Unraveling Ambiguities: Analyzing Student Approaches to Solving Probeable Problems

Viraj Kumar
Indian Institute of Science
Bengaluru, India
viraj@iisc.ac.in

Paul Denny
University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

James Prather
Abilene Christian University
Abilene, TX, USA
james.prather@acu.edu

Stephen MacNeil
Temple University
Philadelphia, PA, USA
stephen.macneil@temple.edu

Abstract

Introductory programming courses often rely on small code-writing tasks with clear and complete problem specifications. Such tasks limit opportunities for students to practice clarifying ambiguous requirements – a critical skill in real-world programming. Given the ubiquity of large language models (LLMs) that can produce accurate solutions for such well-specified tasks, students may question the relevance of learning to write code for such tasks. Probeable Problems, by contrast, deliberately omit essential details, encouraging students to think critically, identify ambiguities, and seek clarifications. This study builds on theories of metacognition, cognitive flexibility, and design thinking to explore the strategies used by human students when tackling ambiguous programming tasks. For three Probeable Problems, we analyze the contents of 40,000 student-written probes to investigate how thoroughly and efficiently students are able to explore ambiguities. Our results offer insights into the effective design of feedback for Probeable Problems, and emphasize the benefits of engaging learners with authentic, ill-structured problems to enhance critical thinking and metacognitive skills.

CCS Concepts

• **Social and professional topics** → *Computing education.*

Keywords

Probeable Problems, CS1, test cases, requirements, ambiguity

ACM Reference Format:
Viraj Kumar, Paul Denny, Juho Leinonen, James Prather, and Stephen MacNeil. 2026. Unraveling Ambiguities: Analyzing Student Approaches to Solving Probeable Problems. In *28th Australasian Computing Education Conference (ACE 2026), February 09–13, 2026, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3786228.3786255

This work is licensed under a Creative Commons Attribution 4.0 International License.
ACE 2026, Melbourne, VIC, Australia
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2352-0/26/02
https://doi.org/10.1145/3786228.3786255

1 Introduction

Introductory programming courses typically use small code-writing tasks with clearly specified requirements. While effective for teaching syntax and basic programming concepts, such tasks offer limited opportunities for students to practice handling ambiguous or incomplete specifications. Clarifying requirements is a critical skill in real-world programming, where requirements are often communicated in natural language and may be incomplete or unclear [16, 45]. Failure to resolve ambiguities early in the development process can lead to costly errors [13].

Traditional programming exercises, which often rely on automated grading tools, reinforce a focus on well-defined problems [3, 17, 27]. However, the rise of large language models (LLMs) presents a new challenge: well-specified tasks can now be reliably solved by LLMs [12, 36]. Studies have already shown that multimodal models can even solve Parsons problems and complete graph and tree traversals based only on an image and problem description [19, 21]. This has the potential to harm student engagement and lead students to question the relevance of manually solving such tasks that some students increasingly perceive as 'busy work' [53].

Probeable Problems introduce ambiguity into programming tasks by deliberately omitting key details. These problems require students to generate 'probes' – test inputs submitted to an 'oracle' that clarify expected behavior – before submitting code and obtaining feedback based on instructor-written test cases. Such tasks not only simulate real-world programming scenarios but also resist trivial solutions from LLMs, because the lack of detail prevents LLMs from producing correct code solutions via a 'copy-and-paste' approach. This aligns with metacognitive theories, which emphasize that tasks requiring active cognitive engagement can help students develop self-regulation and informed decision-making skills [15, 54]. By navigating uncertainty and making strategic choices about when to stop probing and start coding, students enhance their ability to monitor and adapt their learning strategies.

Design thinking and cognitive flexibility theory further support the use of Probeable Problems in educational contexts. Design thinking promotes iterative problem framing and comfort with ambiguity [41]. The process of probing – actively generating test inputs to clarify a problem – is a natural fit for design thinking, as it encourages exploration and reframing of the problem space. Similarly, cognitive flexibility theory [48] highlights the value of

engaging with ill-structured problems, where multiple perspectives and adaptive thinking are required. By encouraging students to think about and consider different interpretations of a problem, Probeable Problems help support the development of flexible problem-solving strategies.

While prior work has analyzed the *quantity* of student probes in both small- and large-scale programming contexts [10, 31], the current study examines the *content* of those probes, as well as incorrect code submissions. Since ambiguous tasks allow for multiple interpretations, our analysis investigates *whether* and *how efficiently* students use probes and feedback from incorrect code submissions to discard some of these interpretations, and thereby hone in on a solution consistent with the desired interpretation. As noted by Denny et al. [10], students should ideally be able to explore the problem space thoroughly enough via probing alone. We provide a definition of thoroughness, drawing inspiration from Wrenn and Krishnamurthi's definition in the context of creating thorough test cases [52].

Our analysis is structured around the following research questions:

- **RQ1:** How thoroughly are students able to explore the input space of Probeable Problems?
- **RQ2:** How efficiently do students explore the input space?
- **RQ3:** What are the strategies that students use to probe and when do they decide to start coding?

By exploring these questions, we aim to provide insights into designing educational tools and tasks that support critical thinking and metacognitive skills, preparing students for real-world programming challenges.

2 Related Work

2.1 Ambiguity in Code Specifications

The idea of incorporating ambiguity into programming assignments dates back to at least 1978, when Schneider [43] argued that CS1 students must develop the ability to "recognize and resolve uncertainties in simple problem statements". To develop this ability, he proposed that "some programming assignments should intentionally be left incomplete, requiring the student to consider the alternatives and to make a reasonable decision on the omitted details". To date, this approach has not been widely adopted in introductory programming education. Instead, Luxton-Reilly et al. [26] highlight the dominance of an exercise-intensive approach to introductory programming pedagogy, supported by the widespread use of automated assessment tools. These tools have reinforced an emphasis on well-defined problems with clearly specified, unambiguous requirements [27, 30].

While code-writing exercises are effective for mastering syntax and basic programming concepts, an excessive emphasis on clearly specified tasks can reduce the opportunity for students to develop higher-order cognitive and metacognitive skills such as handling ambiguous requirements. These skills, which have always been essential for real-world programming [16, 45], are perhaps more relevant in an era with powerful code-generation models [14, 19, 21, 24, 42]. Therefore, the computing education community is keenly interested in exploring new types of learning activities [12, 36, 39].

