Received: 21 April 2025, Accepted: 10 November 2025

DOI: https://doi.org/10.26529/cepsj.2130

Developing Conceptual Programming Knowledge in Pre-Service Computer Science Teachers: The Role of Programming Patterns

Matej Zapušek*[1] and Irena Nančovska Šerbec[2]

This study examines how students enrolled in a two-subject teacher programme (computer science and mathematics) at the Faculty of Education, University of Ljubljana, develop a conceptual understanding of programming knowledge through the implementation, recognition and explanation of programming patterns. Based on over 500 programming solutions completed by first- and second-year students, we focus on four foundational patterns: linear search, guarded search, counting and extreme values. The study involved 70 students across different phases, with 17 of them tracked longitudinally over three to four years, examining their ability to recognise programming patterns, explain underlying logic and design related tasks. The results show that the students gradually improved their use of programming patterns, initially producing many redundant or incorrect solutions, which over time shifted towards correct implementations. However, this development was uneven across pattern types and programming constructs. Tasks involving while loops and guarded searches initially proved more challenging, with higher rates of incorrect or redundant solutions in the early phases. A consistent finding across all of the student groups was a substantial gap between the students' ability to implement patterns and their ability to explain them conceptually. This demonstrates that for loop implementation skills do not automatically transfer to conceptual understanding, especially for more complex cases like guarded search and extreme values. This finding is particularly concerning for prospective educators. Importantly, explanation ability strongly predicted task design quality, underscoring the fact that conceptual mastery directly supports pedagogical competence. These findings highlight the need for explicit instruction on programming patterns in teacher education, not only to support correct implementation but also to build deeper

1 *Corresponding Author. Faculty of Education, University of Ljubljana, Slovenia; matej.zapusek@pef.uni-lj.si.

2 Faculty of Education, University of Ljubljana, Slovenia.

explanatory and pedagogical skills. Emphasising patterns as conceptual tools can help future educators better analyse code, anticipate student difficulties and design effective, pattern-based programming tasks.

Keywords: introductory programming, programming patterns, computational thinking, teachers' education

Razvijanje konceptualnega programerskega znanja pri študentih pedagoških smeri računalništva: vloga programskih vzorcev

Matej Zapušek in Irena Nančovska Šerbec

V raziskavi preučujemo, kako študentje študijskega programa Dvopredmetni učitelj, smer računalništvo in matematika, na Pedagoški fakulteti Univerze v Ljubljani razvijajo konceptualno programersko znanje skozi implementacijo, prepoznavanje in razlago programskih vzorcev. Na podlagi več kot 500 programskih rešitev študentov prvega in drugega letnika se osredinjamo na štiri temeljne programske vzorce: linearno iskanje, zaščiteno linearno iskanje, štetje in iskanje ekstremnih vrednosti. V raziskavi je sodelovalo 70 študentov; 17 smo jih spremljali longitudinalno (od tri do štiri leta) ter analizirali njihovo zmožnost prepoznavanja in razlage logike delovanja izbranih programskih vzorcev ter oblikovanja nalog, ki pri reševanju zahtevajo njihovo uporabo. Izsledki kažejo, da so študentje postopoma izboljševali uporabo programskih vzorcev: v začetnih fazah so pogosto reševali naloge z odvečnimi deli kode ali napačnimi rešitvami, s časom pa so prehajali k pravilnim implementacijam. Napredek je bil neenakomeren med različnimi programskimi vzorci in uporabljenimi programerskimi konstrukti. Naloge, ki so vključevale uporabo zanke »while« pri zaščitenem linearnem iskanju, so se izkazale za zahtevnejše, saj so v zgodnjih fazah pogosteje vodile v napačne rešitve ali odvečne dele kode. Kljub izboljšavam pri implementaciji se je pri vseh skupinah študentov pokazal razkorak med tem, kar znajo implementirati s programsko kodo, in tem, kar znajo konceptualno razložiti. To kaže, da pravilna uporaba programskega konstrukta, na primer zanke »for«, še ne zagotavlja razumevanja ideje koncepta programskega vzorca, kar je še zlasti očitno pri zahtevnejših vzorcih, kot sta zaščiteno linearno iskanje in iskanje ekstremnih vrednosti. Zavedanje obstoja tega razkoraka je pri bodočih učiteljih še posebej pomembno. Poleg tega se je pokazalo, da ima sposobnost razlage delovanja programskega vzorca močen vpliv na kakovost oblikovanih nalog. To potrjuje, da konceptualno razumevanje neposredno podpira razvoj pedagoških kompetenc. Rezultati raziskave kažejo, da bi bilo v izobraževanju učiteljev računalništva smiselno bolj eksplicitno vključiti poučevanje programskih

