and various studies operationalize it in different ways [115]. As a result, findings across the

literature can be difficult to compare, and conclusions drawn from these proxies may be misleading.

In this work, we address this problem by defining code comprehensibility in terms of expert agreement about comprehension difficulty. In the absence of an objective definition, we argue that consistent agreement among experienced software engineers provides a practical and reliable approximation of the construct. Intuitively, if experts consistently agree that certain code is easier or harder to understand, this agreement can serve as a reference point against which proxies can be evaluated.

To operationalize this idea, we conducted two complementary studies on eight shared code snippets. First, we carried out an **expert study** with five senior software engineers using the Delphi expert-consensus protocol [28, 29, 57], a structured method for eliciting and refining expert judgment under uncertainty. Over multiple rounds, participants ranked the snippets by comprehension difficulty and iteratively resolved disagreements through written feedback and discussion. The Delphi protocol has been successfully applied in medicine [42, 66, 84, 94], geopolitical forecasting [49, 107], and other domains [11, 61, 65, 85]. Second, we conducted a **student study** with 44 computer science (CS) undergraduates from two U.S. universities. The students completed a set of tasks derived from commonly used comprehension proxies in the literature, including time-based, accuracy-based, and subjective measures. We used students for this part of the study because approximately 90% of studies in a recent survey of code comprehension research measured their proxies with student participants [115]. We then conducted **a correlation analysis** where we analyzed how well each proxy correlates with the expert-consensus ranking, treating the latter as a ground-truth approximation of code comprehensibility.

Our results reveal three main findings. First, proxies based on **input-output reasoning** about program behavior, including both response time and answer correctness, show the strongest relationship with expert judgments. Second, **time-based measures** consistently outperform direct correctness-based measures; the time required to answer input-output questions was the single best-performing proxy. These results imply that future studies of code comprehensibility can confidently approximate comprehensibility with time measurements of input-output questions. Third, several commonly used proxies, including **subjective Likert-scale ratings and human-judged free-text code summaries**, show weak correlations with expert consensus. Notably, **proxies based on syntactic questions** (e.g., identifying structural properties of the code) exhibit near-zero or negative correlations, suggesting that they may misrepresent comprehension effort. As a result, we can reinterpret some results from the literature that solely rely on syntax-derived proxies (e.g., [9, 105]).

In summary, our contributions are:

- A conceptualization of code comprehensibility based on expert agreement about comprehension difficulty, enabling the construction of ground-truth data (section 3).
- An expert study using the Delphi protocol to construct a consensus-based ground truth (section 4).
- A student study that collects 14 comprehension proxies commonly used in prior work (section 5).
- A correlation analysis comparing these proxies against expert-consensus judgments, showing that time-based and input-output proxies align more closely with expert assessments than other proxies (section 6).
- A discussion of the implications of our findings for interpreting prior code comprehension research (section 7).

2 Background and Related Work

Universal definition for code comprehensibility. Despite its importance, the code comprehension construct still lacks a precise definition that primary studies can rely on in their design, let alone a standardized measurement method or even consensus on which features to include in such measurements [22, 115]. As a consequence, what is measured is often implicitly defined by the method of measurement itself (i.e., the comprehension proxy). A comprehension proxy has two parts: a task and a measure. Common tasks in the literature include reading code [53], answering comprehension questions [4, 41, 73, 75], summarizing functionality [39, 53, 75], or finding or fixing a bug [39, 71]. Common measures in the literature are time [4, 5, 35, 53, 71, 73–75], accuracy [4, 5, 35, 39, 53, 71, 73–75, 82], eye movement [3, 4, 35, 39, 73, 75], or subjective ratings [4, 5, 14, 35, 74, 82]. The task and measure are inseparable when interpreting proxy measurements. For example, accuracy may be meaningless if the task is poorly designed (e.g., due to vague or trivial questions), but it can be informative when paired with a well-designed task. Likewise, a meaningful task may yield little insight when paired with an uninformative measure.

Generalization from students to developers. The problem is further complicated by participant selection. Based on data reported by Wyrich et al. [115], approximately 90% of code comprehensibility studies between 2010 and 2019 included students as participants [17, 21, 36, 89, 90], and this trend has continued in more recent years [4, 53, 71, 74, 82, 88]. These studies often recruit students because they are inexpensive and readily available, yet generalize the results to professional software engineers, who are the intended target population.

We regard the assumption that results obtained from students can be unconditionally generalized to professionals as a methodological flaw because code comprehension is a skill shaped by professional experience. Moreover, multiple studies confirm that professionals and novices differ substantially in how they approach code comprehension [7, 18, 20, 64]. At the neurological level, the two groups activate different brain regions when reading code [64], suggesting that expertise fundamentally reshapes how the brain processes code. These differences also manifest in reading behavior: eye-tracking data reveal that experts do not read code in the order it is written, instead navigating non-linearly toward semantically relevant regions, whereas novices tend to treat source code as a narrative, reading it sequentially [20]. Accordingly, experts concentrate their attention on functional units and object interactions that serve program goals, while novices focus on structural units and literal statements [18]. This targeted behavior is further reflected in visual attention patterns: experts efficiently direct their gaze toward information relevant to an area of interest, whereas novices are more easily distracted by irrelevant content [7, 20]. These findings motivate our study by suggesting that proxies derived from prior

Table 1: Taxonomy of comprehension tasks based on Wyrich et al. [115], with updated study counts through March 2026.

Task Category	Task	# of Studies
Tc1: Provide information about the code	Answer comprehension questions	37
	Determine output of a program	30
	Summarize code verbally or textually	25
	Recall (parts of) the code	7
	Match with diagram or similar code	3
	Determine code trace	1
Tc2: Provide personal opinion	Rate code comprehensibility	13
	Other subjective indications	7
	Rate task difficulty or task load	5
	Rate confidence in answer/understanding	5
	Compare or rank code snippets	5
	Rate own code understanding	2
Tc3: Debug code	Find a bug	9
	Extend the code	4
Tc4: Maintain code	Fix a bug	4
	Cloze test on code	3
	Refactor code	3
	Determine if code is correct	1

studies with students may not accurately capture the comprehension processes of professionals, and so may not serve as reliable indicators of code comprehensibility.

2.1 Comprehension Tasks in Prior Work

Table 1 extends the taxonomy of comprehension tasks in Wyrich et al.'s survey [115] with papers published on or before March 25, 2026 but after their literature search. While there is significant diversity in tasks, the literature has a strong bias toward **Tc1** and, to a lesser extent, **Tc2**. The most common tasks in these categories are answering questions of various types, determining program output given some inputs, free-text summarization, and subjective ratings; together, these tasks dominate the literature.

Open coding for comprehensibility questions. The task "answer comprehension questions" is highly abstract: one can conceive of questions targeting many different aspects of code—semantics, syntax, functionality, and so on. So, we performed an open-coding analysis ourselves on the questions actually asked in the relevant papers [6, 9, 10, 12, 13, 16, 20, 24, 27, 33, 34, 43, 46, 50, 52, 68, 70, 78, 80, 81, 83, 86, 87, 90–92, 101–105, 111, 112].

We first extracted all the questions. Two authors then independently performed open coding to categorize these questions. Following this initial phase, the authors discussed their labels and converged on a unified taxonomy of question categories. To assess inter-rater agreement, each author's initial labels were mapped to the final taxonomy and Cohen's kappa [23] was computed, yielding $\kappa = 0.89$. The resulting taxonomy, simplified for space, appears in fig. 1. It guided the choice of questions in our own study (section 5.3.1). Our replication package has a full list of questions and their categorization [8].

2.2 Comprehension Measures in Prior Work

Table 2 extends the comprehension measure survey data from Wyrich et al. [115] with that from studies published before March 25, 2026. *Correctness* (also referred to as *accuracy* [15]) compares a study participant's (i.e., a subject) answer to a known ground truth [4, 5, 35, 39, 53, 71, 73–75, 82]. *Time* captures how long a subject takes to complete a task, regardless of answer correctness [4, 5, 35, 53, 71, 73–75]. *Subjective ratings* are self-reported perceptions, such as Likert-scale responses to questions [4, 5, 14, 35, 71, 74, 82]. *Physiological* measurements capture responses of the human body or brain when performing code comprehensibility tasks. Eye-tracking [3, 4, 73, 75], fMRI [74] and EEG [14] are the most common measures in this category. *Aggregation* combines two or more other measures [5, 41, 82]; for example, "time to correctly answer a question" aggregates correctness and time measures. *Other* includes measures that do not fit into the above categories, such as using code metrics to measure comprehensibility [20, 71]. This taxonomy guided our choice of measures in our study (section 5.3.2).

```
Question categories (111)
|-- Program Function (27)
|   (e.g., What does the identifier "ListLimit" signify?)
|
|-- Semantics (49)
|   |-- Program Analysis (15)
|   |   (e.g., Does X affect Y?)
|   |
|   `-- Simulated Execution (34)
|       |-- Conditional (7)
|       |   (e.g., Is an error message printed?)
|       |
|       |-- Output (10)
|       |   (e.g., What output is produced for a given input?)
|       |
|       `-- Internal State (17)
|           (e.g., State the final value of "X", given an input "Y"?)
|
|-- Syntax (30)
|   |-- Global (18)
|   |   (e.g., How many loops are there?)
|   |
|   `-- Local (12)
|       (e.g., Is variable "Z" initialized to zero?)
|
`-- Other (5)
```

Figure 1: Simplified taxonomy of question categories derived from prior studies, with the number of instances per category and one representative example per leaf. The full categorization has more levels of detail and is available in our replication package [8].