2.2 Task Comprehension

Even with well-specified tasks, novice programmers often struggle to develop a correct understanding of the problem before attempting to solve it. To address this, Craig et al. [9] and Denny et al. [11] asked students to solve or create test cases before writing code. They showed that this exercise can enhance their students' comprehension of the task and reduce errors in the eventual code. Similarly, the tools developed by Pechorina et al. [32] as well as Wrenn and Krishnamurthi [52] provide metacognitive scaffolding by encouraging students to write and solve test cases before writing code, which again leads to fewer errors in the resulting code. Since the task in Probeable Problems is deliberately vague, asking students to write test cases would force them to guess what the expected behavior should be on certain test inputs. Since this can be frustrating, Pawagi and Kumar propose providing students with an oracle that reveals the expected behavior on any input specified by the student [31]. They evaluated Probeable Problems in a two-hour programming contest where the use of AI tools such as ChatGPT and GitHub Copilot was permitted. However, they found that most students were unable to explore the problem space well enough, often failing to recognize the lack of crucial details such as tie-break mechanisms. In this work, we investigate the probing behavior of students in greater detail, in the context of a large introductory programming course.

3 Methods

3.1 Course Context and Data Collection

To address our research questions, we collect data from a large introductory programming course taught at the University of Auckland during the second half of 2024. A total of 1028 students were enrolled in the course. To integrate Probeable Problems into the curriculum, we selected three consecutive weekly lab sessions. Labs in the course each contribute 1% towards a student's final grade. Although this is a relatively small amount of credit, due to the competitive nature of the course it is typically sufficient to promote a high level of participation.

Each of the three lab sessions featured one Probeable Problem which was included alongside several traditional programming exercises. We will refer to these problems as P7, P8, and P9, where the numbers correspond to the respective week in the course. The problem descriptions and default probes provided to students are summarized in Table 1.

For delivery of the Probeable Problems, we utilized the CodeRunner platform, an open-source auto-grading plugin for Moodle that allows instructors to define problem statements, specify input-output test cases, and include model solutions to validate test cases during problem setup. To implement Probeable Problems, we leveraged CodeRunner's ability to match expected outputs using regular expressions, enabling a wildcard-based oracle system that could return correct outputs for any student-submitted probe. Figure 1 illustrates how probes were entered and evaluated in the system.

Following standard course policies, incorrect code submissions incurred a small grading penalty, increasing by 5% per unsuccessful attempt. However, probes carried no penalties, allowing students to submit as many as needed before writing code. Each Probeable Problem consisted of two phases: first, students submitted probes

Table 1: The Probeable Problem statements, each with 16 interpretations, and the default probe provided to students. The desired interpretation is indicated by choices in bold. Selecting any of the 15 other combinations of choices leads to incorrect implementations. Default probes are carefully constructed so that they do not conflict with any incorrect interpretation.

Problem	Statement	Multiple interpretations (count)	Default Probe
P7	Implement a function to count the number of integers in an array of length n between a and b	Count unique or **all**. Assume $a \leq b$ or **no**. Include endpoint a or **no**. Include endpoint b or **no**. $(2 \times 2 \times 2 \times 2 = 16)$	CountBetween({1, 2, 3}, 3, 0, 5) ⟶ 3
P8	Implement a function to search an array of length n for the smallest even value	Assume at least one even in input or **no**. Assume answer appears exactly once (in which case: report value or index) or **no** (in which case: report value (first or all) or **index** (first, last, all ascending, or **all descending**)). $(2 \times (2 + 2 + 4) = 16)$	SmallestEven({50, 25, 2, 30, 45}, 5) ⟶ 2
P9	Implement a function to find the first vowel in a string	Assume at least one vowel in input or **no**. Assume only lowercase vowels (in which case: vowel or alphabetic order) or **no** (in which case: Index, ASCII, or **alphabetic** order; preserve input case or **no**). $(2 \times (2 + 3 \times 2) = 16)$	FirstVowel("apple") ⟶ 'a'

to explore the problem's behavior, and second, they implemented a code solution based on their findings. The interface for entering and checking code, along with an illustration of the test case feedback, is shown in Figure 2. Students were encouraged (but not required) to start with the probing phase, and were free to switch between probing and coding phases as often as they wished.

As can be seen in Figure 1, to introduce these tasks to students, we adopted the "Ask the Client" framing, inspired by the Pawagi and Kumar paper [31]. Each problem began with a prompt encouraging students to clarify vague requirements before coding. The instructions emphasized the fact that probes were penalty-free, and thus should be used to identify missing details before attempting to write and submit a code solution.

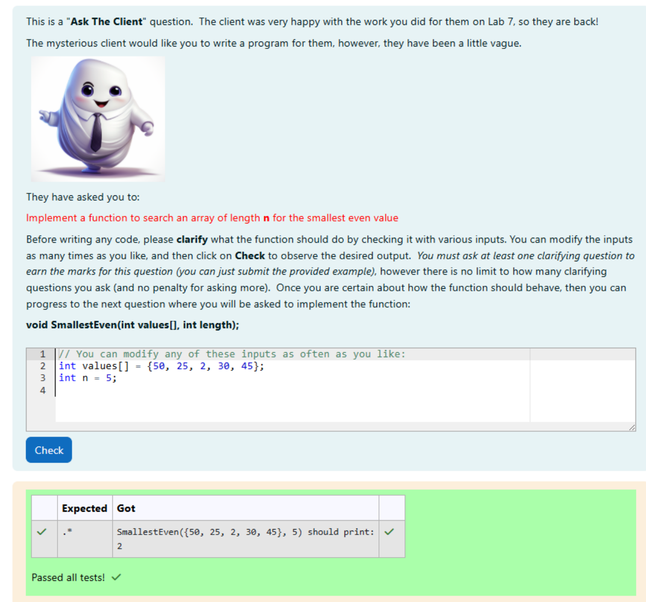

Figure 1: Presentation of the probing interface within the CodeRunner platform. Students can modify the inputs as presented in the editing pane, and then view the expected output by clicking the "Check" button.

3.2 Definitions

A task specification that omits one or more details can be interpreted in multiple ways. For a Probeable Problem, we assume that there is a finite set of interpretations that are reasonable in Scheneider's sense [43]. The correct solution corresponds to one of these interpretations, and the remaining interpretations yield $N \geq 1$ incorrect implementations. As shown in Table 1, each Probeable Problem in this study has $N = 15$ incorrect implementations.

An *attempt* to solve a Probeable Problem is a sequence of probe and code submissions. Each probe is an input to the oracle, and all incorrect implementations under consideration whose behavior differs from the correct solution's behavior (as revealed by the oracle) should be discarded. We say that a probe is *productive* if at least one of the remaining incorrect implementations can be discarded. We have ensured that the default probes we provide (see Table 1) are *unproductive*. Many student attempts began with this default probe, and they occasionally included duplicate probes. To focus our analysis on genuinely unproductive probes, we ignore these two "trivial" types of unproductive probes from each attempt.