vzorcev. S tem bi prispevali k njihovi pravilni implementaciji, prav tako pa bi spodbujali globlje konceptualno razumevanje in s tem povezanih pedagoških kompetenc. Umeščanje programskih vzorcev kot konceptualnih orodij v programe pedagoških smeri računalništva lahko bodočim učiteljem omogoči natančnejšo analizo kode, boljše predvidevanje pogostih učnih težav in oblikovanje učinkovitih, na vzorcih temelječih programerskih nalog.

Ključne besede: uvodno programiranje, programski vzorci, računalniško mišljenje, izobraževanje učiteljev

Introduction

In today's rapidly evolving technological society, understanding core computer science (CS) concepts is essential for meaningful and competent participation in modern life (Webb et al., 2017). High-quality computer science education (CSE) is crucial not only for meeting workforce demands but also for supporting diverse professional fields where digital skills are increasingly vital. Digital literacy enhances efficiency, fosters innovation and enables creative problem-solving (Fojcik & Fojcik, 2021). With appropriate knowledge, students can move from passive consumers to active co-creators of the digital future, becoming agents of cultural and social change rather than merely reacting to technological developments (Gretter & Yadav, 2016). However, critical discourse suggests that the concept of digitalisation in education is often framed through vague or metaphorical narratives that obscure its pedagogical implications. Vivitsou (2019) argues that digitalisation has acquired a kind of mythical status – frequently referenced, yet conceptually underdefined – leading to blurred boundaries between technological ambition and educational purpose. While general digital literacy is important, future CS educators need structured programming knowledge that develops systematic thinking and problem-solving capabilities beyond surface-level approaches. Our study responds to this need by examining how future CS teachers develop a deep understanding of programming structures, which are essential elements for teaching CS effectively.

CSE fosters not only technical skills but also essential cognitive competencies like critical thinking and problem-solving, which are vital across all sectors (Salehi et al., 2020). When integrated with digital technologies, it promotes creativity through experimentation, collaboration and personalised content creation (Weng et al., 2023). A key outcome is the development of computational thinking (CT). Introduced by Wing (2006), this includes decomposition, pattern recognition, abstraction and algorithmic thinking, offering structured strategies applicable across disciplines (Maharani et al., 2019; Shute et al., 2017; Wu et al., 2024).

The role of programming in computer science education

Programming plays a central role in CSE, as it provides the primary means for developing CT and understanding how digital systems operate (Abesadze & Nozadze, 2020; Zeng et al., 2023). While research has documented novice difficulties with reading and tracing code (Lister et al., 2004), less is known about how these foundational challenges manifest in programming pattern

implementation and pedagogical understanding. Importantly, programming has been described not only as a technical skill but also as a cognitive activity that fosters logical reasoning, analytical thinking and problem-solving (Soloway, 1986; Wing, 2006; Xie et al., 2019). These strategies extend beyond the field of CS, equipping learners to address complex challenges across diverse domains (Yadav et al., 2017). Furthermore, programming encourages active creation, interdisciplinary thinking and the development of creativity and innovation (Kiesler, 2022). Its integration into CS curricula is therefore a key strategy for preparing learners with the competencies needed to navigate and shape the demands of the twenty-first century. At the heart of successful implementation of CSE – and a prerequisite for achieving these goals – is the well-trained and competent teacher. Within teacher education programmes that prepare future CS educators, it is essential that students acquire a deep and well-structured conceptual understanding of programming. Such knowledge must go beyond surface-level proficiency, enabling future teachers to meaningfully interpret and convey programming concepts to their own students. As Soloway (1986) articulated, learning to program involves "constructing mechanisms and explanations", requiring teachers to develop both coding ability and skills to explain the conceptual structures underlying effective solutions. Only with this level of understanding can they design effective, developmentally appropriate instruction that fosters CT and supports learners in becoming confident and creative problem solvers. In line with this perspective, studies have explored how hands-on digital making can foster both technical understanding and pedagogical reflection in teacher education. For example, Bosco et al. (2019) describe a multi-year project in which students in education programmes collaboratively designed and fabricated digital artifacts. Through reflective storytelling and process documentation, the students not only acquired technological knowledge but also developed pedagogical insights. These experiential approaches highlight the importance of active knowledge construction but often lack explicit focus on the conceptual structures that underpin effective teaching. In this context, our study contributes by focusing specifically on how pre-service computer science teachers develop conceptual and pedagogical understanding through engagement with programming patterns, an approach that combines algorithmic thinking with didactic purpose.

