A Facet-Based Perspective on Code Comprehension Task Design for Empirical Research

Luisa Bartl
Chemnitz University of Technology, Germany

Marvin Wyrich
Saarland University, Germany

Sven Apel
Saarland University, Germany

Janet Siegmund
Chemnitz University of Technology, Germany

Abstract

Code comprehension is an inevitable part of the software development process and a frequent target of empirical research. However, the field lacks a shared definition of the construct. Studies operationalize code comprehension in diverse ways, often without explicitly stating what aspect of understanding they measure. As a result, researchers design study-specific tasks and measures, limiting comparability across studies and hindering cumulative knowledge building.

To initiate a coordinated approach to assess code comprehension, we introduce a facet-based perspective that distinguishes between analytical skills, abstraction, and critical evaluation. We demonstrate the benefit of explicitly stating the targeted facet(s) of code comprehension and aligning task design accordingly. Building on this perspective, we propose a method to guide researchers from research goals to concrete tasks and study design, including considerations for question formulation and response formats. Rather than immediately prescribing a single standardized instrument, our approach takes a first step toward this long-term goal by enabling researchers to design adaptable yet conceptually grounded measurements. By making operationalizations explicit and tasks reusable, we initiate a cumulative process through which the community can iteratively converge on standardized instruments and, ultimately, a shared understanding of code comprehension.

2012 ACM Subject Classification General and reference → Measurement

Keywords and phrases Code comprehension, Construct validity, Empirical software engineering

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.51

Category Emerging Results, Vision & Reflection Track Paper

Funding This work has been supported by the European Union under ERC Advanced Grant "Brains On Code" (101052182).

1 Introduction

Code comprehension is a core activity in software development and a recurring focus of empirical software engineering research. Developers spend a substantial portion of their working time understanding existing code rather than writing new code [56]. Nevertheless, code comprehension remains a loosely defined construct [53]. Studies operationalize it in various ways, and no generally accepted definition or measurement approach exists [54, 30]. As a consequence, empirical results are often difficult to compare [55], and it is unclear which aspects of "understanding code" are captured by specific tasks [53, 54].

Beyond this conceptual ambiguity, researchers face concrete methodological challenges when selecting or designing code comprehension tasks. Code comprehension studies use a wide range of tasks—such as output prediction, bug detection, or code reproduction [54]—often

without explicitly stating which cognitive demands these tasks are intended to elicit. Task choices are often guided by prior work or practical considerations, while the alignment between research goals and the cognitive demands induced by a task is rarely made explicit [54].

We argue that this problem is partly caused by treating code comprehension as a monolithic construct. In practice, however, empirical studies implicitly target different aspects of understanding depending on their research goals. These aspects can be understood as different facets of code comprehension, which correspond to distinct cognitive activities involved in understanding code. For instance, a study may focus on analytical skills, that is, decomposing a program into smaller functional units and understanding their behavior [40]; on the ability to abstract, that is, integrating these elements into a coherent mental model of program behavior [50]; or on critical thinking, that is, evaluating code with respect to code quality or correctness [9]. These facets reflect qualitatively different cognitive demands, which in turn require different task designs to be appropriately operationalized.

However, because these distinctions are rarely made explicit, it remains unclear why a particular task is suitable for a given empirical context or which aspect of code comprehension it is actually intended to measure. This lack of explicit grounding complicates systematic task selection and contributes to the variability observed in empirical study designs.

To support systematic task-selection decisions, we introduce a pragmatic distinction between three facets of code comprehension that we derive from the literature: analytical skills (decomposing code into functional units), the skill to abstract (integrating these units into a coherent mental model), and critical thinking (evaluating the correctness, efficiency, or comprehensibility of code). These facets can be seen as a common denominator and are not intended as a comprehensive theory of code comprehension, but as an empirically grounded lens that reflects how code comprehension is treated in current research through combinations of bottom-up, top-down, and integrated comprehension models [53]. They are used here to make task design decisions explicit and to link empirical tasks to the specific cognitive demands they are intended to elicit.

Building on this distinction, we propose a method for the systematic construction of code comprehension tasks in empirical research. The method guides researchers from a use case through the selection of relevant facets, appropriate task types, and concrete design choices, such as question formulation and response formats. We illustrate our approach with an empirical use case in which we measure code comprehension in a controlled study with graduate students. The goal of this evaluation is not to validate the proposed facets as psychological constructs, but to examine how different task designs perform in practice and which design choices affect key quality criteria, such as validity, reliability, and feasibility in empirical settings. Rather than proposing a standardized measurement instrument, we provide task templates and study-design considerations that allow for adaptation to specific experimental contexts, for example, when introducing treatments or manipulating program characteristics. In doing so, we support more transparent, comparable, and systematically designed code comprehension studies without constraining experimental flexibility.