In our study, each code submission is tested against a fixed sequence T of instructor-supplied test cases, and a code submission is deemed correct if it passes all tests. By this definition, a student's incorrect code submission will fail some first test case in sequence T, as illustrated in Figure 2. Since we reveal this failing test case, this information may permit some incorrect implementations to be discarded from further consideration. If so, we say that this failing code submission is *productive*; otherwise it is *unproductive*.

At any point during an attempt, we can classify each of the N incorrect implementations into one of three mutually exclusive categories: discarded due to a probe, discarded due to a failing code submission's revealed test case, or remaining under consideration.

Once you are confident you know how the function should work, implement it below.

Implement a function to search an array of length **n** for the smallest even value

void SmallestEven(int values[], int length);

```
#include <limits.h>

void SmallestEven(int arr[], int n) {
    int min_even = INT_MAX; // Initialize with a large value
    int found = 0;

    for (int i = 0; i < n; i++) {
        if (arr[i] % 2 == 0 && arr[i] < min_even) {
            min_even = arr[i];
            found = 1;
        }
    }

    if (found) {
        printf("%d\n", min_even);
    } else {
        printf("No even number found\n");
    }
}
```

Check

	Test	Expected	Got	
✓	int values[5] = {50, 25, 2, 30, 45}; SmallestEven(values, 5);	2	2	✓
✗	int values[5] = {1, 2, 3, 4, 5}; SmallestEven(values, 5);	1	2	✗

Figure 2: Presentation of the programming interface within the CodeRunner platform, a widely used automated assessment tool. The code shown in the editing pane is the verbatim output from an LLM (GPT-4o) when provided with the prompt: *"Write a function in "C" called "SmallestEven" to solve this problem: Implement a function to search an array of length n for the smallest even value. SmallestEven({50, 25, 2, 30, 45}, 5) should print: 2"*, which consists of all of the information provided to a student. When the code fails, as it does in this case, the first failing test case is shown to the student.

At the end of the attempt, if n_p and n_c are the number of incorrect implementations discarded due to probes and code respectively, we define the *probe-thoroughness* and the *code-thoroughness* of this attempt as $\frac{n_p}{N}$ and $\frac{n_c}{N}$ respectively. The *cumulative thoroughness* is the sum of these fractions. Note that all three types of thoroughness lie in the closed interval $[0, 1]$.

3.3 Thematic Analysis of Students Open Responses

At the end of the semester, students had the opportunity to participate in an optional survey about the strategies they used to solve 'Ask the Client' questions. The optional survey resulted in a moderate response ($n = 102$) with some questions left blank. The following question was analyzed for the thematic analysis:

> *How would you describe your approach to tackling these 'Ask the Client' questions? If you were giving advice to a new student who was solving these kinds of tasks for the first time, what strategies would you suggest they use in order to be successful?*

The responses were qualitatively analyzed by a researcher on the team. The researcher followed an iterative and inductive process which was guided by best practices [8] starting with familiarization and followed by open coding and developing the themes. Given the exploratory nature of this analysis and the low sample size, the goal was to provide rich insights to help contextualize our quantitative findings. The themes and observations are therefore intended to be suggestive, but not conclusive or generalizable.

4 Results

We logged a total of 2,900 student attempts (976 for P7, 967 for P2, and 957 for P3). Almost 90% of these attempts culminated in code that passed every instructor-supplied test (for P7: 903 attempts; for P8: 820 attempts; for P9: 817 attempts). Less than 2% of student attempts consisted of a single successful code submission. Since these attempts contained no probes, they were likely dishonest attempts and we discard them from our subsequent analysis.

4.1 Exploration Thoroughness (RQ1)

Our first research question asks: "How thoroughly are students able to explore the input space of Probeable Problems?". We defined two measures of thoroughness in Section 3.2 – essentially we are interested in understanding the extent to which probes or test case feedback on failing code submissions were used to discard the incorrect interpretations of each problem, and how many such interpretations were discarded. In particular, given that probe submissions are not penalized in the same way that code submissions are, the metric of probe thoroughness (the proportion of all incorrect interpretations discarded due to probes) is of particular interest. We find that thoroughness improved as students gained familiarity with Probeable Problems.

Figure 3 shows probe- and code-thoroughness of all attempts for all the problems in our study. Recall that P7 was the first exposure students had to Probeable Problems. Unsurprisingly, the average thoroughness of student attempts for P7 was somewhat low: 68.3% for probes and 86.2% in total. On subsequent labs, students were able to improve both their probe thoroughness (83.8% for P8, 77.9% for P9) and their cumulative thoroughness (96.6% for P8, 91.5% for P9). With experience, students also became more effective at discarding incorrect implementations *before* submitting their first code submission as shown in Figure 4. For P7, 11 out of 15 incorrect interpretations were not discovered before the first code submission in over 50% of attempts. For subsequent problems, this number fell (4 out of 15 incorrect P8 implementations and 9 out of 15 incorrect P9 implementations), suggesting that experience helped some students improve their ability to imagine alternative interpretations.

4.2 Exploration Efficiency (RQ2)

Our second research question asks: "How efficiently do students explore the input space?". On a given attempt (i.e., a sequence of probe and code submissions), a probe or code submission is considered productive if it reveals information that permits one or more of the incorrect interpretations of the problem to be discarded. In contrast, submitting a code or probe is *unproductive* if it does not reveal any useful information. For any two attempts that are equally thorough (i.e., they reveal the same proportion of incorrect implementations), the one that involves fewer unproductive code or probe submissions can be considered more efficient.

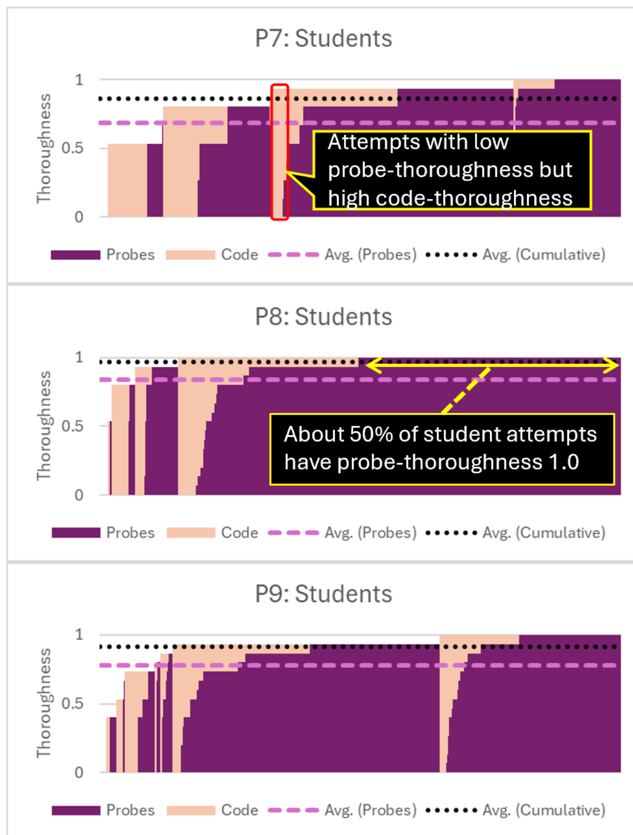

Figure 3: Probe- and code-thoroughness of all attempts by students for problems P7 (top), P8 (middle), and P9 (bottom). In each sub-figure, attempts are ordered in increasing order of cumulative thoroughness.