Table 2: Frequency of comprehension measure types based on the taxonomy of Wyrich et al. [115], with updated study counts through March 2026.

Measure Type	Studies	Measure Type	Studies
Correctness	80	Physiological	30
Time	50	Aggregation	6
Subjective rating	37	Others	6

3 Methodology Overview

Our goal is to evaluate whether proxies commonly used in prior studies (typically measured with student participants) accurately reflect how professional software engineers experience code comprehension difficulty. This goal is important because students are likely to remain a primary study population for practical reasons, such as cost and availability, and the reliability of these proxies is unknown. Our research question is:

RQ1 How well do common comprehension proxies from the literature correlate with expert-determined ground truth?

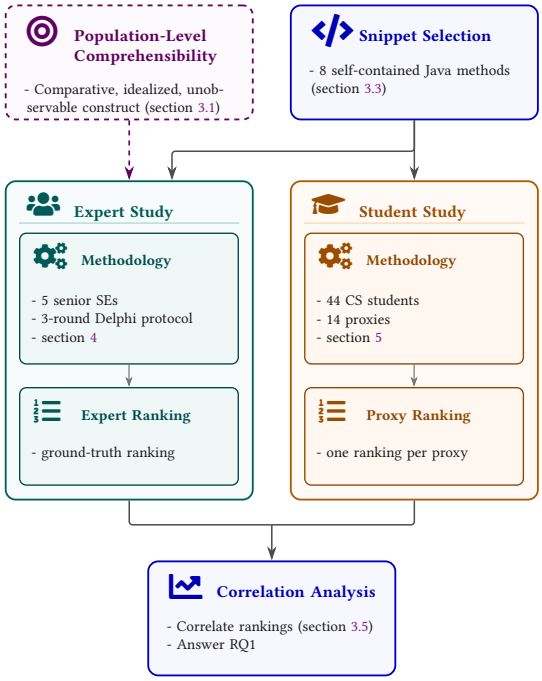

Figure 2: Overview of our methodology.

With this RQ, we are particularly interested in proxies that correlate very well (and are therefore good candidates for future studies) or very poorly (and so prior work relying on them is suspect).

Figure 2 summarizes our overall methodology. We first define an idealized notion of code comprehensibility (section 3.1) and then approximate it empirically via the Delphi expert-consensus protocol (section 3.2) on a set of carefully-selected code snippets (section 3.3). We then collect a range of proxies, drawn from prior work, on these same snippets, using a student population similar to many prior works (section 3.4). Finally, we analyze how well each proxy collected from the student participants correlates with the expert-consensus judgments, treating the latter as an approximation of ground-truth comprehension difficulty (section 3.5).

Note that, in relying on expert judgments as this reference point and collecting proxies from students, we are not claiming that experts and novices comprehend code in the same way. Rather, our goal is to evaluate whether proxies collected from student participants, who are commonly used in the literature, approximate the comprehensibility judgments of professional software engineers, the population such studies ultimately aim to generalize to.

A tempting alternative approach to evaluate the proxies is to conduct parallel studies with students and professionals and compare proxy values directly between the two groups. However, this approach assumes that at least one proxy is inherently valid—that is, that it can be trusted to measure the underlying construct. In our setting, where no proxy has been independently validated, such comparisons are inconclusive: agreement between groups on a proxy does not establish that the proxy itself is a reliable indicator of comprehension difficulty.

Our overall methodology, including participant recruitment procedures, questionnaires, comprehension tasks, and compensation methods for both expert and student studies, was approved by the ethics review boards of our respective institutions.

3.1 Idealized Ground Truth

Comprehension is an internal cognitive process and cannot be directly observed or measured. Moreover, comprehensibility depends not only on the code itself, but also on its human reader. In this section, we explain how we define the ground truth of comprehensibility in the abstract and how we approximate it in practice.

Prior work in psychology and program comprehension [59, 60, 78, 98, 109] characterizes comprehension as the process of constructing a coherent internal representation (or mental model) of the "material" (e.g., text or code), which can support tasks such as recall, reasoning, or modification. We adopt this view and define code comprehensibility in terms of how easily such a representation can be constructed relative to a reference point: that is, one piece of code can be easier or harder to understand than another, but assigning a specific numerical value to "comprehensibility" is not defined. At the level of an individual software engineer, we can thus model comprehensibility as a comparison between two code snippets: either one is easier to understand than the other or they are equally understandable. This captures the intuition that comprehension difficulty is better expressed comparatively than absolutely. This intuition aligns with the law of comparative judgment, which suggests greater consistency in relative than absolute judgments [106]. Empirical evidence shows that absolute ratings suffer from heterogeneous scale interpretation, whereas comparative judgments yield more consistent and reliable measurements [58]. Based on relative code comparisons, we can establish a ranking of snippets by comprehensibility.

Building on this, we introduce an idealized notion of *population-level comprehensibility*, which reflects how a broad population of software engineers would compare two snippets. This construct is not directly observable, but serves as a conceptual target: it represents the aggregate judgment we would obtain if we could study all relevant developers under consistent conditions.

Since such a population-level notion cannot be measured directly, we approximate it using a group of experienced software engineers. We refer to this approximation as *expert-determined comprehensibility*: a consensus-based ranking of code snippets produced by a panel of experts. While this approximation is imperfect, it provides a practical and reliable reference point for evaluating proxies, particularly in the absence of an objective ground truth.

3.2 Establishing Expert Consensus

For reliable expert consensus, we need a clear definition of expertise and a structured method for eliciting and refining judgments.

3.2.1 Expert Definition. We operationalize "experts" as professional software engineers with "senior-level expertise", demonstrated experience in real-world development projects, based on professional progression and mentoring experience. Our recruitment criteria are proxies for sustained exposure to diverse codebases and for an ability to understand and assess how code is understood by others. More details on expert recruitment are in section 4.1.

3.2.2 Delphi Protocol. We employed the Delphi method [28, 29], a structured approach for eliciting and refining expert judgments in settings where objective ground truth is unavailable [11, 42, 49, 61, 65, 66, 84, 85, 94, 107]. Our protocol consisted of three iterative rounds conducted on a custom web platform, designed based on guidance published by Delphi experts [57]. In each round, experts evaluated the same set of code snippets by summarizing their functionality, assessing their comprehensibility, and ranking them relative to one another. After each round, participants received anonymized summaries of group responses, reflected on disagreements and revised their assessments. Anonymity was maintained throughout to reduce bias. This process produced a stable, consensus-based ranking of snippet comprehensibility, which we use as our approximation of ground truth. Further details are provided in section 4.2.

3.3 Snippet Selection

A key methodological decision in code comprehension studies is the choice of code snippets. We use real-world code because our goal is to draw conclusions relevant to professional software practice. At the same time, the snippet set must support controlled comparison by limiting confounding factors such as project-specific background knowledge, code length, and code-formatting differences. Since the snippets from the prior studies did not meet these criteria, we decided to construct a new set of snippets for our study. We constructed the snippet set in three stages: project filtering, method filtering, and redundancy reduction. Figure 3 shows two example snippets; the other snippets are in our replication package [8].

3.3.1 Project Filtering. We identified candidate open-source projects hosted in GitHub with SEART's repository search infrastructure [26]. To favor mature, active, and widely used open source projects, we required candidate repositories to satisfy the following criteria: (1) written in Java (because it is the most common language in prior studies [115] and is popular in industry [99]), (2) at least 1,000 commits, (3) at least 1,000 pull requests, (4) at least 1,000 stars, and (5) at least one recent commit within a three-week window after the data collection date (February 2, 2025). These filters yielded 16 candidate projects (e.g., apache/kafka [1], libgdx/libgdx [2]). The full list of projects is provided in our replication package [8].

3.3.2 Method Filtering. Following prior work [115], we use individual methods as the unit of analysis. Method-level comprehension is common in practice, since developers often need to understand specific methods in order to reuse, modify, or integrate them during development and maintenance tasks. We used these inclusion criteria to select methods: 1) all types in the signature and body must be from the JDK, 2) no annotations in signature, 3) static methods only, 4) must have a Javadoc comment, 5) must have at least one parameter and return a value, and 6) must have 18-22 non-comment, non-blank lines of code. Criteria 1, 2, and 3 ensure the snippets are self-contained, so that participants do not require project-specific knowledge. Criteria 4 and 5 are necessary for our experimental design: Javadoc comments are needed as ground-truth for free-text summary tasks, and input/output question tasks require at least one input and at least one output. Criterion 6 controls for size but keeps each task within a reasonable time budget while still using snippets that implemented self-contained behavior. Applying these criteria produced 21 candidate methods.

