Department of Electrical, Computer, and Software Engineering

Part IV Research Project

Final Research Report

Project Number: 127

“Metacodenition: A Tool to Scaffold the Problem-Solving Process for Novice Programmers”

Yulia Pechorina

Keith Anderson

Assoc. Prof. Paul Denny

14/10/2022

Declaration of Originality

This report is my own unaided work and was not copied from nor written in collaboration with any other person.

[illegible]

Name: Yulia Pechorina

Metacodenition: A Tool to Scaffold the Problem-Solving Process for Novice Programmers

Yulia Pechorina
The University of Auckland
Auckland, New Zealand
ypec413@aucklanduni.ac.nz

ABSTRACT

Problem-solving is a crucial skill for programmers, but one that novices can find challenging to develop. Despite the importance of problem-solving skills and the many frameworks which exist to formalise them, most introductory programming courses do not explicitly teach them. Programming educators find it challenging to teach problem-solving, as students must develop metacognitive awareness, that is, an awareness of one's thinking, to identify their problem-solving state. We contribute a tool called Metacodenition, which provides metacognitive scaffolding for an existing problem-solving framework. Our evaluation shows that Metacodenition's scaffolding improves performance on code-writing tasks and students view Metacodenition to be a helpful tool they would use again voluntarily.

CCS CONCEPTS

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

KEYWORDS

metacognition, problem-solving, software engineering education

1 INTRODUCTION

Learning to program requires students to develop several skills, one of which is problem-solving [1, 12]. It is widely accepted that problem-solving is both a critical skill [9], and one of the most difficult skills for novices to develop [18]. Further, an ITiCSE working group study found that many students in introductory programming courses lacked problem-solving skills [13].

Poor problem-solving skills often persist throughout a student's degree. A study by Radermacher et al. investigating the skill gap between software engineering graduates and industry expectations found that a lack of problem-solving skills is one of the two most common issues which prevents graduate software engineers from being employed [23]. The authors also found that employers placed the importance of non-technical skill, such as problem-solving skills, above technical skills. Some employers were willing to hire graduates who lacked technical ability so long as they demonstrated good problem-solving skills.

A factor for poor problem-solving skills in novices is the curriculum of introductory programming courses. Programming skills taught in introductory programming courses can be divided into programming knowledge and programming problem-solving strategies. Programming knowledge refers to the syntax and semantics of programming, whereas programming problem-solving strategies refer to how one applies programming knowledge to solve a problem [4]. In a typical course curriculum, programming knowledge is explicitly taught, whereas problem-solving strategies are often implicitly taught [19].

Even when problem-solving strategies are explicitly taught, novice programmers often face difficulty learning problem-solving because they lack metacognitive awareness [6, 14, 16]. Metacognitive awareness means being aware of one's own thinking and problem-solving strategies [8], and plays a significant role in programming [20]. When programming, students use metacognition to move between the steps of problem-solving process and decide when to revisit steps [20].

Loksa et al. developed a problem-solving framework, along with interventions for this framework, to promote metacognitive awareness [15]. Multiple studies exist that build on Loksa et al.'s framework and propose other interventions to promote metacognition further. However, current work is generally limited to interventions for a single stage of a problem-solving framework at a time. These interventions individually are effective at teaching students how to problem solve, increasing their metacognitive awareness and performance in their courses. To our knowledge, only one tool, TextCode [2], has been developed to cater to multiple problem-solving stages. However, TextCode does not scaffold the entire problem-solving process; it only scaffolds the process up to the implementing a solution stage; the evaluation stage is missed. The evaluation stage is included in the majority of problem-solving frameworks and is crucial for ensuring the solution meets the problem requirements and making any necessary modifications to the solution.

In this work, we investigate whether combining multiple problem-solving stages into a tool will make it more effective for teaching problem-solving than a tool that only integrates one, by developing a tool that caters to multiple stages Loksa et al.'s framework at once.

Our contribution is an integrated development environment (IDE) called Metacodenition, designed to provide metacognitive scaffolding during code writing exercises. We outline the design of Metacodenition and then evaluate its effectiveness through a study involving 821 first-year engineering students taking an introductory programming course. We investigate the following research questions:

- RQ1: What are students' perceptions toward using Metacodenition for problem-solving activities?
- RQ2: Does the scaffolding provided by Metacodenition improve performance on code-writing tasks?

The remainder of this paper is organized as follows. Section 2 reviews related work on strategies, frameworks, and interventions designed to aid novices in developing problem-solving skills. In Section 3, we present our approach to designing Metacodenition. We then cover our process in evaluating Metacodenition in Section

Table 1: Loksa et al.'s problem-solving stages

#	Stage	Description
1	Reinterpreting the problem prompt	Understanding and interpreting the problem statement and requirements
2	Searching for analogous problems	Drawing upon previously encountered problems
3	Searching for solutions	Seeking solutions that will solve the problem
4	Evaluating a potential solution	Evaluating whether the solution meets problem requirements, often using prototyping techniques such as pseudo code
5	Implementing a solution	Writing the solution code
6	Evaluating an implemented solution	Evaluating whether the current solution meets the problem requirements, usually through testing and debugging

4 and report the results and discussion of this evaluation in Sections 5. Finally, we conclude in Section 6.

2 RELATED WORK

2.1 Problem-solving frameworks

Loksa et al. [15] developed a programming-specific problem-solving framework consisting of the following stages: (1) reinterpreting the problem prompt, (2) searching for analogous problems, (3) searching for solutions, (4) evaluating a potential solution, (5) implementing a solution, and (6) evaluating an implemented solution (see Table 1).

A controlled experiment with 48 students was conducted to measure the effect of teaching Loksa et al.'s framework along with interventions to motivate students to think about their progress along the stages of the problem-solving framework. The experimental and control groups participated in separate coding camps, and the experimental group had multiple interventions in their camp schedule. Intervention (1) was a 1-hour problem-solving lecture. Intervention (2) was a prompt when a student requested help from an instructor, asking them to identify their current problem-solving stage. Intervention (3) was a physical handout of the problem-solving stages. Intervention (4) was hints categorized under the problem-solving stages added to the student's IDE. They found that students in the experimental group had more detailed explanations of their problem-solving strategies, required help with design and selection rather than debugging, relied less on the instructors, and had greater productivity and scores. While these interventions were successful, they can be difficult to implement at scale due to no automation for interventions 1 to 3. Intervention 2 had especially low scalability, as it required 1:1 contact between the instructor and student.