Figure 5 shows that the efficiency of student probing improved as they gained familiarity with Probeable Problems, with a higher proportion of attempts clustering near the upper-left corner of each sub-plot. To achieve perfect probe thoroughness (1.0), several students issued over 60 unproductive probes for problem P7. In contrast, many students achieved perfect probe thoroughness for problems P8 and P9 while issuing fewer than 40 unproductive probes. The high number of unproductive probes issued by some students might indicate maximizing tendencies (see the discussion in Section 5.1). However, the large number of unproductive code submissions by some students may indicate that they were "stuck in a rut".

4.3 Exploration Strategies

Our third research question asks: "What are the strategies that students use to probe and when do they decide to start coding?"

The qualitative analysis investigates the responses students shared in the open-response question about the strategies they used for solving 'Ask the Client' questions. The analysis uncovered a few prominent strategies, including emphasizing planning before writing code and attempts to disrupt their fixated interpretation of the problem description.

4.3.1 Embracing Requirements Elicitation and Test-Driven Development. Most of the participants emphasized the need to get the requirements correct from the start rather than writing code and iteratively uncovering them over time. Participants who focused on getting the requirements right up front described brainstorming all of the possible corner cases before writing the code. For example, P30 wanted to deeply understand the task before writing any code:

> *"Spend as much time as possible asking the client for outputs of all possible values and edge cases that they can think of. This is important because eventually, they will know exactly what the function needs to do in every possible situation - which is extremely important for writing the code correctly." (P30)*

This was a common sentiment from students that testing should take priority over coding to ensure the understanding is correct. For example, P70 described this need to focus on testing:

> *"I enjoyed the Ask the Client questions, but testing for edge cases was incredibly hard due to their sneakiness. My advice would be to just go insane on testing, and not on actually writing the code." (P70)*

In fact, some students did issue an extraordinarily large number of probes, despite the fact that most of these were unproductive (Figure 5).

One student (P56) was so hesitant to submit their code that they described running the code on their local machine to avoid submitting it before it was completely correct. *"...test for all edge cases they can think of, create a function that creates the desired outputs, and test it on your own machine *before* submitting it."*

This emphasis on testing before writing code is a best practice [23], but also not something that most computing students tend to value [2, 6, 7]. Students often test their code later, and their testing behavior varies widely from assignment to assignment [6]. Prior work has shown that this is not only a problem for first-year students, but even graduating seniors lack testing skills [7]. Consequently, the emphasis students are placing on testing is encouraging and more work is needed to understand whether this test-first approach transfers to their other programming activities.

4.3.2 Deliberately Disrupting Their Own Process. Students also described the need to break out of their initial plan to explicitly consider other ideas. Experts tend to have many strategies and heuristics that help them to solve problems efficiently. These shortcuts can help them reduce their cognitive load, but can also be a source of design fixation [22]. Multiple participants in our study described finding it necessary to 'doubt' their own reasoning and 'think outside of the box' to solve these problems. For example, P40 described the need to 'disprove [their] current train of thought' to ensure that their interpretation of the problem did not overshadow the requirements of the client:

> *"Think outside of the box when testing. Rather than trying to prove your theory about how YOU THINK the function SHOULD work, try and find all the ways that you can disprove your current train of thought. Only*

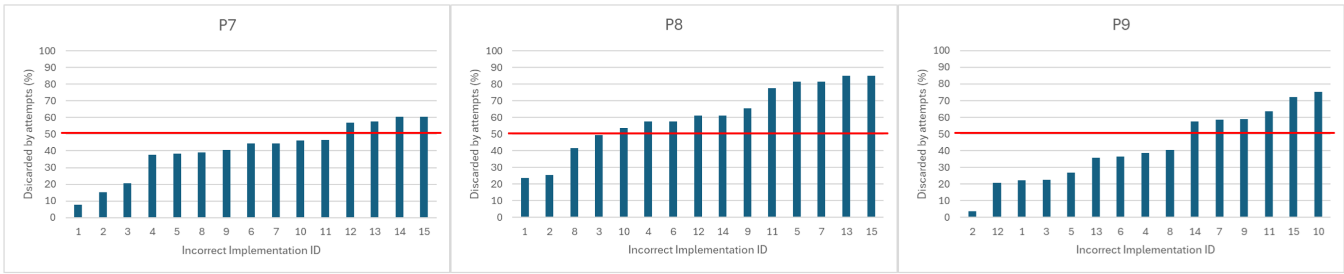

Figure 4: For each Probeable Problem, we show the percentage of student attempts that are able to discard each of the 15 incorrect implementations (numbered 1 to 15) *before* the first code submission. A low percentage suggests that the implementation is difficult to discard whereas a high percentage suggests that it is easy to discard.