In summary, we contribute (1) a pragmatic, facet-based perspective on code comprehension, (2) a structured method for deriving task designs from research goals, and (3) an empirical illustration highlighting effective practices and common pitfalls in designing code comprehension tasks. Furthermore, we provide supplementary materials on the following Web site: `https://doi.org/10.5281/zenodo.21490313`.

2 Code Comprehension and its Heterogeneous Operationalization

Research on code comprehension dates back several decades, with early work distinguishing between top-down and bottom-up comprehension strategies. Top-down comprehension describes a hypothesis-driven process in which developers use prior knowledge to understand source code. They form expectations about the program's purpose (e.g., sorting a list of names) and iteratively test these expectations against observed behavior [7, 48]. This approach is effective when relevant domain knowledge is available.

In contrast, when such prior knowledge is lacking, developers rely on bottom-up comprehension, in which understanding is constructed from the code itself. Developers analyze code line by line and progressively group statements into meaningful semantic chunks that represent coherent functionality [34, 43]. Higher-level understanding emerges through the iterative integration of these chunks. Developers can switch between these strategies as needed, which is described by integrated models [41, 50].

These theoretical models are the base for measuring code comprehension. For a valid measurement, researchers must make design decisions regarding which aspect of comprehension they intend to evaluate; that is, they must *operationalize* the construct [6]. Despite the theoretical models, measuring code comprehension in a valid way remains challenging, as comprehension strategies depend on multiple factors, including the underlying code base, task framing, prior experience, and domain familiarity [46].

The quality of operationalizations is typically discussed in terms of *construct validity*, that is, the extent to which a measurement reflects the intended theoretical construct [8, 47]. Awareness of construct validity has considerably increased in the software engineering research community [47, 36], and several studies devote substantial effort to justifying their operationalization of code comprehension, reflecting the importance of careful measurement design (e.g., as in [52, 44]).

In empirical practice, researchers operationalize code comprehension through a combination of *tasks* and *measures*. Tasks vary widely, but, according to a previous literature survey, can broadly be grouped into four categories [54]: First, information-providing tasks require participants to answer questions about code or predict program output and have historically been the most common type. Variants include code summarization or reproduction tasks (e.g., [37, 14, 29]). Second, subjective assessment tasks ask participants to rate aspects such as perceived code comprehensibility, their own understanding, or task difficulty (e.g., [26, 38, 9]). Third, debugging tasks require participants to identify, fix, or evaluate defects in code (e.g., [49, 28, 10]). Finally, maintenance tasks involve modifying code, such as filling in missing fragments, extending functionality, or refactoring existing code (e.g., [29, 5, 16]).

The corresponding *measures* used to assess performance are often largely independent of task type [54]. The most common measures are correctness and time, frequently used individually or in combination, followed by subjective ratings. To observe the comprehension and reasoning process itself during task execution, researchers sometimes use think-aloud protocols. More recently, neurophysiological and behavioral measures, such as eye tracking, EEG, and fMRI, have been introduced to study cognitive aspects of code comprehension in greater detail [54, 45, 33, 32].

The diversity of operationalizations and the increasing awareness of construct validity also reveal an issue: The software engineering community lacks a shared theoretical definition of code comprehension.

In practice, definitions of code comprehension are typically implicit and context-dependent,

with many empirical studies that measure code comprehension not explicitly defining the construct at all [54, 30, 53]. Instead, the construct is implicitly defined through the chosen combination of tasks and measures. For example, studies on the effect of code smells often claim to investigate code comprehension, but operationalize it through time and accuracy in information-providing or maintenance tasks without specifying which code comprehension strategy is expected or possible [1, 35, 20]. Similarly, studies on large-scale systems sometimes refer to *top-down* or *bottom-up* comprehension without further clarifying which cognitive process is actually being measured [42].

Given the large number of possible task types, measures, and their combinations, this practice makes it difficult to compare results across studies [55]. Put simply, it remains unclear whether the many ways of measuring code comprehension actually capture the same underlying construct. This lack of conceptual grounding motivates the need for a clearer link between the kind of understanding a study investigates and the tasks used to operationalize it—an issue addressed by the facet-based perspective introduced in the following section.

3 A Facet-Based Perspective on Code Comprehension

The variety of operationalizations and (implicit) underlying definitions of code comprehension indicate the research community's diverse view on code comprehension. One contribution of this paper is to provide a unifying umbrella of the research community's current view of code comprehension, as can be deduced from the existing studies on code comprehension.

Distilling the underlying assumptions about what code comprehension is, we derive a multi-faceted view of code comprehension that comprises three facets: analytical, abstract, and critical. To arrive at these facets, our starting point is the systematic mapping study by Wyrich and others [54], which provides an overview of established tasks used in code comprehension studies. Based on these tasks, we can infer the cognitive skills required to solve them, which we group into the three key facets. We analyzed the tasks regarding the cognitive operations required to solve them at a fundamental level, focusing on the skills participants must use to complete them successfully. For example, tasks such as tracing code execution or recalling code elements require participants to focus on individual statements and their effects. This corresponds to analytical or bottom-up thinking. In contrast, tasks such as summarizing code or matching code to higher-level representations (e.g., flowcharts or diagrams) require forming an overall understanding of the program and interpreting details in view of that understanding. This reflects abstract or top-down thinking. Finally, tasks such as debugging, assessing correctness, or rating code quality require participants to evaluate the code, linking comprehension with judgment, which we categorize as critical thinking. By systematically mapping recurring task types to the underlying cognitive skills they require, we observe that the variety of task forms can be reduced to combinations of three key facets.