A framework with a greater focus on algorithms was developed by Hilton et al., called "The Seven Steps" [10]. The framework consisted of the following steps: (1) work an example yourself, (2) write down exactly what you just did, (3) generalize, (4) test your algorithm, (5) translate to code, and (6) test. These steps are designed to help a student design and implement an algorithm to solve a solution. They taught this framework to students in two introductory programming courses – one for graduate engineering students and another for undergraduate computer science students. Lab sessions were held to implement steps (1) to (4). The effectiveness of this framework was measured through student surveys. They found that the graduate students had a much higher rate of use of "The Seven Steps" framework than undergraduate students and that there is no statistically significant relationship between the rate of use of the framework and how confident students were in their programming ability.

The algorithmic focus of "The Seven Steps" framework makes it less generally applicable compared to the framework given by Loksa et al. Steps (1) and (2) would be particularly hard to implement for a larger problem, such as creating a web application. However, for a small algorithmic problem such as the "closest point" problem, as seen in [7, Fig. 2], the "The Seven Steps" framework may be better suited than Loksa et al.'s abstracted framework. While an algorithmically focused framework is likely beneficial for an introductory programming course, we feel that it is important to teach frameworks that a student can apply in future years.

Another framework by the ITiCSE working group [17], consisted of the following steps: (1) abstract the problem from its description, (2) generate sub-problems, (3) transform sub-problems into sub-solutions, (4) re-compose, and (5) evaluate and iterate. They conducted an experiment using this framework across several universities to measure students' performance in introductory programming courses. No data was collected about the effectiveness of this framework in teaching problem-solving. However, they did find that novices often skip the early stages of the problem-solving process. The ITiCSE working group suggested that this could be due to the heavy focus of the taught framework on the later stages of the problem-solving process. In comparison, Loksa et al.'s framework focuses more on the early stages, as stage (1) relates to getting the student to understand the problem.

We chose the problem-solving framework by Loksa et al. to guide our work due to its unique focus on metacognition compared to Hilton et al.'s framework and a greater focus on the early stages of the problem-solving process compared to the ITiCSE working group's framework. Studies that build on Loksa et al.'s framework typically involve automated interventions to increase scalability and target individual stages of the framework rather than all stages. A summary of these existing studies is presented in Table 2.

2.2 Interventions related to "Reinterpreting the problem prompt"

Stage (1) of Loksa et al.'s problem-solving framework corresponds to problem comprehension. Studies of interventions that aim to increase student problem comprehension typically involve test cases. These interventions can be divided into solving test cases (section 2.2.1) and writing test cases (section 2.2.2).

Table 2: Summary of existing Studies.

Author	Stage in Loksa et al.'s Framework	Intervention	Number of Students Involved in Experiment	Year	Result
Prather et al. [21]	1	Solving test cases	36	2019	Inconclusive
Denny et al. [5]	1	Solving test cases	976	2019	Positive
Craig et al. [3]	1	Solving test cases	831	2019	Positive
Janzen et al. [11]	1, 6	Writing test cases	140	2008	Positive
Weinman et al. [25]	2	Faded Parsons problems	237	2008	Positive
Garcia [7]	4	Design-level Parsons problems	10	2018	Positive
Corno et al. [2]	1, 3, 5	Decomposing a problem into sub-problems, linking sub-problems to code fragments, rearranging code fragments into a solution	N/A	2021	N/A
Prather et al. [22]	5	Enhanced compiler error messages	31	2017	Positive

2.2.1 Solving test cases. A study by Prather et al. involved a controlled experiment to measure the effect of solving test cases before coding on student metacognitive development, using an automated assessment tool (AAT) [21]. Students in the experimental group were required to solve and pass a test case quiz before coding, requiring them to reflect on their interpretation of the problem prompt. Students in the experimental group completed the task faster, at a higher completion rate, and required fewer attempts before completion. However, Prather et al. were not able to conclude the effect on the likeliness of solving the problem. They acknowledged that this was likely due to the small scale of the experiment (36 students).

A follow-up study with 976 participants was conducted by Denny et al. [5]. Students in each group used an online tool to solve a programming exercise and were shown a problem statement. Students in the control group were immediately shown the code editor on the same page of the online tool. Students in the experimental group were prompted to solve questions before the code editor was shown, though they did not have to answer correctly to proceed. Denny et al. did not find any differences in completion time, number of submissions, or the total error number between the control and experimental groups. However, they did find that the experimental group made fewer errors caused by an incorrect mental model of the problem. This indicates that solving test cases is successful in helping students' problem comprehension.

Craig et al. conducted a study on the effect of solving test cases before solving programming problems on submission counts [3]. Students were split into two treatment groups and received different programming exercises to solve. The treatment groups were then further split into control and experimental groups. Students in the experimental group were required to solve three test cases before being allowed to code, which acted as the problem-solving intervention. Craig et al.'s students were required to correctly answer test cases before they were shown the code editor. In contrast, Denny et al.'s students only needed to submit an answer to view the code editor; the answer did not necessarily have to be correct. They found successful results in one treatment group, where the correct problem solution relied on a solid understanding of boundary conditions. In this treatment group, the students who received the intervention required fewer submissions to correctly solve the problem compared to the control group. This indicates that solving test cases is beneficial for problems where the difficulty lies in understanding rather than coding. Problems where programming language or library understanding is the greatest challenge, are not well suited to the test-case solving intervention.

2.2.2 Writing test cases. A related approach of requiring students to write test-cases before they solve the problem was taken by Janzen et al.[11]. This study involved students in CS1 and CS2 courses who were taught Test-Driven Development (TDD). Students were given two programming projects for which they were required to write tests. Students were required to adhere to a TDD approach for any one of the two projects. For the CS1 group, they found no difference in grades or productivity between students who wrote tests before they wrote solutions and students who wrote tests afterward. However, for the CS2 group, they did find a difference, where students who wrote tests first achieved higher grades and spent less time on the solution code. The higher grades and faster development time indicate greater problem comprehension. Janzen et al. concluded that requiring students to write their own tests is an effective intervention to increase problem comprehension for more mature students, however, may not be suitable for novice students.

Writing test cases, as in Janzen et al.'s study, can directly assist students in stage (1) of Loksa et al.'s problem-solving framework. However, it is also important to note that these test cases can also later be used by the student in stage (6) – "Evaluating an implemented solution".

