Jigsaw: A Tool for Decomposing and Planning Programming Problems

Heidi Reichert
North Carolina State University
Department of Computer Science
Raleigh, NC, USA
hreiche@ncsu.edu

Benyamin T. Tabarsi
North Carolina State University
Department of Computer Science
Raleigh, NC, USA
btaghiz@ncsu.edu

Thomas Price
North Carolina State University
Department of Computer Science
Raleigh, NC, USA
twprice@ncsu.edu

Tiffany Barnes
North Carolina State University
Department of Computer Science
Raleigh, NC, USA
tmbarnes@ncsu.edu

2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) | 979-8-3503-6613-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/VL/HCC60511.2024.00034

***Abstract*—Many students struggle with decomposition and planning despite the necessity of these skills in computing education. Hence, more tools are needed to scaffold these processes. In this paper, we present Jigsaw, a standalone visual planning tool to help students practice decomposition and planning before writing code. Jigsaw allows students to compose a solution to a new problem based on previously seen "patterns," such as the accumulator pattern for summing values or the filter pattern for conditional input selection. Students can connect these patterns together to see how data flows between them and define a solution plan. Jigsaw's goal is to scaffold students' planning processes by presenting relevant patterns for a given problem. Using a within-subjects design, we evaluated Jigsaw by observing 17 undergraduate students as they planned for and implemented two programming assignments. The experimental task included Jigsaw, and the control task did not. This design aimed to understand how the tool impacted students' planning and programming process. Subsequently, we conducted interviews with these students regarding their planning and programming experiences with and without Jigsaw. Many students explicitly mentioned they would employ Jigsaw for planning and appreciated the scaffolding it provided. Students also admired the Jigsaw's novelty in visualizing programming problems. We conclude with our design takeaways and recommendations for future work.**

***Index Terms*—planning tool, CS1, decomposition, planning scaffolding**

I. Introduction

Decomposition and plan composition are fundamental skills for computing students [1], [2]. Students must learn to solve new problems by *decomposing* them down into smaller subgoals that they already know how to solve, also called "patterns," "recipes," or "tasks," followed by *composing* these pieces into a solution plan [1]. However, these skills are not always practiced explicitly in CS courses, and students are often expected to implicitly *plan* a solution while *implementing* it in code, which can make both tasks more challenging. Previous studies have shown that students struggled to decompose programming problems, recognize when patterns may be useful, and plan out a solution [3]–[5]. These papers have cited the need for more scaffolding for the planning process in order to help students identify the patterns that can be composed into a solution.

To address this, we developed a novel tool called *Jigsaw*, where students can compose canonical programming patterns to plan out the solution to a programming problem. Jigsaw offers novel features, including a palette of relevant patterns, scaffolding to connect these patterns to the given problem, and visualizing how different test inputs change program outputs at each step of the plan. To evaluate Jigsaw, we conducted a pilot study with three high school students in a summer programming internship, as well as a follow-up within-group study with seventeen undergraduate introductory programming students to complete two programming assignments in the language of their choice. One assignment was completed while planning with Jigsaw (experimental), while the other was planned according to the students' preferences (control). After completing each plan, students were asked to program the corresponding task. Finally, they were interviewed regarding their attitudes toward Jigsaw. We did this to understand the impact of this form of scaffolding, as well as to gather suggestions to further improve Jigsaw for future deployment in the classroom.

Through this process, we aimed to answer the following research question:

- How does providing scaffolding via an interactive visual tool like Jigsaw impact students' planning and programming processes?

Our evaluation suggests that students found Jigsaw's visualization facilitated planning and programming through visualization, although the tool needs to be improved impacting students' code correctness or speed. Our results have broader implications for future studies on supporting students in planning and decomposition.

979-8-3503-6613-6/24/$31.00 ©2024 IEEE
DOI 10.1109/VL/HCC60511.2024.00034

II. Background

In this section, we start with the theoretical and practical facets of planning and decomposition. Then, we discuss the importance of supporting planning in programming. We elaborate on this by exploring decomposition and programming patterns as important factors in formulating a successful plan. Finally, we note some tools developed for novices to plan their programming activities.

A. Planning and Decomposition

Planning emphasizes high-level strategies for solutions without including low-level elements of implementation [6]. Effectively, this entails creating more abstract or general goals of a program rather than focusing on the specific details of implementation. In programming, identifying the goal of a program and being able to plan for completing that goal is of high importance, especially for novices [7]. This solution planning step commonly includes the utilization of subgoal decomposition [8].

Decomposition (i.e., breaking down a problem into manageable units) is one of the primary facets of computational thinking (CT) [9]. Plans can be defined as "knowledge structures that organize steps for a particular task (or goal) into chunks," where each chunk represents a subgoal [10]. Research has shown that decomposition and abstraction are strong predictors of algorithmic thinking, evaluation, and generalization [11], and decomposition improves code readability and understandability [12].

However, proper decomposition can be demanding for students [8], [13] and needs to be explicitly taught to some students [3], [14], [15]. Research has indicated that although the quality of decomposition can be enhanced by scaffolding it, many students lack the skills to decompose problems and plan after completing introductory programming courses [16]. Thus, there is good potential for tools and methods focusing on novices' decomposition skills.

B. Supporting Planning and Decomposition in Programming

Planning is a quality that separates novices from accomplished programmers [17]. However, planning as strategic knowledge is more challenging than learning the syntax of a programming language for novices [18]. Even though planning can successfully be taught to programming students [2], the best ways to help novice programmers learn, and practice planning need to be further investigated.

Previous studies have explored different approaches, such as the integration of goals and plans in a visual programming environment [19], explicit illustration of algorithmic planning through displaying think-aloud sessions of expert programmers [18], and designing a planning sheet [7]. A study by Chao et al. has shown novices' learning of planning and solution design can be improved by assisting them in decomposing problems [16].

In decomposition, many pieces or patterns are reusable. Coding patterns (also called recipes [20] or schemas [21]) are powerful tools for teaching programming that enable learners to organize their knowledge [22]. In addition, patterns make complex programming tasks easier to follow, explain, and understand [23]. Given these benefits of patterns, prior studies [24], [25] have developed tools and models that utilize schemas and patterns for teaching programming and assessing students' CT and plan to implementation skills. Rivera et al., who studied students' ability to compose plans using Higher-Order Functions (HOFS), discovered that students can both build and implant plans by using HOFS [6].

East et al. [24] devised a model that incorporates patterns for teaching programming, which gained encouraging preliminary results and indicated that going over patterns besides syntax can be very easy for an instructor. Seiter and Foreman [25] developed a model for examining CT through the lens of project-wide design pattern variables. Particularly, they created a model that categorizes CT through the skill level utilized in design patterns. Once students learn the patterns, they can reuse them in different programming tasks, although merging the patterns and plans is challenging for students [26].

The positive impacts of proper decomposition in programming are manifold. First and foremost, decomposition helps students tackle subproblems rather than dealing with the problem all at once. This helps students build an understanding of the main problem, thus solve it more efficiently. Second, proficiency in decomposition affects students' other CT skills. Research has shown that decomposition and abstraction are strong predictors of algorithmic thinking, evaluation, and generalization [11]. Third, using decomposition improves code readability and understandability [12]. Smaller pieces of code, reasonable flow, and different levels of abstraction are the features brought by decomposition that make the code easier to read and simpler to understand. Finally, the benefits of learning and practicing CT skills are not limited to CS [27], and they can be used in other domains [28]. Decomposition is not an exception; students who learn this skill can transfer it to other areas.