3.3.3 Redundancy Reduction. Thirteen of the 21 methods were identical or near-identical implementations appearing in multiple locations within the same project (e.g., *atan2*, *atan2Deg*, and *atan2Deg360* from libgdx [2]). We randomly retained one representative from each such group, yielding a final set of eight methods. These eight were used for both the expert and student studies.

3.3.4 Code Presentation. To reduce presentation-related variation, which could influence comprehension [79], all snippets were reformatted using Google Java Formatter [44] and displayed with a standardized IntelliJ-style theme. Participants could choose either dark or light mode. We did not show Javadoc comments to participants, to avoid revealing intended functionality [17, 47, 79]. We also removed inline comments relevant to comprehension tasks, since tasks like code summarization and behavior prediction can be biased by natural-language hints. At the same time, we preserved original identifier names, including method, parameter, and variable names. We did so to maintain realism: in practice, developers only ever encounter code with naturally-occurring names. As a result, identifier quality is part of the overall comprehensibility signal rather than a separately controlled factor.

3.3.5 Background Knowledge. Some snippets might require background knowledge beyond general Java familiarity. For example, the *isValidProjectName* snippet (fig. 3) does not require specialized knowledge, whereas the *atan2* snippet may benefit from familiarity with trigonometry and knowledge of floating-point behavior around singularities. To identify such knowledge, the authors reviewed each snippet and enumerated domain concepts that could plausibly influence comprehension. For each such concept, our student study included (1) a self-reported familiarity question on a 5-point scale and (2) a multiple-choice knowledge question shown only when a participant reported at least some familiarity. We analyzed whether prior knowledge was associated with proxy outcomes (see section 6.1.2) and found it to be mildly predictive on proxies using accuracy metrics.

3.4 Proxy Selection and Student Study

It is not practical to evaluate every proxy reported in prior work. Existing studies employ a wide variety of tasks and measures [115], some of which require expensive instrumentation (e.g., fMRI devices) or extensive interaction with code (e.g., debugging, bug fixing, or other maintenance tasks). We therefore focus on 14 proxies that are both common in the literature and feasible in standard human-subject studies without specialized equipment. Our systematic approach for selecting these proxies is described in section 5.3.

We then recruited 44 undergraduate computer science students from our two corresponding universities. These participants provided and allowed us to collect values for the 14 selected proxies on the same eight code snippets evaluated in the expert study. Section 5 provides details on the design of this study.

3.5 Correlation Analysis

Finally, we evaluated how well each proxy collected in the student study aligns with the final ranking from the expert study. For each

```java
public static boolean isValidProjectName(String name) {
   if (name == null) return false;
   if (name.startsWith(".")) return false;
   if ((name.length() < 1) || (name.length() >
        MAX_NAME_LENGTH)) {
      return false;
   }
   for (int i = 0; i < name.length(); i++) {
      char c = name.charAt(i);
      if (!Character.isLetterOrDigit(c) &&
           !VALID_NAME_SET.contains(c)) {
         return false;
      }
   }
   return true;
}
```

```java
public static float atan2(float y, float x) {
   float n = y / x;
   if (n != n)
      n = (y == x ? 1f : -1f);
   else if (n - n != n - n)
      x = 0f;
   if (x > 0)
      return atanUnchecked(n);
   else if (x < 0) {
      if (y >= 0) return atanUnchecked(n) + PI;
      return atanUnchecked(n) - PI;
   } else if (y > 0)
      return x + HALF_PI;
   else if (y < 0)
      return x - HALF_PI;
   return x + y;
}
```

Figure 3: Two example code snippets used in the study: *isValidProjectName* (Snippet #1) and *atan2* (Snippet #8).

proxy, we aggregated student responses at the snippet level by averaging proxy values to rank the eight snippets from easiest to hardest according to that proxy. We then compared the rank correlation between each proxy-based ranking and the expert ranking using **Kendall's τ_b rank correlation coefficient** [55]. Kendall's τ_b is well suited to our setting for three reasons. First, both expert and proxy outputs are ordinal rankings, making a rank-based measure appropriate. Second, with a small number of items (eight snippets), τ_b's pairwise comparison formulation provides a stable and interpretable measure of agreement. Third, τ_b explicitly accounts for ties in the rankings, which can arise in both expert and proxy-derived rankings. Together, these properties make Kendall's τ_b an appropriate choice for quantifying how closely each proxy reflects expert-assessed comprehension difficulty.

4 Expert Study Design

The goal of this study is to create an expert-determined ranking of snippets: the "ground truth" for evaluating the reliability of comprehension proxies. Five professional software engineers with extensive Java experience ranked the eight code snippets from section 3.3 based on their comprehension difficulty using a web-based application that we developed. The study followed a multi-round structured process based on the Delphi method [28, 29, 57].

4.1 Participant Recruitment

We recruited participants through our professional networks. We contacted potential candidates via email and asked them to complete an eligibility form. To be included, participants had to self-report at least three years of experience in Java programming and demonstrate senior-level software engineering expertise. We operationalized "senior-level expertise" as these two criteria: (i) prior promotion to a higher-level engineering role (for example, from entry level to senior roles), and (ii) experience mentoring junior engineers in a professional setting. These criteria ensure that participants not only have substantial technical experience, but have also been entrusted with increased responsibility and technical leadership within their organizations. Candidates who reported substantial professional programming experience but did not report prior promotion or mentoring experience were not automatically excluded. Instead, we evaluated them based on additional background information, including job title, total years of development experience, and descriptions of the most complex systems they had worked on. Final inclusion decisions for such cases were made through discussion and consensus among the authors. Using this process, we included one participant who had substantial programming experience (10–14 years overall and 6–9 years in Java) and mentoring experience in diverse software projects, but did not explicitly report a promotion.

We do not claim that these criteria capture all aspects of expertise. Rather, they provide a practical way to identify participants with substantial, relevant experience in code comprehension.

We invited approximately 30 candidates to complete the eligibility form. Of these, 6 met the inclusion criteria, and 5 ultimately participated in the study after providing informed consent. Four of these experts had between 10 and 20 years of professional software development experience (6 to 14 years of Java experience), three were currently working as senior software engineers and one as a non-senior engineer. The remaining expert had more than 20 years of experience in both general software development and Java, and was working as a Vice President of Engineering. Several participants have held technical leadership roles, including leading projects, mentoring engineers, and contributing to architectural decisions in large-scale systems.

The participants had experience working in organizations of varying sizes, including large-scale enterprise environments, and have contributed to software systems deployed in real-world production settings with substantial user bases. Their work includes systems that require high reliability, performance, and maintainability, such as backend services, distributed systems, and production-grade applications. The experts primarily worked on proprietary or commercial software (5 experts), with one expert also reporting experience contributing to open-source projects. They reported experience building diverse types of systems, including web applications (5 experts), AI based systems (4), mobile applications (2), libraries or frameworks (2), middleware (1), and desktop applications (1). They had worked across a broad range of domains, including healthcare (3), government (3), finance or banking (2),

cloud and development tools (2), as well as game development, retail or legal services, transport or airline systems, education, energy, bioinformatics, or postal systems (1 each). We compensated each expert with a $50 USD Amazon gift card.

4.2 Snippet Ranking Procedure

We employed a three-round Delphi-style process to elicit and iteratively refine expert judgments, with the goal of obtaining a stable consensus ranking of the snippets by comprehension difficulty. Each round was assigned a nominal one-week completion window. However, we granted extensions upon request to accommodate participants' professional schedules and ensure thoughtful, high-quality responses. Participants were also allowed to revise and resubmit their responses before the deadline. All interactions were anonymized to reduce potential bias arising from factors such as seniority or personality. The stopping criteria were either: all experts converged to the same ranking; or after three completed rounds, following established recommendations that additional rounds typically yield diminishing returns [57].

4.2.1 Round 1: Independent Ranking. In the first round, experts independently evaluated the snippets through the study website. This round consisted of three phases:

- **Phase 1 (Independent understanding)**: Experts were presented with the eight snippets in randomized order. For each snippet, they (i) described its primary functionality and (ii) explained their assessment of its comprehension difficulty. Both tasks were completed using textual responses through the web interface. This phase ensured that experts engaged deeply with each snippet before ranking.
- **Phase 2 (Ranking)**: Experts ordered all snippets by comprehension difficulty using a drag-and-drop interface with eight slots. Participants were allowed to place multiple snippets in the same slot to indicate similar levels of comprehension difficulty.
- **Phase 3 (Ranking justification)**: Experts provided written justifications for their rankings, including cases where snippets were ranked as having similar difficulty.