2.3 Interventions related to "Searching for analogous solutions"

Programming solutions are analogous when they follow similar programming patterns. Consequently, teaching reusable programming patterns can help a student build a bank of analogous problems to search for during this problem-solving stage. The use of Faded Parsons problems as a method to teach reusable programming patterns was investigated by Weinman et al. [25]. Faded Parsons problems are where students are given a block of code with lines in the wrong order and containing blank spaces. The students rearrange these lines of code and fill in the blank spaces with expressions. Adding blank spaces in lines distinguishes Faded Parsons problems from Parsons problems. This study involved three experiments in an introductory programming course, which involved solving Faded Parsons problems, code writing problems, and code tracing problems. Students' improvement in learning reusable patterns was measured through three metrics. Pattern Exposure measured whether students were first exposed to programming patterns through viewing instructor solutions (Faded Parsons problems or code-tracing exercises) or through writing their own code. Active Pattern Exposure measured whether students' solutions adhered to a programming pattern. Pattern Acquisition measured whether students applied programming patterns they had learned in previous exercises to subsequent code-writing exercises. The authors found that Faded Parsons problems have a higher rate of Pattern Exposure, Active Pattern Exposure, and Pattern Acquisition than code-writing problems. The results show that Faded Parsons problems effectively teach students reusable programming patterns and how to apply them to code-writing exercises. Further, Faded Parsons problems are more effective in teaching these patterns than traditional code-writing exercises and code-tracing problems and are preferred by students.

2.4 Interventions related to "Evaluating a potential solution"

Students often use prototyping techniques such as pseudo code to evaluate a potential solution. Parsons problems as a prototyping technique were investigated by Garcia [7]. Parsons problems are where students are given fragments of code in the wrong order and asked to arrange them to form a solution. In this experiment, rather than being provided fragments of code, students were given fragments of the algorithm's strategy as sentences. Arranging these fragments prompted students to design a solution. Data was collected through an initial self-assessment, think-aloud sessions for three assignments, and a final interview per assignment. They found that Parsons problems were an effective intervention to help students develop high-quality solutions, though the scale of the experiment was small.

2.5 Interventions related to "Implementing a solution"

One of the difficulties that novices encounter when implementing their solutions is interpreting error messages. Prather et al. addressed this in a study on the helpfulness of enhanced compiler error messages in an automated assessment tool (AAT) which involved 2 experiments [22]. The first experiment was a controlled experiment that involved daily quizzes. These daily quizzes contained a program with errors and compiler error messages, and students were required to solve the error. The students in the experimental group received enhanced compiler error messages, where error messages had wording and structure better suited to novices. They found that students in the experimental group had significantly fewer incorrect answers. The last experiment was a think-aloud session where students solved a programming problem under time constraints. They found that students performed worse on the given problem than students in other semesters who did not receive enhanced compiler error messages. They suggested that these results could be due to students being able to solve this problem offline in previous semesters. Students' verbalizations indicated that they believed the enhanced error messages were useful. The results of these experiments indicate that enhanced compiler error messages are an effective intervention for stage (5) of Loksa et al.'s problem-solving framework.

2.6 Designing tools to support problem-solving

Sáenz and Russis investigated how ten novice students envision a tool to support their metacognitive awareness during problem-solving [24]. The study involved an interview where students' problem-solving strategies and difficulties were investigated, followed by an exercise where students designed and sketched their vision for a tool to support problem-solving. Based on student sketches, the authors outlined some implications for designing a tool for supporting problem-solving strategies. These design implications included:

- supporting artifacts different from code
  Generally, in programming assessments, only students' final code is assessed. Consequently, many novice students consider developing artefacts such as annotations or pseudocode a wasted effort, especially when time constrained. A tool should capture all other artefacts that students produce rather than just their final code.
- integrate and link the programming exercise text to the code constructs
  Most students stated that they kept the problem description next to them while coding. Additionally, one student's sketch of a problem-solving tool showed the problem text placed next to their code. A tool should allow students to link fragments of the problem text to code constructs and create comments and annotations for these links.
- supporting the choice of algorithms and the data structure
  Some students stated that their biggest challenge was choosing the correct algorithm and data structure for their solution. One student's sketch presented an area

which provided documentation on various data structures.

- formatting and commenting features for the text and code
  The most common problem-solving strategy students used was highlighting keywords in the problem statement. A problem-solving tool should allow students to format the problem text, including highlighting or adding comments.

These design implications directly informed design choices that we made when developing Metacodenition, and are documented in section 3.

2.7 Tools for scaffolding the problem-solving process

To scaffold the problem-solving process, Corno et al. developed a tool called TextCode, which supports multiple problem-solving stages: understanding the problem before coding, decomposing the problem into sub-problems, solving each sub-problem individually, and arranging the solution to each sub-problem to build a general solution [2]. In Loksa et al.'s framework, these stages correspond to *reinterpreting the problem prompt*, *designing a solution* and *implementing a solution*. TextCode allows students to highlight parts of the problem statement, effectively identify sub-problems, and then create code fragments to solve the subproblems. Students then rearrange the code fragments to form a solution to the problem. The functionality of TextCode is similar to Parsons Problems, except that students rearrange their own code fragments rather than pre-defined fragments. Unfortunately, the authors did not evaluate TextCode, so it is unclear whether TextCode is effective as a problem-solving tool.

3 DESIGN

Metacodenition's design provides scaffolding for Loksa et al.'s problem-solving framework. The problem-solving interventions included in Metacodenition are based upon existing literature which has shown positive results previously:

- solving test cases before programming [3, 5, 21],
- design parsons problems [7],
- test-driven development for education [11].

Additionally, Metacodenition's designed is informed by the design implications for problem-solving tools proposed by Sáenz and Russis [24]. Specifically, the design implications we incorporate in Metacodenition are:

- supporting artefacts different from code
- integrate and link the programming exercise text to the code constructs
- formatting and commenting features for the text and code

Metacodenition omits Sáenz and Russis' design implications of *supporting the choice of algorithms and the data structure*, as our target users are first-year engineering students taking an introductory programming course at the University of Auckland. In this course, students only learn basic data structures in the C language, such as arrays and structs. Additionally, Metacodenition is designed for use in a lab activity rather than in a test or assignment. Since labs are typically focused on practising using a specific data structure, students rarely need to choose a data structure themselves.

