Computer Science Education

ISSN: 0899-3408 (Print) 1744-5175 (Online) Journal homepage: www.tandfonline.com/journals/ncse20

Unraveling novices' code composition difficulties

Renske Weeda, Sjaak Smetsers & Erik Barendsen

To cite this article: Renske Weeda, Sjaak Smetsers & Erik Barendsen (2024) Unraveling novices' code composition difficulties, Computer Science Education, 34:3, 414-441, DOI: 10.1080/08993408.2023.2169067

To link to this article: https://doi.org/10.1080/08993408.2023.2169067

Published online: 24 Jan 2023.

COMPUTER SCIENCE EDUCATION
2024, VOL. 34, NO. 3, 414–441
https://doi.org/10.1080/08993408.2023.2169067

Routledge
Taylor & Francis Group

ARTICLE

OPEN ACCESS Check for updates

Unraveling novices' code composition difficulties

Renske Weeda[a], Sjaak Smetsers[b] and Erik Barendsen[a,c]

[a]Institute for Science Education Radboud University, Nijmegen, The Netherlands; [b]Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands; [c]Department of Computer Science, Open University, The Netherlands

ABSTRACT

Background and Context: Multiple studies report that experienced instructors lack consensus on the difficulty of programming tasks for novices. However, adequately gauging task difficulty is needed for alignment: to select and structure tasks in order to assess what students can and cannot do.

Objective: The aim of this study was to examine perceived and observed difficulties that novice programmers endure during code composition.

Method: This study reports on the in-depth research combining multi-faceted data from observations to students' perceptions via interviews, think aloud, task ranking and task rating.

Findings: Some interesting findings were observed which would not have been revealed through solution analysis on its own. For example, that nesting two loops is perceived as easier than nesting a selection in a loop and that the abstraction mechanism of an enhanced for loop makes it a particularly challenging concept to initially grasp.

Implications: Our results augment related work, which are primarily based on instructors' perceptions, providing insights into difficulties endured by novice programmers and recommendations for task (de)composition and curricular alignment.

ARTICLE HISTORY
Received 2 September 2022
Accepted 12 January 2023

KEYWORDS
Programming assessment; code composition; difficulty; novice; conceptual knowledge; strategic knowledge

1. Introduction

'*. . . programming educators may be systemically underestimating the cognitive difficulty in their instruments for assessing programming skills of novice programmers*' Whalley et al. (2006).

The assessment of Computer Science students' programming knowledge and skills is a problematic endeavour (Robins et al., 2003) for which educators lack valid and reliable assessment instruments (Tew, 2010) and clear frameworks and tools (Whalley et al., 2006). In particular, assessment alignment, rather than the subject itself, may be responsible for poor results (Luxton-Reilly et al., 2017). Assessment alignment requires understanding the complexity and difficulty of tasks. To this end, McCracken et al. (2001) call for data analysis using qualitative approaches to refine current assessment tools.

CONTACT Renske Weeda renske.weeda@ru.nl Institute for Science Education, Radboud University, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

One issue in assessment alignment is gaps between instructors' estimations of complexity and novices' actual difficulties (Shuhidan et al., 2009). To illustrate this, Sheard et al. (2011) report a mere 43% consensus between experienced instructors on whether questions were easy, moderate or difficult. These results are in line with earlier researchers such as Petersen et al. (2011). Nathan and Petrosino (2003) note that this gap may be attributed to an 'expert blind spot' effecting educators' perceptions of task difficulty. Moreover, classification of difficulty is subjective and highly dependent on expected student capabilities (Sheard et al., 2011). Although several researches report on the complexity of code comprehension tasks (e.g. Whalley et al. (2006)), there is little consensus about indicators for task difficulty of code composition tasks (Lopez et al., 2009).

Qian and Lehman (2017) classify programming concepts and skills into three categories of knowledge, namely syntactical, conceptual and strategic. Of these, **conceptual knowledge** (Robins et al., 2003) and **strategic knowledge** (Ginat et al., 2013; Spohrer et al., 1985) have been reported to be the most difficult to master. Individual concepts require full internalization before students can reason about their interactions (Petersen et al., 2011) or deal with more abstract concepts and larger composite tasks (Teague & Lister, 2014b). When mastering new concepts, CS students generally go through different stages of development. While some students may demonstrate high performance on typical tasks that integrate multiple concepts and skills (Luxton-Reilly & Petersen, 2017), requiring both conceptual and strategic knowledge, others struggle to even commence. As a result of a deficient attempt, instructors are provided with no evidence whatsoever of the student's capabilities or which difficulties they endure. To this end, small code writing tasks that focus on single concepts help disentangle conceptual and strategic sources of difficulty (Luxton-Reilly & Petersen, 2017) and are furthermore more accurate predictors of performance (Zingaro et al., 2012). It makes sense to decompose and order tasks in terms of difficulty in order to help identify which aspect of a programming task a particular student struggles with. Along this line, to assess knowledge of individual atomic concepts, Luxton-Reilly et al. (2017) proposed a set of decomposed CS1 benchmark tasks. Although these tasks give insight into the mastery of single concepts, CS1 students are also expected to plan, develop and debug novel programs (Du Boulay, 1986). Therefore, fundamental to learning to program, students learn plans or stereotypical solutions (Soloway, 1986) and how to combine these into larger programs (De Raadt et al., 2009) which should be assessed accordingly. However, difficulties pertaining to individual constructs and their relationship to program difficulty are not yet fully understood (Lopez et al., 2009). This is problematic because, in order to classify and adapt tasks, instructors need to understand what makes one multi-conceptual programming task more difficult than another and how these characteristics influence each other (Kalyuga, 2011). Concluding, as both conceptual and strategic knowledge play an important role in programming, the perceived difficulties of both should be considered during assessment alignment.

A possible approach to predict difficulty could be to consider the program's elementary structures (Luxton-Reilly et al., 2017) or compositions of plan schemata (Duran et al., 2018). To our knowledge, these models have not been empirically validated. From another perspective, error analysis of students' solutions can provide insight into particular difficulties endured by students. However, on its own, these data do not provide insight into the root-causes of errors or omissions (Spohrer et al., 1985), the cognitive

processes that drive task solving process (Teague & Lister, 2014a), nor which specific aspects contribute to difficulty (Morrison et al., 2014).

To this end, we pursue a qualitative in-depth approach based on empirical evidence from a wide range of data sources, including solution analysis, think-aloud sessions, interviews and task rating and ranking. The goal is to gain a deeper understanding about which aspects that contribute to novices' difficulty during code composition and why and thus answer the following research questions:

RQ1: From a student's perspective, which aspects contribute to the difficulty of programming tasks through the lens of **conceptual knowledge**, and why?

RQ2: From a student's perspective, which aspects contribute to the difficulty of programming tasks through the lens of **strategic knowledge**, and why?

The rest of this paper is organized as follows: Section 2 presents a literature review pertaining to task difficulty and the elicitation thereof. Subsequently, in Section 3, we describe how we identified difficulties. In Section 4 we present our results and then discuss these in light of our research questions in Section 5, followed by a conclusion and description of future work in Section 6.

2. Background

2.1. Task difficulty

Aligning assessment of learner capabilities requires understanding the complexity and difficulty of tasks. Following an extensive review, Liu and Li (2012) present complexity as an objective measure concerning the intrinsic properties of a task (such as structure, components and their relations), whereas difficulty is subjective and directly related to learners' cognitive development (Sweller, 2010) and prior knowledge. The focus of our research pertains primarily to task difficulty.

Based on the Neo-Piagetian perspective, the Structure of the Observed Learning Outcome (SOLO) model (Biggs & Collis, 1982) has been used to describe a student's cognitive development based on their solutions to code comprehension (Whalley et al., 2006) or code composition (Ginat & Menashe, 2015) tasks. SOLO distinguishes five achieved levels of abstraction: prestructural, unistructural, multistructural, relational and extended-abstract. In light of conceptual and strategic knowledge, Ginat and Menashe (2015) used SOLO to assess algorithmic design, placing straightforward implementations of plans (and thus conceptual knowledge) on lower levels of the hierarchy while combining and tailoring plans (and thus strategic knowledge) on higher levels.

We now discuss the conceptual and strategic knowledge and their relationship to the program difficulty in more detail.

Conceptual knowledge pertains to programming constructs and basic plans (De Raadt et al., 2009). The analysis of student errors (Du Boulay, 1986) and misconceptions (Sirkiä & Sorva, 2012) has shed light on difficulties pertaining to conceptual knowledge. Novice programmers may suffer from fragile knowledge that is incomplete, hard to

retrieve and often misused (Qian & Lehman, 2017). Students may misunderstand the mechanism of code execution or lack coherent mental models of how particular construct work (Milne & Rowe, 2002). During learning, in order to try to understand how programs are run, students may make a (simplified) model of a computer, described by (Du Boulay, 1986) as the 'notional machine' (Duran et al., 2018). Consequently, some constructs may be perceived more difficult than others. Schulte and Bennedsen (2006) surveyed 243 educators and found program design and control structures to be the hardest topics. Some control structures are harder to trace (Lopez et al., 2008) or write than others. For example, selection statements with compound conditionals or multiple branches are harder to write (Ebrahimi, 1994), and loop structures are perceived as slightly more difficult than selection structures (Lahtinen et al., 2005). As expertise develops, programmers incorporate code interactions into schemas, also known as plans or design patterns. Such plans are described by Soloway (1986) as stereotypical solutions to problems and are essential for coping with complex problems (Muller et al., 2007). First, when students fail to recognize and select similar plans, they often endure difficulties on task initiation (Muller et al., 2007). Second, they allow a person to hold more information in working memory (Morrison et al., 2014), thus lowering the cognitive load (Ginat & Menashe, 2015). Such a higher-level abstraction skill, to '*see the forest and not just the trees*", corresponds to a higher SOLO level Lister et al. (2009). Contrastingly, novices who lack plan knowledge use much of their working memory to process the information required to complete the task (Luxton-Reilly et al., 2017), increasing the task's difficulty.