4.2.2 Round 2: Structured Feedback and Reranking. After the first round, we analyzed the rankings to identify disagreements between experts. We constructed a set of *disagreement points*, each capturing a specific conflict in the relative ordering of snippets, along with the differing justifications provided by experts.

In the second round, experts were presented with anonymized summaries of other participants' assessments and the identified disagreement points. They then reviewed the rankings and justifications of other experts from the previous round. For each disagreement point, they examined the alternative perspectives and could revise their rankings and justifications. This structured feedback phase encouraged reflection and convergence while preserving independent judgment. The reranking and revised justifications followed the same protocol and web interface as Phases 2 and 3 (ranking and justification) from the first round.

4.2.3 Round 3: Discussion and Reranking. The third round was conducted as a moderated, anonymized discussion using a web forum interface, as recommended by modern guidance on conducting online Delphi panels [57]. We again analyzed the rankings and justifications from Round 2 to identify disagreement points. These were grouped and presented to participants in the forum. Experts first reviewed the rankings and justifications from the previous round, and then engaged in discussions with one another on each group of related disagreement points. The forum allowed experts to tag one another to send email notifications about updates. During this process, experts could revise their rankings at any time. The moderator (the first author) facilitated the discussion by organizing disagreement points, prompting participation when needed, and encouraging experts to clarify and reflect on their reasoning, while avoiding commenting on the content of their judgments.

Table 3: Final expert rankings of snippets after the last Delphi round.

Expert	Final Ranking (Easier → More difficult to understand)
Expert A	$(1 = 2) \rightarrow 3 \rightarrow (4 = 5) \rightarrow (7 = 8) \rightarrow 6$
Expert B	$1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 7 \rightarrow 3 \rightarrow 8 \rightarrow 6$
Expert C	$1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 6$
Expert D	$1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 6$
Expert E	$1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 3 \rightarrow 6 \rightarrow 8$

Table 4: Evolution of expert agreement across Delphi rounds. Disagreements indicate opposite pair orderings, while soft agreements capture ties versus strict ordering.

Round	Agreements	Soft Agreements	Disagreements	Total
Round 1	13	2	13	28
Round 2	17	1	10	28
Round 3	21	3	4	28

4.3 Ranking Results

4.3.1 Final Individual Rankings. We encouraged but did not require the experts to reach consensus to avoid biasing their final rankings. Each expert's individual final ranking is in table 3. While the experts did not reach full consensus, their final rankings are highly aligned. Specifically, there is strong agreement on several relative positions; for example, snippets 1 and 2 are consistently ranked among the easiest, and snippets 6 and 8 among the most difficult. However, variability remains in the ordering of middle-ranked snippets and in the use of ties (by Expert A).

For the main correlation analysis, we obtained a single expert ranking by averaging ranks across experts. We tested that our results are robust under other aggregation methods (section 6.1.1).

4.3.2 Expert Agreement Across Rounds. To analyze how expert agreement evolved across rounds, we examined all pairwise comparisons between snippets (28 total pairs) and categorized them into three types based on how consistently the experts ordered them. For a snippet pair (a, b), we define:

- *Agreement*: all experts assign the same relation between a and b, either $a < b$, $b < a$, or $a = b$.
- *Soft agreement*: there is no direct conflict in ordering, but not all experts assign the same relation. In particular, some experts assign a strict ordering (either $a < b$ or $b < a$, but all experts agree on the same direction) while others assign a tie ($a = b$).
- *Disagreement*: there is a direct conflict in ordering, with some experts assigning $a < b$ and others assigning $b < a$.

Table 4 shows how experts converged towards more consistent, strict orderings in later rounds. Inter-expert agreement, measured using Kendall's coefficient of concordance (W) [56] computed using the corresponding Friedman test formulation [40], shows a steady increase across rounds: $W = 0.7358$ in Round 1, $W = 0.8472$ in Round 2, and $W = 0.9338$ in the final round. These results indicate that expert rankings became increasingly concordant after each round, as expected from the Delphi method.

Overall, the results suggest that the Delphi process reduces conflicts and leads to a largely consistent ordering with high agreement, while retaining limited areas of ambiguity rather than enforcing complete consensus across all snippets.

5 Student Study Design

The goal of this study is to collect well-established proxies like those in prior comprehension studies, on the same set of code snippets used in the expert study. The study uses a within-subjects design: each participant interacted with each code snippet. For each snippet, participants completed a set of comprehension tasks, with measures including response time, answer correctness, and subjective ratings.

5.1 Participant Recruitment

Participant recruitment targeted undergraduate Computer Science (CS) students from our two universities. Using flyers, email announcements, and in-class invitations, we recruited 44 students: 37 from one institution and 7 from the other. Students were compensated with food after the study, and an entry into a raffle for two $50 USD Amazon gift cards. Participants were required to be at least 18 years old, have prior experience programming in Java, and have completed or be enrolled in advanced programming courses. The participant pool consisted of 33 male, 10 female, and one non-binary individual. On average, participants reported 3.1 years of Java programming experience. There were 16 4th-year students, 15 third-years, 11 second-years, and 2 first-years.

5.2 Experimental Procedure

Participants completed the study individually using their own laptops via a dedicated study website we developed, in-person in a controlled lab environment. Sessions lasted at most 90 minutes, consistent with prior comprehensibility research on sustained attention [92, 97, 101], though most finished within 70 minutes.

Participants were first verbally introduced to the study and gave informed consent. They then completed a short demographic and programming background questionnaire, which asked about their perceived programming experience relative to their classmates, their experience with Java and object-oriented programming generally, and their overall programming experience; we analyze whether these factors impact the main results in section 6.1.2. Participants then completed a brief practice task to familiarize them with the web interface and question format and reduce learning effects unrelated to code comprehension.

Participants then completed the main study. The eight Java code snippets were presented in a different random order to each participant. For each snippet, participants first read the code and then answered four comprehension questions presented sequentially in randomized order. Once a participant advanced, they could not return to previous questions or snippets, preventing answer revision and reducing potential carryover effects across tasks. The code remained visible while answering the questions to avoid introducing memory-related confounds into our results.

Participants were instructed not to use external resources (e.g., search engines or AI tools) so that the collected measures reflect their own comprehension effort. An on-screen timer timed all interactions. Participants were instructed to pause the timer whenever they were not actively working on the tasks, and to resume it when continuing. Researchers monitored sessions in-person to ensure compliance with these instructions and to assist with technical or logistical issues. To reduce fatigue, the study system enforced short breaks after every two snippets, during which participants could resume the study after 30 to 60 seconds. A longer break was also enforced at the midpoint of the session, with a duration ranging from one to seven minutes.

After completing the comprehension tasks, participants completed a post-study survey assessing their familiarity with the background concepts required for each snippet (section 3.3.5).

5.3 Comprehension Proxies

We wanted a broad set of common comprehension proxies from the literature that are feasible for students in a single study session. Recall that a comprehension proxy is a pair of a specific task (e.g., writing a summary) and a measure (e.g., time or correctness).

5.3.1 Comprehension Tasks. The taxonomy proposed by Wyrich et al. [115] organizes tasks into four categories: providing information about code (Tc1), providing subjective opinions (Tc2), debugging (Tc3), and maintenance (Tc4) (see table 1). As discussed in section 2.1, prior work is heavily concentrated on Tc1 and Tc2 tasks, so we focus on those and exclude Tc3 and Tc4 tasks. Tc1 and Tc2 tasks are widely-used and low-cost to administer, unlike Tc3 and Tc4 tasks. Tc1 tasks are much more common than Tc2 tasks in the literature, so we include several Tc1 tasks and one Tc2 task. We selected representative tasks within Tc1 via our open-coding analysis of comprehension questions from prior studies (see section 2.1), which organizes questions into categories such as program function, semantics, and syntax. To ensure conceptual coverage while keeping the study feasible, we selected one representative task from each of these categories. Specifically, we included these tasks:

- **Tc1: Program function**: describe what the code does.
- **Tc1: Semantics**: determine the code's output for a given input.
- **Tc1: Syntax**: identify syntactic elements in the code.
- **Tc2: Subjective rating**: rate comprehension and task difficulty.

Question Design. For the **Program function** and **Subjective rating** tasks, all snippets share a fixed question format. For the *program function* task, participants described the overall functionality of the method in their own words. The *subjective task* used two 5-point Likert scales to assess perceived comprehension and task difficulty, ranging from "Very difficult" to "Very easy."

For the **Semantics** and **Syntax** tasks, we used a semi-structured design process informed by our open-coding analysis of prior studies (section 2.1) to design unique questions for each snippet. We first derived question templates from the literature: for the *semantics*

Table 5: Comprehensibility proxies used in our study. Direction indicates whether higher values correspond to harder or easier comprehension difficulty. sec = seconds.