Programming patterns as pedagogical tools

Introductory programming is widely recognised as a difficult subject, especially for students in teacher education, who often lack prior programming

experience. The main challenge lies not only in understanding individual constructs, such as loops or conditionals, but also in learning to integrate these constructs into coherent solutions for complex problems. Addressing this challenge requires a shift from surface-level syntactic knowledge to a deeper conceptual understanding of algorithmic structures (Proulx, 2000). Programming patterns, defined as conceptual templates for recurring algorithmic problems, have been proposed as an effective pedagogical approach because they capture expert strategies, support systematic problem-solving and encourage good programming practices (Zapušek, 2022). Prior research shows that engagement with patterns strengthens algorithmic thinking and helps bridge the gap between theory and practice (Proulx, 2000). Despite these advantages, programming patterns are rarely addressed explicitly in introductory programming courses (Weinman et al., 2021).

Few studies have investigated how novices acquire and apply programming patterns across different stages of learning. In one such study, Nurollahian et al. (2024) compared CS1 and CS2 students and found that although more advanced students produced better implementations, their ability to recognise expert structures remained limited. This indicates that implementation proficiency does not necessarily align with recognition skills and highlights the need for a broader understanding of how different dimensions of programming knowledge develop. Complementary perspectives, such as the work of Xie et al. (2019), emphasise the fact that instruction should explicitly and sequentially address distinct novice skills, moving from tracing and writing correct syntax to recognising and applying reusable templates. In this way, sequencing skills reduce cognitive load and improve both performance and conceptual understanding.

Within this broader context, pre-service computer science teachers represent a particularly under-researched population. They face a dual challenge: acquiring the ability to implement programming patterns effectively, and developing the competence to evaluate their quality, explain the underlying logic to learners and design instructional tasks that require their use. To the best of our knowledge, no study has systematically examined both the correctness and quality of programming pattern implementations among pre-service computer science teachers, nor has any study explored how implementation experience informs their ability to explain such patterns and apply them in design. The present study addresses this gap by analysing pre-service computer science teachers' ability to implement, recognise, explain and design with fundamental programming patterns.

Fundamental programming patterns in the present study

In order to study how future CS teachers develop conceptual programming knowledge, the present research focuses on the implementation and recognition of four fundamental programming patterns: linear search, guarded linear search, counting and extreme values (Astrachan et al., 1998; Astrachan & Wallingford, 1998). These four programming patterns represent fundamental algorithmic strategies that novice programmers encounter early in their learning process and offer insight into how students move from isolated constructs to structured problem-solving strategies.

Linear search sequentially examines elements in a collection to locate the first element that satisfies a given condition. It assumes that at least one matching element exists.

Guarded linear search is a variation of linear search that also considers the possibility of no element meeting the condition. It includes an explicit safeguard to handle the case where no match is found.

Counting traverses the entire collection, maintaining a counter that increments whenever an element satisfies the condition. The final value of the counter represents the total number of elements meeting the criterion.

Extreme values identify the most extreme element according to a defined criterion. During iteration, the variable storing the current extreme value represents the best candidate found so far and is updated whenever a more extreme element is found.

Research problem and questions

While programming patterns offer a structured approach to solving recurring algorithmic problems, it remains unclear how future CS teachers develop the ability to implement, recognise and explain these patterns. The present study explores how students' understanding of the target set of patterns progresses from implementation to conceptual comprehension. We examine the development of these skills across multiple years, highlighting a notable gap between pattern implementation and the ability to explain underlying logic. We also investigate how explanation skills relate to pedagogical abilities such as task design, which is critical for effective CS teaching. By tracing these cognitive developments, the study provides insights into how teacher education programmes can better prepare future educators to both apply programming solutions and convey the underlying concepts.