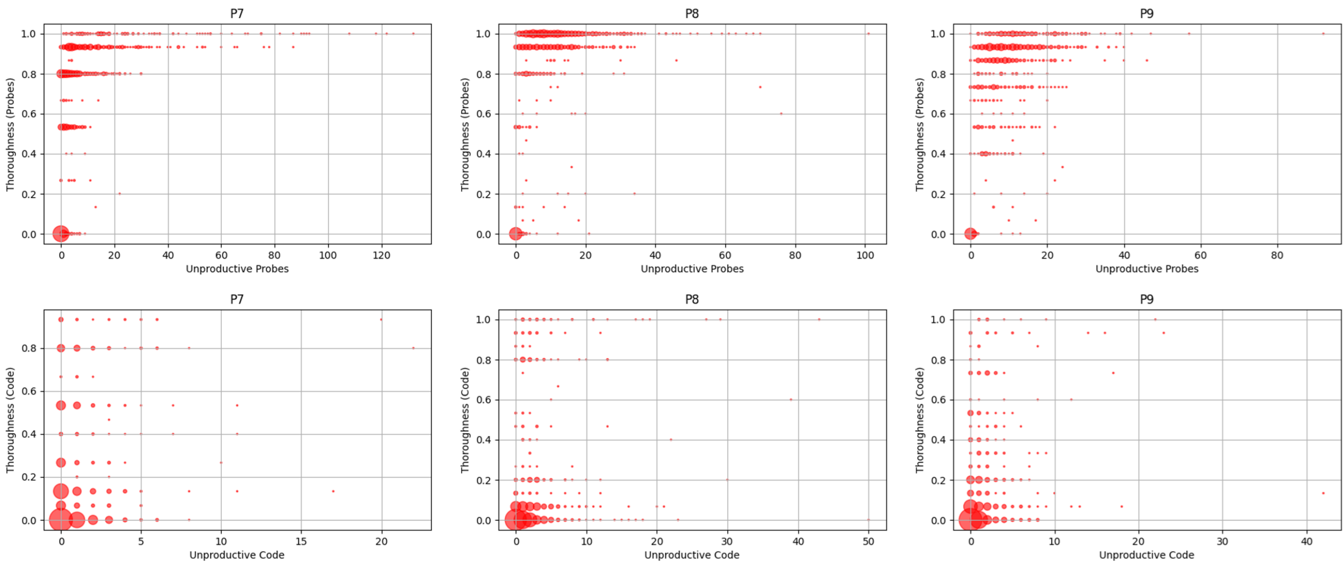

Figure 5: Probing efficiency of student attempts across problems P7 (left), P8 (middle), and P9 (right). For each problem, the upper sub-plot shows the number of unproductive probes (x-axis) against probe-thoroughness (y-axis) whereas the lower sub-plot shows the number of unproductive code submissions (x-axis) against code-thoroughness (y-axis). The size of each circle is proportional to the fraction of student attempts that lie at its center. Ideal attempts are productive and thorough i.e., they lie near the upper-left of each sub-plot.

> *when you have exhausted all other feasible options, then can you be sure that you have the right idea. Also try and ensure to test all edge cases and inputs that a person trying to break the function would test. Place in negative numbers, empty strings, extremely long arrays, etc. to see how the code functions." (P40)*

One student also talked about how the problem description could be misleading and prime them to think about the problem in the wrong way. Instead, they advocated for asking many clarifying questions about the expected outputs:

> *"...take [the client's] description of what they think they want with little regard, instead ensure you have a good idea of what they want by asking them about expected outputs." (P5)*

These quotes are encouraging, as students appear to be adopting a critical design perspective, in which they question assumptions, clarify user needs, and remain open to new information. This process of clarifying user needs rather than passively accepting design requirements is a fundamental aspect of the design process. Embracing these design aspects and appreciating users' needs has been a consistent challenge for CS educators [1, 18, 29, 33].

5 Discussion

5.1 Student Strategies in Probing

Our findings reveal a range of student behaviors when engaging with Probeable Problems, which raise interesting questions about strategy, efficiency, and support – each of which warrants further discussion and points to directions for future work. Figure 5 shows

that while issuing more probes tends to lead to more thorough exploration of the problem space, this is not always the case. Across all problems, student attempts with high probe-thoroughness scores often involved a large number of probes, but not all such attempts were efficient. Many student attempts that were not thorough involved 10 or more unproductive probes, suggesting that simply encouraging students to probe more is not an effective strategy on its own. Students may be issuing redundant probes or focusing too narrowly on one part of the design space – a phenomenon known as design fixation [22]. Design fixation occurs when individuals become overly anchored to initial ideas or examples, and find it hard to think about alternative possibilities [50].

In creative foraging theory [20], expert designers tend to explore a design space broadly before exploiting local solutions. However, novice designers may sometimes need help to make a 'creative leap' [28] to other parts of the design space if they get stuck exploiting a local maximum without realizing it. Similarly, students may need nudges to help avoid wheel spinning [4, 5] and break out of probing unproductive parts of the design space. In our setting, students may have developed early assumptions about how the program should behave. Prior work on approaches for overcoming design fixation could form the basis for hinting strategies in Probeable Problems, such as context shifting or analogical reasoning [47]. Siangliulue et al.'s 'IdeaHound' system also provides an interesting model to support creative exploration at scale [46]. They propose constructing a model of a collective solution space by clustering ideas submitted by users during ideation tasks. A similar approach could be applied to Probeable Problems, where we could globally cluster all submitted probes and identify regions of the behavioral space that represent distinct insights – even ones that do not fit within the instructor-defined set of N incorrect interpretations. This would allow us to evaluate an individual's probe set not only in terms of breadth (how broadly did they probe?) and discovery (did they find the test cases we intended?), but also in terms of coverage (did they find the clusters others in the class found?).

There were many students who achieved a perfect thoroughness score by issuing 60 probes or more. These students may reflect maximizing tendencies [44], where they aim to explore every possible option rather than stopping once they have explored to a satisfactory extent. While thorough, this approach may be cognitively taxing for students and they might benefit from guidance to improve their efficiency. This mirrors findings related to black-box testing activities in software engineering education. For example, Bai et al. found that most students were ineffective at discovering known defects when testing, and one of their most common challenges was deciding how much testing was enough [2]. Many students defaulted to writing test cases that cover only expected or "happy path" behavior, with little emphasis on edge cases or fault discovery. On the one hand, this overlap suggests that probing tasks might offer a useful, low-barrier entry point into fundamental software engineering concepts like test design, fault detection, and stopping criteria in black-box testing. However, future work should explore techniques to help students monitor their probing process and better evaluate when they are ready to stop. In particular, although our current implementation of Probeable Problems does not support this, it would clearly be possible to provide a hint to students who attempt to repeatedly submit unproductive probes.

Probeable Problems are also closely related to the work of Wrenn and Krishnamurthi, who proposed using executable input–output examples as a way to assess students' understanding of a programming problem before they begin implementation [52]. In their Examplar system, the thoroughness of a student's test suite is evaluated by checking it against a curated set of buggy implementations, in order to provide early feedback on their understanding of the problem. Our definitions of probe- and code-thoroughness, which is based on their notion of thoroughness, share a common weakness: they rely on instructors to identify a representative set of incorrect implementations. Prasad et al. have proposed classsourcing as one way to address this weakness in the context of Examplar [34], and a similar approach could be effective for Probeable Problems.

Probeable Problems share the goal of clarifying the problem to be solved, but differs in that students only supply inputs, not expected outputs. In this way, probing is more similar to the process of asking a clarifying question of a client or user to determine what the program should do. This distinction reinforces the framing of probing as a method of requirements elicitation rather than verification.

5.2 Implications for Teaching

Requirement elicitation – the skill of clarifying specifications – is typically taught in upper-level software design courses. However, the increasing ability of LLMs to solve well-specified programming tasks suggests that CS1 curricula could benefit from integrating these skills earlier. Probeable Problems offer a low-cost, scalable way to introduce some basic aspects of this skill without requiring major changes to existing grading infrastructure.