On a higher cognitive level, **strategic knowledge** pertains to problem-solving strategies. This involves selecting, combining and tailoring constructs and plans to solve the task at hand (Ginat & Menashe, 2015). As such, plans are used as basic building blocks, integrated by means of abutment (concatenation), merging (interleaving) or nesting and then tailoring to create (novel) solutions (Soloway, 1986). A recent study by Costantini et al. (2020) verified results from previous studies that selecting (De Raadt et al., 2009), combining (Fisler, 2014; Ginat et al., 2013) and tailoring (Cant et al., 1995) plans are among the most difficult skills for novices to master. To facilitate plan recognition, Sajaniemi (2002) expresses that recognizing variable roles and role changes is a valuable skill for program comprehension. This view, which explicitly links the variables to their role in plan-goal achievement, may also facilitate plan composition and tailoring and thus builds on strategic knowledge. They found a comprehensive set of nine generic variable roles that describe almost all variables appearing in generic plans and in novice-level programming in general. Understanding variable roles could also aid students during code composition, facilitating construction and plan selection and implementation.

2.2. Cognitive load

Cognitive load can be explained as the load on an individual's working memory by a particular task (Paas, 1992), and overload should be avoided to maximize the effectiveness of learning. According to Cognitive Load Theory, three sources can impose cognitive load: intrinsic (IL), extraneous (EL) and germane (GL) (Sweller, 2010). IL is defined as a combination of the innate difficulty of the material and the learner's knowledge, and EL is the (undesirable) load placed on working memory due to aspects, such as unclear or lengthy instructions while, on the other hand, GL is related to processes such as schema formation

which are beneficial for learning (Leppink et al., 2013). From a cognitive load perspective, a task is difficult because the number of elements has to be processed simultaneously (Sweller, 2010). Along these lines, Çakiroğlu and Bilgi (2022)'s results suggest that element interactivity may be used as a proxy for cognitive load. From another perspective, Ginat and Menashe (2015) attribute low-level unistructural errors in programming to limitations in working memory, rather than a lack of knowledge. Taking this further, Izu et al. (2016) distinguish between persistent errors, from misconceptions, and inconsistent errors caused by cognitive load.

2.3. Determining difficulty

Several researches report on eliciting difficulties, from analyzing errors in programming tasks (Ebrahimi, 1994; Sirkiä & Sorva, 2012) or examinations (Lopez et al., 2008), effort to solve errors Çakiroğlu and Bilgi (2022), using concept classification or inventories (Luxton-Reilly et al., 2017; Petersen et al., 2011; Sheard et al., 2011), eye-tracking (Kather et al., 2021), counting 'help-calls' (Robins et al., 2003), surveys among practitioners (Campbell, 2018), academics (D'Souza et al., 2012; Petersen et al., 2011; Sheard et al., 2011; Shuhidan et al., 2009), novices (Ihantola et al., 2014; Milne & Rowe, 2002; Morrison et al., 2014), both academics and novices (Lahtinen et al., 2005) and think-aloud sessions or interviews (Ebrahimi, 1994; Teague & Lister, 2014b).

Mirroring the expectations of programming courses, an interesting perspective by Luxton-Reilly et al. (2017), is to consider the effect on task complexity when concepts are combined into (progressively) more complex tasks. The hierarchy is based on pre-requisite knowledge, e.g. understanding conditionals is precursory to loops, inherently making the latter more complex. Recognizing that tasks consist of multiple concepts is relevant to difficulty as it contributes to cognitive load (Petersen et al., 2011). Additionally, the manner in which structures are combined contributes to complexity, such as nesting (Shuhidan et al., 2009) and plan merging and tailoring (Kather et al., 2021). Along this line, Duran et al.'s theoretic models seem promising as they draw on Cognitive Load Theory combined with Soloway's plans to predict the difficulty and complexity of a program. They propose two metrics: maximum plan depth (MPD) and maximum plan interaction (MPI). MPD reflects the overall complexity of the elementary plans out of which a program is built up of. From a cognitive load perspective, MPI considers the complexity of interactions between plans (element interactivity) that arise from program composition, estimating how many plans must simultaneously be held in working memory in order to solve a task. Both Luxton-Reilly et al.'s and Duran et al.'s work may be enriched by considering the (sources of) difficulties of the individual concepts and plans used.

Focusing on the specific types and sources of errors made can also provide insight into difficulties. As such, Ginat and Menashe (2015) categorized errors as either minor local errors or plan-composition-based errors, and Weeda et al. (2020) decomposed tasks into primary and secondary goals to describe the grievousness of errors on an abstract plan level. To illustrate, an incorrect plan selection or composition is affiliated to strategic knowledge and reflects a lower cognitive level. On the other hand, a correct plan selection (reflecting a high cognitive level) which is then implemented incorrectly (e.g. an incorrect boundary) may be caused by either fragile construct knowledge or cognitive load.

However, rather than considering instructors' perceptions, which may be subject to the 'expert blind spot' (Nathan & Petrosino, 2003), gauging students' mental effort could be used to proxy students' difficulties. Subjective mental effort measures are quick and reliable (Sweller, 2010) and have been employed in CS with positive results. For example, Morrison et al. (2014) used validated surveys to approximate difficulty, and Ihantola et al. (2014) asked students to indicate the perceived difficulty of programming tasks on a Likert scale. However, Lahtinen et al. (2005) warn that students' self-perceptions may be skewed and Morrison et al. (2014) express that survey data alone cannot clarify which specific aspects contribute to increased difficulty. On the bright side, interviews and think-aloud sessions have the potential of providing rich data (Ericsson & Simon, 1980) yielding insights into cognitive processes, thoughts, strategies, misconceptions and furthermore help remove interpretation about the rationale behind participant's behavior (Spohrer et al., 1985). To affirm, Kather et al., 2021 attribute their insightful results to having complemented their eye-tracking data with think-aloud and rating tasks. Along the same line, Ebrahimi (1994) and Ginat and Menashe (2015) augmented their error analysis with interviews to clarify the causes of errors. Likewise, Çakiroğlu and Bilgi (2022) analyzed students' efforts to solve errors using both think-aloud sessions and screen recordings and combined these findings with teachers' predictions of difficulty level. We note, however, that difficulty does not invariably lead to errors, as such, justifying an approach which additionally considers other difficulty-related aspects.

2.4. Task design: code comprehension and composition

As students tend to avoid using unfamiliar constructs, Ebrahimi (1994) advocates the use of comprehension, rather than composition tasks, to assess individual construct knowledge. Code comprehension tasks have also been used to assess knowledge of elementary plans, such as *count*, *sum*, and a *one-way Boolean* plan (De Raadt et al., 2009). On the other hand, strategic knowledge (such as the ability to select, combine, and tailor constructs and plans) can be assessed using modifications of standard plans in code composition tasks. More specifically, Fisler et al. (2016) note that students' preferences for relational versus multistructural solutions reflect their strategic knowledge regarding efficiency. Thus, the integration of several basic plans using diverse approaches (i.e. abutment or merging) is an aspect to consider during task design.

2.5. Summary

Summarizing, in CS, several methods have been employed to gain insight into what makes a programming task difficult for novices. Where objective complexity measures may provide insight into difficulty, they do not consider student's cognitive development or prior knowledge. On the other hand, individual subjective measures, such as surveying instructors or students, may provide skewed data. In this light, a qualitative approach using data from multiple sources, including solution analysis, interviews, observations and student perceptions, could provide a more profound insight into program code characteristics, which effect difficulty and why and how they do so. To our knowledge, there are no studies using such a wide range of empirical data sources to uncover novices'

difficulties considering the conceptual and strategic knowledge needed for code composition tasks.

Based on the literature review discussed above, we group indicators related to difficulty into the following four categories, pertaining to:

(1) **Conceptual knowledge**: constructs and basic plans.
(2) **Strategic knowledge**: the design of a (novel) solution, such as recognizing (variable roles in) plans and combining and tailoring plans.
(3) **Code complexity**: a program's intrinsic attributes that can be measured objectively, such as syntactic structure, program size, control flow and decision structures.
(4) **Cognitive load**: the cognitive load imposed on working memory by a particular task.

3. Method

3.1. Participants

The study targeted CS1 students from Computer Science or Artificial Intelligence in their first year of university. All students had completed a CS0 course on imperative programming and were currently taking a JAVA course on Object-Oriented programming. We were familiar with the course contents of both courses that gave insight into the previous knowledge we could expect from the students. For this research 14 students, 2 females and 12 males, were recruited on a voluntary basis. Students are referred to by shorthand notation as follows: S<number>. They were instructed on the nature of the research, completed a consent form and received a gift voucher as an incentive.

3.2. Tasks

For this study, we developed a set of comprehension (i.e. code reading) and composition (i.e. code writing) tasks. Tasks are referred to by shorthand notation as follows: RT-<task_name> and WT-<task_name> for comprehension and composition tasks, respectively.[1] We will now discuss the rationale for each task in detail.

3.2.1. Code comprehension tasks

The goal of the code comprehension tasks was a three-fold. Firstly, to confirm that students are familiar with the individual concepts and elementary plans needed for the code composition tasks. Also, the tasks are meant to reduce the 'cold start effect' (Gibson, 1997) by getting the students acquainted with the context and feel more comfortable thinking aloud. In particular, three 'Explain in Plain English' (EiPE) tasks were used:

- Minimum [*RT-ForEach*]: Determine the smallest value in an array of integers and print it, implemented with a **for each** loop.
- Maximum [*RT-While*]: Determine the largest value in an array of integers and print it, implemented with a **while**.