RQ1. To what extent do first- and second-year students accurately apply expected programming patterns when solving targeted tasks?
RQ2. How does the type of control structure (for vs. while loop) influence the correctness of students' implementations of linear and guarded linear search patterns?
RQ3. Do guarded search patterns present greater implementation challenges than unguarded patterns, regardless of loop type?
RQ4. How does students' experience with implementing programming patterns in earlier academic years influence their ability to recognise and explain the same patterns in subsequent years?
RQ5. What is the relationship between students' ability to recognise programming patterns and their ability to explain the conceptual functioning of these patterns, and how might this relationship inform pedagogical approaches in introductory programming education?
RQ6. To what extent does students' ability to conceptually explain programming patterns predict their capacity to design appropriate programming tasks that require the use of these patterns?

In order to address these questions, we relied on two primary data sources: students' programming submissions from the first and second academic year, and a mixed-method questionnaire administered in the third and fourth year. The programming submissions enabled analysis of implementation accuracy and the influence of control structures (RQ1–RQ3), while the questionnaire provided insights into students' recognition, explanation and task design abilities (RQ5–RQ6). Importantly, RQ4 was addressed through a combination of both data sources, linking earlier programming submissions with later questionnaire results to examine how prior implementation experience influenced subsequent recognition and explanation. Inspired by Robins (2019), this multi-stage design allowed us to trace the development of both implementation skills and conceptual understanding, revealing trajectories from surface-level strategies towards abstract, pattern-based reasoning. In doing so, the study seeks to connect novice programming performance with the growth of conceptual knowledge and to inform targeted instructional interventions in teacher education.

Method

This study employed non-experimental, multi-phase longitudinal educational research design.

Participants

A total of 70 students enrolled in a two-subject teacher programme with a specialisation in computer science and mathematics at the Faculty of Education, University of Ljubljana, participated in the study. The students, of which 23 were male and 47 female, were regularly enrolled in the 2020/21 and 2024/25 academic years. All of the students enrolled in the aforementioned programme during the specified period were included in the study. Participation was part of regular coursework, with the students providing informed consent for the anonymised use of their submissions and questionnaire responses. Prior to analysis, all of the data were anonymised. The study was approved by the Ethics Committee.

Instruments

Three instruments were employed to examine the students' knowledge and use of programming patterns across different stages of their study. Instrument 1 measured the students' ability to implement programming patterns in authentic tasks. Instrument 2 measured their ability to transfer the same patterns to a new context. Instrument 3 measured their ability to recognise and explain programming patterns from given code and to translate their understanding into didactic task design. All of the programming tasks and code snippets were written in Python, a programming language consistently used in the curriculum. Although the use of AI assistants could not be technically prevented for homework assignments, the students signed a declaration of non-use and oral defences were organised to verify independent work, a practice applied to cohorts since 2023, when generative AI tools became widely available.

Instrument 1 — Year 1 programming tasks

In the first year (the course Introduction to Programming), the students completed four tasks, each corresponding to one of four target patterns: linear search, guarded linear search, counting and extreme values. For linear search and guarded linear search, the students were required to provide two versions of each solution, one using a for loop and one using a while loop.

For the counting and extreme values patterns, any loop construct was permitted. The tasks were embedded in a context that used COVID-19 case data from Slovenia, making the data relevant to real-world problems encountered at the time. The assignments were administered as homework with one week for completion and were followed by a mandatory oral defence in which the students demonstrated their understanding of their own code. Two instructors, with 15 and over 30 years of teaching experience respectively, independently graded all of the submissions using a three-level rubric that distinguished between correct solutions, redundant solutions (i.e., functionally correct but unnecessarily complex or containing superfluous functionality) and incorrect solutions. The rubric was developed jointly in advance. Ratings were made independently and any disagreements were resolved in consensus discussions, thus ensuring the reliability of scoring.

Instrument 2 — Year 2 programming tasks

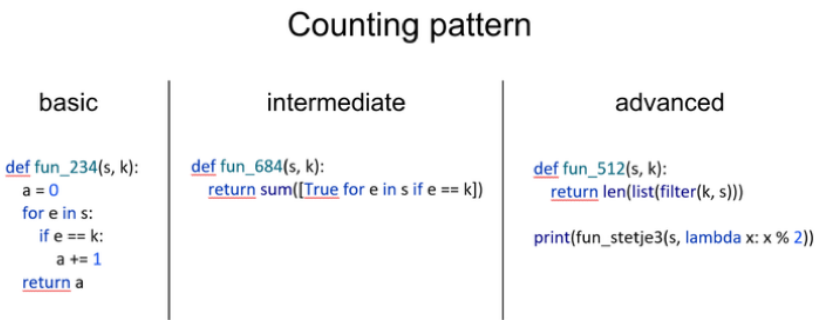
In the second year (the course Environments for CS Education), the students solved tasks that were isomorphic in algorithmic structure to those in Year 1 but embedded in a different context, namely the analysis of social media data. The same four patterns were targeted, and the same implementation requirements and rubric were applied. Evaluation followed the same procedure as in Year 1, with two experienced instructors grading independently and then resolving disagreements by consensus.