C. Programming Planning Tools

Scaffolding the planning process can encourage students to learn this skill. Many tools have been developed to aid students in constructing plans for their problems.For example, Metacodenition [29] is a programming environment that offers students metacognitive scaffolding for programming exercises throughout the problem-solving process. They found that students who received scaffolding in the initial steps of problem-solving (i.e., understanding the problem, designing a solution, and evaluating a solution) experienced fewer test case failures compared to those who only had scaffolding in the implementation stage. TextCode [30] is another IDE designed for novices that pivot their focus toward understanding the problem statement, decomposing it, writing and trying different solutions for each subproblem, and combining the subproblems' code to compose the complete solution. PlanIT! [31], a planning tool integrated within the Snap*!* programming environment, aimed to assist students in making plans for open-ended projects. Students could describe their projects

briefly, make a step-by-step to-do list, and determine the main elements of their projects, such as variables. Students' interviews revealed that their plans became more actionable and precise following the use of the tool.

Despite the importance of decomposition, these tools depended on students' recall of patterns for creating a plan rather than presenting them with relevant programming patterns, which necessitates students' recognition. They also did not have the feature allowing students to run their plans before starting the implementation. Wrenn et al. [32] considered this issue by developing a tool that allowed students to create and evaluate their own examples without any need to do the implementation, although it still required students to create the examples by themselves.

Previous studies [33]–[35] have focused on creating flowcharts from natural language prompts entered by the user, which improved students' performance and problem-solving skills. However, they did not decompose prompts into programming patterns. Cunningham et al. [36] proposed purpose-first programming, which was intended to help conversational programmers who needed the skill to communicate about technical programming topics rather than knowing the syntax. Their approach, which provided scaffolding through code patterns that were mergeable into their code, showed that programmers were motivated to learn and work more with this approach, allowing them to meet their goals and feel successful.

Tseng et al. [37] proposed a system that presents prefabricated plans and requires minimal effort by students. It offered students an example of the concepts being taught, some programming plans, and the code related to each plan. Jin et al. [38] assessed the effectiveness of guided planning and assisted coding in an intelligent tutoring system, which showed that students' learning gains were higher with this tool in comparison to coding-only, planning-only, and interleaved planning-only and planning-coding environments.

The primary limitation of most of these tools is that they have predominantly relied on students' ability to build a plan from scratch without offering any pre-defined patterns. While this might not be challenging for experienced developers, novice programmers may learn more from recognition than recall of suitable planning components. Another limitation is that students may be expected to create plans by writing pseudocode or code snippets, which cannot be tested and may not fit as a reusable pattern that benefits students in future implementations. In designing Jigsaw, elaborated upon in Section III-B, we aimed to address these gaps.

III. Tool Design

Jigsaw is a standalone program planning tool designed to 1) separate the challenges of program design from implementation, 2) scaffold the planning process using best practices from literature and 3) allow students to execute their plan with different inputs before getting into actual implementation. In this section, we first describe Jigsaw's user experience, and then we highlight the specific design goals of Jigsaw and how these connect to prior literature on planning and computing education.

A. User Experience

Learning Context: Jigsaw is designed to be used in a formal learning context, where students have been explicitly taught programming patterns (e.g., a design recipe approach [39] or pattern-oriented instruction [40]). The tool assumes students have practiced implementing these patterns (e.g., filter, reduce, accumulate) as code, but may struggle to *decompose* larger problems into patterns, and to *compose* those patterns together into a solution. Students can use therefore Jigsaw at the start of a programming problem, to form a plan using these programming patterns, and then reference that plan throughout programming. Jigsaw is hosted online and can be accessed via a direct link through any web browser[1], although it is most effective on a computer or laptop rather than a mobile device. Accordingly, it can be used within any class that has Internet access.

As an example, Figure 1 shows Jigsaw's user interface, and a completed plan for a simplified version of the rainfall problem [41] (described in more detail in Section IV-A3). When planning out a program in Jigsaw, students compose their plan out of *pattern blocks*, each representing a common programming pattern (e.g. "read input until value," or "filter" or "sum"). Students can browse through relevant patterns from a palette at the bottom, and drag patterns they wish to use into their workspace, similar to many block-based programming environments.

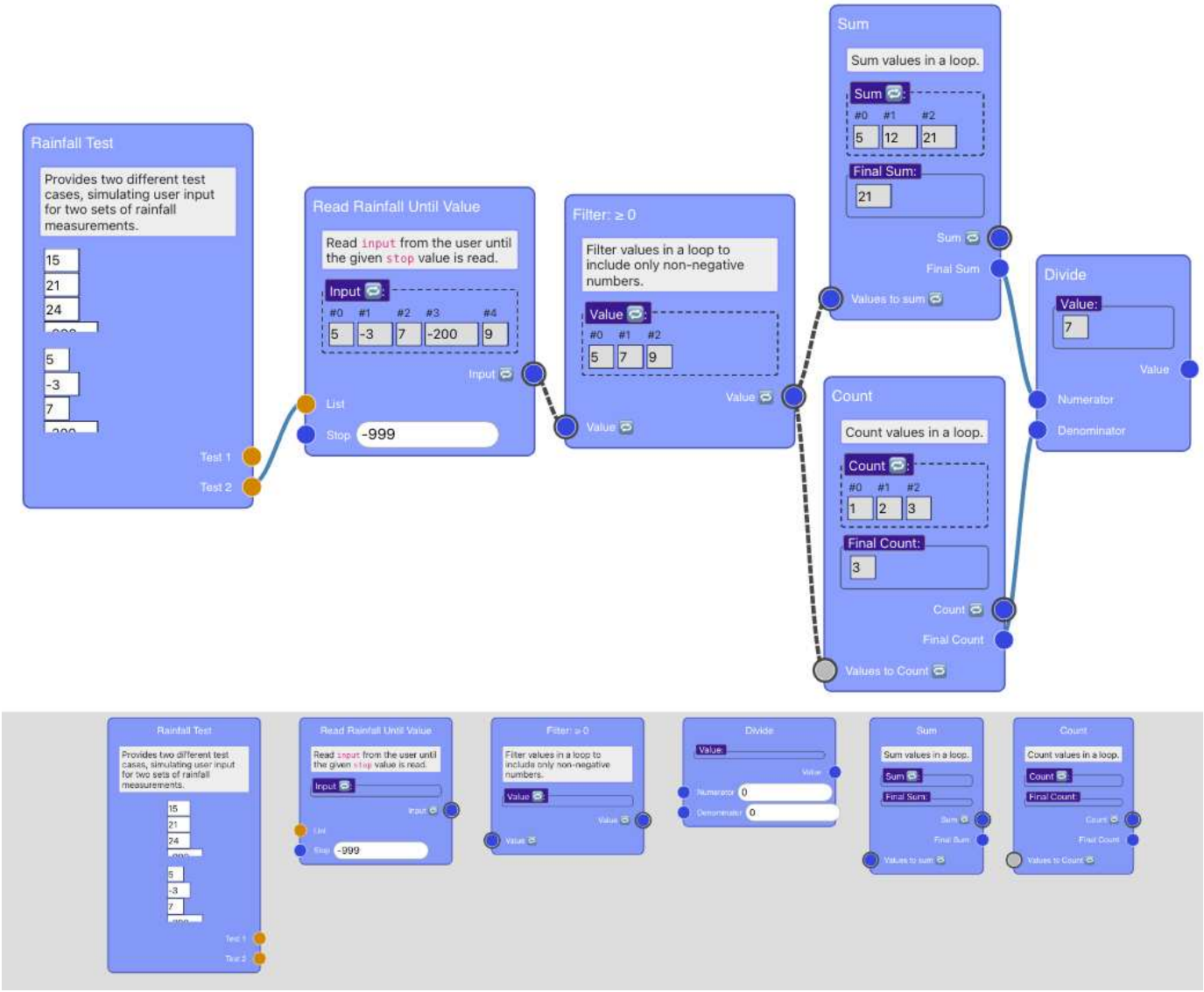
Each pattern block consists of a title ("Filter ≥ 0"), a natural language description ("Filter values in a loop to include only non-negative numbers"), a set of inputs (left) and outputs (right). Inputs and outputs are represented as circles, and students can connect the output of one block to the input of another by clicking and dragging. The inputs and outputs of each block are color-coded to indicate the data type of the input/output, such as a list (orange), a number (blue), or any type (gray).