Proxy	Description	Scale	Direction
Time-based			
readTime	Reading time before answering questions	sec	↑ harder
time_function	Time to write function summary	sec	↑ harder
time_output	Time to predict output given an input	sec	↑ harder
time_syntaxBL	Time to answer syntax question	sec	↑ harder
time_correct_function	Time to write summary graded 4 or 5	sec	↑ harder
time_correct_output	Time for correct output predictions	sec	↑ harder
time_correct_syntaxBL	Time for correct answers to syntax question	sec	↑ harder
time_scaleSM	Time to decide snippet difficulty rating	sec	↑ harder
time_scaleST	Time to decide task difficulty rating	sec	↑ harder
Correctness-based			
acc_function	Function summary grade	1–5	↑ easier
acc_output	Output prediction correctness	0/1	↑ easier
acc_syntaxBL	Syntax question correctness	0/1	↑ easier
Rating-based			
scaleSM	Snippet difficulty rating	1–5	↑ harder
scaleST	Task difficulty rating	1–5	↑ harder

task, the template required computing the return value for a specific input; for the *syntax task*, the template required identifying or locating a syntactic element within a local code region (e.g., conditions, variable usage, or control structures). We then used ChatGPT to fill in these templates to generate candidate questions for each snippet. All generated questions were manually reviewed by the research team to ensure correctness, clarity, and consistency across snippets. We excluded questions that were ambiguous, misleading, or that revealed the intended functionality of the code. From the remaining candidates, we randomly selected one question per task category for each snippet.

Ground-truth answers for all questions were established by the research team and verified through manual inspection of the code to support correctness-based measures used in our analysis.

As a concrete example, for the *isValidProjectName* snippet (fig. 3), participants were asked: (*output*) *"What value does the method return when called with input* `"a1_"`*, given* `MAX_NAME_LENGTH = 10` *and* `VALID_NAME_SET = {'_', '-', '.'}`*?"*; (*syntaxBL*) *"Is* `MAX_NAME_LENGTH` *used in a conditional expression?"*; (*function*) *"What does this code snippet do? Briefly describe its main functionality."*; and (*scaleSM, scaleST*) *"How easy or difficult was this snippet to understand?"* and *"How easy or difficult was this task?"* (Very easy – Very difficult). The full set of tasks and questions for all eight snippets is available in our replication package [8].

5.3.2 Comprehension Measures. We used three types of measures:

- **Time**: reading time per snippet and time spent on each task.
- **Correctness**: correctness of responses to Tc1 questions.
- **Subjective ratings**: answers to Tc2 Likert-scale questions.

These are the three most common measures in the literature (section 2.2). We excluded physiological measures as they require specialized equipment and expertise, and are less common. We also included combined measures based on the time for *correct* answers to questions, which have been used in prior work [46, 110].

Correctness was determined differently depending on the question type. For the semantics and syntax questions, which were binary, we defined correctness based on predefined ground-truth answers validated by the research team. For program function questions, we used a 5-point ordinal scale (adapted from prior work [108]) to assess the quality of free-response answers.

Grades of 4 or 5 were considered "correct" for combined measures. We used each method's Javadoc to create the initial grading rubric. To ensure grading consistency, all authors first independently graded a shared sample of responses and then discussed discrepancies to finalize the rubric. Subsequently, each response was graded independently by two authors, and disagreements were resolved through discussion. All grades, justifications, and discussion logs are in our artifact [8]. Inter-rater agreement for program function grading, based on the 5-point ordinal scale for free-response function summaries, was Cohen's $\kappa = 0.57$.

5.3.3 Summary of Proxies. The final set of proxies includes 14 combinations of the selected task types (program function, semantics, syntax, subjective rating) and measures (time, correctness, and derived measures). For time-based and subjective-rating proxies, higher values indicate greater comprehension difficulty, whereas for correctness-based proxies, lower values indicate greater difficulty. Table 5 lists all proxies used in our study.

6 Results

Figure 4 presents the results of the correlation analysis between proxies and the expert-determined ranking. Time-based semantic proxies exhibit strong alignment with expert-defined comprehensibility, with *time_output* achieving a correlation of 0.86, followed by *time_correct_output* (0.79) and *time_function* (0.71). These values indicate that these proxies approximate expert judgment well. The five top proxies are statistically significant at $\alpha = 0.05$ (e.g., *time_output*, $p = 0.002$). The remaining proxies do not reach statistical significance at this sample size, but their relative ordering remains stable across the secondary analyses in section 6.1 (alternative aggregation strategies and individual-level data), increasing confidence in the overall proxy-reliability pattern.

In contrast, syntax-based proxies fail to capture expert judgments. Measures such as *acc_syntaxBL* (−0.37) and *time_syntaxBL* (−0.21) show weak or even negative correlations, suggesting that the syntax tasks reflect a different construct than expert-perceived comprehensibility. More interestingly, proxies in the middle show moderate but consistent alignment. Perception-based measures such as *scaleST* (0.64) and *scaleSM* (0.50), as well as general effort measures such as *readTime* (0.50), exhibit meaningful correlations with expert rankings, though substantially weaker than semantic time-based proxies. These results suggest that while subjective ratings and general reading effort capture some aspects of perceived difficulty, they do not reflect the same depth of understanding as time spent solving semantic tasks. Similarly, time spent on task difficulty judgments (e.g., *time_scaleST*, 0.18) shows only weak alignment, indicating that deliberation about perceived difficulty is a less reliable signal. Overall, these results demonstrate a clear gradient: semantic time-based proxies provide the strongest approximation of expert-defined comprehensibility, perception-based and general effort measures provide moderate signals, and syntax-based measures correlate poorly.

6.1 Secondary Analyses

This section details secondary analyses that we performed to ensure that the result is stable, especially with respect to possible

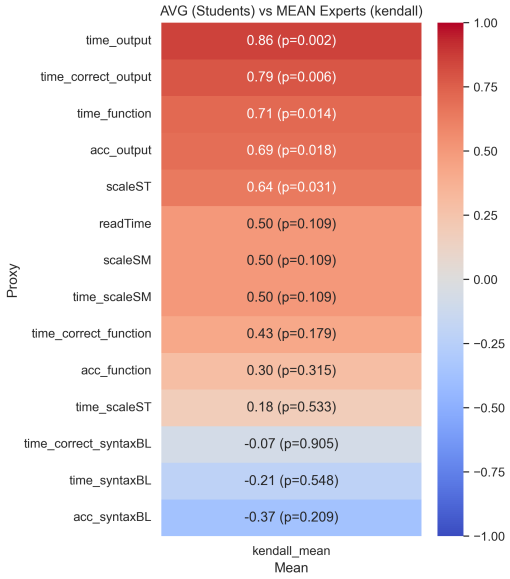

Figure 4: Correlation between proxies and expert-determined rankings, ordered by strength of correlation (strongest at the top). Descriptions of the proxies are in table 5 and section 5.3.

confounders and degrees of freedom in our experimental design. More details on all of these analyses are in Appendix A in our supplementary material.

6.1.1 Aggregation Strategies. Our experts did not reach complete agreement, so we aggregated their rankings into a single reference ordering by assigning each snippet its average rank across experts. To ensure that the results were stable, we also considered alternative aggregation strategies: requiring full agreement, allowing ties without contradiction, and permitting a small number of inconsistencies. The overall pattern of which proxies are well- or poorly-correlated with the expert-derived ranking remains stable across all aggregation strategies. We also confirmed that using individual experts' rankings instead of an aggregated ranking leads to consistent results: the relative performance and ordering of proxies remain largely unchanged across experts.

Our correlations are also based on aggregated student data. We tested whether analyses using individual student data yield similar patterns of correlations. They do: the relative ordering of proxies remains consistent. Per-student correlations do exhibit greater variability and are lower, due to individual differences and noise.

6.1.2 Student Characteristics and Fatigue. Differences in participant background or task order could explain variation in the comprehension proxies, and therefore potentially confound our main results. To assess possible effects, we fit regression models relating participant characteristics and task position to each proxy. Overall, participant characteristics exhibit limited and task-specific associations with comprehension performance. There are a few statistically significant relationships for accuracy on output and program function tasks, particularly for *Programming Experience Relative to Peers* ($\beta \approx 0.33$, $p = 0.024$ for program function tasks; $\beta \approx 0.36$, $p = 0.045$ for output tasks), *Object-Oriented Experience* ($\beta \approx 0.39$, $p = 0.026$ for program function tasks; $\beta \approx 0.45$, $p = 0.023$ for output tasks), and *Background Knowledge* (for program function tasks, $\beta \approx 0.35$, $p = 0.010$). In contrast, there are no consistent effects for syntax tasks or for most general experience measures like *Java Experience*.

Time-based proxies show little systematic relationship with participant characteristics, with effect sizes generally small and non-significant across tasks. However, *Fatigue* (operationalized as task order) has a strong and consistent effect on timing: participants complete tasks more quickly as the study progresses (e.g., $\beta \approx -7.9$ for *time_function*, $p < 10^{-3}$; $\beta \approx -15.6$ for *snippet_total_time*, $p < 10^{-8}$). This effect is not accompanied by a corresponding decrease in accuracy (all $p > 0.05$).