3.1 Scaffolding

Scaffolding for the problem-solving steps should be presented in a logical order. Loksa et al's problem-solving stages are intended to be visited in loose sequential manner. Stages should be revisited often while iteratively implementing a solution. Metacodenition provides the scaffolding for the problem-solving steps in a navigation bar, which shows all stages in order from top to bottom, and the current stage highlighted. This is designed to equip the student to visualise their problem-solving state. Upon opening the question, the user is navigated to the first stage. Students can freely visit stages, allowing them to revisit previous stages, or skip ahead as needed.

3.2 Problem-solving stages

Metacodenition presents an adapted version of Loksa et al.'s problem-solving stages, more suitable for display in a navigation bar:

(1) reinterpreting the problem prompt → understanding the problem,
(2) searching for solutions → designing a solution,
(3) evaluating a potential solution → evaluating a solution,
(4) implementing a solution,
(5) evaluating implemented solution

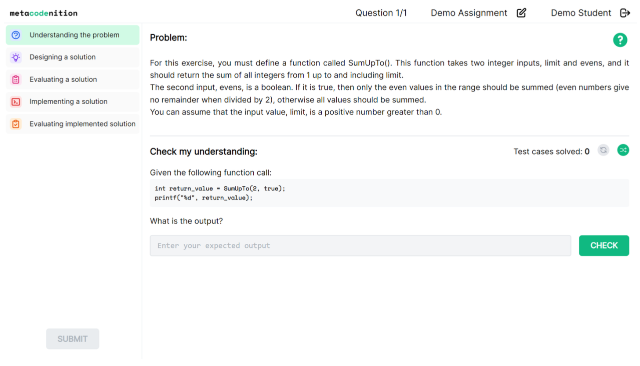
We chose not to support Loksa et al.'s stage 2, *searching for analogous problems*, because Metacodenition was designed to be used in a single lab. In Loksa et al.'s study, students are introduced to the stages through a lecture, while Metacodenition is designed to be used standalone, so wording must be more intuitive. For this reason, stages 1 and 2 are renamed to have greater simplicity. Stage 3 is renamed to be more concise.

A problem-solving tool should support artefacts different from code [24]; therefore, we capture students' solutions to activities for each stage in Metacodenition, not just their final solution.

3.3 Interventions

3.3.1 Understanding the problem. The *understanding the problem* stage is where a student should read and understand the problem. In Metacodenition, this stage consists of the problem description and a section to solve test cases, as shown in Figure 1. The section to solve test cases is informed by previous literature on solving test cases before programming. The student is shown a function call and is asked to enter their expected output. They can check their answer using the *Check* button or by pressing an enter key. Students can shuffle through test cases using the *Shuffle* button if they get stuck on one test case for too long. Test cases the student solves are added to a bank of solved test cases to be used in the *Evaluating implemented solution* stage.

3.3.2 Designing a solution. The *designing a solution* stage is where a student reads through the problem and identifies programming patterns they have used or encountered in the past which may be relevant. Figure 2 shows the *designing a solution stage* on Metacodenition. A problem-solving tool should integrate and link the programming exercise text to the code constructs and have formatting and commenting features for the text and code [24]. In



Problem:

For this exercise, you must define a function called SumUpTo(). This function takes two integer inputs, limit and evens, and it should return the sum of all integers from 1 up to and including limit.
The second input, evens, is a boolean. If it is true, then only the even values in the range should be summed (even numbers give no remainder when divided by 2), otherwise all values should be summed.
You can assume that the input value, limit, is a positive number greater than 0.

Check my understanding:

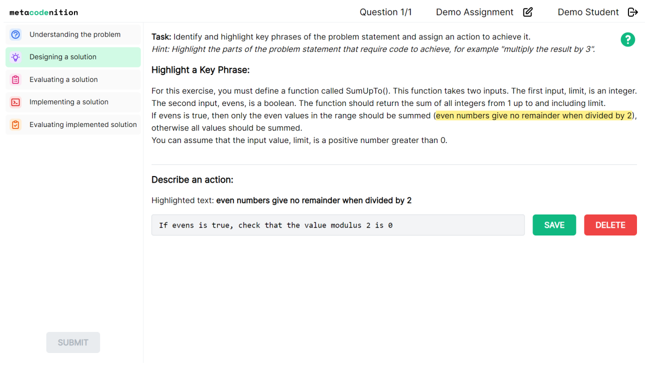
Given the following function call:

```
int return_value = SumUpTo(2, true);
printf("%d", return_value);
```

What is the output?

Figure 1: Understanding the problem page.

Metacodenition, this is achieved by highlighting parts of the problem that the student thinks require code to achieve and assigning an action to it. By assigning an action to a part of the problem, the student identifies a programming pattern or code construct that they can use to address that part of the problem. For example, the student may identify that a modulo operator can be used to check if a value is even.



Task: Identify and highlight key phrases of the problem statement and assign an action to achieve it.
Hint: Highlight the parts of the problem statement that require code to achieve, for example "multiply the result by 3".

Highlight a Key Phrase:

For this exercise, you must define a function called SumUpTo(). This function takes two inputs. The first input, limit, is an integer.
The second input, evens, is a boolean. The function should return the sum of all integers from 1 up to and including limit.
If evens is true, then only the even values in the range should be summed (even numbers give no remainder when divided by 2), otherwise all values should be summed.
You can assume that the input value, limit, is a positive number greater than 0.

Describe an action:

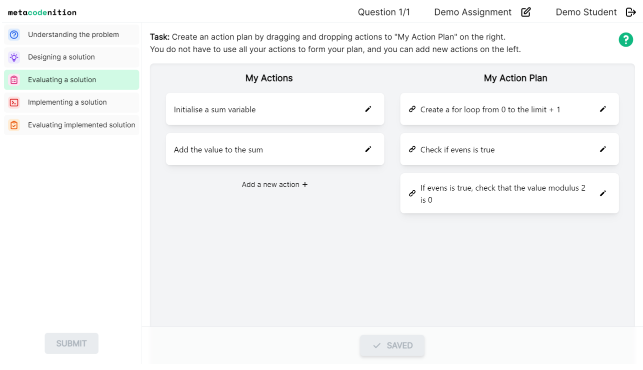
Highlighted text: **even numbers give no remainder when divided by 2**

```
If evens is true, check that the value modulus 2 is 0
```

Figure 2: Designing a solution page.