One of our primary design goals (see below), which sets Jigsaw apart from existing planning tools, is that students' plans are fully executable. Each plan block shows its current output given its current input; for example, the "Filter ≥ 0" block takes as input numbers `5, -3, 7, -200, 9`, and has output `5, 7, 9`. As students connect new blocks, Jigsaw updates their outputs dynamically. For each problem, Jigsaw includes a test block with multiple test inputs and students can visualize how these inputs affect both intermediate and final outputs of the program, in order to verify the correctness of their plan, and spot potential missed edge cases. While test cases are technically not a part of the *program* plan itself, prior work shows that testing input/output pairs is an important step in program comprehension and planning [29], [42].

[1]A version of Jigsaw is accessible via the following link: isnap.csc.ncsu.edu/pbp-planner/dist/#/editor/test.

Note that one may replace 'test' with 'rainfall' or 'bottled_water' to find versions tailored to the problems referenced in Section IV-A4

Fig. 1. Jigsaw's user interface, consisting of a palette (bottom) and a work space where students connect pattern blocks together (middle). Each block has inputs and outputs (circles), which can be connected, and a preview of the outputs' values.



Jigsaw's executable plans are represented similarly to a dataflow programming language (e.g. LabView [43]), with inputs connecting to outputs. This choice enables us to represent plans as a composition of reusable patterns. This would be much more difficult in an imperative representation, where programming patterns often correspond to lines of code that might be spread throughout a student's program. For example, in an imperative language like Java, a "sum" pattern (i.e., accumulator) requires a loop to be present, and includes code before the loop (initializing the variable), in the loop (adding something to the variable), and after (using the sum value). Additionally, multiple patterns might use the same loop (e.g. the "sum" and "count" patterns in Rainfall use the same "read input until" loop).

Because Jigsaw is a standalone program planning tool and not a language itself, it is consequently language-agnostic and can be used in conjunction with most common introductory programming languages, such as Java, Python, MATLAB and Snap*!* (which were all used in our evaluation in Section IV-A4). Since these languages are imperative, Jigsaw introduces a special "loop" input/output type, indicated by a dark circle, connected by dashed lines, and a symbol, as seen in Filter's input/output (see Figure 1). This indicates a variable that changes values in a loop, such as the "input" read from the user in the "Read Rainfall Until Value" block. This helps students connect Jigsaw's functional plan representation to imperative program code. When visualizing the output of these blocks, Jigsaw shows all values that the variable takes, in order of iteration (e.g. `5, 7, 9` in the Filter block). If students hover over any of these values, Jigsaw highlights other

values that were calculated in the same iteration. For example, hovering over Filter's "7" would highlight Sum's "12" (the sum after the second iteration of the loop: 5 + 7), and Count's "2". Jigsaw's interface can also support 2 nested loops (using a 2-dimensional table to represent values), as well as lists (and lists of lists), and future work may explore how to represent more complex data types. However, an important principle of Jigsaw's design is *not* to be a complete programming language, capable of representing arbitrary programs. Jigsaw's planning blocks are scoped for CS1 problems, and some CS2 problems, where students are still learning basic programming patterns; they cannot be exported into a programming environment.

The version of Jigsaw used in this study is inherently problem-specific (aligning with our design goals, below), and therefore appropriate patterns for each problem must be designed by a researcher or instructor. However, Jigsaw has since been extended to allow students to define their own custom pattern blocks, which can map a set of input values to specified outputs values. This enables Jigsaw to be used on a wider variety of problems, which cannot be fully mapped to a set of reusable patterns.

B. Design Goals

Jigsaw's design was focused around four key design goals that we identified from prior literature:

1) Center planning on programming patterns: Prior studies have found that recognizing and implementing reusable programming patterns is a critical skill for students learning to program [22], [40]. Therefore, Jigsaw defines planning as organizing and connecting reusable pattern blocks.

2) Use purpose-first language to contextualize patterns: By definition, reusable programming patterns are generic (e.g., "filter"), not context-specific (e.g., "keep positive rainfall values"), and this can make it difficult for students to understand how to apply and combine patterns in the specific context of a given problem [22], [36]. To address this, Cunningham et al. argue that programming patterns should be presented "purpose-first," where each pattern is described with "domain-specific," contextualized natural language [36]. Similarly, Jigsaw labels patterns with both a reusable label (e.g. "filter"), with the goal of helping students recognize a pattern, and a context-specific label ("numbers $\geq$ 0") that connects the pattern to the problem.

3) Enable recognition over recall of patterns: Prior work has shown that students are much better able to *recognize* relevant information, when it is presented to them, than they are at *recalling* it without a prompt [44]. Jigsaw therefore provides students with a palette of relevant programming patterns, such as "sum," "count," "filter," "always," and "never," which students can insert into their plans as needed. Lidwell et al. [45] name this as one of their principles of universal design.

This builds on prior work in block-based programming environments (BBPEs), which present students with a palette of blocks, allowing them to easily recognize, search and browse for relevant functions, rather than recalling them from memory. However, while BBPEs offer a different style of programming, Jigsaw is intended to guide students in making plans based on programming patterns.

In the problems we used in our evaluation of Jigsaw (Section IV-A3), we only included relevant pattern blocks, since prior work in Parsons problems has suggested that irrelevant "distractor" blocks can inhibit learning and increase problem solving time [46], [47], though future work could explore the effect of such blocks.

4) Make plans testable and self-evident: Prior work has shown that a major barrier students face when planning and programming is that they form an incorrect conceptual model of what the problem is asking [42], [48]. However, a major limitation in prior work on tools to scaffold planning and decomposition, such as flowcharts and pseudocode, is that there is no way for the student to self-check whether their plan is correct before moving on to implementation. We, therefore, designed Jigsaw to allow students to fully execute their plans and provide test inputs that reveal potential problems in these plans (e.g. forgetting to filter out negative rainfall readings). To facilitate debugging incorrect plans, Jigsaw also *shows the output of each plan block*, allowing students to easily identify where their plan went awry.