Analytical thinking is the skill to focus on individual parts of code to understand their functionality and, as a result, gradually develop an understanding of the entire program. It maps to the strategy of bottom-up comprehension [34, 43]. Several studies include this view. For example, Huang and others describe a tool to support students in acquiring programming skills, for which they provide different views on code [21]. One of those views focuses on detailed parts of the code to understand the effect of individual statements on the overall program, which is in line with the analytical skill. Barke and others found that programmers focus on details of the code to decide whether to accept or reject generated code [3]. Similarly, during code reviews, Gonçalves and others found that, as part of the code review process, developers understand code in a bottom-up way [13]. Benjamini and others let participants

understand and modify functions, in which variable names were replaced with single-letter names, enforcing a bottom-up comprehension process [5]. Izu and Mirolo evaluated students' skill to compare the equivalence of two small programs that had only single-letter identifiers, thereby enforcing bottom-up comprehension [23]. Several studies using eye-tracking and neuro-imaging methods assess the analytical thinking skill. For example, Abid and others found that developers examine specific code elements, such as call terms, when asked to summarize Java methods [2]. Siegmund and others, followed by Peitek and others, let participants predict the outcome of source code with obfuscated identifier names, enforcing that participants focus on details of code, while being observed in an fMRI study [44, 31]. These examples highlight the relevance of analytical skills in code comprehension studies.

Abstract thinking describes the skill to understand individual code elements in the context of an overarching understanding of a program. It is in line with top-down comprehension, which requires knowledge of the domain and the skill to recognize code elements as beacons [7, 48]. Several studies focus on this facet. For example, Kang and others assessed the effect of readability rules on top-down comprehension by giving participants a natural-language description of what a subsequent code snippet is supposed to do [22]. Similarly, Kallia gave students a natural-language description to observe the extent to which students can employ a top-down process [24]. Hassan and others evaluated how to enable students to employ a top-down comprehension strategy when understanding code to explain it in plain English [19]. Sedano indirectly promotes an abstract, overarching understanding by encouraging readers to gain an overview of the code functionality during think-aloud sessions and to interpret individual code elements within this context [39]. The program comprehension tasks used in the study of Konopka and others [25] promote abstract or top-down thinking indirectly, as they require participants to interpret local code elements in the context of a global understanding of the program and the data flow.

Critical thinking is the skill to evaluate code in terms of its correctness, efficiency, and comprehensibility. This evaluation of code does not take place after a program has been fully understood, but takes place in parallel to the understanding process and is a fundamental part of code comprehension. In the context of debugging, Gilmore [12] shows that evaluating code takes place parallel to the comprehension process and not, as assumed in earlier models, only after a complete understanding of the program [17]. Votipka and others demonstrate that reverse engineers evaluate code during security evaluations [51]. Basili and Mills [4] also report from observations of programmers that they simultaneously attempt to evaluate the correctness of code while reading it, thus providing early evidence that understanding and evaluation are closely linked.

4 Our Vision

The task landscape described in the previous section suggests that many existing code comprehension tasks can be interpreted as instantiations of a smaller set of underlying cognitive demands. In particular, the three facets—analytical thinking, abstract thinking, and critical thinking—provide a way to systematically relate these tasks to the cognitive skills they are intended to elicit. Importantly, this perspective does not require replacing existing tasks; instead, it provides a structure for interpreting and organizing them.

Based on this perspective, we argue that code comprehension tasks can serve as reusable building blocks across empirical studies, provided that their intended cognitive focus is made explicit. Rather than designing study-specific instruments from scratch, researchers could select tasks based on the facet(s) they aim to investigate while adapting the underlying code

content to their specific experimental context. This would allow tasks to remain flexible regarding treatments while still being anchored in a shared conceptual framework.

Our long-term vision is to gradually develop a shared pool of empirically grounded and validated code comprehension tasks, organized by facet. In such a setting, researchers would explicitly specify which facet(s) of code comprehension they target in a study and select corresponding tasks from this pool. This would substantially reduce the overhead of designing and justifying bespoken tasks for each individual study, while at the same time improving comparability across studies by enabling the reuse of task structures across contexts.