Taken together, these results suggest that while certain background factors are weakly associated with performance on specific comprehension tasks, overall comprehension outcomes are not strongly driven by participant characteristics. The observed changes in timing over the course of the study are more consistent with adaptation or increasing familiarity with the task format than with fatigue-related degradation in performance.

7 Discussion and Implications

7.1 Implications for Interpreting Prior Work

Our findings on proxy reliability have important implications for how prior work on program comprehension should be interpreted.

Syntax-only proxies. A subset of prior studies relies on syntax tasks like identifying structural properties or locating code elements, typically with response time or accuracy measures. Although some studies frame the construct as *readability*, they operationalize it the same way as studies of program comprehension [115]. For example, Asenov et al. [9] aim to evaluate whether richer code visualizations improve comprehension, measuring how quickly and accurately developers answer yes/no questions about code structure; similarly, Tenny [105] investigates readability by assessing how accurately participants answer questions about program properties after reading the code. These task–measure combinations are syntax proxies (*time_syntax* and *acc_syntax*) that our results show are unreliable signals of comprehensibility. The improvements these studies report may reflect increased efficiency in detecting or extracting structural features and not genuine improvements in understanding program behavior, and so should be interpreted with caution.

Aggregated syntax-semantic proxies. Another group of studies [13, 17, 78, 114] uses a mix of syntax and semantics tasks, but aggregates performance across them into one correctness or time measure. These studies include reliable tasks that ask about program behavior, but the aggregation implicitly treats all task–measure combinations as equally valid proxies of comprehensibility. Our findings indicate that this assumption does not hold, since proxies vary a lot in their alignment with expert-determined comprehensibility: syntax proxies provide weaker signals than semantic ones. Aggregating across heterogeneous proxies can obscure meaningful differences, and so reported effects may reflect limitations of the measurement approach rather than the absence of an effect on comprehensibility itself.

Accuracy-only proxies. Some studies [37, 54, 63] employ tasks that directly target program understanding, such as explaining the purpose of code, but rely primarily on correctness as the sole

measure. These task–measure combinations are proxies similar to *acc_function*. While these tasks are well aligned with the construct of comprehensibility, our findings show that accuracy alone provides a weaker and less consistent signal compared to a time-based measure of the same task (i.e., *time_function*). As a result, these studies may detect whether participants reach a correct interpretation, but may underestimate differences in how difficult the code is to understand.

Semantic proxies. In contrast, some studies [25, 45, 48, 72, 77, 113] use tasks that require reasoning about program behavior, such as predicting outputs or determining intermediate values during execution, measured via both response time and accuracy. These task–measure combinations align closely with proxies that show strong and consistent agreement with expert-determined comprehensibility in our evaluation (e.g., *time_output*). This alignment indicates that such proxies more effectively capture the cognitive effort required to understand program behavior. Therefore, findings from these studies are more likely to reflect genuine differences in comprehensibility rather than artifacts of the measurement approach.

Overall, these observations highlight that the validity of conclusions in program comprehension research depends on the alignment between the chosen task–measure combinations and the underlying construct of comprehensibility. Reliable and meaningful results require careful selection and interpretation of proxies.

7.2 Implications for Future Studies

Our results suggest that future studies can approximate code comprehensibility using simple and cost-effective proxies based on response time for input-output reasoning tasks. These proxies show strong and consistent alignment with expert judgments, indicating that they capture key aspects of the cognitive effort required to understand program behavior. Importantly, when using such reliable proxies, studies with student participants meaningfully approximate professional engineers' assessments of comprehensibility. So, researchers can continue to use students as a practical and scalable study population, as long as appropriate task–measure combinations are selected—that is, (expensive) expert participants are not necessary for reliable results. But, our findings caution against relying on proxies that focus on syntax or use accuracy alone, as these provide weaker, less consistent signals of the underlying construct.

7.3 Limitations and Threats to Validity

Construct validity. Our study operationalizes code comprehensibility using expert-derived rankings as a proxy for ground truth. While expert agreement provides a practical and principled approximation, it is not perfect. Different expert populations or elicitation procedures may yield different rankings, though the Delphi protocol mitigates the latter. Our selected proxies may miss aspects of comprehension, such as long-term retention or maintainability. Correctness for function summaries is based on human grading on a 5-point scale, which introduces subjectivity despite calibration and moderate inter-rater agreement ($\kappa = 0.57$). Grading disagreements were resolved through a rigorous protocol; however, only *acc_function* and *time_correct_function* depend on these grades, and neither was among the top-performing proxies. Our taxonomy of comprehension tasks is based on an open-coding process over prior studies. Although we followed a systematic procedure and achieved high inter-rater agreement ($\kappa = 0.89$), the resulting categorization may be subjective and may not capture every task in the literature.

Internal validity. Several factors may influence the observed proxy values independent of comprehension. For example, participant fatigue, learning effects, or familiarity with the study interface may affect response times, although our analysis in section 6.1.2 suggests that time decreases are more consistent with adaptation than degradation in performance. Noise may also arise from self-monitored timing (e.g., pausing the timer).

External validity. Our study uses just eight Java methods selected under controlled criteria (e.g., size, use of standard libraries, absence of project-specific context). To reduce required background knowledge, we only included methods that exclusively use standard Java types. To reduce dependence on external project context and object state, and consequently enable more controlled comparisons across snippets, we focused on static methods. While these choices improve experimental control, they exclude methods that depend on custom types, frameworks, or richer context, and thus may limit generalizability. Similarly, although our expert panel consists of experienced professional developers, the number of experts is limited and may not fully represent the diversity of industry practices. Nevertheless, we verified that, when correlating student-derived proxies with individual expert rankings (rather than aggregated rankings), the overall trends remain consistent. Finally, we did not evaluate all possible comprehension proxies, but instead focused on a representative subset of frequently used task–measure combinations from prior work.

8 Conclusion

We investigated the reliability of comprehension proxies from the literature with two complementary studies: a Delphi study of expert software engineers to establish ground truth, and a study of students with common proxies on the same code snippets. We correlated the results of the two studies, and showed that time measures and tasks that require semantic reasoning are the most reliable, while syntax tasks are unreliable. Future code comprehension studies should use our results to guide their choices of proxies.

Acknowledgments

We thank the experts and students who participated in our study. This work was supported in part by National Science Foundation grants CCF-2239107, CCF-2414110, and CCF-2414111. The views expressed herein are the authors' own and do not necessarily reflect those of our funders.

Data Availability Statement

Our replication package is publicly available [8].

References

[1] 2025. Apache Kafka. https://github.com/apache/kafka.
[2] 2025. libGDX. https://github.com/libgdx/libgdx.
[3] Amine Abbad-Andaloussi, Thierry Sorg, and Barbara Weber. 2022. Estimating developers' cognitive load at a fine-grained level using eye-tracking measures. In *ICPC*. 111–121.
[4] Youssef Abdelsalam, Norman Peitek, Annabelle Bergum, and Sven Apel. 2026. The effect of comments on program comprehension: an eye-tracking study. *Empir. Softw. Eng.* 31, 4 (2026), 94.

[5] Tarek Alakmeh, David Reich, Lena Jäger, and Thomas Fritz. 2024. Predicting code comprehension: a novel approach to align human gaze with code using deep neural networks. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1982–2004.

[6] Ahmed S. Alardawi and Agil M. Agil. 2015. Novice comprehension of object-oriented OO programs: an empirical study. In *WCITCA*. 1–4.

[7] Salwa D. Aljehane, Bonita Sharif, and Jonathan I. Maletic. 2023. Studying developer eye movements to measure cognitive workload and visual effort for expertise assessment. *Proc. ACM Hum.-Comput. Interact.* 7, ETRA (2023), 1–18.

[8] Erfan Arvan, Nadeeshan De Silva, Oscar Chaparro, and Martin Kellogg. 2026. Online Replication Package for "On the Reliability of Code Comprehension Proxies". https://doi.org/10.5281/zenodo.21218362. Zenodo.

[9] Dimitar Asenov, Otmar Hilliges, and Peter Müller. 2016. The effect of richer visualizations on code comprehension. In *CHI*. 5040–5045.

[10] Ronald Baecker. 1988. Enhancing program readability and comprehensibility with tools for program visualization. In *ICSE*. 356–357.

[11] Ann Barcomb, Klaas-Jan Stol, Brian Fitzgerald, and Dirk Riehle. 2022. Managing Episodic Volunteers in Free/Libre/Open Source Software Communities. *IEEE Trans. Softw. Eng. (TSE)* 48, 1 (2022), 260–277. doi:10.1109/TSE.2020.2985093

[12] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? an empirical study. *Empir. Softw. Eng.* 20 (2015), 1052–1094.

[13] Roman Bednarik, Carsten Schulte, Lea Budde, Birte Heinemann, and Hana Vrzakova. 2018. Eye-movement modeling examples in source code comprehension: a classroom study. In *Koli Calling*. 1–8.