IV. Methodology

A. Study Design

*1) **Pilot Study**:* This study was administered in two parts: a pilot study and a formal one. The pilot study was conducted in the summer of 2022, while the formal study was conducted in the fall and winter of 2022. For our pilot study, three high school students who were engaged in a summer internship at a public university in the US volunteered to participate without compensation. These students were proficient in Snap*!*, which was the language used for their programming sessions. The pilot study allowed the researchers to smooth out issues with the assignments and protocol, as well as to develop an initial codebook for future thematic analysis, to be elaborated upon in Section IV-B. The pilot study consisted of the same assignments, protocols, and interview questions that we discuss below. However, the pilot population was not in our target demographic, and we did not consequently include their results in this study.

*2) **Participants and Recruitment**:* Our target population for using Jigsaw consisted of students in introductory CS courses—this included CS majors, non-majors, and minors. Since Jigsaw was designed to be language agnostic, we intentionally recruited students from diverse introductory programming courses. Thus, for our formal study, all participants were recruited from 100-level CS courses. Seventeen students signed up to participate in the study. They were enrolled in either of the introductory programming classes in Java, Python, or Matlab during the fall of 2022. Recruitment advertisements were made via email and Moodle, and participants were compensated with a $30 Amazon gift card.

Students were asked to schedule a 90-minute study session over Zoom, during which either one or two researchers were present. Sessions were conducted remotely due to the ongoing

COVID-19 pandemic. In the event where two researchers were present, one conducted the study while the other took notes. Participants were given anonymous IDs for use during the study and were instructed to turn off their videos before recording. Data collection was limited to students' screens and voices.

*3) **Assignments**:* During the study session, students were tasked with working on two assignments. Both assignments involved elements of reading input, accumulators, and filters. Both assignments were formatted as a Google document in view-only mode that was shared with the student and included starter code to prompt the user, as well as several test cases for assessing the correctness of the program.

a) Rainfall: The first assignment was a modified version of Rainfall, a task first designed by Elliot Soloway in 1986 [41]. This was chosen due to its status as a common programming task that has been repeatedly studied for difficulty [49], [50]. Rainfall asked students to take a set of numeric inputs from a user one at a time and average and output the non-negative values. However, we modified the task to remove the possibility of invalid inputs or no inputs (i.e., a student could assume all potential inputs would be valid and there would be no division by zero), thus making it less challenging.

b) Water/Bottle: The second assignment was entitled Water/Bottle. This task was based on the balanced delimiter problem featured as a free response question in the 2019 AP Computer Science A exam [51]. This task asked students to take a set of string inputs from a user one at a time. However, in our modification of the delimiter problem, we required students to assess at every iteration of the loop whether there was a "bottle" to hold "water" within.

These two assignments' prompts, starter code, and solutions were available in the programming languages Python, Java, and MATLAB.

*4) **Programming Session**:* Before programming, each student was randomly assigned to use the tool for either Rainfall or Water/Bottle. The session was structured as follows:

1) Prior to planning and programming, students were instructed to verbalize their thoughts as they completed their tasks. Students were shown an approximately one-minute-long video demonstrating the process of "thinking-aloud,", in which one of the researchers spoke their thoughts aloud while solving a math problem.
2) Before beginning programming, all students were given an approximately 5-minute presentation via a slide deck shared and presented by the researcher. The researcher went over the programming recipes "Read input until," "Accumulator," and "Filter," which are described in Table I. Each pattern was explained in plain terms and included functional sample code in the student's programming language of choice. This code was available for students to copy and paste.
3) After this, each student worked on Rainfall. Students whose experimental task was Rainfall first received an approximately 7-minute tutorial on using Jigsaw. This tutorial was an interactive demonstration within a demo-version of the tool, including how to use the tool (i.e., dragging out and connecting blocks), and the functionalities of the blocks. Students were instructed to follow along on their end using the same demo-version of the tool, as well as encouraged to ask clarifying questions during this process. Students whose experimental task was Water/Bottle were *not* given the tutorial at this point.
4) Before starting to program the Rainfall, the researcher shared a link via Zoom chat to a Google document containing the Rainfall instructions and test cases, as well as a link to the planning area. For students using Jigsaw, this was a link to a version of Jigsaw that was tailored to Rainfall. For students whose control task was Rainfall, this was a link to a Google document with embedded links to the programming pattern slide deck.
5) Students were instructed to read the instructions and plan for at least five minutes. Once five minutes had passed, the researcher informed the student that they could begin programming or continue to plan. However, in the event that a student completed planning before time had finished, students were given permission to continue with programming, thus allowing for some students to begin programming before five minutes had passed.
6) Students were then allowed 25 minutes to either continue planning or programming. After 25 minutes, students were informed that they could either continue programming for up to five more minutes or move on to the next assignment.

After the completion of Rainfall, this process was repeated for Water/Bottle. Students who had *not* used Jigsaw by this point received the tutorial and scaffolded version of the tool, while students who had used Jigsaw already received only a link to a planning document.

*5) **Interview**:* Following the completion of their second assignment, students were requested to stay for an approximately 10-minute interview using a semi-structured approach. Students were asked about their planning and programming experiences, as well as their general attitudes toward planning. They were also asked to elaborate on their experiences using the tool, what they found useful and difficult about the tool, and how they planned differently with and without the tool. From here on, participants are referred to as P*X*, where *X* represents the numerical order in which their interview was conducted.

B. Analysis

We utilized both qualitative and quantitative approaches for data analysis. The former primarily used students' interview data and the latter was mainly based on students' times during programming and planning.

*1) **Qualitative Analysis**:* In order to best understand how students perceived the usefulness of Jigsaw for planning scaffolding, we performed qualitative analysis using thematic analysis on the transcripts [52], [53]. Because of the open-ended nature of our research question and limited sample size,

TABLE I
THE PATTERNS PRESENTED TO STUDENTS DURING THE STUDY SESSION, ALONGSIDE THEIR SHORT DESCRIPTIONS.

Pattern	Description
Read Input Until	Reading input taken from a user until the user states the stopping criterion; examples include the user typing 'stop.'
Accumulator	Using variables within a loop; examples include summing a set of numbers, counting the number of times something happens, or checking conditionals.
Filter	Keeping input that only meets certain conditions; an example is non-negative numbers

we considered thematic analysis to be the appropriate avenue for our analysis.

First, we utilized an inductive approach [53] for creating our preliminary codebook. This was done via open coding of the interviews of our pilot study. All three of the interviews conducted during this study were re-watched and coded by the two researchers who later conducted the study sessions. From this open coding and a discussion between the researchers, during which their respective tags and notes were compared, a preliminary codebook was created.

This codebook was then refined through the further open coding of three students in our final study. One of these interviews, P6, was chosen at random; the other two, P2 and P9, were chosen by the researchers because of their identified relevant and interesting themes. All codes were chosen to relate generally to decomposition, planning, programming, and attitudes toward Jigsaw.