Instrument 3 — Year 3/4 questionnaire

In the third and fourth years, the students completed a paper-based questionnaire within a 45-minute classroom session. The questionnaire consisted of three parts. In the first part, the students were presented with twelve Python code snippets, three for each of the four patterns, equally distributed across basic, intermediate and advanced levels of difficulty. Figure 1 shows an example item for the counting pattern from the questionnaire. For each snippet, the students had to identify which programming pattern was implemented. In the second part, they provided written explanations of how the same twelve snippets worked. Their explanations were scored as correct, partially correct, incorrect or vague/over-general (answers that were broadly correct but too superficial to capture the actual mechanism). In the third part, the students selected one of the four patterns and designed a programming task suitable for primary school students. These tasks were scored as adequate or inadequate; in addition, for adequate tasks, the raters noted qualitatively whether the task was original or generic. As with the programming tasks in Year 1 and 2, the two

instructors first developed criteria together, then graded independently, and finally discussed discrepancies until consensus was reached.

Figure 1
Example questionnaire item for the counting pattern in Instrument 3. The students were asked to identify the programming pattern and explain the code



Research design

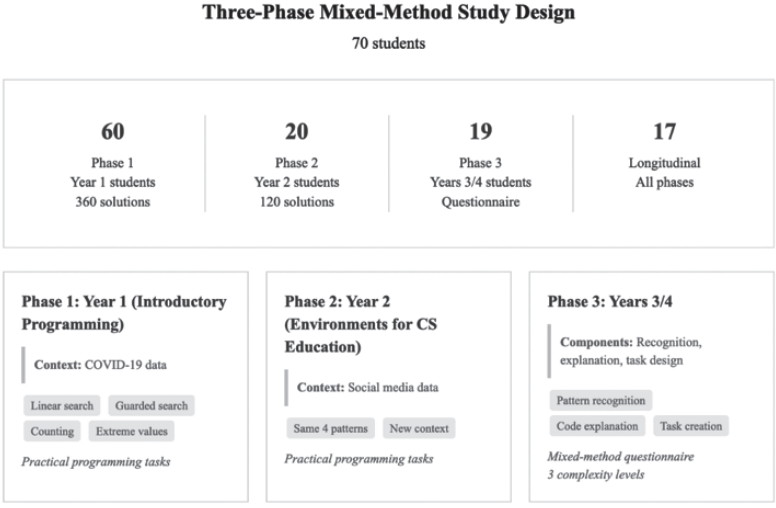
The study employed a mixed design, combining a longitudinal component with cross-sectional cohorts. This approach was necessary because the research instruments were embedded in different courses across the curriculum, which are taught in separate study years. The design thus enabled both the tracking of individual progress across multiple years and the comparison of different student cohorts. The research was carried out during the 2020/21 and 2024/25 academic years and was organised into three distinct phases (Figure 2), each with a specific aim.

- Phase 1 (Year 1, Introduction to Programming) included a total of 60 students across the target academic years and provided 360 solutions.
- Phase 2 (Year 2, Environments for CS Education) included 20 students across the target academic years and provided 120 solutions.
- Phase 3 (Years 3 and 4, questionnaire) completed by 19 students.

Within this overall sample, a longitudinal subsample of 17 students participated in all three phases, providing the opportunity to observe individual developmental trajectories over several years. At the same time, the inclusion of different cohorts across study years allowed for broader cross-sectional comparisons. This combination of longitudinal and cross-sectional data was a

deliberate methodological choice. It reflects the curricular structure, in which relevant instruments could only be administered in specific courses, while at the same time ensuring both developmental insights into individual students' learning and broader cohort-level comparisons.