3.3.3 Evaluating a solution. The *evaluating a solution* stage (see Figure 3) is where a student forms an action plan for their program by selecting and rearranging actions they created previously in the *designing a solution* stage. Students can do this by dragging and dropping blocks from the *My Actions* section to the *My Action Plan* section. There is also functionality to create new actions if a student feels that the actions they generated previously are insufficient. This stage is inspired by design parsons problems [7], where instead of students being provided with fragments to rearrange, students use previously created fragments. In designing Metacodenition, we particularly wanted to have good interconnection of stages to support a good flow through the problem-solving process. Using users' previously created actions, we found a more suitable solution for our goal of interconnection than pre-defined fragments.

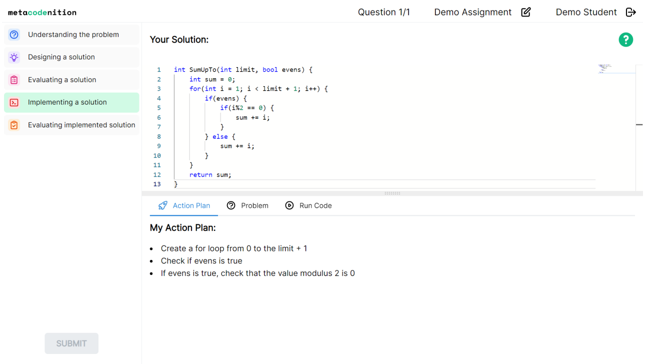
3.3.4 Implementing a solution. The *implementing a solution* stage is where students write their code, as shown in Figure 4. This screen is split into two sections: an area to write their code and a panel with three different tabs. A problem-solving tool should integrate and link the programming exercise text to the code constructs [24], which we address in the page's tab design. Tab 1, the default tab, shows the previous action plan the student created, which can be used to inform their solution. Tab 2 shows the problem statement along with its highlights and assigned actions. Tab 3 is an area where students can run and debug their code.



Task: Create an action plan by dragging and dropping actions to "My Action Plan" on the right.
You do not have to use all your actions to form your plan, and you can add new actions on the left.

My Actions	My Action Plan
Initialise a sum variable	Create a for loop from 0 to the limit + 1
Add the value to the sum	Check if evens is true
	If evens is true, check that the value modulus 2 is 0

Figure 3: Evaluating a solution page.



Your Solution:

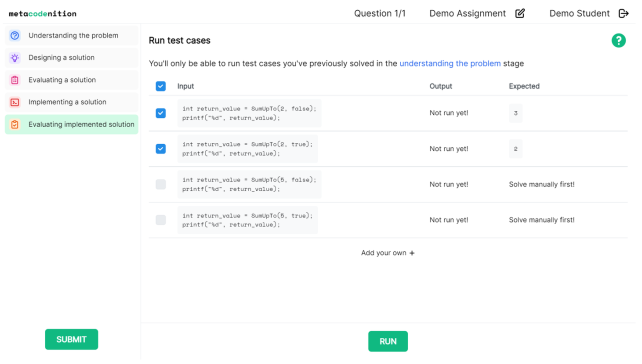
```
int SumUpTo(int limit, bool evens) {
    int sum = 0;
    for(int i = 1; i < limit + 1; i++) {
        if(evens) {
            if(i%2 == 0) {
                sum += i;
            }
        } else {
            sum += i;
        }
    }
    return sum;
}
```

My Action Plan:

- Create a for loop from 0 to the limit + 1
- Check if evens is true
- If evens is true, check that the value modulus 2 is 0

Figure 4: Implementing a solution page.

3.3.5 Evaluating implemented solution. In the *evaluating implemented solution* stage, students can test their code against test cases they previously solved in the *understanding the problem* stage. Test cases they did not solve previously are disabled. Students also have the option to add test cases. Figure 5 shows the user interface for this stage 5. The functionality of solving test cases before programming and then running these test cases after programming is influenced by test-driven development (TDD) [11]. Like TDD, students convert requirements from the problem statement into test cases before developing and testing their solution against all test cases. The main difference is that the students' test suites are built by solving pre-defined test cases rather than writing their own. While traditional TDD for education has proven successful in isolation, Metacodenition provides multiple non-code-writing activities for students to complete. Consequently, keeping each activity short to minimise impedance becomes particularly important to keep students motivated.



Run test cases

You'll only be able to run test cases you've previously solved in the understanding the problem stage

	Input	Output	Expected
✓	`int return_value = SumUpTo(2, false); printf("%d", return_value);`	Not run yet!	3
✓	`int return_value = SumUpTo(2, true); printf("%d", return_value);`	Not run yet!	2
	`int return_value = SumUpTo(5, false); printf("%d", return_value);`	Not run yet!	Solve manually first!
	`int return_value = SumUpTo(5, true); printf("%d", return_value);`	Not run yet!	Solve manually first!

Figure 5: Evaluating implemented solution page.

3.4 Technologies

Metacodenition was chosen to be developed into a web-based integrated development environment. A web based development environment is easier for students to access than a desktop application, and no configuration or installation is required. We chose to work with React.js, Mantine UI, and Tailwind CSS as the main front-end libraries, as we had familiarity with them. We selected Monaco editor as the code editor for the IDE as it is the same one used in VS Code, and offers a wide suite of functionality. For hosting our application, we chose the Firebase and Google Cloud Platform services due to their ease of use and suite of features. JobeInABox was chosen as the code-running server as this is the same server that is used to run student's code submissions in the course, and was simple to deploy to Google's Cloud Run service.

4 EVALUATION

Metacodenition was deployed and used by 821 students in a graded lab for a first-year software engineering course at the University of Auckland. The lab had six questions about loops and arrays in the C programming language. The first three problems served as warm-up exercises to ensure that students were comfortable with functions that *return* values, functions that *do not return* values, and iterating through an array. These warm-up problems were provided through the regular platform used for labs, which students were instructed to solve in their own development environments. After question three, the platform asked students to solve problems four to six on Metacodenition and provided them with a link. For simplicity, we will refer to problems four to six of the lab as Metacodenition problems one to three. Table 3 displays the descriptions students were provided with for each of these problems.