After creating the codebook, we utilized a deductive approach [53] to tag the remaining fourteen interviews. The two researchers who conducted the study sessions acted as coders. Each researcher watched every student's interview and separately tagged the interview according to the codebook. After repeating this procedure for three or four students' interviews, the researchers reconvened to compare and discuss their respective tags. This process was completed within two days of completing individual coding to ensure the researchers accurately recalled why they made their coding decisions. During instances in which tags or quotes did not align with each other, the researchers would re-assess, discuss, and then tag the relevant quote together. In all instances of initial disagreement, the researchers reached a mutually agreed-upon decision regarding tagging. Because all seventeen interviews were coded by two researchers and by the fact that every tagging was discussed almost immediately after the interviews were re-watched, inter-rater reliability was deemed unnecessary and thus not calculated.

When the two researchers found it necessary, the codebook was further refined. We then grouped these codes into five sections, roughly aligning with our design goals: these concerned the scaffolding Jigsaw provided, the relationship between planning and implementation, the use of Jigsaw to test and execute code, the visual interface of Jigsaw, and the impact of Jigsaw-scaffolded plans on students' programming. We align our results in Section V-A accordingly.

*2) **Quantitative Data Collection**:* As we wanted to understand how scaffolding during planning impacted the ability to implement students' programs, we recorded some quantitative data regarding students' planning and programming processes. Because we had only 17 participants and 34 plans/programs, we do not include any statistical tests in this evaluation and do not expect to reach any quantitative conclusions; however, we chose to include descriptive statistics in order to present a more complete picture of students' use of Jigsaw.

The amount of time that students spent planning and programming for each assignment was recorded by one researcher while rewatching all seventeen session recordings. The researcher also listened to students' verbalized thoughts to ensure accuracy, particularly in differentiating planning and programming times. Programming times did not begin until a student began typing in a programming environment. All times were recorded in seconds for precision.

We developed a 5-point rubric for assessing each assignment. Rubrics were completed for each assignment and concerned whether the program ran/compiled, accepted correct input and produced correct output, implemented the counting pattern, implemented filtration (for Rainfall) or accumulators (for Water/Bottle), and successfully completed all provided test cases (which encompassed all potential edge cases).

All 17 students were able to complete their plans within Jigsaw successfully. In order to better see how students leveraged their plans during implementation necessitated collecting information on how students engaged with these plans, we also recorded the number of times that students referred back to their plans was thus recorded for each student. Students' references to their plans were considered when they clicked on or interacted with their plan, or when their verbalized thoughts indicated they were referring to their plans. Six students held their plans side-by-side on their screen (i.e., with their plan adjacent to their IDE); in these cases, the durations during which they held their plans on their screens were recorded in seconds. For all cases in which both the plan and the IDE were on the screen, students did so for both assignments.

V. RESULTS

A. Qualitative

Through our thematic analysis, we found themes relating to scaffolding (via a palette of blocks, as in Jigsaw), testing and executability, and visual interface.

a) Scaffolding: Jigsaw was designed to scaffold learning decomposition, and seven students discussed the tool's ability to do so. P2 mentioned Jigsaw would be helpful for new programmers, as the tool "*kind of gives them the basic blocks [...] even if they don't understand why the blocks are there.*" P4 explicitly noticed the relationship between Jigsaw's pattern blocks and the patterns presented at the beginning of the

evaluation session, stating that the tool encouraged them to consider "*Which of those [patterns] I need to implement?*"

Jigsaw's scaffolding may reduce planning effort, which 11 students noted. P2 stated that the tool "*[D]id help [me] figure out how I needed to plan it so it wasn't all on me.*" P6 said that "*[Jigsaw] could be more helpful because it separates the planning and actual writing of the code.*" P3, with regard to the possibility of using the tool in the future, expressed "*I think I would probably use it because it's easier than typing my thoughts. So because you have something to play around with, it's easy. It doesn't take up much time planning.*" A common sentiment was the simplicity of being able to use the already created blocks; P15 shared that the tool "*[D]efinitely helped, being able to have something quick and easy that was literally just drag and drop.*" This reduction of planning effort may also have reduced so-called blank page syndrome; as P17 stated, "*It's sort of intimidating to have just like a blank sheet in front of you, and you have to be like, 'Oh, should I draw? Should I write when I do this?' Versus [with] the tool is like, 'Okay, I just got to start dragging stuff in and figure out how to link them.*"'

Two students, P4 and P7, expressed that they felt restricted or limited by Jigsaw. Both identified in their interviews as more advanced programmers and felt that the tool was too limited in scope to be of use in their planning processes. These students believed that the extra scaffolding restricted their freedom to guide their own approach. For example, P4 noted that "*[...] I feel that this type of planning [using blank document] gives me a little bit more freedom, um, just to like do things a little different like, you know, whereas here [with Jigsaw], I only have certain options to do [...].*" P7 explicitly connected this to the lack of blocks, however, which suggests that a less project-specific design may mitigate these concerns.

Another drawback that at least one student noted is that Jigsaw's scaffolding could present too much support. This was explicitly noted by P17: "*I think if you were first starting, [Jigsaw] would be really helpful, but then I think it would probably be good practice to not use it all the time. I think it could end up being like a crutch – like, I could see myself using this as a crutch.*" This is a trade-off inherent in scaffolding: it is helpful for the less experienced and less for the more experienced.

b) Planning-implementation relationship: Several students suggested that Jigsaw allowed for improved implementation. This was predominantly because of its structuring of ideas. P10 stated, "*Because it helped me see what the outputs should have been [...] I was a little bit more prepared for, like, you know, I could run through it a little bit easier.*" In another example, P17 said that "*[Jigsaw] made it more simple for me to just go through and be like, okay. I did this. I'm done with this.*" In P8's case, they expressed a belief that the tool would have helped them catch a significant error that impeded their progress overall, thus causing the overall process to have been faster.

c) Testing and Executability: Some students enjoyed using the provided input blocks to check their work and see in real time how changes to the input affected the output. P13 enjoyed how "*[Y]ou could edit the inputs and then see everything else change.*" P10 explicitly mentioned the tool as being useful for testing: "*I think it would be really helpful for test functions like that – that helped a lot, so I didn't have to run it a bunch of times.*" P11 referred to troubleshooting, stating "*I was able to use it to see if my program was right or wrong, which is really useful.*"

For P12, the program provided two main benefits regarding testing. One was that using the test cases allowed them to see, "*Oh, this test case doesn't work, and this one does.*" The other was that the test cases actually helped them clarify their understanding of the problem. In this case, the interactive input/output directly helped improve their mental model for the problem.