- Count [*RT-For*]: Determine the frequency of the value 0 in an array of integers, using a **for** loop.

3.2.2. Code composition tasks

With regard to the composition tasks, we specifically searched for tasks with characteristic features which may effect perceived difficulty:

- Frequency of maximum [*WT-MaxFreq*]: Given an array of integers, each representing daily rainfall, print on how many days the maximum amount of rain fell. Also used by Ginat and Menashe (2015), this task requires abutting or merging three basic plans: maximum, count and output. This task allows us to identify misconceptions and to explore difficulties when combining plans without the need for tailoring.
- First and last rainy day [*WT-FirstLast*]: Given an array of integers, print the index of the first and the last rainy day. Inspired by Ginat and Menashe (2015), this task requires merging, abutting and tailoring a linear search plan to maintain an index (rather than the value). The problem has several solutions where it can be noted that the search for the last index is best done in the opposite direction (back to front). The previously determined first index can then serve as a bound for the iteration. This task provides us with the opportunity to investigate the effect of plan tailoring and augmented control flow on difficulty.
- Bar chart [*WT-BarChart*]: Given an array of integers, print a horizontal bar graph of asterisks, with each row showing daily rainfall. Inspired by J. Whalley et al. (2011), this task provides us with the opportunity to investigate the effect of nesting loops and tailoring a loop termination condition (i.e. to be dependent on the element value in the outer loop).
- Merging lists [*WT-Merge*]: Given two lists of rainfall values, create and print a list in which the values are merged, alternating from each list. Used by Izu et al. (2016), this task requires tailoring the list traversal plans in such a way that the two input lists are traversed simultaneously.
- Average [*WT-Average*]: Given an array of integers, if it can be computed, then print the average value of the non-negative values, otherwise print an error message. Originally described by Soloway (1986), this task requires abutting or merging count, sum and filter plans, then subsequently guarding (no division by zero), casting to a double and outputting the result. This task allows us to focus on the number of non-tailored plans and the manner in which they are combined.

4. Data collection

During both the code comprehension and the code composition tasks, students were asked to think aloud, followed by a semi-structured interview, task rating and task ranking; see Figure 1.

- *Tasks* For the code comprehension tasks, students were asked to summarize their understanding of the code by choosing a meaningful name for the main method. For

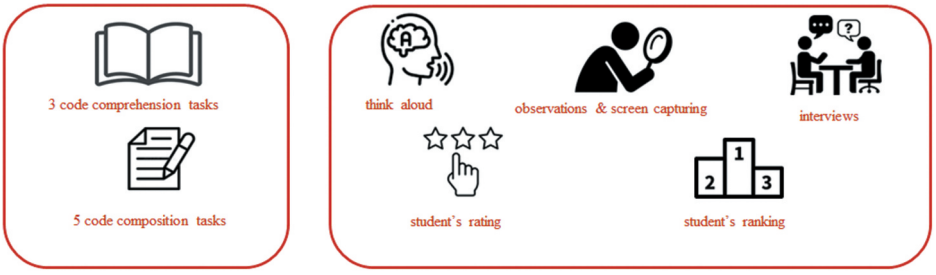

Figure 1. Data sources.

the composition tasks, students were asked to design and implement a solution in a text editor, without the use of a compiler.

- *Think-aloud* Students were asked to think aloud while solving tasks. Whenever necessary, students were probed to continue verbalizing their thoughts (Ericsson & Simon, 1980).
- *Observation notes* The observer took notes of the student's behaviour during each task.
- *Semi-structured reflective interview* Students were then asked to rate the difficulty of the task (on a scale from 0 to 10, where 10 is most difficult), in terms of (1) the solution algorithm, (2) the programming concepts in the solution and (3) the problem statement. These rating questions were taken from Morrison et al. (2014) and used to establish self-reported (intrinsic) cognitive load in CS. Then, likewise to Fisler et al. (2016), students were given paper snippets with the task descriptions and asked to rank the tasks from most to least difficult, which was then photographed. The ratings and rankings were used as conversation starters: students were invited to explain which aspects contributed to their perception of difficulty and why (J. Whalley & N. Kasto, 2014b). To elaborate on the students' verbalizations, the interviewer referred to specific events observed in the students' problem-solving process. Example questions are: '*I noticed you hesitated while writing this loop. Can you explain what happened?*' and '*At that moment you went back to a piece of code you wrote earlier. Please explain why*'. As to (1), follow-up questions focused on the principle loop structure, constructs, plans or strategies used, especially when these differed from the expert solution. Students were invited to explain their thoughts and choices. As to (2), students were asked to compare and contrast the task's difficulty to previous tasks in terms of required concepts. The goal of (3) was to uncover any extraneous load that may unintentionally ffect difficulty.
- *Screen capturing and audio recording* Using screen capturing software, students' (code) editing activities and verbal reports were simultaneously recorded. The voice recordings were transcribed verbatim.

5. Data analysis

For the purpose of identifying difficulties and errors and to explore any underlying patterns, we employed qualitative content analysis (QCA), '*a data analysis technique within a rule guided research process, and this research process is bound to common*

(qualitative and quantitative) research standards' (Mayring, 2014). The coding of the student responses and those of the transcripts will be discussed below independently.

5.1. Data from think aloud and interviews

The response data from the audio transcriptions, from both the think aloud sessions and the interviews, were coded in ATLAS.ti. One researcher initially encoded the texts inductively and then on a second pass grouped the resulting codes into code-categories based on the literature. Then, on a third pass, the same researcher encoded the texts again, but this time using the developed category system. To ensure the validity and reliability of the interpretation category system, part of the data was individually coded by a second researcher. For inter-rater reliability seven transcriptions containing 215 quotations (of the 524 total) were selected, which is sufficient for a Krippendorff's alpha of 0.800 at a 0.05 level of significance. An inter-rater reliability score of 70% was reached for the transcripts. Our resulting category system for the transcripts consists of the following codes:

- **Conceptual knowledge**. This category is assigned to comments about constructs or elementary plans. It concerns statements about construct compactness, intuitiveness, elegance and being prone to errors or about loop-specific characteristics, such as the components of a **for** loop. It also concerns knowledge about basic plans and variable roles within those plans. Comments expressing construct or plan recognition or recall, lack of confidence about execution details of a construct, not knowing how a certain loop works, which a construct's components, can be tailored or how to do so, uncertainty about (implementation) details of the plan or in which situation a particular loop can best be applied belong to this category.
- **Strategic knowledge**. This category pertains to solution design. It encompasses comments pertaining to selecting, combining or tailoring plans to solve the problem at hand. Typical verbalizations include incorrect initial plan selection, considering hardcoding, getting caught up in details, trouble tailoring a plan to the task at hand or uncertainty about the execution of their solution (possibly as a result of patchwork).
- **Code complexity**. This category includes student comments related to complexity metrics, such as program flow. Typical statements refer to the number of variables, code length, number and types of constructs, determining conditional statements and boundary values or misconceptions pertaining to control flow. It encompasses comments pertaining to data flows, such as commenting on variable interactions or dispersion, explicitly tracing values or combining constructs through nesting or sequencing (e.g. '*the same thing happens at each iteration*').
- **Cognitive load**. This category is applied when the participant specifically verbalizes an increased cognitive effort ('*I have to think more*' or '*Arg. now I'm lost*'.). Comments include losing overview, forgetting solution parts or needing to backtrack their program code, particularly when several plans are involved.

5.2. Data from students' solutions

Students' solutions were coded in ATLAS.ti. For both construct and error coding, the code categories were borrowed from related work (Ebrahimi, 1994; Weeda et al., 2020). The solutions were first coded by one researcher, and then half of the solutions were independently coded by a second researcher. The resulting inter-rater reliability was high (more than 90%). The following category system for the students' solutions was used:

(1) **Coding constructs**: W (**while**), FE (**for each**), F (**for**), S (selection, i.e. **if** statements), MS (multiple selection, i.e **if-then-else** statements) and CS. (compound selection, i.e. conditions comprising and/or)
(2) **Coding errors**: Omit:Design (used hard coding instead of selecting a plan), Omit:Major (missing major parts of the plan), Omit:Minor (missing a minor part of the plan), Error:Misplaced (part of the plan in the wrong place), Error:Spurious (irrelevant, inappropriate or superfluous construct), Error:Logic (a logical error), Error:Bound (wrong iteration bound), Error:Other. (any other minor mistake that is made, e.g. updating an incorrect variable).
(3) **Coding transcript–solution relationships**: Inconsistent:Interview (the solution was correctly described in the interview but incorrectly implemented), Inconsistent:Sloppy (construct that has been correctly applied elsewhere, but is used incorrectly once), Error:Misconception (concept or construction that is consistently misused), Error:Specialisation (variable that is incorrectly initialized or updated, e.g. a loop control variable or other tailoring issues).

6. Results

Table 1 depicts the distribution of codes across the different coding categories. We now summarize the results in relation to (1) conceptual knowledge, (2) strategic knowledge, (3) code complexity or specifically and (4) cognitive load. Where relevant, we specify each student (S<number>) or reading/writing task (RT/WT-<task_name>) concerned (see sections 3.1 and 3.2).

6.1. (1) Conceptual knowledge

6.1.1. Constructs and basic plans

Students recognize and recall taught (sub)plans such as *min*/*max* (linear traversal and update most-wanted value), *count* (linear traversal and update gatherer value) and *output* plan. They describe variable roles using appropriate terminology, hence linking them to plans. All students provide a correct abstract answer to at least one of the comprehension tasks. Of the five students with incorrect answers to the comprehension tasks, two correctly implement the exact same plan later and the other three correctly answered the *RT-While* (*count* plan), and either the *RT-ForEach* (*min* plan) or the *RT-For* (*max* plan) task. Thus, despite making mistakes, all students show coherent understanding of the basic plans and constructs appearing in the comprehension tasks. However, not all students have all of the other expected plans available in their repertoires, such as the

Table 1. Distribution of codes.