[14] Annabelle Bergum, Norman Peitek, Maurice Rekrut, Janet Siegmund, and Sven Apel. 2026. On the influence of the baseline in neuroimaging experiments on program comprehension. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 35, 3 (2026), 1–27.

[15] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empir. Softw. Eng.* 18, 2 (2013), 219–276.

[16] Scott Blinman and Andy Cockburn. 2005. Program comprehension: investigating the effects of naming style and documentation. In *AUIC*. 73–78.

[17] Jürgen Börstler and Barbara Paech. 2016. The role of method chains and comments in software readability and comprehension: an experiment. *IEEE Trans. Softw. Eng. (TSE)* 42, 9 (2016), 886–898.

[18] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. 2002. Object-oriented program comprehension: effect of expertise, task, and phase. *Empir. Softw. Eng.* 7, 2 (2002), 115–156.

[19] Raymond P. L. Buse and Westley R. Weimer. 2009. Learning a metric for code readability. *IEEE Trans. Softw. Eng. (TSE)* 36, 4 (2009), 546–558.

[20] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye movements in code reading: relaxing the linear order. In *ICPC*. 255–265.

[21] Celia Chen, Reem Alfayez, Kamonphop Srisopha, Lin Shi, and Barry Boehm. 2017. Evaluating human-assessed software maintainability metrics. In *NASAC*. 120–132.

[22] Kyle D. Chin and Reid Holmes. 2026. Put the "Code" Back in "Code Comprehension Study". In *Proc. 34th IEEE/ACM Int. Conf. Program Comprehension (ICPC)*. 146–158. doi:10.1145/3794763.3794806

[23] Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences* (2 ed.). Routledge. doi:10.4324/9780203771587

[24] Ricardo Couceiro, Raul Barbosa, João Durães, Gonçalo Duarte, João Castelhano, Catarina Duarte, Cesar Teixeira, Nuno Laranjeiro, Júlio Medeiros, and Paulo Carvalho. 2019. Spotting problematic code lines using nonintrusive programmers' biofeedback. In *ISSRE*. 93–103.

[25] Igor Crk, Timothy Kluthe, and Andreas Stefik. 2015. Understanding programming expertise: an empirical study of phasic brain wave changes. *ACM Trans. Comput.-Hum. Interact. (TOCHI)* 23, 1 (2015), 1–29.

[26] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling projects in GitHub for MSR studies. In *MSR*. 560–564.

[27] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *ESEC/FSE*. 107–118.

[28] Norman Dalkey and Olaf Helmer. 1963. An experimental application of the Delphi method to the use of experts. *Manage. Sci.* 9, 3 (1963), 458–467.

[29] Norman C. Dalkey. 1969. *The Delphi method: an experimental study of group opinion.* RAND Corp., Santa Monica, CA. doi:10.7249/RM5888

[30] Nadeeshan De Silva, Martin Kellogg, and Oscar Chaparro. 2025. Relative code comprehensibility prediction. *arXiv* (2025). arXiv:2510.03474

[31] Nadeeshan De Silva, Martin Kellogg, and Oscar Chaparro. 2026. Verifier Warnings Do Not Improve Comprehensibility Prediction. In *EASE'26*. to appear.

[32] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. 2025. Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests. In *ICSE*. 1449–1461. doi:10.1109/ICSE55347.2025.00032

[33] Bart Du Bois, Serge Demeyer, and Jan Verelst. 2005. Does the "refactor to understand" reverse engineering pattern improve program comprehension?. In *CSMR*. 334–343.

[34] Aruna Duraisingam, Ramaswamy Palaniappan, and Samraj Andrews. 2017. Cognitive task difficulty analysis using EEG and data mining. In *ICEDSS*. 52–57.

[35] Yasmine Elfares, Gül Çalikli, and Mohamed Khamis. 2025. GazeCopilot: evaluating novel gaze-informed prompting for AI-supported code comprehension and readability. *arXiv* (2025). arXiv:2511.08177 [cs.HC]

[36] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The effect of poor source code lexicon and readability on developers' cognitive load. In *ICPC*. 286–296.

[37] Sarah Fakhoury, Devjeet Roy, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2020. Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empir. Softw. Eng. (ESE)* 25 (2020), 2140–2178.

[38] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2012. Measuring programming experience. In *Proc. IEEE/ACM Int. Conf. Program Comprehension (ICPC)*. 73–82.

[39] Samuel W. Flint, Robert Dyer, and Bonita Sharif. 2026. Do Developers Read Type Information? An Eye-Tracking Study on TypeScript. In *Proc. IEEE/ACM Int. Conf. Program Comprehension (ICPC)*. 73–84.

[40] Milton Friedman. 1937. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *J. Am. Stat. Assoc.* 32, 200 (1937), 675–701.

[41] Hao Gao, Haytham Hijazi, Júlio Medeiros, João Durães, Chan Tong Lam, Paulo de Carvalho, and Henrique Madeira. 2025. NRevisit: a cognitive behavioral metric for code understandability assessment. In *Proc. Int. Conf. Evaluation Assessment Softw. Eng. (EASE)*. 908–918.

[42] Ileana Gefaell Larrondo et al. 2026. Strengthening Primary Health Care in Europe: A Delphi study towards accessibility, equity and continuity of care. *Eur. J. Gen. Pract.* 32, 1 (2026), 2619226.

[43] David J. Gilmore and Thomas R. G. Green. 1984. Comprehension and recall of miniature programs. *Int. J. Man-Mach. Stud.* 21, 1 (1984), 31–48.

[44] Google. 2024. Google Java Formatter. https://github.com/google/google-java-format. Accessed: 2024-11-20.

[45] Michael Hansen, Andrew Lumsdaine, and Richard L. Goldstone. 2013. An experiment on the cognitive complexity of code. In *Proc. Annu. Conf. Cogn. Sci. Soc. (CogSci)*.

[46] Dean Hendrix, James H. Cross, and Saeed Maghsoodloo. 2002. The effectiveness of control structure diagrams in source code comprehension activities. *IEEE Trans. Softw. Eng. (TSE)* 28, 5 (2002), 463–477.

[47] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. 2019. Shorter identifier names take longer to comprehend. *Empir. Softw. Eng.* 24 (2019), 417–443.

[48] Errol R. Iselin. 1988. Conditional statements, looping constructs, and program comprehension: an experimental study. *Int. J. Man-Mach. Stud.* 28, 1 (1988), 45–66.

[49] Oleksandra Ishchenko et al. 2025. Barriers and opportunities for Demand Response Aggregation in Ukraine and Norway: A Delphi-based study. *Energy* 328 (2025), 136296.

[50] Toyomi Ishida and Hidetake Uwano. 2019. Synchronized analysis of eye movement and EEG during program comprehension. In *EMIP*. 26–32.

[51] Ahmad Jbara and Dror G. Feitelson. 2017. How programmers read regular code: a controlled experiment using eye tracking. *Empir. Softw. Eng.* 22 (2017), 1440–1477.

[52] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. 2019. An empirical study assessing source code readability in comprehension. In *ICSME*. 513–523.

[53] Zachary Karas, Aakash Bansal, Yifan Zhang, Toby Li, Collin McMillan, and Yu Huang. 2024. A tale of two comprehensions? analyzing student programmer attention during code summarization. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 33, 7 (2024), 1–37.

[54] Nadia Kasto and Jacqueline Whalley. 2013. Measuring the difficulty of code comprehension tasks using software metrics. In *Proc. Australas. Comput. Educ. Conf. (ACE)*. 59–65.

[55] Maurice G. Kendall. 1945. The treatment of ties in ranking problems. *Biometrika* 33, 3 (1945), 239–251.

[56] Maurice G. Kendall, Sheila F. H. Kendall, and B. Babington Smith. 1939. The distribution of Spearman's coefficient of rank correlation in a universe in which all rankings occur an equal number of times. *Biometrika* (1939), 251–273.

[57] Dmitry Khodyakov, Sean Grant, Jack Kroger, and Melissa Bauman. 2023. *RAND methodological guidance for conducting and critically appraising Delphi panels.* RAND Corp., Santa Monica, CA. doi:10.7249/TLA3082-1

[58] George Kinnear, Ian Jones, and Ben Davies. 2025. Comparative judgement as a research tool: A meta-analysis of application and reliability. *Behavior Research Methods* 57 (2025), 222.

[59] Walter Kintsch. 1988. The role of knowledge in discourse comprehension: a construction-integration model. *Psychol. Rev.* 95, 2 (1988), 163–182.

[60] Walter Kintsch and Teun A. Van Dijk. 1978. Toward a model of text comprehension and production. *Psychol. Rev.* 85, 5 (1978), 363–394.