This vision intentionally builds on existing literature and established task design practices to provide a concrete and actionable starting point. Rather than aiming to fully capture the nature of code comprehension, the proposed facets reflect how the construct is currently operationalized in empirical research. As such, they are intended as a foundation that can be refined, extended, or revised, as further empirical evidence and alternative perspectives emerge. We therefore see this work as a starting point for a broader community discussion on how to conceptualize code comprehension. In what follows, we take initial steps toward this long-term vision by outlining a structured process for designing code comprehension tasks and illustrating the considerations that influence design decisions through a controlled experiment that we conducted for demonstration purposes.

5 From Research Question to Reusable Code Comprehension Tasks

Measuring (facets) of code comprehension requires careful design choices regarding tasks, participants, and materials to ensure valid and interpretable results. Feitelson [11] frames experiments on code comprehension as a "challenge–response game", where understanding is demonstrated through task performance, and shows that different task types probe different aspects of comprehension. He also argues that combining multiple tasks can improve validity and reveal otherwise hidden effects.

We build on this perspective and treat it as a starting point. In particular, we agree that task choice strongly shapes what aspect of comprehension is measured. At the same time, we add that these considerations benefit from a more explicit and structured link between cognitive targets, study goals, and practical constraints.

To make this concrete, we move from a set of abstract considerations to a step-by-step walkthrough: from research question to construct definition, to task design, to an instrument ready to be used in a study. The proposed process was developed through careful reflection within the author team, including perspectives from empirical software engineering and psychology, to ensure that the considerations align with established principles of measurement and test design [27]. We illustrate each step with our own case to show how design decisions are made in practice and what their consequences are.

Step 1: Clarify the Measurement Goal

The first step is to define what exactly should be measured in a given study. In our case, this starts from a typical empirical research question, for example, assessing whether a treatment (such as AI programming support or a teaching intervention) affects code comprehension. However, instead of treating code comprehension as a single construct, we explicitly specify which facet(s) are of interest.

Using a facet-based perspective, we identify whether the study targets analytical thinking, abstract thinking, critical thinking, or a combination of these. This decision directly determines what kind of cognitive processes tasks need to elicit. In our example, we deliberately

include all three facets to demonstrate how the full design space can be operationalized within a single set of tasks.

This step results in an explicit and structured definition of the construct that guides all subsequent design decisions. Additionally, it makes studies comparable, because the concrete definition is provided explicitly, so researchers can directly understand what studies measure the same facet(s).

Step 2: Define Context Constraints

After defining the measurement goal, the next step is to specify the context in which the tasks will be used. This includes both the **use case** of the measurement and the **characteristics of the participant population**, as both strongly constrain task design decisions.

Different use cases impose different requirements on how tasks should be constructed and interpreted. In controlled experiments, for example, the primary goal is typically to detect the effect of a treatment while maintaining controlled conditions and clear evaluation criteria. In contrast, educational settings, such as exams or training, introduce additional requirements, such as fairness, prevention of guessing, and efficient grading. In self-assessment or professional contexts, aspects like feedback value, time constraints, and robustness against strategic cheating behavior become more important.

In our case, we consider a typical empirical study setting in which code comprehension is measured as an outcome variable (e.g., to assess the effect of an intervention such as LLM support or a teaching approach). This implies that tasks need to be sensitive to differences in the targeted facet(s) while still being sufficiently controlled to allow meaningful comparison between conditions.

The participant population further constrains task design. We focus on graduate students with basic Python programming experience. This sets a medium level of expected proficiency: tasks that are too simple would not discriminate between participants, while overly complex tasks (e.g., large-scale real-world systems) would exceed reasonable cognitive limits.

These constraints together define the boundary conditions for task design. They determine what level of complexity is appropriate and how tasks should be framed to align with both the intended facet(s) of code comprehension and the practical realities of the study context. In other words, Step 2 is necessary to understanding the confounding factors that might bias the results. Explicitly defining them ensures a suitable task selection.

Step 3: Design Tasks Based on Facet Alignment

Given the construct definition (Step 1) and the constraints of the study context (Step 2), the next step is to design concrete tasks that operationalize the selected facet(s) of code comprehension. In our case, this means translating analytical, abstract, and critical thinking skills into observable task requirements.

Each facet implies different cognitive demands. To operationalize all three facets, we design a set of tasks that combines closed and free-text formats. Closed formats allow for controlled and comparable measurement, while free-text formats provide insight into participants' reasoning and help verify whether the intended facet is actually elicited. Free-text formats can potentially reveal the most details about the level of code comprehension of participants, as they require participants to actively recall and apply knowledge. However, they are also the most difficult to analyze and interpret. A combination of these formats enables internal consistency checks between selected answers and explanations.

We further select a familiar programming context based on standard algorithms typically covered in undergraduate courses. This reduces unnecessary domain-specific difficulty while still requiring genuine code comprehension. At the same time, we avoid overly familiar or trivial examples to ensure that participants engage in actual reasoning rather than recall. In general, contextualization can be an important aspect. A real scenario, perhaps even one to which participants can relate, could improve the participants' motivation. A fictitious scenario is especially useful when working with young participants.