4.1 User groups

To address our research question about the effect of scaffolding on the amount of failed test cases in final submissions, we split students into one of two user groups (A or B) when they logged in to Metacodenition. Of the 821 students who used Metacodenition, 392 were in group A, and 429 were in group B. For problem one, students in user group A were provided with all scaffolding, while students in user group B were only given the *implementing a solution* stage and vice versa for problem two. Table 4 summarises this configuration. When students got to problem three, they were prompted to select which problem-solving stages they would like to use. This setup was designed to ensure all students had a fair chance to try all interventions while still allowing us to answer our research question. Final submissions for all problems were recorded. We hypothesise that submissions from the user group who received all scaffolding will have a smaller percentage of test cases that failed for each question.

Table 3: The problems presented on Metacodenition.

#	Problem description
1	Define a function called SumPositiveValues() which is passed two inputs: an array of integers, and an integer indicating how many elements are in the array. The function should return the sum of all positive integers in the input array.
2	Define a function called PrintSummary() which is passed one input: an array of integers of unknown length. This function should determine whether there are more positive or more negative values in the array. You should only examine values in the array until the first occurrence of the value 0. If there are more positive than negative values, you should print the word "Positive". If there are more negative values than positive values, you should print the word "Negative". If there are an equal number of positive and negative values, you should print the word "Equal".
3	Define a function called PrintAverageRainfall() which is passed an array of integers representing daily rainfall amounts. The function should calculate and print the average rainfall. The daily rainfall amounts are occasionally corrupted, represented by negative values, so any negative value should be ignored from your calculation. The array also contains a special value, -999, to indicate the end of the valid rainfall data, so you should only examine values in the array up to the first occurrence of the value -999. Print the average of the valid rainfall data, rounded to 2d.p. If there are no valid rainfall values then print "no rain".

Table 4: Summary of the configuration of the scaffolding made available to each user group for each problem.

Problem #	Group A	Group B
1	All scaffolding	Only *implementing a solution*
2	Only *implementing a solution*	All scaffolding
3	Choice of scaffolding	Choice of scaffolding

4.2 Questionnaire

At the end of the lab, an optional questionnaire was displayed for students to answer. The following questions were asked to address our research question of student perceptions of Metacodenition:

(1) How helpful was the problem-solving assistance?
(2) Which assistance was the most helpful?
(3) I would use this tool again if it was optional (True or False).
(4) What worked well?

Question one was a five-point scale question which asked students to rank how easy it was to use Metacodenition from zero (extremely difficult) to five (extremely easy). Question two was close-ended, allowing students to select a problem-solving stage. Question three was a close-ended question presented as a checkbox, allowing students to check if they agreed with the statement. Question four was an open-ended question intended to collect student anecdotes.

The questionnaire also included a number of questions to identify room for improvement or any bugs that students encountered:

- How easy was it to use this tool?
- What went wrong?

5 RESULTS AND DISCUSSION

5.1 Test cases failed in final submissions

The metric we used to measure a user group's performance was the percentage of test cases that a user group's final submissions failed. A final submission refers to the source code that a student submits to Metacodenition (only one submission is allowed per student). We ran all final submissions against the test suite, which consists of the test cases available to students in the *evaluating implemented solution* stage and additional "hidden" test cases covering edge cases. Tables 3 and 4 present the test suites we run final submissions against and the count of submissions which failed each test case per user group. Table 3 presents the test suite for problem 1, while Table 4 presents the test suite for problem 2.

Figure 6 shows a bar graph of the percentage of test cases that failed in final submissions per user group per problem. We see that for problem 1, the percentage of test cases failed by group A, which received scaffolding for this problem, is approximately 2%, compared to approximately 22% by group B. For problem 2, the percentage of test cases failed by group B is approximately 8%, in contrast to approximately 13% by group A. These results show that students who receive all scaffolding tend to fail fewer test cases than students who receive no scaffolding. This result could be attributed to students who received all interventions having the ability to view the test suite (understanding the problem) and test their code against it (evaluating implemented solution) before submission. However, as we see in Tables 3 and 4, this trend still holds for hidden test cases, which neither students with scaffolding nor without could view or test their code against. Additionally, students could still manually run test cases when they did not have scaffolding. The difference is that when students had scaffolding, they were explicitly prompted to run their code against test cases. When they did not have scaffolding, they were not made aware of their problem-solving state or the corresponding actions to take. We believe this indicates that Metacodenition's scaffolding positively affects test case failure rates. We also see that the gap between the percentage of failed test cases is smaller for problem two than for problem one. A possible explanation is that the group of students who did not receive scaffolding for problem two had previously received scaffolding for problem one and had become more metacognitively aware. Hence, they self-initiated the task of running test cases.

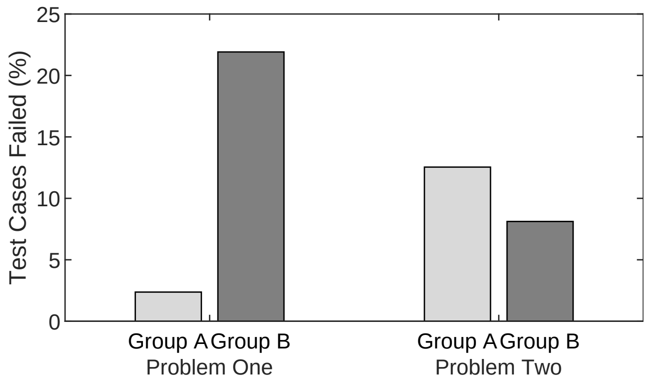


Figure 6: Percentage of test cases failed for final submissions per user group per question.

5.2 Student perceptions of Metacodenition's scaffolding

Most students found the scaffolding provided by the *evaluating implemented solution* stage the most helpful. The second most helpful stage deemed by students was the *understanding the problem* stage. Figure 7 presents a bar graph of the number of students who indicated each stage to be the most helpful. It is no surprise that students found *evaluating implemented solution* to be the most helpful, as this stage appears to contribute the most to the final solution. The popularity of *understanding the problem* is also expected, as this intervention is backed up by past studies showing positive results. Still, it is good to see that this stage was well received by students when combined with the other scaffolding in Metacodenition, as past work studied it in isolation.

Figure 8 shows a histogram of student responses to "How helpful was the problem-solving assistance?". We see that the mean rating is approximately 3.18 out of 5, which corresponds to "helpful". Of the 427 students who answered the questionnaire, 304 student agreed with the statement "I would use this tool again if it was optional". Together, these results indicate that most students perceived Metacodenition as a helpful tool they would use again, even if not required.