[61] Martin F Krafft, Klaas-Jan Stol, and Brian Fitzgerald. 2016. How do free/open source developers pick their tools? A Delphi study of the Debian project. In *Proceedings of the 38th International Conference on Software Engineering Companion.*

232–241.
[62] Luigi Lavazza, Sandro Morasca, and Marco Gatto. 2023. An empirical study on software understandability and its dependence on code characteristics. *Empir. Softw. Eng.* 28, 6 (2023), 155.
[63] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective identifier names for comprehension and memory. *Innov. Syst. Softw. Eng.* 3, 4 (2007), 303–318.
[64] SeolHwa Lee, Andrew Matteson, Danial Hooshyar, SongHyun Kim, JaeBum Jung, GiChun Nam, and HeuiSeok Lim. 2016. Comparing programming language comprehension between novice and expert programmers using EEG analysis. In *BIBE*. 350–355.
[65] Brady D Lund. 2020. Review of the Delphi method in library and information science research. *J. Doc.* 76, 4 (2020), 929–960. doi:10.1108/JD-09-2019-0178
[66] Sarah B Maness, Stacey B Griner, and Erika L Thompson. 2025. Expert Consensus on Indicators of Social Determinants of Health: A Modified Delphi Study. *J. Prim. Care Community Health* 16 (2025).
[67] Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand, and Andrzej Wasowski. 2017. Variability through the eyes of the programmer. In *Proc. IEEE/ACM Int. Conf. Program Comprehension (ICPC)*. 34–44.
[68] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. 1983. Program indentation and comprehensibility. *Commun. ACM* 26, 11 (1983), 861–867.
[69] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I know what you did last summer: an investigation of how developers spend their time. In *ICPC*. 25–35.
[70] Russell Mosemann and Susan Wiedenbeck. 2001. Navigation and comprehension of programs by novice programmers. In *IWPC*. 79–88.
[71] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2019. Commenting source code: is it worth it for small programming tasks? *Empir. Softw. Eng.* 24, 3 (2019), 1418–1457.
[72] Pavel A. Orlov and Roman Bednarik. 2017. The role of extrafoveal vision in source code comprehension. *Perception* 46, 5 (2017), 541–565.
[73] Kang-il Park, Jack Johnson, Cole S. Peterson, Nishitha Yedla, Isaac Baysinger, Jairo Aponte, and Bonita Sharif. 2024. An eye tracking study assessing source code readability rules for program comprehension. *EMSE* 29, 6 (2024), 160.
[74] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program comprehension and code complexity metrics: an fMRI study. In *ICSE*. 524–536.
[75] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What drives the reading order of programmers? an eye tracking study. In *ICPC*. 342–353.
[76] Norman Peitek, Janet Siegmund, Sven Apel, Christian Kästner, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2018. A look into programmers' heads. *IEEE Trans. Softw. Eng. (TSE)* 46, 4 (2018), 442–462.
[77] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, and André Brechmann. 2018. Beyond gaze: preliminary analysis of pupil dilation and blink rates in an fMRI study of program comprehension. In *Proc. Workshop Eye Movements Program. (EMIP)*. 1–5.
[78] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cogn. Psychol.* 19, 3 (1987), 295–341.
[79] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. 2011. A simpler model of software readability. In *MSR*. 73–82.
[80] Vennila Ramalingam and Susan Wiedenbeck. 1997. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Empir. Stud. Programmers*. 124–139.
[81] Gerard K. Rambally. 1986. The influence of color on program readability and comprehensibility. In *SIGCSE*. 173–181.
[82] Talita Vieira Ribeiro, Paulo Sérgio Medeiros dos Santos, and Guilherme Horta Travassos. 2023. On the investigation of empirical contradictions: aggregated results of local studies on readability and comprehensibility of source code. *Empir. Softw. Eng.* 28, 6 (2023), 148.
[83] Talita Vieira Ribeiro and Guilherme Horta Travassos. 2018. Attributes influencing the reading and comprehension of source code: discussing contradictory evidence. *CLEI Electron. J.* 21, 1 (2018), 5–1.
[84] Jennifer Roggenbuck, Breda HF Eubank, Joshua Wright, Matthew B Harms, Stephen J Kolb, and the ALS Genetic Testing and Counseling Guidelines Expert Panel. 2023. Evidence-based consensus guidelines for ALS genetic testing and counseling. *Ann. Clin. Transl. Neurol.* 10, 11 (2023), 2074–2091.
[85] Daniel Russo, Paolo Ciancarini, Tommaso Falasconi, and Massimo Tomasi. 2017. Software quality concerns in the Italian bank sector: the emergence of a meta-quality dimension. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 63–72.
[86] Jonathan A. Saddler, Cole S. Peterson, Patrick Peachock, and Bonita Sharif. 2019. Reading behavior and comprehension of C++ source code: a classroom study. In *Augmented Cognition (AC)*. 597–616.
[87] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. 2014. An empirical study on program comprehension with reactive programming. In *FSE*. 564–575.
[88] Isabel Braga Sampaio and Alberto Sampaio. 2024. Replication of a study about the impact of method chaining and comments on readability and comprehension. In *ICCQ*. 35–52.
[89] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. Automatically assessing code understandability: how far are we?. In *ASE*. 417–427.
[90] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2019. Automatically assessing code understandability. *IEEE Trans. Softw. Eng. (TSE)* 47, 3 (2019), 595–613.
[91] David A. Scanlan. 1989. Structured flowcharts outperform pseudocode: an experimental comparison. *IEEE Softw.* 6, 5 (1989), 28–36.
[92] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. 2013. Does the discipline of preprocessor annotations matter? a controlled experiment. In *GPCE*. 65–74.
[93] Todd Sedano. 2016. Code readability testing: an empirical study. In *CSEET*. 111–117.
[94] Zhida Shang. 2023. Use of Delphi in health sciences research: a narrative review. *Medicine* 102, 7 (2023), e32829.
[95] Bonita Sharif, Michael Falcone, and Jonathan I. Maletic. 2012. An eye-tracking study on the role of scan time in finding source code defects. In *Proc. Symp. Eye Tracking Res. Appl. (ETRA)*. 381–384.
[96] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *ICSE*. 378–389.
[97] Janet Siegmund and Jana Schumann. 2015. Confounding parameters on program comprehension: a literature survey. *EMSE* 20 (2015), 1159–1192.
[98] Catherine Snow. 2002. *Reading for understanding: toward an R&D program in reading comprehension*. RAND Corp.
[99] TIOBE Software. 2026. TIOBE index. https://www.tiobe.com/tiobe-index/.
[100] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A human study of comprehension and code summarization. In *ICPC*. 2–13.
[101] Andreas Stefik, Christopher Hundhausen, and Robert Patterson. 2011. An empirical investigation into the design of auditory cues to enhance computer program comprehension. *Int. J. Hum.-Comput. Stud.* 69, 12 (2011), 820–838.
[102] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Program. Lang.* 4, 3 (1996), 143–167.
[103] Barbee E. Teasley. 1994. The effects of naming style and expertise on program comprehension. *Int. J. Hum.-Comput. Stud.* 40, 5 (1994), 757–770.
[104] Ted Tenny. 1985. Procedures and comments vs. the banker's algorithm. *ACM SIGCSE Bull.* 17, 3 (1985), 44–53.
[105] Ted Tenny. 1988. Program readability: procedures versus comments. *IEEE Trans. Softw. Eng. (TSE)* 14, 9 (1988), 1271–1279.
[106] Louis L. Thurstone. 2017. A law of comparative judgment. In *Scaling*. Routledge, 81–92.
[107] Abbie Tingstad et al. 2025. Divergent trajectories of Arctic change: Implications for future socio-economic patterns. *Ambio* 54, 2 (2025), 239–255. doi:10.1007/s13280-024-02080-x
[108] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. 2014. An eye-tracking study assessing the comprehension of C++ and Python source code. In *Proc. Symp. Eye Tracking Res. Appl. (ETRA)*. 231–234.
[109] Anneliese von Mayrhauser and A. Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55.
[110] Anneliese von Mayrhauser and A. Marie Vans. 1995. Program understanding: models and experiments. In *Adv. Comput.* Vol. 40. 1–38.
[111] Jacqueline Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 2006. An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. *Australas. Comput. Soc.* (2006).
[112] Susan Wiedenbeck, Vennila Ramalingam, Suseela Sarasamma, and Cynthia L. Corritore. 1999. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interact. Comput.* 11, 3 (1999), 255–282.
[113] Eliane S. Wiese, Anna N. Rafferty, and Armando Fox. 2019. Linking code readability, structure, and comprehension among novices: it's complicated. In *ICSE-SEET*. 84–94.
[114] Andreas Wulff-Jensen, Kevin Ruder, Evangelia Triantafyllou, and Luis Emilio Bruni. 2018. Gaze strategies can reveal the impact of source code features on the cognitive load of novice programmers. In *Proc. Int. Conf. Appl. Hum. Factors Ergon. (AHFE)*. 91–100.
[115] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 2023. 40 years of designing code comprehension experiments: a systematic mapping study. *ACM Comput. Surv.* 56, 4 (2023), 1–42.