Category	Code	Freq.
Constructs	**while**	26
	for each	19
	for	83
	if	111
	Multiple-branch Sel.	26
	Compound Sel.	33
Errors	Omit:Design	5
	Omit:Major	5
	Omit:Minor	22
	Error:Misplaced	6
	Error:Spurious	6
	Error:Logic	4
	Error:Bound	5
	Error:Other	7
Relationship coding	Inconsistent:Interview	4
	Inconsistent:Sloppy	24
	Error:Misconception	7
	Error:Specialisation	35

Table 2. Use, errors and rated difficulty per construct.

Construct	%Ranked most difficult	Freq. construct used	Construct-specific error rate	Construct-specific errors	Students avoiding construct
RT-While	57%	27	22.2%	stepper initialization, stepper update, termination condition	36%
RT-For	0%	84	13.1%	stepper initialization, stepper update, termination condition	0%
RT-ForEach	43%	19	5.3%	misconception	50%

search plan (*WT-FirstLast*). Also, one student expressed that it was not possible to break out of a **while** (S10) and thus unable to call the **one-way Boolean** plan.

6.1.2. Selection statements

Selection statements with multiple branches (i.e. **if-then-else** statements) require more thought than concatenated selections (two **if** statements) as these require simultaneously considering multiple (preceding) conditions (S4, S7, S9, S10, S11, S12, S14) and the code jumps that follow (S7, S10, S11) *'Because if you have to do many statements and like you exclude conditions, and then your final statement then it's really hard [to reason] if you're gonna make it there'*.(S14). Along the same line, conditionals containing logical operators such as **and**/**or** contribute to difficulty (S4, S12, S14). One student held the misconception that every selection statement requires an else-branch (*WT-FirstLast* : S7).

6.1.3. Loops

Table 2 depicts the use, errors and rated difficulty per construct.

The **for** loop was ranked as the easiest looping construct, implemented most frequently and used by each and every student. It is remarked that the construct can be used without deep knowledge of the notional machine '*you can also understand the for-loop without knowing what a for-loop does*'(*RT-For* : S10). Contrasting, ranked as the most difficult, the **while** loop had the highest error rate (22%) and was avoided by more than one-third of the students. Considering where to place the stepper initialization and update statements is reported to make the **while** error prone (*RT-While* : S6, S10, S14). Regarding the **for** and the **while** alike, students indicate that formulating looping requirements (stepper initialization, termination and update) is prone to errors '*you have to be careful how you make the counter*' (*RT-BarChart* : S14) and increases difficulty (*RT-While* : S3, S7, S8) '*I have to think about it more*' (*RT-While* : S8), which is affirmed by the construct-related errors made. Additionally, described to augment the difficulty of the **for** is considering all three parameters simultaneously (S9). Verbally described construct-specific misconceptions are that loops in general cannot be stopped prematurely, and also that the termination condition of a **for** cannot be extended with an additional condition.

Closely trailing the **while**, 43% of the students rank the **for each** as the most difficult construct. Though students comprehend the **for each** (except S11), only half the students use it. Verbal reports indicate that those students lack a coherent mental model of the **for each** construct and are thus unsure of how the construct works exactly. Yet, the students who employ the construct during composition, do so successfully. The only error was a misconception, consistently mistaking the element value for an index (S10).

6.1.4. Combining constructs

The manner in which constructs are combined effects difficulty. Nesting a conditional in a loop augments difficulty (*WT-BarChart* : S6, S14; *WT-FirstLast*) as '*something different can happen during each iteration*' (*WT-Merge* : S4). However, a double nested **for** is not perceived as much more difficult than a single **for** (*WT-BarChart* : S2, S3, S5, S6, S10) as '*it's the same thing for each element*' (*WT-BarChart* : S6), and a nested **for** is even considered easier than two abutted **for** s (*WT-BarChart* : S10). As such, solutions with nested conditionals require considering different cases and their corresponding actions (e.g. *WT-MaxFreq* : S2, S3, S5, S9; *WT-Average* : S2), making them harder to implement than nested loops, '*another thing I have to check*' (*RT-While* : S8). As another example of difficulties that arise depending on how constructs are combined, one student expressed understanding the effect of a **break** in a loop, but not how it behaves within a selection nested in a loop (*WT-FirstLast* : S7).

6.2. (2) Strategic knowledge

As a part of plan design, most students explicitly consider which constructs or variables to select, matching plan requirements such as array traversal (*RT-While* : S2, S13; *WT-Merge* : S2, S5, S8), producing a table (*WT-BarChart*), selecting which array to iterate over (*WT-Merge* : S2), prematurely terminating an iteration (*WT-FirstLast* : S2, S6, S13) or selecting viable constructs, for example when an index value is needed (*WT-FirstLast* : S5, S9, S12). Some students skip over the step of explicitly considering which construct to select, for example always start off using a **while** and changing it later if necessary (*WT-FirstLast* : S12), or with a **for** before considering what needs to precede or follow it (*WT-FirstLast* : S7).

From the perspective of plan design, students describe that thinking in terms of variable roles in relationship to program goals helps decompose the task and thus manage complexity (*WT-MaxFreq* : S2, S3, S12). Also, control structures such as the **for** are reported to help devise an appropriate novel algorithm '*Thinking about the problem in terms of structures maybe made it easier because they're already defined to do exactly that*' (*WT-FirstLast* : S5), and another student reports that the explicit consideration of boundaries helps envision and devise a solution (*WT-FirstLast* : S10).

Students mention that a task becomes more straightforward when they recognize and recall a familiar plan or solution (*RT-For* : S1, S3, S5, S10, S13, S14) or have previously practiced a similar task (*RT-For* : S2, S3, S10). The composition tasks rated easiest and implemented with a high degree of accuracy, are those which can be implemented using familiar plans, with minimal tailoring. This is the case for *WT-FirstLast* and *WT-Average* tasks '*those two that can already be combined [as is]*' (*WT-MaxFreq* : S9) and the *WT-BarChart* task which became straightforward when students recognized the need for a nested loop as being similar to an example discussed in class.

Though familiar plans are reported to simplify some tasks, not all students have a complete set of plans readily available in their repertoire. Students struggling to select appropriate plans or constructs may reside to hardcoding (*WT-FirstLast* : S12, S14), consider it as a preliminary option (*WT-FirstLast* : S1, S2, S11) or have to switch constructs halfway through implementation (*WT-FirstLast* : S1, S5, S8), all of which are detrimental for code quality. Rather than producing a coherent solution, this results in a patchwork process: *'you're not writing the code down, you just try to fix your own problems. . . OK, I have some trouble, so how can I fix this, such that it gives me in the end the result I need'*.(*WT-FirstLast* : S10). In turn, this patchwork impedes reasoning about control flow (*WT-FirstLast* : S8, S10, S11) decreasing confidence about correctness (*WT-FirstLast* : S12), introducing issues which generate rework '*I did something unhandy*' (*WT-FirstLast* : S3), trial-and-error debugging (*WT-FirstLast* : S8, S11) and confusion (*WT-FirstLast* : S1)*'when I changed it to the* ***for*** *loop, I had the condition and the iteration, but, but then I didn't know what I was trying to do because maybe I was imagining it for a* ***while****. I don't know'*.(*WT-FirstLast* : S5). To illustrate, in the *WT-FirstLast* task, students who lack a plan for *search* proceed with a patchwork process, increasing the number of (compound) selection constructs, driving deep nesting and complex control flow, and in effect making the task more difficult than the model solution proposed.

In some tasks, plans could be either abutted or merged. Though aware of the more efficient and elegant merged approach, some students report feeling more confident with plan abutments despite the lengthier code. They express that multiple independent statements or blocks are not harder than each block on its own (*WT-MaxFreq* : S3, S9), as long as they do not interact or share variables (*WT-MaxFreq* : S2).

Tailoring efforts needed to solve the task at hand increase difficulty. Examples of tailoring aspects include adjusting the termination condition of the inner loop to be dependent on array values (*WT-BarChart*), considering how to combine plans (*WT-MaxFreq*, *WT-FirstLast*) and determining specific termination conditions (*WT-FirstLast*). Students explicitly attribute the difficulty of tailored solutions to design aspects rather than implementation effort (*WT-Merge* : S5, S12). Plan-tailoring related mistakes include misplaced statements (*WT-FirstLast* : S10, S14; *WT-BarChart* : S14), use of wrong variables (*WT-FirstLast* : S1, S7, S8, S9, S11; *WT-BarChart*), major/relevant (i.e. other than output)

subplan omission (*WT-FirstLast* : S1, S4, S10; *WT-BarChart* : S10), spurious code (*WT-FirstLast* : S1, S10; *WT-BarChart* : S1), logical errors (*WT-FirstLast* : S3), boundary errors (*WT-Merge* : S6; *WT-BarChart* : S10), control flow errors (*WT-FirstLast* : S14, *WT-BarChart* : S1, S14) or loop-control variable initialization and update errors (*WT-BarChart* : S4, S10; *WT-Merge* : S6; *WT-FirstLast* : S6, S7).

In addition to construct-specific misconceptions, students may also hold misconceptions which impede plan tailoring, for example that a nested **for** loop can only be used for numeric tables and thus cannot be modified for the purpose of printing a bar chart (*WT-BarChart* : S1, S12, S14) or pertaining specifically to tailoring of constructs.