One promising area for future research is whether early exposure to Probeable Problems helps students develop better debugging and testing strategies in later courses. Since debugging inherently requires recognizing gaps in one's understanding of a program, the ability to systematically probe ambiguities may transfer well to debugging practices. Additionally, further studies could investigate whether students who engage deeply with probing exhibit better long-term retention of fundamental programming concepts.

In this work, the problems were presented using the CodeRunner automated assessment platform. On the one hand, there are benefits in being able to make use of existing course software to support these new kinds of problems. On the other hand, there are limitations based on the way the platform can be configured. A custom platform that provides more fine-grained control for configuring the problems and the feedback could be useful. As an example, one area where a custom tool could support further exploration involves grading incentives. While the current implementation allows unlimited probes without penalty, future iterations could experiment with:

- threshold-based systems (requiring students to submit a minimum number of clarifying probes, potentially that reveal a minimum number of ambiguities, before coding).
- penalties for redundant or trivial probes, encouraging more strategic questioning.
- feedback systems that track probe diversity, helping students recognize when they are overly fixated on a single aspect of the problem.

5.3 Design Implications

Programming at the introductory level is often perceived as a rule-based, convergent discipline, where the goal is to find a single correct solution. However, real-world programming requires significant divergent thinking, particularly when faced with ambiguous requirements. In this study, we observed that students who performed best on Probeable Problems were those who explored the problem space broadly before committing to a solution. This aligns with theories of design thinking and cognitive flexibility, which suggest that experts in ill-structured domains tend to reframe problems multiple times before converging on an answer.

Theories of creative problem-solving emphasize the importance of flexibility in shifting between exploratory (divergent) and refining (convergent) modes of thinking [49], which is rarely mentioned in computing education contexts [40]. Some students in our study appeared to struggle with this balance, either over-exploring the ambiguity space (generating too many probes without consolidating findings) or under-exploring (skipping probes and relying on incorrect code submissions to reveal missing constraints). This behavior parallels the fixation effect in design studies [22], where individuals become trapped in a narrow interpretation of a problem, failing to consider alternative possibilities.

Future work could consider how to model these local fixations on related probes. The work of Noy et al. [28] on creative leaps and Hart et al. on creative foraging [20] both help conceptualize the theoretical foundations of the student behavior observed in our study. What causes a student who is stuck in one semantic group of probes to suddenly take a creative leap to another different group of probes? This could be a useful step in detecting wheel spinning and nudging students to different parts of the problem space. Siangliulue et al. provide a practical model for semantic mapping between related groups/ideas that could be useful in clustering probe similarity and measuring the extent to which all clusters have been hit [46]. Encouraging students to adopt a design-thinking mindset – where ambiguity is seen as a natural part of problem-solving rather than an obstacle – could enhance their ability to handle incomplete specifications and make a creative leap. Therefore, future work could also explore explicit instructional and user interface scaffolding that teaches students when to diverge and when to converge while probing, ensuring they develop both creativity and precision in problem formulation.

5.4 Metacognition

One of the most striking aspects of Probeable Problems is how they engage students in metacognitive reflection – the process of thinking about one's own thinking. Prior work suggests that metacognition plays a critical role in programming success [25, 35], as students must monitor their understanding, recognize gaps in their knowledge, and adjust their problem-solving approach accordingly. However, novice programmers often struggle with metacognitive regulation, leading to issues such as misinterpreting problem requirements, failing to evaluate their own errors, or persisting in ineffective strategies [37].

Probeable Problems provide a natural mechanism for scaffolding metacognition by forcing students to explicitly consider what is missing from a problem before attempting to solve it. In contrast to traditional assignments, where students often rush into coding, these problems encourage deliberate questioning, boundary exploration, and iterative refinement – key metacognitive behaviors. The process of formulating probes mirrors metacognitive self-questioning strategies [51], which have been shown to improve learning outcomes in computing education. Even without direct measurement, student reflections on their probing strategies suggest an increased awareness of problem constraints and edge cases, reinforcing the idea that structured ambiguity can promote metacognitive growth.

Furthermore, self-regulated learning (SRL) theory [54] provides a useful lens through which to view students' interactions with Probeable Problems. Effective self-regulators tend to plan their approach, strategically test hypotheses, and adjust their methods when encountering difficulties – all behaviors that align with the probing process. However, less effective self-regulators may struggle to recognize when they have gathered enough information to transition from probing to coding, leading to unproductive behaviors such as "wheel spinning" – repeatedly issuing probes without meaningful progress. Future work could explore interventions that help students regulate their probing strategies, such as feedback systems that identify when they are stuck in local maxima and prompt them to broaden their exploration.

Recent work by Prather et al. reported that students using generative AI can become distracted or misled by it, struggling through multiple metacognitive difficulties [38]. The kind of metacognitive scaffolding described above that Probeable Problems offers could be one part of a solution. For students using generative AI, probing first could help students better grasp the problem, edge cases, and viable approaches to the solution. Prather et al. also recommended students be taught generative AI via *negative expertise*, which is being taught what to do as much as what *not* to do. Probeable Problems allow students to explore the space in such a way that they learn which tests are helpful as well as unnecessary or redundant, leaning into this recommendation to support negative expertise.

5.5 Limitations

While our work sheds light on student approaches to solving Probeable Problems in an introductory programming context, there are several limitations to acknowledge when interpreting our results.

While it was possible to configure the CodeRunner platform to deploy the Probeable Problems at scale, CodeRunner is a generic automated assessment tool for programming tasks and does not necessarily present an ideal interface for these tasks. For instance, students could not view a history of all of their previously submitted probes. A purpose-built interface for Probeable Problems could support richer forms of feedback, exploration tracking, and visualization to better support student reasoning.

Another limitation of this work is that we explored only three Probeable Problems. While these problems were carefully designed to capture a range of ambiguity types, they represent a tiny fraction of the possible design space. Additional studies involving a broader variety of tasks and ambiguity dimensions would help validate our findings, as would replication across diverse contexts and institutions.

6 Conclusions

In this work, we examined how students approach Probeable Problems by analyzing the thoroughness and efficiency of their attempts, as well as the strategies students use to decide when to probe and when to code. This is the first study to analyze the content of student-generated probes to evaluate how effectively they discard incorrect interpretations of a Probeable Problem. Probeable Problems deliberately present ambiguous programming tasks that require learners to generate clarifying probes to uncover desired behavior and edge cases before submitting a code solution. Our quantitative analysis revealed that after an initial exposure to Probeable Problems, many students were able improve their probing behavior in terms of both thoroughness and efficiency, despite the limitations of the programming environment and feedback. Our qualitative analysis revealed two key themes in students' approaches. First, many students embraced test-driven development, choosing to deeply explore edge cases before writing any code. Second, students described deliberately disrupting their own reasoning to challenge assumptions and explore alternative interpretations – demonstrating a form of cognitive flexibility aligned with expert design thinking. These insights suggest that Probeable Problems not only offer a novel pedagogical response to AI-driven changes in computing education, but also serve as a valuable tool for fostering metacognitive awareness and critical problem-solving skills.