In the open-ended responses for "What worked well?", many students expressed that they liked a certain stage of Metacodeniton:

- "Understanding the problem was a huge help. Not only did it allow for easy testing later but often there is a bit of ambiguity of what to do in a specific case so its really nice to have that sorted before I started coding."
- "I liked the Test cases and the highlighting of the question sections. I found I fully understood all the edge cases and sentences in the question that I would typically miss."

Table 5: Test cases failed in final submissions for problem one.

Input Array	Input Length	Expected Output	Hidden (Y/N)	Failed Count	
				A	B
{1, 2, 3}	0	0	N	7	20
{1, 2, 3}	2	3	N	7	25
{1, 2, 3}	3	6	N	8	23
{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}	6	0	N	9	124
{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}	11	15	N	9	125
{-150, 150}	2	150	N	10	124
{125, 125, -256, 328}	4	578	Y	11	125
{10000, 30000, -100000}	3	4000	Y	11	124
{-250000, -100000, -14, 3}	4	3	Y	10	125
{1024, 1024, 2048, 4096, -8192}	5	8192	Y	11	125
{1, 2, 3, 0}	0	0	Y	7	20

Table 6: Test cases failed in final submissions for problem two.

Input Array	Expected Output	Hidden (Y/N)	Failed Count	
			A	B
{1, 2, 3, 0}	Positive	N	46	32
{0, 1, 2, 3}	Equal	N	46	34
{-1, 0, 1, 2, 3}	Negative	N	49	34
{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}	Negative	N	47	33
{-5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 0}	Equal	N	50	34
{-150, 15, 10, 0}	Positive	N	50	37
{1000000000, -1, -1, 0}	Negative	Y	52	32
{1, -1, 1, -1, 1, -1, 1, -1, 0, 1, -1, 1}	Equal	Y	50	36
{1, -1, 1, -1, 1, -1, 1, -1, 1, 0, -1, 1}	Positive	Y	50	37
{1, -1, 1, -1, 1, -1, 1, -1, -1, 0, 1, 1}	Negative	Y	51	35
{0}	Equal	Y	46	33
{-4123, -58182, -16, 3, 1, 7, 8, 0}	Positive	Y	53	36

- "Requiring the user to highlight and explain parts of the brief makes it much easier to make a pseudo code."
- "It was exactly like writing a pseudocode, except you could change the order of your plan easily. It was just super helpful to type down notes and clear my head before I started coding."
- "I really liked how I could write out a solution plan which outlined what I needed to do, and how it remained visible as I was writing my code."
- "I found the evaluating part helpful as it allowed me to see all the tests together and see patterns in the tests that failed."

Other students indicated that they particularly liked how the *understanding a problem* and *evaluating implemented solution* stages linked together:

- "It was good to be able to manually solve the expected outputs and then test my code on those same test cases."
- "I like how we are able to check our code solution to something we hand worked first."
- "Manually working out answers and then comparing them to the code's actions was very useful."

Multiple students expressed interest in enhanced compiler error messages (ECEMs), with one stating: "Some of the error messages

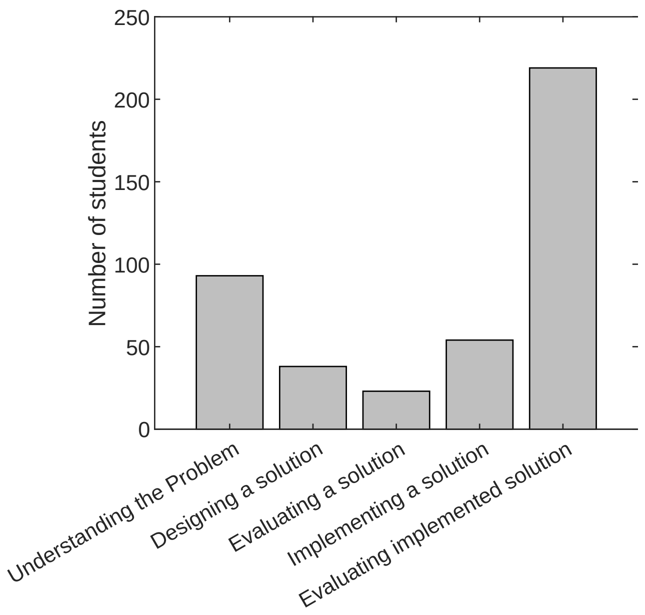


Figure 7: Responses to "Which assistance was the most helpful?".

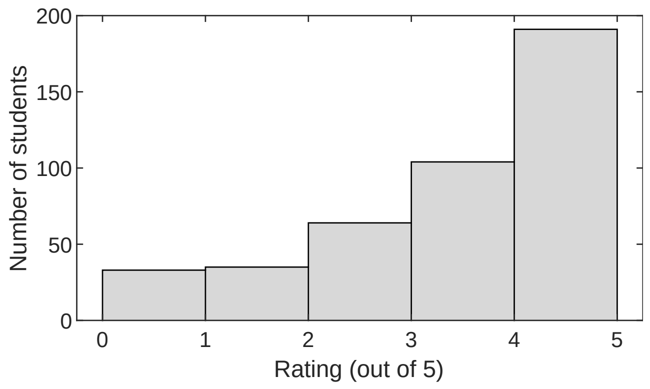


Figure 8: Responses to "How helpful was the problem-solving assistance?".

are vague or hard to understand to inexperienced coders". Multiple students indicated an interest in a specific ECEM, which is being informed when their code enters an infinite loop:

- "If a while loop prints something infinitely, it makes the output text box stretch out."
- "Occasionally when my code didn't work, it didn't produce any output or give any error message so it was hard to move further."

Interestingly, one student responded, "Felt like learning a new skill" to the question on what went wrong. Though this student felt that this went wrong, we consider their response to be a positive indicator that Metacodenition achieved what we intended it to do - which is to teach students problem-solving skills.

5.3 Limitations

One limitation is that our evaluation was conducted with a limited scope, taking place in a single lab over a week. Problem-solving skills take time to develop, so a longer time frame is needed to evaluate Metacodenition's interventions accurately. Future studies should investigate Metacodenition's benefits over a more extended period, such as over a semester.