As P8 mentioned, they believed that having Jigsaw would have improved their bug-catching ability. "*If I had to write another program similar to that, I would plan, and I would probably use the flowchart tool first. If for no other reason than it would be easier to look back at and understand literally follow my logic to what I was trying to do earlier, and then maybe understand why I went wrong.*"

d) Visual interface: We found, as prior flowchart tools have, that visualizing plans is helpful for students. 13 students noted the tool's capability to improve visualization, commonly referring to the tool's ability to present a physical representation of the decomposed parts of the program. As P6 stated, "*I feel like this would be more useful than just writing it on paper, because it visually separates everything, which makes it super easy and quick to reference back to, especially when you're using a single screen like I am right now, and you're switching between tabs.*" P15 mentioned that visualization improved organization: "*The second time using the tool was definitely more organized, with an organized plan [...] Being able to look back, see what things were supposed to be in order.*"

Additionally, we found an easy-to-use UI is important for overcoming learning curves. Some students like P5 felt that the tool's interface, while helpful, took some time to learn: "*There's a bit of a -– I want to say, a learning curve. It is not hard to use, but I guess there's a little bit of a learning curve just to figure out.*" Other students felt the program was fairly intuitive, particularly given the colors and highlights embedded within the program. P13 noted that "*[...] it took a while to get used to [Jigsaw]. But then, once you realize the colors are actually helpful, and you have to use them to connect the objects, it helps get the flow of what you need to do.*"

e) Plans' impact on programming: Students engaged with their Jigsaw-made plans differently than with their personally designed plans. P11 stated that, "*As far as I remember, from the one where I had [Jigsaw], it was very smooth. It was like I did it step by step versus the one where I didn't have it.*" P10 stated that "*It definitely helped me figure out what order everything needed to be in, and how I needed to think of things, so it made that process a lot faster.*" P13 shared, "*Without the tool, it's definitely more like a high level of like,*

'Okay, what do I need to have in my code to make it function?' [...] And then with the tool, it's more interactive [...] I think the visualization of how the data flows through the program is also kind of helpful." P10 expressed that the scaffolding was highly beneficial, and touched on the decompositional scaffolding of Jigsaw: "*Honestly, when we have to write that out ourselves, I would rather just do the code first, and then put things in afterwards, because I really am not sure at all, versus [Jigsaw] gave a lot more of a better skeleton without really telling me what I was supposed to do. It got me in the mindset of what I thought I should do. I was already thinking, 'Okay, I'm gonna sum this and count this. [...] I would rather do that flowchart before I code, versus trying to make my own little plan where I'm not even sure what the code looks like yet.*" P1 shared that Jigsaw streamlined programming because it encouraged a better understanding of the programming prompt than planning alone: "*I think the plan with [Jigsaw] made things more fleshed out. So when I got to writing the program, there [weren't] more steps to think through when I was writing the program. And that made it easier. When I was planning with just a blank document [...] I sort of had an idea of where to go, but it was definitely more difficult.*"

B. Quantitative

In Table V-B, we present the average planning and programming times for all four conditions, as there were two assignments (Rainfall and Water/Bottle) and two scaffolding options (with or without Jigsaw). 13 students engaged with Jigsaw equivalently to or more than with their non-scaffolded plans. Additionally, students who planned on their own for Rainfall were more likely to engage with their link to the recipes slide deck they had been presented earlier than those who planned on their own for Water/Bottle. As mentioned earlier, all students had access to the link in their provided documents for planning. Six in eight students who manually planned for Rainfall referred back to the recipes slides, as opposed to two in nine students who manually planned for Water/Bottle. This was the primary difference in planning between these groups, as approximately equal proportions of students chose to continue planning regardless of the plan's scaffolding.

VI. Discussion

We found that Jigsaw was generally effective at fulfilling our design goals for three reasons. First, its design centered purpose-first and clear language, which students noted and appreciated. Second, Jigsaw supported recognition over recall, as indicated in part by students' preference to engage with Jigsaw over a slide deck presenting the same recipes. Third, Jigsaw made plans testable, allowing students to debug and more thoroughly understand the task at hand as they worked.

A. Takeaways

Here, we summarize what we have learned during the process of tool design and evaluation. We note that these takeaways are *our* drawn conclusions from having run the study and are primarily derived from our participants' feedback.

a) ***Intuitive design****:* Jigsaw was designed to be easy to use, with an intuitive visual interface for students and simple, understandable language and descriptions for patterns, and executable plans. Our participants stated that they did find the tool to be easy to use. This design facilitated quick adoption of the tool and allowed students to focus on planning rather than learning the nuances of a new interface. It also enabled students to ensure their plans functioned desirably with different inputs before diving into implementation.

b) ***Student understanding****:* Jigsaw helped students reason about their understanding of the program and debug it. They could see where their expectations diverged from the actual plan execution. In some cases, Jigsaw streamlined their programming, as they began programming with a firmer understanding than they otherwise would have.

c) ***Planning****:* Students largely preferred not to plan and generally noted that they wouldn't have if they had not been required to do so. When left to plan themselves, students largely wrote pseudocode, which corresponded closely with program code – essentially coding on paper rather than in an IDE. This allowed those students to sometimes implement their programs faster, as they had effectively written out a first draft. By contrast, Jigsaw takes additional time to plan, and the result of planning with Jigsaw is a flowchart that is new to students and may be challenging to translate into a working implementation on the first try. However, this increased cognitive effort may translate to more engagement with planning process, the benefits of which we discussed earlier.

B. Limitations

This study has several limitations. First, we had a limited data sample for evaluating a prototype of the tool. Only 17 students' data was analyzed, resulting in 36 programs available for analysis. When analyzing assignments and the condition of planning (e.g., Rainfall with Jigsaw), we did not have sufficient samples to make definitive conclusions regarding the quantitative effectiveness of the tool on impacting programming outcomes. We conducted a qualitative analysis of students' interviews to address this limitation, but a larger sample size is needed to corroborate our findings. Also, we did receive feedback on improving the tool and hope that this may improve its future use.

Second, social desirability bias[2] might have impacted our results. More specifically, some students could have provided positive feedback on Jigsaw because of their assumptions that researchers were interested in the success of the tool. To mitigate this effect, future research evaluating this tool may ask for students to provide feedback via a more impersonal

[2]Social desirability bias is "the tendency to present oneself and one's social context in a way that is perceived to be socially acceptable, but not wholly reflective of one's reality. In research, the bias denotes a mismatch between participants' genuine construction of reality and the presentation of that reality to researchers." [54]

TABLE II
THE AVERAGE TIME (IN SECONDS) AND GRADE (GRADED FROM A RUBRIC CONSISTING OF FIVE POINTS) FOR EACH OF THE FOUR CONDITIONS.

Assignment	Average planning time (s)	Average programming time (s)	Average grade (out of 5)
Rainfall with Jigsaw (n=9)	344	670	4.78
Rainfall without Jigsaw (n=8)	332	909	4.625
Water/Bottle with Jigsaw (n=8)	328	1438	4
Water/Bottle without Jigsaw (n=9)	326	918	4.72

survey instead of engaging directly with one of the researchers via an interview.

Third, Jigsaw was designed to be used in a formal, introductory learning context with an emphasis on programming patterns, as discussed in Section III-A. Consequently, this within-users study, while providing insights into how the tool can be used and may benefit students, does not accurately capture the context in which we would like this tool to be used. This study design also prevented us from understanding to what extent Jigsaw may have impacted the transfer of knowledge to other problems or situations. Additionally, due to the nature of the tool, scaling problems can provide issues in terms of both creating more complex plans as well as taking up more physical space; consequently, we recommend the tool's use particularly for novice programmers and introductory programming environments.