6.3. (3) Code Complexity

Students report multiple aspects pertaining to code complexity which contribute to increased difficultyin particular, related to data and their interactions induced by control flow. Examples are the number of variables to keep track simultaneously (*RT-While* : S2, S5, S6, S8; *RT-For* : S9), tracing the interaction among variables (*RT-While* : S8; *WT-BarChart* : S4), reuse of variables for multiple purposes (*WT-BarChart* : S2, S3, S4, S5, S10, S13; *WT-FirstLast* : S3; *WT-Merge* : S2, S9), co-dependency of variables with other parts of the code (e.g. variable effecting loop behavior in another code chunk) (S8, S14) or with other plans (S5, S6), and variables updated repeatedly (*WT-FirstLast* : S3, S12), successively (*WT-Merge* : S2, S9) or conditionally (*WT-FirstLast* : S3, S4, S6, S12, S13; *WT-Average* : S2, S4) '*index doesn't increment simultanously*' (*WT-Merge* : S9). Also, a conditional statement dependent on a variable is more difficult to reason about than a literal (*RT-For* : S4). In order to figure out what is going on (*WT-Merge* : S2), these aspects require more thought and (mental) tracing iterations, in effect, working on a low level of abstraction.

This interactivity goes hand in hand with (specialisation) errors such as incorrectly updating (*WT-MaxFreq* : S1, S6, S10; *WT-FirstLast* : S6, S7) or using the wrong variable (*WT-MaxFreq* : S1, S5; *WT-FirstLast* : S7). Though the reuse of variables increases difficulty, on the upside it also decreases the number of variables to keep track of (*WT-Merge* : S5) '*you have less variables for which you have to puzzle out their meaning*' (*WT-BarChart* : S3). The difficulty of the **while** construct was explicitly related to code length by all but one student (S9). Clarification during the interviews revealed that specifically the span of code over which a loop's variables remain active (S12; *RT-While* : S8, S11) makes code time-consuming to understand (*RT-While* : S10).

6.4. (4) Cognitive Load

Cognitive load, as coded here, encompasses aspects pertaining to losing overview or forgetting details, which are not attributed to conceptual or strategic knowledge or covered by the complexity metrics discussed above. Students express that cognitive load is elevated by tasks comprising multiple goals, for example '*when I can't keep the entire task in my head, I tend to forget things'*(*WT-FirstLast* : S9), '*I usually read the whole problem, but I don't always remember everything. . .*'(*WT-FirstLast* : S6), 'there *are so many possible cases to think about '*(*WT-FirstLast* : S14). Specifically, induced by the number of plans, students express they need to switch between working on an abstract-task level with working meticulously on subplan details: '*when you are programming you are often busy with quite abstract things, and*

then you have to focus very much on that, oh yeah, that you also have to work very precisely. . .'(*WT-FirstLast* : S9), which can lead to losing track of other aspects: *'. . . you're so deeply engaged, focused on that, that you don't really, uhm, remember, or well, don't pay attention to the rest anymore'*.(*WT-FirstLast* : S9).

We consider careless errors, for which no particular conceptual or strategic difficulty is related for that student, likely to be caused by cognitive load. Three types of (minor) careless errors were observed, namely (1) correctly describing an algorithm verbally followed by an erroneous implementation (e.g. incorrect update (*WT-FirstLast* : S6, S7; *WT-MaxFreq* : S4, S6), printing a different variable than asked in the task (*WT-MaxFreq* : S1), subplan omission (*WT-FirstLast* : S9)), (2) minor omissions (e.g. a missing print (all tasks) or return statement (*WT-Average* : S2, S13)) and (3) inconsistent errors correctly implemented elsewhere (e.g. re-initializing a variable, mistaking '=' for '==' (*WT-FirstLast* : S3, S8), omitting an index (*WT-FirstLast* : S7) or length in a conditional statement (*WT-Merge* : S9). Table 3 depicts performance, in terms of fully correct solutions, those involving minor errors at most (those mentioned above) or with major errors.

No students ranked *WT-Average* as a difficult task, yet due to the predominance of careless mistakes (66.7%) none were able to provide a fully correct answer. Contrastingly, the fewest careless errors (18.2%) were identified in the *WT-BarChart* task.

7. Discussion

Summarizing from the results, not all constructs are perceived as equally difficult, and students tend to avoid using constructs that they feel uncomfortable with. The number of variables and variable roles, the interactions between elements and the span of code over which a variable remains active disrupt students' overview and ability to abstractly reason about program behavior, augmenting difficulty. Similarly, the number of (sub)plans is noted to increase cognitive load, which can lead to careless errors. From a strategic point of view, recognizing learned plans as-is from repertoire facilitates solution design and implementation. However, when tailoring is required to solve a task, students endure difficulties selecting, combining and manipulating plans and constructs indicating that adequate plan knowledge alone is not sufficient for solving the task successfully. Furthermore, construct misconceptions can also effect tailoring strategies. However, our results show that students endure more construct difficulties and misconceptions than those uncovered from error analysis alone, emphasizing the need for qualitative data.

Table 3. Code composition tasks: performance and perception, rating on a scale from 0 (easy) – 10 (difficult).

Task	n	%Ranked most difficult	%Fully correct	%Minor errors only	%Major errors	Avg code rating	Avg concept rating
WT-FirstLast	14	71.4%	21.4%	21.4%	57.1%	4.6	3.1
WT-Merge	8	25.0%	12.5%	75.0%	12.5%	4	4.1
WT-BarChart	11	9.1%	45.5%	18.2%	36.4%	3.5	2.5
WT-MaxFreq	14	7.1%	21.4%	57.1%	21.4%	3.6	2.2
WT-Average	3	0%	0%	66.7%	33.3%	3	3

Through an in-depth analysis of the results from multiple sources (solutions, think-aloud sessions, interviews and student ratings), we gained a deeper understanding into the sources of difficulties, errors and misconceptions, which we will now discuss more profoundly.

7.1. (1) Conceptual difficulties

Although complexity ratings which merely count constructs do not distinguish between the types of constructs used, students describe differences in terms of perceived difficulty. The **while** is considered the most difficult looping construct, accompanied with the highest construct-specific error rate. Students attribute errors to the cognitive burden imposed by explicitly considering looping conditions as well as where to place them. Another plausible explanation is that because the components of the **while** can readily be modified (Shackelford & Badre, 1993), it is used in the more complex tasks such as those requiring tailoring, where cognition is already overloaded.

According to the verbal reports in our study, students perceive a difference between loops and conditionals. The effect of a loop on the rest of the program is similar for each iteration, thus the code can be chunked. In contrast, code in each conditional branch can have different effects, for example a different variable update, making program behavior more difficult to predict. This is in line with previous work which reports that individual control structures, such as looping constructs and different types of branched selections, are not equally difficult to trace (Lopez et al., 2008), comprehend (Cant et al., 1995) or write (Ebrahimi, 1994), yet contradicts Lahtinen et al. (2005)'s findings that loop structures are perceived as slightly more difficult than selection structures. More profoundly, our results suggest that the number of (implicit) conditional evaluations brought about by selection statements with multiple branches plays a role in perceived difficulty, extending Ebrahimi's reports on **if** statements. This finding could augment complexity measures which do not consider compound conditionals or multiple paths, such as Magel's regular expression (Magel, 1981).

A deep understanding of the **for each** seems to be related to a threshold. Students either value the **for each** 's abstraction mechanism, find it easy to use and use it faultlessly, or they lack a coherent mental model and rate it as difficult. The perceived difficulty may be attributed to its hidden traversal mechanism, making it hard to understand where and how the loop control variable is updated (Du Boulay, 1986) and thus a difficult concept for novices to initially grasp. These findings are also in line with Milne and Rowe (2002) who report that creating a clear mental model of the execution of abstract concepts is problematic. As a result of lacking a coherent mental model and deep understanding, some students resort to memorizing how the **for each** work, which may explain why students can comprehend the loop but avoid using it. The difficulty of mastering such abstraction mechanisms may be linked to the 'expert blind spot' and thereby underestimated by instructors. Though our results are in line with Lopez et al. (2008) who suggested an association between comprehension and composition, we believe that in order to confidently employ (abstraction) mechanisms flexibly during composition, the comprehension level must specifically be profound (understanding how it works) rather than superficial (understanding that it works). Particularly for abstract

mechanisms, fine-grained tasks should be developed to practice and assess students' knowledge.

Summarizing, not all constructs are perceived as equally difficult and a proxy for difficulty should differentiate between types of constructs, even if they are similar. Abstract mechanisms which cannot be readily traced are more difficult to grasp. Moreover, multiple evaluations, such as compound conditional statements and conditionals with multiple branches, drive difficulty. Finally, we emphasize the need for uni-constructed tasks to practice and assess construct-specific knowledge and tailoring skills.

7.2. (2) Strategic difficulties

Strategic knowledge encompasses selecting, combining and tailoring plans and the constructs used within them.

7.2.1. Construct and plan selection

Strategic knowledge involves deciding which construct to select in order to implement a specific plan, e.g. using a **for** or **for each** for array traversal, or a **while** or **for** when requiring an index. Though students express that considering appropriate construct-plan matches burdens cognitive load, our results concur with Ebrahimi's report that students tend to choose familiar constructs, rather than the most appropriate constructs. Avoiding particular constructs impacts the quality of the solution as well as the plan composition process, as evident in the *WT-FirstLast* (find the index of the first and last non-zero value in an array) and *WT-BarChart* (make a bar chart of an array) tasks.