Acknowledgments

This work was supported by the Research Council of Finland grant #356114.

References

[1] Johan Aberg. 2010. Challenges with teaching HCI early to computer students. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education* (Bilkent, Ankara, Turkey) *(ITiCSE '10)*. Association for Computing Machinery, New York, NY, USA, 3–7. doi:10.1145/1822090.1822094

[2] Gina R Bai, Justin Smith, and Kathryn T Stolee. 2021. How students unit test: Perceptions, practices, and pitfalls. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. 248–254.

[3] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. 2021. STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1062–1068.

[4] Joseph E Beck and Yue Gong. 2013. Wheel-spinning: Students who fail to master a skill. In *Artificial Intelligence in Education: 16th International Conference, AIED 2013, Memphis, TN, USA, July 9-13, 2013. Proceedings 16*. Springer, 431–440.

[5] Anthony F Botelho, Ashvini Varatharaj, Thanaporn Patikorn, Diana Doherty, Seth A Adjei, and Joseph E Beck. 2019. Developing early detectors of student attrition and wheel spinning using deep learning. *IEEE Transactions on Learning Technologies* 12, 2 (2019), 158–170.

[6] Kevin Buffardi and Stephen H Edwards. 2013. Effective and ineffective software testing behaviors by novice programmers. In *Proceedings of the ninth annual international ACM conference on International computing education research*. 83–90.

[7] Jeffrey C Carver and Nicholas A Kraft. 2011. Evaluating the testing ability of senior-level computer science students. In *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 169–178.

[8] Victoria Clarke and Virginia Braun. 2017. Thematic analysis. *The journal of positive psychology* 12, 3 (2017), 297–298.

[9] Michelle Craig, Andrew Petersen, and Jennifer Campbell. 2019. Answering the Correct Question. In *Proceedings of the ACM Conference on Global Computing Education*. ACM, New York, NY, USA, 72–77.

[10] Paul Denny, Viraj Kumar, Stephen MacNeil, James Prather, and Juho Leinonen. 2025. Probing the Unknown: Exploring Student Interactions with Probeable Problems at Scale in Introductory Programming. In *Proceedings of the 2025 on Innovation and Technology in Computer Science Education V. 1* (Nijmegen, Netherlands) *(ITiCSE 2025)*. Association for Computing Machinery, New York, NY, USA, 7 pages.

[11] Paul Denny, James Prather, Brett A. Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research* (Koli, Finland). ACM, New York, NY, USA, Article 11, 10 pages.

[12] Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves, Eddie Antonio Santos, and Sami Sarsa. 2024. Computing Education in the Era of Generative AI. *Commun. ACM* 67, 2 (Jan. 2024), 56–67. doi:10.1145/3624720

[13] D Méndez Fernández, Stefan Wagner, Marcos Kalinowski, Michael Felderer, Priscilla Mafra, Antonio Vetrò, Tayana Conte, M-T Christiansson, Des Greer, Casper Lassenius, et al. 2017. Naming the pain in requirements engineering: Contemporary problems, causes, and effects in practice. *Empirical software engineering* 22 (2017), 2298–2338.

[14] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A. Becker. 2023. My AI Wants to Know if This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In *Proceedings of the 25th Australasian Computing Education Conference* (Melbourne, VIC, Australia). ACM, New York, NY, USA, 97–104.

[15] John H Flavell. 1979. Metacognition and cognitive monitoring: A new area of cognitive–developmental inquiry. *American psychologist* 34, 10 (1979), 906.

[16] Vincenzo Gervasi, Alessio Ferrari, Didar Zowghi, and Paola Spoletini. 2019. *Ambiguity in Requirements Engineering: Towards a Unifying Framework*. Springer International Publishing, Cham, 191–210.

[17] Vincent Gramoli, Michael Charleston, Bryn Jeffries, Irena Koprinska, Martin McGrane, Alex Radu, Anastasios Viglas, and Kalina Yacef. 2016. Mining autograding data in computer science education. In *Proceedings of the Australasian Computer Science Week Multiconference* (Canberra, Australia) *(ACSW '16)*. ACM, New York, NY, USA, Article 1, 10 pages.

[18] Colin M. Gray, Shruthi Sai Chivukula, Kassandra Melkey, and Rhea Manocha. 2021. Understanding "Dark" Design Roles in Computing Education. In *Proceedings of the 17th ACM Conference on International Computing Education Research* (Virtual Event, USA) *(ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 225–238. doi:10.1145/3446871.3469754

[19] Sebastian Gutierrez, Irene Hou, Jihye Lee, Kenneth Angelikas, Owen Man, Sophia Mettille, James Prather, Paul Denny, and Stephen MacNeil. 2024. Seeing the Forest and the Trees: Solving Visual Graph and Tree Based Data Structure Problems using Large Multimodal Models. *arXiv preprint arXiv:2412.11088* (2024).

[20] Yuval Hart, Avraham E Mayo, Ruth Mayo, Liron Rozenkrantz, Avichai Tendler, Uri Alon, and Lior Noy. 2017. Creative foraging: An experimental paradigm for studying exploration and discovery. *PloS one* 12, 8 (2017), e0182133.

[21] Irene Hou, Owen Man, Sophia Mettille, Sebastian Gutierrez, Kenneth Angelikas, and Stephen MacNeil. 2024. More Robots are Coming: Large Multimodal Models (ChatGPT) can Solve Visually Diverse Images of Parsons Problems. In *Proceedings of the 26th Australasian Computing Education Conference* (Sydney, NSW, Australia) *(ACE '24)*. ACM, New York, NY, USA, 29–38.

[22] David G Jansson and Steven M Smith. 1991. Design fixation. *Design studies* 12, 1 (1991), 3–11.

[23] Ayaan M Kazerouni, Clifford A Shaffer, Stephen H Edwards, and Francisco Servant. 2019. Assessing incremental testing practices and their impact on project outcomes. In *Proceedings of the 50th acm technical symposium on computer science education*. 407–413.

[24] Natalie Kiesler and Daniel Schiffner. 2023. Large Language Models in Introductory Programming Education: ChatGPT's Performance and Implications for Assessments. arXiv:2308.08572 [cs.SE] https://arxiv.org/abs/2308.08572

[25] Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2022. Metacognition and Self-Regulation in Programming Education: Theories and Exemplars of Use. *ACM Trans. Comput. Educ.* 22, 4, Article 39 (Sept. 2022), 31 pages. doi:10.1145/3487050

[26] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018 Companion)*. ACM, New York, NY, USA, 55–106.