Fourth, students' use of the tool was occasionally limited by technical issues. Although all students were able to use and access the tool successfully, sometimes troubleshooting extended the length of the study and affected students' interactions and satisfaction with the tool. Jigsaw cannot be used on all browsers (e.g., Safari), and an overly sensitive zoom feature discouraged some students from engaging with the tool further. We intend to improve these components of the tool in future work.

Fifth, we did not conduct pre- and post-assessments with the students and consequently could not assess specific learning outcomes. Since we built Jigsaw to be a scaffolding tool, we hypothesize that our qualitative results in Section V-A bridge our theory to our execution. However, further studies will be needed to understand the specific impacts of the theory of scaffolding on our implemented practices.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we presented and detailed the creation of Jigsaw, a tool designed to explicitly teach decompositional techniques for planning in programming. Because of the importance of planning and decomposition, we believe that the design of planning tools like Jigsaw, especially those that can be used in multiple contexts and programming languages, could improve the design and quality of the software that students create and help them develop transferable skills in decomposition and planning. These skills are critical for the computational thinking (CT) process, and consequently may be particularly valuable for more novice programmers.

Explicit planning, whether with a tool like Jigsaw or another approach, may or may not quantitatively improve programming outcomes like reducing programming time or improving programming correctness, particularly for simpler tasks. However, the open question seems to be whether explicit planning can create better long-term learning and encourage good habits when it comes to approaching more complex tasks; this would require further studies to explore. Our preliminary results suggest that a visual scaffolding tool like Jigsaw may encourage good programming habits.

The versions of Jigsaw given to the students in our evaluation were inherently problem-specific. However, we have modified Jigsaw to allow students to define their own custom pattern blocks. This enables Jigsaw to be used on a wider variety of problems and potentially scale to more complex problems.

Ultimately, our goal with Jigsaw is to incorporate an improved version among a wider group of undergraduate students within the context of an introductory programming class. Specifically, Jigsaw is intended to exist within a pedagogy in which decompositional patterns are explicitly taught and repeatedly practiced. Such an approach may help change student attitudes toward planning and its usefulness, and requiring its practice, even in small cases, may help students develop more skills and intentions to use planning and planning tools more regularly.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant 1917885. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] K. Fisler, S. Krishnamurthi, and J. Siegmund, "Modernizing plan-composition studies," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 2016, pp. 211–216.

[2] F. E. V. Castro, S. Krishnamurthi, and K. Fisler, "The impact of a single lecture on program plans in first-year CS," in *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*. Koli Finland: ACM, Nov. 2017, pp. 118–122. [Online]. Available: https://dl.acm.org/doi/10.1145/3141880.3141897

[3] F. E. V. Castro and K. Fisler, "Qualitative analyses of movements between task-level and code-level thinking of novice programmers," in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 2020, pp. 487–493.

[4] E. S. Wiese, M. Yen, A. Chen, L. A. Santos, and A. Fox, "Teaching students to recognize and implement good coding style," in *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*, 2017, pp. 41–50.

[5] K. Fisler and F. E. V. Castro, "Sometimes, rainfall accumulates: Talk-alouds with novice functional programmers," in *Proceedings of the 2017 acm conference on international computing education research*, 2017, pp. 12–20.

[6] E. Rivera, S. Krishnamurthi, and R. Goldstone, "Plan composition using higher-order functions," in *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, 2022, pp. 84–104.

[7] J. Tsan, D. Eatinger, A. Pugnali, D. Gonzalez-Maldonado, D. Franklin, and D. Weintrop, "Scaffolding Young Learners' Open-Ended Programming Projects with Planning Sheets," in *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, ser. ITiCSE '22. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 372–378. [Online]. Available: https://doi.org/10.1145/3502718.3524769

[8] R. Nata, *New Directions in Higher Education*. Nova Publishers, 2005.

[9] V. J. Shute, C. Sun, and J. Asbell-Clarke, "Demystifying computational thinking," *Educational research review*, vol. 22, pp. 142–158, 2017.

[10] C.-C. Yu and S. P. Robertson, "Plan-based representations of Pascal and Fortran code," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1988, pp. 251–256.

[11] M.-J. Tsai, J.-C. Liang, S. W.-Y. Lee, and C.-Y. Hsu, "Structural Validation for the Developmental Model of Computational Thinking," *Journal of Educational Computing Research*, vol. 60, no. 1, pp. 56–73, Mar. 2022, publisher: SAGE Publications Inc. [Online]. Available: https://doi.org/10.1177/07356331211017794

[12] A. Bogdanovych and T. Trescak, "Coding Style and Decomposition," in *Learning Java Programming in Clara's World*, A. Bogdanovych and T. Trescak, Eds. Cham: Springer International Publishing, 2021, pp. 83–100. [Online]. Available: https://doi.org/10.1007/978-3-030-75542-3_4

[13] C. Charitsis, C. Piech, and J. C. Mitchell, "Detecting the Reasons for Program Decomposition in CS1 and Evaluating Their Impact," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. Toronto ON Canada: ACM, Mar. 2023, pp. 1014–1020. [Online]. Available: https://dl.acm.org/doi/10.1145/3545945.3569763

[14] C. C. Selby, "Promoting computational thinking with programming," in *Proceedings of the 7th Workshop in Primary and Secondary Computing Education*. Hamburg Germany: ACM, Nov. 2012, pp. 74–77. [Online]. Available: https://dl.acm.org/doi/10.1145/2481449.2481466

[15] A. Keen and K. Mammen, "Program Decomposition and Complexity in CS1," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: Association for Computing Machinery, Feb. 2015, pp. 48–53. [Online]. Available: https://doi.org/10.1145/2676723.2677219

[16] P.-Y. Chao, "Exploring students' computational practice, design and performance of problem-solving through a visual programming environment," *Computers & Education*, vol. 95, pp. 202–215, Apr. 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360131516300161

[17] F. K. Bailie, "Improving the modularization ability of novice programmers," *ACM SIGCSE Bulletin*, vol. 23, no. 1, pp. 277–282, Mar. 1991. [Online]. Available: https://dl.acm.org/doi/10.1145/107005.107065

[18] Y.-T. Lin, M. K.-C. Yeh, and S.-R. Tan, "Teaching Programming by Revealing Thinking Process: Watching Experts' Live Coding Videos With Reflection Annotations," *IEEE Transactions on Education*, vol. 65, no. 4, pp. 617–627, Nov. 2022, conference Name: IEEE Transactions on Education.

[19] M. Hu, M. Winikoff, and S. Cranefield, "Teaching novice programming using goals and plans in a visual notation," in *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123*, ser. ACE '12. AUS: Australian Computer Society, Inc., Jan. 2012, pp. 43–52.

[20] K. Beck, R. Crocker, G. Meszaros, J. O. Coplien, L. Dominick, F. Paulisch, and J. Vlissides, "Industrial experience with design patterns," in *Proceedings of IEEE 18th International Conference on Software Engineering*. IEEE, 1996, pp. 103–114.

[21] M. E. Caspersen and J. Bennedsen, "Instructional design of a programming course: a learning theoretic approach," in *Proceedings of the third international workshop on Computing education research*, 2007, pp. 111–122.

[22] K. Cunningham, "The novice programmer needs a plan," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2018, pp. 269–270.