Finally, we design tasks with potential reuse in mind by keeping the underlying task and cognitive demand stable while allowing researchers to adapt the code content, for example, to introduce experimental treatments such as different code formatting or naming styles.

In practice, this step results in a set of tasks, and each task can be traced back to a specific facet (or a combination) and a specific cognitive demand.

Step 4: Construct the Task Set

In this step, the individual tasks are assembled into a coherent test instrument. The main goal is to ensure that the selected tasks jointly cover the intended facet(s) while remaining balanced regarding difficulty, complexity, and time requirements.

We start by ensuring facet coverage: each selected task is explicitly linked to at least one of the selected facets. This mapping is not only conceptual but also guides practical decisions such as wording, expected reasoning steps, and response format. Where multiple facets are included, we ensure that they are not conflated within a single task unless explicitly intended so that their effects remain distinguishable.

Second, we consider task diversity and redundancy. Tasks targeting the same facet should differ in surface structure (e.g., code context, formulation, or response type) to avoid training or repetition effects while still probing the same underlying cognitive demand. At the same time, the overall set should not become redundant or overly long, especially in time-constrained experimental settings.

Third, we balance difficulty and coverage. Tasks are arranged so that no single facet is systematically easier or more difficult than others, as this would bias comparisons across conditions. In our case, we iteratively adjusted task complexity based on pilot solving to ensure that participants could engage with all tasks within the available time while still exhibiting meaningful variation in performance.

Finally, we ensure that the task set remains adaptable to the experimental context. This includes keeping code snippets modular and avoiding overly context-specific dependencies, so that tasks can be reused across studies or modified to include experimental manipulations (e.g., different code representations or annotations) without changing the underlying facet.

In practice, this step results in a structured collection of tasks where each task can be traced back to a specific facet and a specific cognitive demand, while the full set provides balanced coverage of all facets of code comprehension.

We give an overview of all tasks, including what cognitive process they require, the answer format, and the targeted facet, in Table 1. In total, we defined twelve tasks that cover fundamental programming concepts common within an academic setting, such as arrays, loops, conditionals, and simple algorithmic structures. Tasks were summarized into five groups, with each group using the same source code. They require students to trace and understand code step by step, recognize underlying patterns, and transfer these into more abstract representations. In addition, later tasks address evaluation and improvement of code, thereby assessing students' critical thinking with regard to code quality and readability.

Table 1 Code comprehension tasks. All task types that are not free-text formats are different variants of closed formats. Rows with the same background denote tasks with the same source code.

Activity	Task Type	Skills Tested
Loop & Index Understanding	Multiple Choice	Analytical, Abstraction
Edge Case Handling	Multiple Choice	Abstraction
Flowchart Comparison	Dichotomous Task	Analytical, Abstraction
Detect Output	Multiple Choice	Analytical, Abstraction
Assigning Inputs to Outputs	Matching Task	Analytical, Abstraction
Detecting Correct Statements on Behavior	Multiple Choice	Analytical, Abstraction
Method Naming	Free-text format	Analytical, Abstraction
Error Detection	Multiple Choice	Analytical, Abstraction, Critical Thinking
Code Snippet Comparison	Multiple Choice	Analytical, Abstraction, Critical Thinking
Functionality Check	Multiple Choice	Analytical, Abstraction, Critical Thinking
Code Evaluation	Free-text format	Critical Thinking
Code Improvement	Free-text format	Critical Thinking

Due to space constraints, we focus on three tasks (see the project Web site for details). The first task covered a loop with non-descriptive identifiers, requiring step-by-step tracing (analytical) while also allowing pattern recognition (abstract). It used a multiple-choice format in which the correct answer could only be determined by understanding the code; participants also explained their choice to assess the required thinking processes. The final two tasks required students to evaluate the quality and readability of the code, and based on that, suggest improvements, targeting critical thinking through free-text responses.

Step 5: Check What the Tasks Actually Measured

After constructing the task set, it is essential to examine what the instrument actually measures in practice. This step should be guided by established quality criteria from psychometrics, in particular *validity*, *reliability*, and *objectivity* [18, 27]. Validity concerns whether tasks measure the intended facet(s) of code comprehension; reliability refers to the consistency of measurement; and objectivity captures the extent to which results are independent of subjective interpretation. Secondary criteria such as fairness, economy, or robustness can further inform design decisions depending on the use case.

To assess these criteria, we applied our tasks in a controlled setting with 34 participants and complemented the results with a qualitative debriefing to collect feedback on the tasks. This revealed several important issues. Most notably, many tasks did not measure the intended facets; although designed to target multiple facets, they often elicited primarily analytical behavior. In some cases, participants focused on low-level details instead of the intended higher-level reasoning, directly threatening construct validity. In addition, certain response formats (especially free-text answers) introduced ambiguity and reduced objectivity, as we had not anticipated the range of valid responses.