Another potential limitation could have been introduced by some students knowing this study was conducted by fourth-year engineering students for a part four project. These students could have had a courtesy bias, preventing them from giving negative feedback in our questionnaire.

Additionally, a limitation of our evaluation could have stemmed from the bugs reported by students, which were fixed during the first two days. Students who accessed the tool on different days may have had different experiences. These students likely had a cognitive bias where the bugs they experienced were heavily weighted in their overall judgement of Metacodenition, resulting in more negative feedback (peak-end rule).

6 CONCLUSIONS

This paper presented our approach to designing and evaluating a tool for novice programmers called Metacodenition, which provides metacognitive scaffolding for a problem-solving framework. By scaffolding the problem-solving process from start to finish, Metacodention aims to promote the development of problem-solving skills in novices. Results from our evaluation show fewer test cases failed in the final submissions of students who received all of Metacodenition's scaffolding compared to students who received only the *implementing a solution* stage. We also found that most students perceived Metacodenition as a helpful tool they would voluntarily use in the future.

Student feedback has highlighted a potential direction for improving Metacodenition, specifically in adding enhanced compiler error messages. Loksa et al.'s problem-solving framework also included the *searching for analogous problems* stage that Metacodenition omitted. Here, students draw upon problems they have encountered previously. Additional future work could include adding an intervention for this stage to Metacodenition. We suggest a "problem bank" of past problems a student has solved and their corresponding solutions, allowing them to identify and filter relevant problems (ones with similar patterns).

REFERENCES

[1] Christine Bruce, Lawrence Buckingham, John Hynd, Camille Mcmahon, Michael Roggenkamp, and Ian Stoodley. 2003. Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. (2003).

[2] Fulvio Corno, Luigi De Russis, and Juan Pablo Sáenz. 2021. TextCode: A Tool to Support Problem Solving Among Novice Programmers. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–5. https://doi.org/10.1109/VL/HCC51201.2021.9576164

[3] Michelle Craig, Andrew Petersen, and Jennifer Campbell. 2019. Answering the Correct Question. In *Proceedings of the ACM Conference on Global Computing Education* (Chengdu,Sichuan, China) *(CompEd '19)*. Association for Computing Machinery, New York, NY, USA, 72–77. https://doi.org/10.1145/3300115.3309529

[4] Michael de Raadt, Richard Watson, and Mark Toleman. 2009. Teaching and Assessing Programming Strategies Explicitly. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95* (Wellington, New Zealand) *(ACE '09)*. Australian Computer Society, Inc., AUS, 45–54.

[5] 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) *(Koli Calling '19)*. Association for Computing Machinery, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/3364510.3366170

[6] Anneli Eteläpelto. 1993. Metacognition and the Expertise of Computer Program Comprehension. *Scandinavian Journal of Educational Research* 37, 3 (1993), 243–254. https://doi.org/10.1080/0031383930370305

[7] Rita Garcia. 2021. Evaluating Parsons Problems as a Design-Based Intervention. In *2021 IEEE Frontiers in Education Conference (FIE)*. 1–9. https://doi.org/10.1109/FIE49875.2021.9637175

[8] Roseanne Gibson. 1996. *Developmental aspects : Metacognition and problem solving*. Bachelor's Thesis. https://ro.ecu.edu.au/theses_hons/882/

[9] Anabela Gomes and António José Mendes. 2007. Problem Solving in Programming. (2007).

[10] Andrew D. Hilton, Genevieve M. Lipp, and Susan H. Rodger. 2019. Translation from Problem to Code in Seven Steps. In *Proceedings of the ACM Conference on Global Computing Education* (Chengdu,Sichuan, China) *(CompEd '19)*. Association for Computing Machinery, New York, NY, USA, 78–84. https://doi.org/10.1145/3300115.3309508

[11] David Janzen and Hossein Saiedian. 2008. Test-Driven Learning in Early Programming Courses. *SIGCSE Bull.* 40, 1 (mar 2008), 532–536. https://doi.org/10.1145/1352322.1352315

[12] Antti-Jussi Lakanen and Ville Isomöttönen. 2015. What Does It Take to Do Computer Programming? Surveying the K-12 Students' Conceptions. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 458–463. https://doi.org/10.1145/2676723.2677229

[13] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) *(ITiCSE-WGR '04)*. Association for Computing Machinery, New York, NY, USA, 119–150. https://doi.org/10.1145/1044550.1041673

[14] Dastyni Loksa and Amy J. Ko. 2016. The Role of Self-Regulation in Programming Problem Solving Process and Success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 83–91. https://doi.org/10.1145/2960310.2960334

[15] Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 1449–1461. https://doi.org/10.1145/2858036.2858252

[16] Murali Mani and Quamrul Mazumder. 2013. Incorporating Metacognition into Learning. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 53–58. https://doi.org/10.1145/2445196.2445218

[17] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Canterbury, UK) *(ITiCSE-WGR '01)*. Association for Computing Machinery, New York, NY, USA, 125–180. https://doi.org/10.1145/572133.572137

[18] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. 2019. A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education* 62, 2 (2019), 77–90. https://doi.org/10.1109/TE.2018.2864133

[19] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-Oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. *SIGCSE Bull.* 39, 3 (jun 2007), 151–155. https://doi.org/10.1145/1269900.1268830

[20] Jennifer Parham, Leo Gugerty, and D. E. Stevenson. 2010. Empirical Evidence for the Existence and Uses of Metacognition in Computer Science Problem Solving. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) *(SIGCSE '10)*. Association for Computing Machinery, New York, NY, USA, 416–420. https://doi.org/10.1145/1734263.1734406

[21] James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 531–537. https://doi.org/10.1145/3287324.3287374

[22] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 74–82. https://doi.org/10.1145/3105726.3106169

[23] Alex Radermacher, Gursimran Walia, and Dean Knudson. 2014. Investigating the Skill Gap between Graduating Students and Industry Expectations. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 291–300. https://doi.org/10.1145/2591062.2591159

[24] Juan Pablo Saenz and Luigi De Russis. 2022. On How Novices Approach Programming Exercises Before and During Coding. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 361, 6 pages. https://doi.org/10.1145/3491101.3519655

[25] Nathaniel Weinman, Armando Fox, and Marti A. Hearst. 2021. Improving Instruction of Programming Patterns with Faded Parsons Problems. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 53, 4 pages. https://doi.org/10.1145/3411764.3445228