[23] L. Sterling, "Patterns for Prolog Programming," in *Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski Part I*, ser. Lecture Notes in Computer Science, A. C. Kakas and F. Sadri, Eds. Berlin, Heidelberg: Springer, 2002, pp. 374–401. [Online]. Available: https://doi.org/10.1007/3-540-45628-7_15

[24] J. P. East, S. R. Thomas, E. Wallingford, W. Beck, and J. Drake, "Pattern Based Programming Instruction," Jun. 1996, pp. 1.349.1–1.349.10, iSSN: 2153-5965. [Online]. Available: https://peer.asee.org/pattern-based-programming-instruction

[25] L. Seiter and B. Foreman, "Modeling the learning progressions of computational thinking of primary grade students," in *Proceedings of the ninth annual international ACM conference on International computing education research*, 2013, pp. 59–66.

[26] E. Soloway, "Learning to program = learning to construct mechanisms and explanations," *Communications of the ACM*, vol. 29, no. 9, pp. 850–858, Sep. 1986. [Online]. Available: https://doi.org/10.1145/6592.6594

[27] M. Mohaghegh and M. Mccauley, "Computational Thinking: The Skill Set of the 21st Century," *nternational Journal of Computer Science and Information Technologie*, vol. 7, pp. 1524–1530, Jun. 2016.

[28] L. A. Mesiti, A. Parkes, S. C. Paneto, and C. Cahill, "Building Capacity for Computational Thinking in Youth through Informal Education," *Journal of Museum Education*, vol. 44, no. 1, pp. 108–121, Jan. 2019. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/10598650.2018.1558656

[29] Y. Pechorina, K. Anderson, and P. Denny, "Metacodenition: Scaffolding the Problem-Solving Process for Novice Programmers," in *Proceedings of the 25th Australasian Computing Education Conference*, ser. ACE '23. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 59–68. [Online]. Available: https://doi.org/10.1145/3576123.3576130

[30] F. Corno, L. De Russis, and J. Pablo Sáenz, "TextCode: A Tool to Support Problem Solving Among Novice Programmers," in *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct. 2021, pp. 1–5, iSSN: 1943-6106.

[31] A. Milliken, W. Wang, V. Cateté, S. Martin, N. Gomes, Y. Dong, R. Harred, A. Isvik, T. Barnes, T. W. Price, and C. Martens, "PlanIT! A New Integrated Tool to Help Novices Design for Open-ended Projects," in *SIGCSE '21: The 52nd ACM Technical Symposium on Computer Science Education, Virtual Event, USA, March 13-20, 2021*, M. Sherriff, L. D. Merkle, P. A. Cutter, A. E. Monge, and J. Sheard, Eds. ACM, 2021, pp. 232–238.

[32] J. Wrenn and S. Krishnamurthi, "Executable Examples for Programming Problem Comprehension," in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ser. ICER '19. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 131–139. [Online]. Available: https://dl.acm.org/doi/10.1145/3291279.3339416

[33] D. Hooshyar, R. B. Ahmad, R. G. Raj, M. H. N. M. Nasir, M. Yousef, S.-J. Horng, and J. Rugelj, "A flowchart-based multi-agent system for assisting novice programmers with problem solving activities," *Malaysian Journal of Computer Science*, vol. 28, no. 2, pp. 132–151, 2015.

[34] D. Hooshyar, R. B. Ahmad, M. Yousefi, F. D. Yusop, and S.-J. Horng, "A flowchart-based intelligent tutoring system for improving problem-solving skills of novice programmers," *Journal of computer assisted learning*, vol. 31, no. 4, pp. 345–361, 2015.

[35] D. Hooshyar, R. B. Ahmad, M. Yousefi, M. Fathi, S.-J. Horng, and H. Lim, "Sits: A solution-based intelligent tutoring system for students' acquisition of problem-solving skills in computer programming," *Innovations in Education and Teaching International*, vol. 55, no. 3, pp. 325–335, 2018.

[36] K. Cunningham, B. J. Ericson, R. Agrawal Bejarano, and M. Guzdial, "Avoiding the Turing tarpit: Learning conversational programming by starting from code's purpose," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–15.

[37] C.-C. Tseng, P.-Y. Chao, and K. R. Lai, "An Analysis of Goal Orientation Pattern and Self-Efficacy for Explanation of Programming Plans," in *2015 IEEE 15th International Conference on Advanced Learning Technologies*, Jul. 2015, pp. 76–77, iSSN: 2161-377X.

[38] W. Jin, A. Corbett, W. Lloyd, L. Baumstark, and C. Rolka, "Evaluation of Guided-Planning and Assisted-Coding with Task Relevant Dynamic Hinting," in *Intelligent Tutoring Systems*, ser. Lecture Notes in Computer Science, S. Trausan-Matu, K. E. Boyer, M. Crosby, and K. Panourgia, Eds. Cham: Springer International Publishing, 2014, pp. 318–328.

[39] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to design programs: an introduction to programming and computing*. MIT Press, 2018.

[40] O. Muller, D. Ginat, and B. Haberman, "Pattern-oriented instruction and its influence on problem decomposition and solution construction," in *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, 2007, pp. 151–155.
[41] E. Soloway, "Learning to program= learning to construct mechanisms and explanations," *Communications of the ACM*, vol. 29, no. 9, pp. 850–858, 1986.
[42] J. Prather, R. Pettit, B. A. Becker, P. Denny, D. Loksa, A. Peters, Z. Albrecht, and K. Masci, "First things first: Providing metacognitive scaffolding for interpreting problem prompts," in *Proceedings of the 50th ACM technical symposium on computer science education*, 2019, pp. 531–537.
[43] J. Kodosky, "Labview," *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 1–54, 2020.
[44] W. D. Hoyer and S. P. Brown, "Recognition over recall," *Journal of Consumer Research*, vol. 17, pp. 141–148, 1990.
[45] W. Lidwell, K. Holden, and J. Butler, *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub, 2010.
[46] K. J. Harms, J. Chen, and C. L. Kelleher, "Distractors in parsons problems decrease learning efficiency for young novice programmers," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, 2016, pp. 241–250.
[47] D. H. Smith IV and C. Zilles, "Discovering, autogenerating, and evaluating distractors for python parsons problems in cs1," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 924–930.
[48] P. Denny, J. Prather, B. A. Becker, Z. Albrecht, D. Loksa, and R. Pettit, "A closer look at metacognitive scaffolding: Solving test cases before programming," in *Proceedings of the 19th Koli Calling international conference on computing education research*, 2019, pp. 1–10.
[49] O. Seppälä, P. Ihantola, E. Isohanni, J. Sorva, and A. Vihavainen, "Do we know how difficult the rainfall problem is?" in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, 2015, pp. 87–96.
[50] K. Fisler, "The recurring rainfall problem," in *Proceedings of the tenth annual conference on International computing education research*, 2014, pp. 35–42.
[51] C. Board, "Ap® computer science a free-response questions," 2019.
[52] M. Maguire and B. Delahunt, "Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars." *All Ireland Journal of Higher Education*, vol. 9, no. 3, 2017.
[53] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.
[54] N. Bergen and R. Labonté, ""everything is perfect, and we have no problems": detecting and limiting social desirability bias in qualitative research," *Qualitative health research*, vol. 30, no. 5, pp. 783–792, 2020.