For illustration, we discuss the results for the three tasks described in the previous step. For the first task, participants reported that it was clearly phrased and matched their skill

level. However, looking at the free-text responses, participants often confused the loop index with the array element and struggled with list construction, suggesting a mismatch between skill level and task complexity. Moreover, participants focused on individual code elements rather than higher-level patterns, indicating that the task did not capture abstract thinking as intended. For the two final tasks, the responses indicated the they indeed require critical thinking, such that students evaluated readability and correctness and also made useful suggestions for improvements. However, we noticed that students had difficulties separating the two tasks (which were based on the same source code), such that they suggested improvements already in the evaluation task. Thus, the improvement task does not add anything to the measurement of the construct and may not fulfill the validity criterion.

In this light, evaluating reliability of the tasks does not make much sense, as it does not matter how accurate the tasks measure something that we do not know. If it is unclear what a task measures, consistency metrics, such as internal agreement provide limited insight. However, once validity is reasonably ensured, reliability analyses (e.g., internal consistency) can help identify tasks that do not align well with the overall instrument.

Regarding objectivity, evaluating multiple-choice responses is straightforward, as only one option is correct. For free-text responses, objectivity is lower. To improve it, the first author evaluated the responses, and two other authors reviewed the evaluation. For essay-based tasks, we defined possible solutions in advance. However, some tasks were phrased imprecisely, and unexpected responses emerged, threatening objectivity. To mitigate this, 3 of the 4 authors reviewed the responses. Thus, objectivity could be improved by phrasing the questions more precisely and clarifying how unexpected responses should be evaluated.

Overall, the evaluation showed that 10 out of 12 tasks required revision. Importantly, the explicit mapping between tasks and facets allowed us to detect these issues directly. Without such transparency, we might have drawn incorrect conclusions about the absence of effects in a study when, in reality, the targeted facet was not measured at all. If, for example, we had evaluated a teaching method to improve abstraction during code comprehension, and the results did not show any improvement in abstract thinking skills, we would assume that the teaching method had no effect, when in reality, we just did not *measure* abstraction with our tasks. Thus, we would draw incorrect conclusions, and nobody would even notice.

Based on these results, we iteratively improved the instrument: we removed or adapted overly complex or ambiguous tasks, refined response formats to improve objectivity, incorporated unexpected but valid answers, and rebalanced the task set across facets. A second application with a comparable group of eight participants indicated clear improvements.

Such iterative refinement of tasks is still rare in code comprehension research, where instruments are often used only once. However, as our experience shows, it is a necessary step for developing robust measurement instruments and aligns with standard practice in test development [15]. This also includes continuous evaluation of secondary quality criteria, such as fairness (i.e., that there is no systematic discrimination of demographic groups) or scaling (i.e., that scores reflect the actual skill level).

Step 6: Make Your Instrument Openly Available

The final step is to make the tasks and their design rationale openly accessible, including their descriptions, mapping to intended facets, design decisions, and observed empirical behavior.

Open availability is a prerequisite for building a shared and reusable pool of code comprehension tasks. Without access to concrete instruments and their documented properties, it remains difficult for other researchers to reuse, compare, or systematically improve existing tasks. In contrast, making tasks and their associated metadata publicly available enables

incremental refinement across studies.

In line with our long-term vision, such a shared repository allows researchers to explicitly specify the facet(s) of code comprehension they aim to investigate and select corresponding validated tasks from a growing pool. In our case, we therefore provide all tasks, code snippets, and response formats on the project's Web site, together with documentation of their intended facet alignment and empirical observations. This is a first step toward a community-driven infrastructure for more systematic and transparent measurement of code comprehension.

6 Conclusion

In this vision paper, we introduce a facet-based perspective on code comprehension and a structured process for designing corresponding tasks. Together, both provide a practical foundation for making task design more explicit, systematic, and aligned with what researchers intend to measure. Our case illustrates both the challenges and the value of this approach: Even carefully designed tasks may fail to capture the intended facets, but making assumptions explicit enables targeted improvement through iteration. In this sense, task design should be understood as instrument development rather than a one-off activity.

This vision paper does not aim to provide a final definition or a complete measurement framework for code comprehension. Instead, it provides a starting point for a more systematic and cumulative approach to designing and evaluating code comprehension tasks. The proposed process builds on established principles of test construction in psychology, but it should be refined and extended through future empirical work and community discussion. In particular, future research can explore how the facet-based perspective applies to more realistic software engineering activities, such as maintenance tasks involving larger code bases, documentation, and incomplete information, as well as to emerging development contexts involving AI-based tools and assistance.

Our long-term vision is a shared pool of empirically grounded and validated code comprehension tasks, organized by facets and reusable across studies. The process outlined in this paper is a step toward this goal, as it supports transparent design decisions, facilitates comparison and reuse, and enables the community to iteratively improve how code comprehension is measured. We hope that future work adopts and extends this perspective by making task design and validation more explicit, contributing to a cumulative and more reliable measurement practice in code comprehension research.