Students recognize variable roles and recall taught (sub)plans. They can reason about plans on an abstract level and use them as building blocks to create more complex programs, albeit with (minor) errors. This is affirmed by the ease with which students apply familiar plans (e.g. *WT-MaxFreq* (determine the frequency of the maximum value of an array), *WT-Average* (compute the average value of an array), double nested loop in *WT-BarChart*), contrasting the troubles which some students have with unfamiliar plans (e.g. *search* in *WT-FirstLast*). Students acknowledge that their ability to recognize and recall appropriate plans and constructs helps reduce task difficulty and increase accuracy and is enhanced by ample instruction and practice. This is in line with Muller et al. (2007)'s results who describe that recognizing and recalling standard plans and relevant constructs from memory can drastically simplify tasks.

Rist (1989) explains this in terms of design strategies: when plans can be retrieved from memory, typically higher-level top-down strategies are employed. Contrasting, improper construct or plan selection can result in poor quality patchwork code invoked by quasi-random code changes (Lister, 2011) or local fixes (Robins et al., 2003), which can have disastrous results on code-quality (Shackelford & Badre, 1993). As a direct result of patchwork, students report an impediment to abstractly reason about code and an increased cognitive burden. Along the same line, Çakiroğlu and Bilgi (2022) interpret patchwork as an indicator of high effort and thus increased task difficulty. We emphasize the need for appropriate construct selection and ratify Ginat and Menashe (2015)'s results reporting on the benefits of explicitly teaching plans.

7.2.2. Combining plans and constructs

Our results show that the type of construct and the manner in which they are combined effects the element interactivity, and in turn the perceived difficulty. For example, a loop requires less thought than a selection because '*the same thing happens at each iteration*'. As a consequence, a selection statement nested in a loop is cognitively more demanding than vice versa because it requires reasoning about the selection condition at each iteration, while a loop nested in a selection statement requires considering the selection only once and then the chunked effect of the loop once.

7.2.3. Construct and plan tailoring

Another aspect of plan composition is tailoring. Focusing on loops, construct-specific errors were made in all three components of the **for** and **while** constructs alike (stepper initialization, termination and update). This suggests that determining looping conditions specific to the task at hand is equally difficult irrespective of the construct and thus a strategic design issue. Construct tailoring difficulties are augmented by misconceptions, for example, that the termination condition of a **for** cannot be extended with extra criteria (*WT-FirstLast* : S5).

Although most students select appropriate plans for straightforward tasks (i.e *WT-MaxFreq*, *WT-Average*), some exhibit problems recognizing candidate plans which require (minor) tailoring in order to solve the specified task. For example, the *search* plan (*WT-FirstLast* task) can be implemented using a one-way Boolean (Lopez et al., 2008) or as a tailored validation plan (De Raadt et al., 2009). Despite having received instruction on both, albeit in a slightly different context, some students were unable to select either plan from their repertoire. An example of a misconception pertaining to plan tailoring is that a nested **for** -loop cannot be tailored to print asterisks instead of values. Remarkably, rather than employing the easy-ranked **for** -loop in the *WT-BarChart* task, students implemented **while** -loops. This is striking because the **while** -loop was explicitly described as an unusual choice for array traversal. Students' reasoning is that the **for** is harder to tailor. Consequentially, instead of using a common plan as a template, students improvise a novel algorithm, which in itself demands higher cognition (Lahtinen et al., 2005). Thus, problems recognizing candidate plans for tailored solutions, or flexibly tailoring constructs, effect plan design, making the task more difficult.

Specifically, the number of aspects which have to be modified, as well as the extent of each modification augments perceived difficulty. Tailoring difficulties encompass modifying variables updates (*WT-BarChart*, *WT-Merge* (merge two arrays in alternating fashion)), using indices rather than values (*WT-FirstLast*) or combining multiple plans in a non-trivial fashion (*WT-FirstLast*). In each case, students are forced to keep track of each requirement and mentally trace code to ensure that each is fulfilled. Similarly, Kather et al. (2021) observed that comprehension of tailored plans took significantly longer than comprehension of standard plans, and we hypothesize the same to be true for code composition. Along the same line, Çakiroğlu and Bilgi (2022) noticed plan tailoring errors such as using a wrong variable for the problem at hand. Thus, in addition to requiring knowledge and plans (Ginat et al., 2013), our results call for specific knowledge pertaining to the recognition and identification of possible construct and plan modifications, in Educational Psychology termed *variability* (Van Merriënboer & Kirschner, 2017).

Summarizing about strategic knowledge, our results show that construct selection is driven by familiarity and is further influenced by misconceptions which impact design strategies, code quality and cognitive load. Interestingly, despite increased cognitive load, the explicit consideration of the looping conditions in **for** and **while** loops has also been reported to guide students through the design of more complicated tasks. The strategic knowledge needed for plan composition involves deep understanding of constructs and plans (Petersen et al., 2011). Students should therefore first focus on a strong foundation of necessary syntactic and conceptual knowledge, to then be followed up with strategic knowledge for combining (De Raadt et al., 2009) and tailoring constructs and plans Ginat and Menashe (2015). We emphasize that construct tailoring may benefit from explicit instruction. Our results support Costantini et al. (2020) and Ginat et al. (2013)'s findings that students have problems manipulating basic algorithmic patterns. We believe that similar to learning plans as canned solutions, students could benefit from learning tailoring strategies, such as which parts of plans and constructs can be (or are often) tailored, and how. Along the same line, Lister et al. (2009) express that flexible manipulation of plans in various contexts is essential and emphasizes that educators need knowledge and know-how about the design of appropriate tasks of incremental difficulties.

7.3. Code complexity-related difficulties

Based on our results, code complexity aspects such as data flow and element interactivity impact difficulty. Specifically, according to students' performance and verbal queues, variables being updated repeatedly, simultaneously, conditionally, reused for multiple purposes or variables which interact with each other or effect program flow contribute to difficulty. Furthermore, students express that merged plans induce element interactivity (in line with Kather et al. (2021)'s results), making them more difficult than abutted or nested plans, as also concluded by Shuhidan et al. (2009). This is in line with Duran et al. (2018) and confirmed by Çakiroğlu and Bilgi (2022), from a cognitive load perspective, argue that due to a higher element interactivity, merged programs are more difficult to comprehend than abutted programs. According to students' verbalizations, our results indicate that element interactivity itself impedes the ability to reason about code at an abstract level, as students have to transition to a lower level to manage the details. The implications of our results are depicted in Figure 2. Variable dispersion, or the span of code over which to keep track of active variables, further increases cognitive demand, which is in line with Sweller (2010)'s findings. As conditional and looping structures regulate control flow, whether, and how often code blocks are executed, they control how the variables within their span of control are updated or interact. Thus, difficulty is correlated to the abundance of each type of control structure.

Our results show that the reuse of variables for multiple purposes, such as in the *WT-Merge* and *WT-FirstLast* tasks, makes the code harder to manipulate, which is in line with reports on cognitive complexity (Cant et al., 1995). As opposed to abutted solutions, students describe an increased difficulty with merged solutions, which is in line with error observations made by Spohrer et al. (1985). Students express that the burden from an increased element interactivity in merged code is higher than that of lengthier abutted code. As solutions become more complex, the number of variables increases, as well as the different roles that these variables play and how they interact.

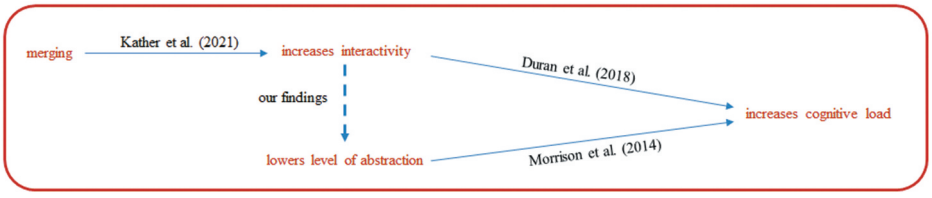

Figure 2. Our findings indicate a(n) (inverse) relationship between element interactivity and abstraction level.

We concur with Lister et al. (2009) that the concept of roles and variables could be a promising technique to help students reason about code on an abstract level. Furthermore, we believe that a thorough understanding of the variable roles in relation to plans (Sajaniemi & Navarro-Prieto, 2005) may help students tailor plans, and more specifically, tailor constructs, to the task at hand. An interesting perspective could be from the Block model (Schulte, 2008), which describes how identifying text surface denotations can help reason about connotations, such as variable roles and program goals. As interactivity increases, plans become harder to identify and denotations become clouded (Kather et al., 2021). In turn, this impairs reasoning about connotations. To exemplify this, we discuss the (lack of) transparency of the **for each** loop. In terms of Sajaniemi (2002)'s variable roles, the *walker* and *stepper* variable roles of the **for each** are both implicitly fulfilled by just one variable, making it harder to comprehend. As such, the impediment to deeply understand how the **for each** works, and thus also tailor the construct, may be instigated by the absence of explicit beacons (see Brooks (1983)) revealing the construct's operation on a text surface level.

Similarly, as a result of increased nesting depth and data interactivity, we expected the nested loops in the *WT-BarChart* task to augment task difficulty (Falah & Magel, 2015), yet the task was ranked as one of the easiest and implemented by all students without hesitation. We attribute the ease of implementing nested loops to familiar plan recall.

Summarizing about program flow, our findings are in line with J. Whalley et al. (2011)'s research who concluded that interacting code is more difficult than code with an encapsulated goal. This also makes sense according to Cognitive Load Theory, as the entire block of code over which data interacts must be held in memory in order to reason about it. Our results affirm J. Whalley and N. Kasto (2014a)'s conclusions that complex control flow, for example inherent to merged code, contributes to interactivity, and Çakiroğlu and Bilgi (2022) who attribute malformed plans to cognitive load imposed by element interactivity. Specifically, we relate element interactivity to augmented cognitive load when students are engaged in tailored, merged and patched solutions. Another view of the perceived difficulty is that when variables take on multiple roles (Sajaniemi, 2002), the code becomes harder to comprehend and manipulate.