[27] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2024. Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Trans. Comput. Educ.* 24, 1, Article 10 (Feb. 2024), 43 pages.

[28] Lior Noy, Yuval Hart, Natalie Andrew, Omer Ramote, Avi E Mayo, and Uri Alon. 2012. A quantitative study of creative leaps.. In *International Conference on Computational Creativity (ICCC '12)*. Association for Computational Creativity, 72–76.

[29] Alannah Oleson, Meron Solomon, and Amy J Ko. 2020. Computing students' learning difficulties in HCI education. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–14.

[30] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Trans.*

Comput. Educ. 22, 3, Article 34 (June 2022), 40 pages.

[31] Mrigank Pawagi and Viraj Kumar. 2024. Probeable Problems for Beginner-level Programming-with-AI Contests. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1* (Melbourne, VIC, Australia) *(ICER '24)*. ACM, New York, NY, USA, 166–176.

[32] Yulia Pechorina, Keith Anderson, and Paul Denny. 2023. Metacodenition: Scaffolding the Problem-Solving Process for Novice Programmers. In *Proceedings of the 25th Australasian Computing Education Conference* (Melbourne, VIC, Australia) *(ACE '23)*. ACM, New York, NY, USA, 59–68.

[33] G. Michael Poor, Laura M. Leventhal, Julie Barnes, Duke R. Hutchings, Paul Albee, and Laura Campbell. 2012. No User Left Behind: Including Accessibility in Student Projects and the Impact on CS Students' Attitudes. *ACM Trans. Comput. Educ.* 12, 2, Article 5 (April 2012), 22 pages. doi:10.1145/2160547.2160548

[34] Siddhartha Prasad, Ben Greenman, Tim Nelson, John Wrenn, and Shriram Krishnamurthi. 2022. Making Hay from Wheats: A Classsourcing Method to Identify Misconceptions. In *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 7 pages. doi:10.1145/3564721.3564726

[35] James Prather, Brett A. Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. 2020. What Do We Think We Think We Are Doing? Metacognition and Self-Regulation in Programming. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (Virtual Event, New Zealand) *(ICER '20)*. ACM, New York, NY, USA, 2–13.

[36] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Petersen, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 2023. The Robots Are Here: Navigating the Generative AI Revolution in Computing Education. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education* (Turku, Finland) *(ITiCSE-WGR '23)*. ACM, New York, NY, USA, 108–159.

[37] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. ACM, New York, NY, USA, 41–50.

[38] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1* (Melbourne, VIC, Australia) *(ICER '24)*. ACM, New York, NY, USA, 469–486.

[39] Arun Raman and Viraj Kumar. 2022. Programming Pedagogy and Assessment in the Era of AI/ML: A Position Paper. In *Proceedings of the 15th Annual ACM India Compute Conference* (Jaipur, India). ACM, New York, NY, USA, 29–34.

[40] Patil Deepti Reddy, Sridhar Iyer, and M Sasikumar. 2016. Teaching and learning of divergent and convergent thinking through open-problem solving in a data structures course. In *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. IEEE, 178–185.

[41] Horst W. J. Rittel and Melvin M. Webber. 1973. Dilemmas in a general theory of planning. *Policy Sciences* 4, 2 (1973), 155–169. doi:10.1007/BF01405730

[42] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. 2023. Thrilled by Your Progress! Large Language Models (GPT-4) No Longer Struggle to Pass Assessments in Higher Education Programming Courses. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (Chicago, IL, USA) *(ICER '23)*. ACM, New York, NY, USA, 78–92.

[43] G. Michael Schneider. 1978. The introductory programming course in computer science: ten principles. *SIGCSE Bull.* 10, 1 (Feb. 1978), 107–114.

[44] Barry Schwartz, Andrew Ward, John Monterosso, Sonja Lyubomirsky, Katherine White, and Darrin R Lehman. 2002. Maximizing versus satisficing: happiness is a matter of choice. *Journal of personality and social psychology* 83, 5 (2002), 1178.

[45] Unnati S. Shah and Devesh C. Jinwala. 2015. Resolving Ambiguities in Natural Language Software Requirements: A Comprehensive Survey. *SIGSOFT Softw. Eng. Notes* 40, 5 (Sept. 2015), 1–7. doi:10.1145/2815021.2815032

[46] Pao Siangliulue, Joel Chan, Steven P. Dow, and Krzysztof Z. Gajos. 2016. IdeaHound: Improving Large-scale Collaborative Ideation with Crowd-Powered Real-time Semantic Modeling. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 609–624. doi:10.1145/2984511.2984578

[47] STEVEN M. SMITH and JULIE LINSEY. 2011. A Three-Pronged Approach for Overcoming Design Fixation. *The Journal of Creative Behavior* 45, 2 (2011), 83–91. doi:10.1002/j.2162-6057.2011.tb01087.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.2162-6057.2011.tb01087.x

[48] Rand J. Spiro, Richard L. Coulson, Paul J. Feltovich, and Daniel K. Anderson. 1994. Cognitive flexibility theory: Advanced knowledge acquisition in ill-structured domains. In *Theoretical models and processes of reading, 4th ed.*, International Reading Association (Ed.). International Reading Association, Newark, DE, US, 602–615.

[49] Donald J Treffinger. 1995. Creative problem solving: Overview and educational implications. *Educational psychology review* 7 (1995), 301–312.

[50] Vimal K. Viswanathan and Julie S. Linsey. 2013. Design Fixation and Its Mitigation: A Study on the Role of Expertise. *Journal of Mechanical Design* 135, 5 (04 2013), 051008. doi:10.1115/1.4024123 arXiv:https://asmedigitalcollection.asme.org/mechanicaldesign/article-pdf/135/5/051008/6222030/md_135_5_051008.pdf

[51] Cui-Yu Wang, Bao-Lian Gao, and Shu-Jie Chen. 2024. The effects of metacognitive scaffolding of project-based learning environments on students' metacognitive ability and computational thinking. *Education and Information Technologies* 29, 5 (2024), 5485–5508.

[52] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. ACM, New York, NY, USA, 131–139.

[53] Cynthia Zastudil, Magdalena Rogalska, Christine Kapp, Jennifer Vaughn, and Stephen MacNeil. 2023. Generative ai in computing education: Perspectives of students and instructors. In *2023 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–9. doi:10.1109/FIE58773.2023.10343467

[54] Barry J Zimmerman. 2000. Attaining self-regulation: A social cognitive perspective. In *Handbook of self-regulation*. Elsevier, 13–39.