Data Availability

Supplemental materials are available on the project's Web site: `https://doi.org/10.5281/zenodo.21490313`.

References

1 Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Europ. Conf. Software Maintenance and Reengineering (CSMR)*, pages 181–190. IEEE CS, 2011.

2 Nahla J. Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I. Maletic. Developer reading behavior while summarizing Java methods: size and context matters. In *Proc. Int. Conf. Software Engineering (ICSE)*, ICSE '19, page 384–395. IEEE CS, 2019.

3 Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.*, 7(OOPSLA1):85–111, 2023.

4 Victor R. Basili and Harlan D. Mills. Understanding and Documenting Programs. *IEEE Transactions on Software Engineering*, SE-8(3):270–283, 1982.

5 Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G. Feitelson. Meaningful Identifier Names: The Case of Single-Letter Variables. In *Proc. Int. Conf. Program Comprehension (ICPC)*, pages 45–54, 2017.

6 Percy Bridgman. *The Logic of Modern Physics*. The Macmillan Company, 1927.

7 Ruven Brooks. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

8 J. Cronbach and P. E. Meehl. Construct Validity in Psychological Tests. *Psychological Bulletin*, 52(4):281–302, 1955.

9 Sarah Fakhoury, Devjeet Roy, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Softw. Eng.*, 25(3):2140–2178, 2020.

10 Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Softw. Eng.*, 18(4):699–745, 2013.

11 Dror G Feitelson. Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension. *Empirical Software Engineering*, 27(6):123, 2022.

12 David J. Gilmore. Models of Debugging. *Acta Psychologica*, 78(1):151–172, 1991.

13 Pavlína Wurzel Gonçalves, Pooja Rani, Margaret-Anne Storey, Diomidis Spinellis, and Alberto Bacchelli. Code Review Comprehension: Reviewing Strategies Seen Through Code Comprehension Theories. In *Proc. Int. Conf. Program Comprehension (ICPC)*, pages 589–601. IEEE CS, 2025.

14 Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 129–139, New York, NY, USA, 2017. Association for Computing Machinery.

15 Daniel Graziotin, Per Lenberg, Robert Feldt, and Stefan Wagner. Psychometrics in behavioral software engineering: A methodological introduction with guidelines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–36, 2021.

16 Latifa Guerrouj, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An experimental investigation on the effects of context on source code identifiers splitting and expansion. *Empirical Softw. Eng.*, 19(6):1706–1753, 2014.

17 L. Gugerty and G. Olson. Debugging by Skilled and Novice Programmers. In *Conf. Human Factors in Computing Systems (CHI)*, CHI '86, pages 171–174, New York, NY, USA, 1986. ACM Press.

18 Harold Gulliksen. *Theory of Mental Tests*. John Wiley & Sons, 1 edition, 1950.

19 Mohammed Hassan, Kathryn Cunningham, and Craig Zilles. Evaluating Beacons, the Role of Variables, Tracing, and Abstract Tracing for Teaching Novices to Understand Program Intent. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, pages 329–343. ACM Press, 2023.

20 Felienne Hermans and Efthimia Aivaloglou. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *Proc. Int. Conf. Program Comprehension (ICPC)*, pages 1–10. IEEE CS, 2016.

21 Sora Chi-Fang Huang, Zhi-Hong Chen, Yu-Tzu Lin, Martin K. Yeh, and Yi-Wei Chen. Exploring the Link Between Problem-Solving Strategies and Programming Performance: A Comparative Analysis of High and Low Performers. *Journal of Educational Computing Research*, 64(2):493–523, 2026.

22 Kang il Park, Jack Johnson, Cole S. Peterson, Nishitha Yedla, Isaac Baysinger, Jairo Aponte, and Bonita Sharif. An Eye Tracking Study Assessing Source Code Readability Rules for Program Comprehension. *Empirical Softw. Eng.*, 29(160), 2024.

23 Cruz Izu and Claudio Mirolo. Comparing Small Programs for Equivalence: A Code Comprehension Task for Novice Programmers. In *Annual Conf. Innovation and Technology in Computer Science Education (ITiCSE)*, ITiCSE '20, page 466–472, New York, NY, USA, 2020. Association for Computing Machinery.

24 Maria Kallia. The Search for Meaning: Inferential Strategic Reading Comprehension in Programming. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, pages 1–14. ACM Press, 2023.

25 Martin Konopka, Adam Talian, Jozef Tvarozek, and Pavol Navrat. Data flow metrics in program comprehension tasks. In *Proc. Int. Workshop on Eye Movements in Programming*, EMIP '18. ACM Press, 2018.

26 Makrina Viola Kosti, Kostas Georgiadis, Dimitrios A. Adamos, Nikos Laskaris, Diomidis Spinellis, and Lefteris Angelis. Towards an affordable brain computer interface for the assessment of programmers' mental workload. *International Journal of Human-Computer Studies*, 115:52–66, 2018.