7.3.1. (4) Cognitive load issues

Although it cannot be determined with certainty whether certain errors are due to cognitive overload, we congregate the remaining issues into the cognitive load category for the sake of convenience. Compared to singular-goal tasks (e.g. *WT-Merge*), tasks comprising multiple goals (e.g. *WT-Average*) are riddled with careless errors. Specifically,

switching between abstraction levels (i.e. plan and implementation-level), which occurs for each subgoal, increases the tendency to forget things. Costantini et al. (2020) attribute such subplan omission to either cognitive overload or working hastily. From a student's perspective, tasks involving multiple goals elevate cognitive load. Similarly, Ihantola and Petersen (2019), Petersen et al. (2011) and Duran et al. (2018) suggest that concept counts relate to task complexity. As concept counts are inherently associated with plan counts, it would be interesting to further empirically investigate the relationship between difficulty and number of subgoals (or plans) involved in their solution.

7.4. Summary

Students generally recognize, recall and apply taught plans as well as composition strategies such as merging. A prominent difficulty is students' lack of knowledge and strategies for effective construct and plan tailoring. Element interactivity is elevated in merged, patched and tailored solutions. When combining plans, the difficulty is furthermore effected by the (nesting) order in which constructs are combined (e.g. a selection in a loop is more difficult than vice-versa). The degree to which variables and constructs interact and the span of code over which they do so effects the capacity to chunk code and thus impedes ability to reason about code abstractly. In addition to the difficulty of each type of construct, its effect on element interaction should be considered when creating tasks. Likewise is true for plans. Plan knowledge and understanding variable roles in relationship to program goals could help students manage complexity. More research is needed to determine how the increase in difficulty related to the order of nesting constructs and plans can be integrated into existing models (e.g. Duran et al. (2018)'s MPI).

8. Conclusion

This study reports on an in-depth research combining data from multiple sources using code comprehension and composition tasks. In addition to solution analysis, students' perceived difficulty was elicited using task ranking, rating, think-aloud sessions and interviews. We augment results from related work, which are primarily based on instructors' expectations, by shedding light on the way we teach and assess from the perspective of students. Specifically, the verbal reports provided supplementary insight into the sources which brought about difficulties, as well as misconceptions which could not have been uncovered merely through the analysis of students' solutions. As such, we provide insights to help educators compose, decompose and order tasks. We now review our research questions in light of this new cognition:

RQ1: From a student's perspective, which aspects contribute to the difficulty of programming tasks through the lens of **conceptual knowledge**, and why?

Some constructs are perceived as more difficult than others. Students avoid constructs which they find difficult to use or tailor. Subsequently, misconceptions and knowledge gaps can go undetected. Furthermore, construct avoidance interferes with appropriate construct selection during plan design, leading to improvised, poor quality solutions. To

help students boost confidence to flexibly apply and tailor constructs, we emphasize embedding uni-construct tasks in curricula in order to practice and assess required construct-specific knowledge in an isolated fashion. Also, construct variants differ in difficulty. For example, the number of (implicit) conditional evaluations brought about by selection statements with multiple branches plays a role in perceived difficulty. Furthermore, abstract mechanisms (such as the **for each** -loop) are difficult to initially grasp. This has implications for curriculum planning, postponing the more abstract concepts until after students have demonstrated a deep understanding of the more explicit constructs alternatives and are capable of reasoning about those on an abstract level.

From the perspective of plans, our results indicate that students who can retrieve a relevant plan from their repertoire significantly decrease task difficulty. Taking this into consideration, we call on the community to add onto the set of plans now available as educational material (e.g. De Raadt et al. (2009)) and to explicitly instruct and assess these.

RQ2: From a student's perspective, which aspects contribute to the difficulty of programming tasks through the lens of **strategic knowledge**, and why?

Students struggle with plan tailoring. Students are unaware of ways in which to tailor plans and the constructs within them. Bearing in mind that teaching and assessing plans explicitly has been shown to be an effective didactic strategy, instruction could be augmented to include construct and plan tailoring strategies. We note that instruction from the viewpoint of variable roles in relation to plans may also be a viable option, helping students recognize, select and tailor-appropriate combinations of plans and constructs for the task at hand.

Constructs and plan compositions comprising multiple variables and plans, which lead to variable dispersion (i.e. the span of code over which a variable is active) as well as increased element interactivity (co-dependency between variables and plans) impede the ability to reason about code at an abstract level and augment cognitive load. More research is needed to understand the implications of the way in which constructs and plans are combined and how to incorporate this aspect into existing complexity models.

9. Limitations and future work

The goal of this study was to determine differences in novices' perceived difficulty across constructs, plans and compositions thereof. Preceding this study, a pilot was carried out in secondary education with students just learning how to program. However, as many of these students were not able to complete the tasks, they were also unable to compare their difficulty. As knowledge of certain plans and constructs was required, our study thus excluded beginners. It is worthwhile to research whether an adapted research setup would yield similar results for true beginners.

When selecting tasks for the study, the complexity metrics of the theoretical models were based on an expected solution rather than students' solutions. During code-composition, students create unique solutions, resulting in a larger solution space. In order to refine common complexity metrics, it would be interesting to further investigate students'

solutions. We have already experimented with this idea yet ascertain that for reliability's sake, data from a larger set of students' solutions is needed.

While our exploratory small-scale setup enabled us to complete a thorough in-depth analysis, such as triangulating data from various data sources, we are careful not to make claims regarding generalizability. Undoubtedly, investigation on a larger scale of novice programmers and tasks could provide more composure. For example, as our research focuses on elementary programming instructions (looping over arrays), it is interesting to contemplate tasks created specifically for the coverage of more advanced topics. As to other future work, in light of our results, we call on the community to examine existing strategies for teaching and assessing plans, such as those proposed by De Raadt et al. (2009), for completeness and propose additional plans where appropriate (e.g. include a plan which implements a nested loop). Furthermore, strategies for construct and plan tailoring should be developed.

A wide range in quality of solutions has been identified, in terms of the amount and types of errors as well as demonstrated SOLO levels. Similarly, multiple reports recognize a large variability in students' abilities in CS1 (e.g. Ginat et al. (2013)) or even a bimodal distribution (McCracken et al., 2001), i.e. students perform either poorly or very well. Whereas the use of average scores (D'Souza et al., 2012) or percent correct (Lister et al., 2009) are practical measures for difficulties, perhaps a more fine-grained analysis which considers error types and their severity could provide illuminating insights into perceived difficulty. To illustrate, different types of students may endure different difficulties. For example, being able to work at an abstract level reduces cognitive load and thus helps manage the difficulty of a task. It would be interesting to research if students working at different cognitive levels experience different difficulties. To this end, in the near future we would like to explore the variations among students in relation to the specific difficulties. This could be of scientific value, clarifying contradictions in the previous research where students' results have been 'pitch-forked' into one pile, as well as have pedagogical implications about the way in which we teach and assess students (including differentiation, scaffolding, as well as for use in automated practice systems).

Note

1. The tasks and canonical solutions are available through: https://doi.org/10.5281/zenodo.7413072.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

The work was supported by the Stichting Leerplan Ontwikkeling, Netherlands.

ORCID

Erik Barendsen http://orcid.org/0000-0003-4684-4287

Data availability

The data that support the findings of this study are openly available in the DANS EASY archive at https://doi.org/10.17026/dans-xnn-x5p3.

References

Biggs, J., & Collis, K. (1982). *Evaluating the quality of learning: The solo taxonomy (structure of the observed learning outcome)*. Academic Press.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*(6), 543–554. https://doi.org/10.1016/S0020-7373(83)80031-5

Çakiroğlu, Ü., & Bilgi, Ş. (2022). Exploring intrinsic cognitive load in the programming process: A two dimensional approach based on element interactivity. *Interactive Learning Environments*, 1–19. https://doi.org/10.1080/10494820.2022.2137527

Campbell, G. A. (2018). Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 International Conference on Technical Debt*, May 27 - 28, 2018. Gothenburg Sweden: Association for Computing Machinery (pp. 57–58).

Cant, S., Jeffery, D. R., & Henderson-Sellers, B. (1995). A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, *37*(7), 351–362. https://doi.org/10.1016/0950-5849(95)91491-H

Costantini, U., Lonati, V., & Morpurgo, A. (2020). How plans occur in novices' programs: A method to evaluate program-writing skills. In *Proceedings of the 51st acm technical symposium on computer science education*, Portland, OR (pp. 852–858).

De Raadt, M., Watson, R., & Toleman, M. (2009). Teaching and assessing programming strategies explicitly. In *Proceedings of the eleventh australasian conference on computing education-volume 95*, Koli, Finland (pp. 45–54).

D'Souza, D., Sheard, J., Harland, J., Carbone, A., & Laakso, M. -J. (2012). Can computing academics assess the difficulty of programming examination questions? In *Proceedings of the 12th koli calling international conference on computing education research* ,Koli, Finland (pp. 160–163).

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

Duran, R., Sorva, J., & Leite, S. (2018). Towards an analysis of program complexity from a cognitive perspective. In *Proceedings of the 2018 acm conference on international computing education research*, Espoo Finland (pp. 21–30).

Ebrahimi, A. (1994). Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies*, *41*(4), 457–480. https://doi.org/10.1006/ijhc.1994.1069

Ericsson, K. A., & Simon, H. A. (1980). Verbal reports as data. *Psychological Review*, *87*(3), 215. https://doi.org/10.1037/0033-295X.87.3.215

Falah, B., & Magel, K. (2015). Taxonomy dimensions of complexity metrics. In Int'l conf. software eng. research and practice, Las Vegas, NEV.