Figure 2
Three-phase mixed-method study design showing participant distribution and data collection procedures across academic years



Results and discussion

This section presents the findings of our longitudinal study, structured around the six research questions (RQ1–RQ6), and interprets them in the context of existing research and pedagogical implications. Quantitative and qualitative data are combined to highlight the students' progression in implementing, recognising and explaining programming patterns, as well as their development of pedagogical task design skills.

Pattern implementation accuracy (RQ1)

In order to address RQ1, we analysed the students' ability to implement target programming patterns within the assigned tasks. For search-related patterns, the students submitted both a for-loop and a while-loop version;

however, a pattern was considered correctly implemented if at least one solution was correct, reflecting our focus on conceptual understanding rather than syntactic form. Table 1 shows the distribution of correct, redundant and incorrect implementations for each pattern in the first and second research phase.

Table 1

Distribution of correct, redundant and incorrect solutions by programming pattern and academic year

Pattern	Correct			Redundant			Incorrect		
	P1 (%)	*p*	P2 (%)	P1 (%)	*p*	P2 (%)	P1 (%)	*p*	P2 (%)
Counting	88	.259	100	6	.640	0	6	.640	0
Linear search	75	.794	81	23	.907	19	2	1.000	0
Guarded search	70	.117	90	20	.426	10	9	.335	0
Extreme values	69	1.000	72	3	1.000	0	28	1.000	28

Overall, across all of the programming tasks, the percentage of correct implementations increased from 64% in Phase 1 to 79% in Phase 2, while redundant solutions decreased from 18% to 11% and incorrect implementations declined from 18% to 10%. However, this improvement was not uniform. While linear and additive patterns (counting, linear search) and conditional patterns such as guarded search showed clear improvements, the extreme values pattern remained persistently challenging.

It should be noted, however, that none of these improvements reached statistical significance (all $p > .05$), most likely due to the small sample size in Phase 2 ($n = 20$). Therefore, the observed changes should be interpreted as trends rather than conclusive differences. Even with this limitation, the overall pattern suggests that instructional scaffolding may be particularly needed for state-tracking patterns, including both variable initialisation and the continuous updating of values (e.g., extreme values).

Since our descriptive results suggested that some programming patterns were more difficult than others, it was important to test whether these differences were statistically significant. Therefore, we conducted pairwise χ^2 tests of correct versus not-correct implementations between patterns within Phase 1, where the larger sample size ($n = 60$) provides sufficient basis for comparison. Table 2 presents the resulting p-values in a matrix format. Each cell indicates whether the difference in the proportion of correct implementations between two patterns was statistically significant.

Table 2

Pairwise χ^2 tests of correct implementation rates in Phase 1, showing that the counting pattern was significantly easier than the guarded linear search and extreme patterns, while other differences were not significant

Pattern	Counting	Linear	Guarded search	Extreme values
Counting	---	.113	0.030	0.019
Linear search		---	0.692	0.555
Guarded search			---	1.000
Extreme values				---

The results show that the counting pattern was significantly easier than the guarded search (p = .030) and extreme values patterns (p = .019), confirming that these patterns posed greater cognitive challenges, which is consistent with the findings of Astrachan and Wallingford (1998). The difference between counting and linear search was not statistically significant (p = .113), indicating that both patterns were comparably accessible to students. Linear search, guarded search and extreme values patterns were statistically indistinguishable from each other (all $p \geq .555$), which supports the interpretation that conditional and state-tracking patterns are consistently more difficult than additive patterns, thus aligning with the results of Lahtinen et al. (2005). This statistical evidence strengthens our earlier descriptive claim that not all programming patterns are equally accessible, and that state-tracking tasks such as extreme values and conditional tasks such as guarded search require additional instructional scaffolding (Astrachan & Wallingford, 1998; Fisler, 2014).

Influence of loop construct on pattern implementation accuracy (RQ2)

In order to address RQ2, we examined whether the type of loop construct (for vs. while) influenced the correctness of the students' implementations of linear and guarded search patterns. These patterns were selected because both involve iteration and were explicitly assigned with both loop types. Novice programmers often approach for and while loops differently: for loops offer structured iteration, whereas while loops require explicit control of loop variables and conditions (Soloway, 1986). By analysing the distribution of correct, redundant and incorrect solutions by loop type, we aimed to identify potential cognitive or syntactic challenges associated with each construct and how these evolved over time.