27 Helfried Moosbrugger and Augustin Kelava. *Testtheorie und Fragebogenkonstruktion*. Springer eBook Collection. Springer, Berlin, Heidelberg, 3 edition, 2020.

28 Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. Commenting source code: is it worth it for small programming tasks? *Empirical Softw. Eng.*, 24(3):1418–1457, 2019.

29 A. F. Norcio. Indentation, documentation and programmer comprehension. In *Conf. Human Factors in Computing Systems (CHI)*, CHI '82, page 118–120, New York, NY, USA, 1982. Association for Computing Machinery.

30 Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In *Int. Conf. Software Maintenance & Evolution (ICSME)*, pages 348–359, 2020.

31 Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 524–536. IEEE CS, 2021.

32 Norman Peitek, Janet Siegmund, and Sven Apel. What Drives the Reading Order of Programmers? An Eye Tracking Study. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 342–353. ACM, 2020.

33 Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, Johannes C. Hofmeister, and André Brechmann. Simultaneous Measurement of Program Comprehension with fMRI and Eye Tracking: A Case Study. In *Int. Symp. Empirical Software Engineering and Measurement (ESEM)*. ACM Press, 2018.

34 Nancy Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19(3):295–341, 1987.

35 Cristiano Politowski, Foutse Khomh, Simone Romano, Giuseppe Scanniello, Fabio Petrillo, Yann-Gaël Guéhéneuc, and Abdou Maiga. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. *Information and Software Technology*, 122:106278, 2020.

36 Paul Ralph and Ewan Tempero. Construct Validity in Software Engineering Research and Software Metrics. In *Int. Conf. Evaluation and Assessment in Software Engineering (EASE)*, EASE '18, page 13–23, New York, NY, USA, 2018. Association for Computing Machinery.

37 Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An empirical study on program comprehension with reactive programming. In *Proc. ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE)*, FSE 2014, page 564–575, New York, NY, USA, 2014. Association for Computing Machinery.

38 Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically Assessing Code Understandability. *IEEE Transactions on Software Engineering*, 47(3):595–613, 2021.

39 Todd Sedano. Code Readability Testing, an Empirical Study. In *Proc. IEEE Int. Conf. Software Engineering Education and Training (CSEET)*, pages 111–117, 2016.

40 Matheus Segalotto, Willian Bolzan, and Kleinner Farias. Effects of modularization on developers' cognitive effort in code comprehension tasks: A controlled experiment. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, SBES '23, page 206–215, New York, NY, USA, 2023. Association for Computing Machinery.

41 Teresa Shaft and Iris Vessey. The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information Systems Research*, 6(3):286–299, 1995.

42 Anshul Shah, Thanh Tong, Elena Tomson, Steven Shi, William G. Griswold, and Adalbert Gerald Soosai Raj. Students' Program Comprehension Processes in a Large Code Base. In *Proc. Int. Conf. Program Comprehension (ICPC)*, pages 182–193. IEEE CS, 2025.

43 Ben Shneiderman and Richard Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer & Information Science*, 8(3):219–238, 1979.

44 Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 378–389. ACM Press, 2014.

45 Janet Siegmund, Norman Peitek, André Brechmann, Chris Parnin, and Sven Apel. Studying Programming in the Neuroage: Just a Crazy Idea? *Commu. ACM*, 63(6):30–34, 2020.

46 Janet Siegmund and Jana Schumann. Confounding Parameters on Program Comprehension: a Literature Survey. *Empirical Softw. Eng.*, 20:1159–1192, 2015.

47 Dag I.K. Sjøberg and Gunnar R. Bergersen. Construct Validity in Software Engineering. *IEEE Transactions on Software Engineering*, 49(3):1374–1396, 2022.

48 Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984.

49 Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. An eye-tracking study assessing the comprehension of c++ and Python source code. In *Proc. Symposium Eye Tracking Research and Applications (ETRA)*, ETRA '14, page 231–234, New York, NY, USA, 2014. Association for Computing Machinery.

50 Anneliese von Mayrhauser and Marie Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.

51 Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An Observational Investigation of Reverse Engineers' Processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1875–1892, 2020.

52 Stefan Wagner and Marvin Wyrich. Code comprehension confounders: A study of intelligence and personality. *IEEE Transactions on Software Engineering*, 48(12):4789–4801, 2021.

53 Marvin Wyrich. Source Code Comprehension: A Contemporary Definition and Conceptual Model for Empirical Investigation, 2023.

54 Marvin Wyrich, Justus Bogner, and Stefan Wagner. 40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study. *ACM Computing Surveys*, 56(4):1–42, 2023.

55 Marvin Wyrich, Marvin Munoz Baron, and Justus Bogner. Apples, Oranges, and Software Engineering: Study Selection Challenges for Secondary Research on Latent Variables. In *Proc. IEEE/ACM Int. Workshop Methodological Issues with Empirical Studies in Software Engineering*, pages 42–47, 2024.

56 Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018.