Fisler, K. (2014). The recurring rainfall problem. In *Proceedings of the tenth annual conference on international computing education research* (pp. 35–42). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/2632320.2632346

Fisler, K., Krishnamurthi, S., & Siegmund, J. (2016). Modernizing plan-composition studies. In *Proceedings of the 47th acm technical symposium on computing science education*, Memphis, Tennessee (pp. 211–216).

Gibson, B. (1997). Talking the test: Using verbal report data in looking at the processing of cloze tasks. *Edinburgh Working Papers In Applied Linguistics , 8*, 54–62.

Ginat, D., & Menashe, E. (2015). Solo taxonomy for assessing novices' algorithmic design. In *Proceedings of the 46th acm technical symposium on computer science education* (pp. 452–457). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/2676723.2677311

Ginat, D., Menashe, E., & Taya, A. (2013). Novice difficulties with interleaved pattern composition. In *International conference on informatics in schools: Situation, evolution, and perspectives*, Berlin, Heidelberg (Springer) (pp. 57–67).

Ihantola, P., & Petersen, A. (2019). Code complexity in introductory programming courses. In *Proceedings of the 52nd hawaii international conference on system sciences*, Hawaii, USA.

Ihantola, P., Sorva, J., & Vihavainen, A. (2014). Automatically detectable indicators of programming assignment difficulty. In *Proceedings of the 15th annual conference on information technology education* (p. 33–38). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2656450.2656476

Izu, C., Weerasinghe, A., & Pope, C. (2016). A study of code design skills in novice programmers using the solo taxonomy. In *Proceedings of the 2016 acm conference on international computing education research*, Melbourne, VIC, Australia (pp. 251–259).

Kalyuga, S. (2011). Cognitive load theory: How many types of load does it really need? *Educational Psychology Review*, *23*(1), 1–19. https://doi.org/10.1007/s10648-010-9150-7

Kather, P., Duran, R., & Vahrenhold, J. (2021, Nov). Through (tracking) their eyes: Abstraction and complexity in program comprehension. *ACM Transactions on Computing Education*, *22*(2), 1–33. Retrieved from https://doi.org/10.1145/3480171

Lahtinen, E., Ala-Mutka, K., & Järvinen, H. -M. (2005). A study of the difficulties of novice programmers. *Acm Sigcse Bulletin*, *37*(3), 14–18. https://doi.org/10.1145/1151954.1067453

Leppink, J., Paas, F., Van der Vleuten, C. P., Van Gog, T., & Van Merriënboer, J. J. (2013). Development of an instrument for measuring different types of cognitive load. *Behavior Research Methods*, *45* (4), 1058–1072. https://doi.org/10.3758/s13428-013-0334-1

Lister, R. (2011). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Proceedings of the thirteenth australasian computing education conference-volume 114*, Perth, Australia (pp. 9–18).

Lister, R., Fidge, C., & Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Proceedings of the 14th annual acm sigcse conference on innovation and technology in computer science education* (pp. 161–165). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/1562877.1562930

Liu, P., & Li, Z. (2012). Task complexity: A review and conceptualization framework. *International Journal of Industrial Ergonomics*, *42*(6), 553–568. https://doi.org/10.1016/j.ergon.2012.09.001

Lopez, M., Sutton, K., & Clear, T. (2009). Surely we must learn to read before we learn to write! In *Proceedings of the eleventh australasian conference on computing education-volume 95*, Wellington, New Zealand (pp. 165–170).

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*, Sydney, Australia (pp. 101–112).

Luxton-Reilly, A., Becker, B. A., Cao, Y., McDermott, R., Mirolo, C., Mühling, A., Whalley, J. . . . , Petersen, A., Sanders, K., Simon, (2017). Developing assessments to determine mastery of programming fundamentals. In *Proceedings of the 2017 iticse conference on working group reports* (pp. 47–69). Bologna, Italy: ACM. Retrieved from https://doi.org/10.1145/3174781.3174784

Luxton-Reilly, A., & Petersen, A. (2017). The compound nature of novice programming assessments. In *Proceedings of the nineteenth australasian computing education conference* (p. 26–35). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3013499.3013500

Magel, K. (1981). Regular expressions in a program complexity metric. *ACM SIGplan Notices*, *16*(7), 61–65. https://doi.org/10.1145/947864.947869

Mayring, P. (2014). Qualitative content analysis: Theoretical foundation, basic procedures and software solution.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B. -D., Wilusz, T. (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* (pp. 125–180). ACM. Retrieved from https://doi.org/10.1145/572133.572137

Milne, I., & Rowe, G. (2002). Difficulties in learning and teaching programming—views of students and tutors. *Education and Information Technologies*, *7*(1), 55–66. https://doi.org/10.1023/A:1015362608943

Morrison, B. B., Dorn, B., & Guzdial, M. (2014). Measuring cognitive load in introductory cs: Adaptation of an instrument. In *Proceedings of the tenth annual conference on international computing education research*, Glasgow, Scotland (pp. 131–138).

Muller, O., Ginat, D., & Haberman, B. (2007). 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*, Dundee, Scotland (pp. 151–155).

Nathan, M. J., & Petrosino, A. (2003). Expert blind spot among preservice teachers. *American Educational Research Journal*, *40*(4), 905–928. https://doi.org/10.3102/00028312040004905

Paas, F. G. (1992). Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach. *Journal of Educational Psychology*, *84*(4), 429. https://doi.org/10.1037/0022-0663.84.4.429

Petersen, A., Craig, M., & Zingaro, D. (2011). Reviewing cs1 exam question content. In *Proceedings of the 42nd acm technical symposium on computer science education*, Dallas, TX, USA (pp. 631–636).

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, *18*(1), 1–24. https://doi.org/10.1145/3077618

Rist, R. S. (1989). Schema creation in programming. *Cognitive science*, *13*(3), 389–414. https://doi.org/10.1207/s15516709cog1303_3

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings ieee 2002 symposia on human centric computing languages and environments*, Arlington, VA, USA (pp. 37–39).

Sajaniemi, J., & Navarro-Prieto, R. (2005). Roles of variables in experts' programming knowledge. In *Ppig* (p. 13).

Schulte, C. (2008). Block model: An educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the fourth international workshop on computing education research*, Sydney, Australia (pp. 149–160).

Schulte, C., & Bennedsen, J. (2006). What do teachers teach in introductory programming? In *Proceedings of the second international workshop on computing education research*, Canterbury, United Kingdom (pp. 17–28).

Shackelford, R. L., & Badre, A. N. (1993). Why can't smart students solve simple programming problems? *International Journal of Man-Machine Studies*, *38*(6), 985–997. https://doi.org/10.1006/imms.1993.1045

Sheard, J., Carbone, A., Chinn, D., Laakso, M. -J., Clear, T., De Raadt, M., Warburton, G., Warburton, G. (2011). Exploring programming assessment instruments: A classification scheme for examination questions. In *Proceedings of the seventh international workshop on computing education research*, Providence, Rhode Island, USA (pp. 33–38).

Shuhidan, S., Hamilton, M., & D'Souza, D. (2009). A taxonomic study of novice programming summative assessment. In *Proceedings of the eleventh australasian conference on computing education-*, Wellington, New Zealand volume *95* (pp. 147–156).

Sirkiä, T., & Sorva, J. (2012). Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th koli calling international conference on computing education research*, Koli, Finland (pp. 19–28).

Soloway, E. (1986). Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM*, *29*(9), 850–858. https://doi.org/10.1145/6592.6594

Spohrer, J. C., Soloway, E., & Pope, E. (1985). A goal/plan analysis of buggy pascal programs. *Human–Computer Interaction*, *1*(2), 163–207. https://doi.org/10.1207/s15327051hci0102_4

Sweller, J. (2010, Jun). Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, *22*(2), 123–138. https://doi.org/10.1007/s10648-010-9128-5

Teague, D., & Lister, R. (2014a). Blinded by their plight: Tracing and the preoperational programmer. In *Ppig* (p. 8).

Teague, D., & Lister, R. (2014b). Longitudinal think aloud study of a novice programmer. In *Conferences in research and practice in information technology series*, Auckland, New Zealand

Tew, A. E. (2010). *Assessing fundamental introductory computing concept knowledge in a language independent manner* (Unpublished doctoral dissertation). Georgia Institute of Technology.

Van Merriënboer, J. J., & Kirschner, P. A. (2017). *Ten Steps to complex learning: A systematic approach to four-component instructional design*. Routledge.

Weeda, R., Izu, C., Kallia, M., & Barendsen, E. (2020). Towards an assessment rubric for eipe tasks in secondary education: Identifying quality indicators and descriptors. In *Koli calling'20: Proceedings of the 20th koli calling international conference on computing education research*, Koli, Finland (pp. 1–10).

Whalley, J., Clear, T., Robbins, P., & Thompson, E. (2011). Salient elements in novice solutions to code writing problems. In *Proceedings of the thirteenth australasian computing education conference-volume 114*, Perth, Australia (pp. 37–46).

Whalley, J., & Kasto, N. (2014a). How difficult are novice code writing tasks?: A software metrics approach. In *Proceedings of the sixteenth australasian computing education conference* - volume *148* (pp. 105–112). Darlinghurst, Australia, Australia: Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=2667490.2667503

Whalley, J., & Kasto, N. (2014b). A qualitative think-aloud study of novice programmers' code writing strategies. In *Proceedings of the 2014 conference on innovation & technology in computer science education*, Uppsala, Sweden (pp. 279–284).

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., & Prasad, C. (2006). An Australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies. In *Proceedings of the 8th australasian conference on computing education* - volume *52* (pp. 243–252). Darlinghurst, Australia, Australia: Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1151869.1151901

Zingaro, D., Petersen, A., & Craig, M. (2012). Stepping up to integrative questions on cs1 exams. In *Proceedings of the 43rd acm technical symposium on computer science education*, Raleigh, North Carolina, USA (pp. 253–258).