Table 3

Distribution of correct, redundant and incorrect solutions by loop type, pattern and phase. Patterns are labelled with (F) for implementations using a for loop and (W) for implementations using a while loop

Pattern	Correct			Redundant			Incorrect		
	P1 (%)	*p*	P2 (%)	P1 (%)	*p*	P2 (%)	P1 (%)	*p*	P2 (%)
Linear search (F)	73	1.000	76	23	1.000	24	3	1.000	0
Linear search (W)	66	.963	62	16	.148	33	19	.232	5
Guarded search (F)	58	.940	62	27	.393	14	16	.600	24
Guarded search (W)	63	.196	81	11	.982	14	27	.070	5

Table 3 shows that the students' performance differed when using for versus while loops to implement linear and guarded search patterns, with distinctions in correctness, redundancy and errors across both research phases. For the linear search pattern, for loops were consistently more accurate: 73% correct in Phase 1 vs. 66% with while, and 76% vs. 62% in Phase 2. While loop solutions became more verbose over time: redundancies rose from 16% to 33% and errors declined from 19% to 5%. This suggests growing caution with while loops, possibly to avoid errors, whereas for loop solutions remained stable and efficient, with no errors by Phase 2.

Guarded search results were more nuanced. In Phase 1, correctness was low for both loop types (58% for, 63% while), with more errors in while (27%) than for (16%). By Phase 2, while loop performance had improved markedly (81% correct, 5% incorrect), whereas for loop correctness had increased only slightly (62%) and errors had risen to 24%. This suggests that the students developed a stronger conceptual grasp of while loops over time, particularly for more complex scenarios like guarded search, but reliance on for loops may have limited adaptability in handling edge cases.

While the differences observed did not reach statistical significance (all $p > .05$), the data provide suggestive evidence of distinct learning trajectories. For loops appeared to offer stable support for early mastery of simpler search tasks, minimising errors and redundancies, whereas while loops, although initially associated with higher error rates, seemed to encourage more deliberate reasoning in complex tasks as experience accumulated (Bonar & Soloway, 1983). Taken together, these tendencies emphasise the pedagogical value of engaging students with both loop types in varied problem contexts, as such exposure may strengthen their conceptual adaptability in algorithmic problem-solving.

Figure 3

Comparison of implementation correctness across loop types and phases, showing overall improvement in Phase 2, with while-based linear search shifting from incorrect to redundant solutions

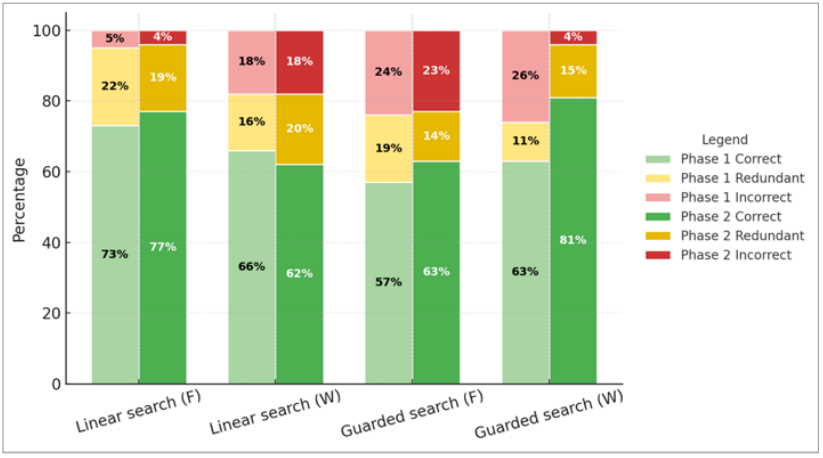


Figure 3 offers a visual comparison of implementation quality across loop types and research phases. It highlights the rise in correct implementations and the decline in incorrect ones, especially in Phase 2. Notably, in linear search with a while loop, incorrect solutions decrease while redundant ones increase, partly replacing correct implementations. This suggests that second-year students, who are still novice programmers, may adopt cautious or overly verbose coding strategies to avoid errors, which is consistent with research showing that novices often make small local fixes rather reformulating programs (Robins et al., 2003).

Learning complexity in search pattern implementation (RQ3)

Guarded search builds on basic linear search by adding control logic to handle cases where no match is found, increasing its cognitive complexity. In RQ3, we examine whether this added complexity leads to systematically lower implementation accuracy compared to linear search. The focus is on whether the pattern's logic – not its syntax – poses a greater challenge for novice programmers. In order to isolate the effect of pattern complexity from loop syntax (analysed in RQ2), we aggregated performance data across both for and while implementations. Table 4 presents the average proportions of correct, redundant and incorrect implementations for each pattern across the two research

phases, calculated by averaging the results from both loop types for each pattern-phase combination.

$$Linear_{year} = \frac{(Correct_{for} + Correct_{while})}{2}$$

This approach was used because all of the students implemented both loop variants of each pattern. Averaging across loop types allowed us to assess whether the students consistently struggled more with the logic of guarded search, particularly in terms of incorrect or structurally flawed solutions. The comparison highlights differences in the cognitive demands of each pattern, offering a more pattern-focused view of the students' algorithmic understanding.

Table 4

Average distribution of solution types for linear and guarded search patterns across loop constructs

	Correct			Redundant			Incorrect		
Pattern	**P1 (%)**	***p***	**P2 (%)**	**P1 (%)**	***p***	**P2 (%)**	**P1 (%)**	***p***	**P2 (%)**
Linear search	70	1.000	70	20	0.324	28	10	0.069	2
Guarded search	60	0.213	71	19	0.648	14	21	0.377	15

The data suggests that guarded search posed greater implementation challenges than linear search, regardless of loop type. In Phase 1, average correctness for guarded search was 60%, compared to 70% for linear search, with nearly twice as many incorrect solutions (21% vs. 10%). In Phase 2, correctness for linear search remained at 70%, whereas guarded search rose to 71%; incorrect implementations were still more frequent for guarded search than for linear search (15% vs. 2%). However, none of the Phase 1 vs. Phase 2 differences reached statistical significance with the available sample sizes (χ^2 tests, all $p \geq .318$; see Table 4). Figure 4 provides a visual representation of these tendencies: the students continued to struggle more with guarded search logic, particularly with formulating guarding conditions that prevent out-of-bounds checks or premature exit. Redundant or semantically inconsistent checks were also more common in guarded implementations in Phase 1. These patterns align with Lister et al. (2004), who emphasised that guarding conditions introduce additional complexity and require reasoning about collection bounds and logical safety.

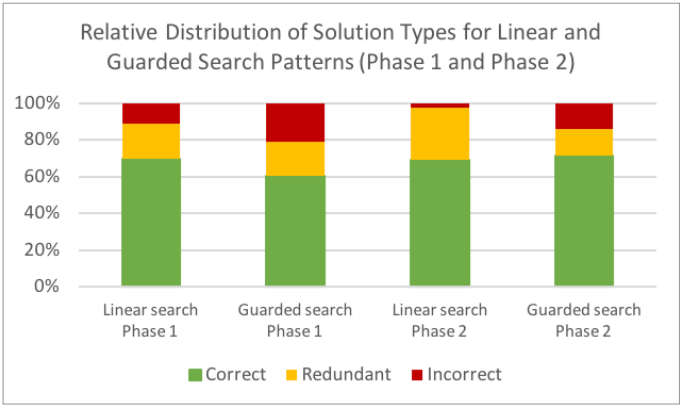
At the same time, Phase 2 shows a higher rate of redundancy in linear search than in guarded search (28% vs. 14%). Although this difference was not

statistically significant, it is pedagogically noteworthy and may reflect several mechanisms: increased caution regarding edge cases, leading to overly defensive coding practices; over-generalisation of guarding strategies to contexts where a simple linear search would suffice; and assessment practices that emphasise functional correctness over structural simplicity. Taken together, the results should be regarded as indicative rather than conclusive: they suggest that experience may reduce severe errors in linear search, whereas guarded search continues to impose higher cognitive demands; at the same time, redundancy in linear search appears to persist, or even increase, unless explicitly addressed.

Given these findings, we propose that instructional support should focus on both issues: providing scaffolding for reasoning about guarding conditions and collection bounds, thereby addressing the residual logic-safety challenges in guarded search highlighted by Lister et al. (2004); and explicitly emphasising simplicity and avoidance of redundancy in linear search, through approaches such as contrastive examples, refactoring activities and rubrics that reward clarity and minimal yet sufficient checks. This dual focus acknowledges that correctness in guarded search appears to improve naturally with experience, whereas redundancy in linear search tends to persist or even increase unless it is deliberately incorporated into instruction and assessment.

Figure 4

Guarded linear search remains more error-prone than linear search despite improvements in Phase 2, while redundancy in linear search increased



Linking implementation to pattern recognition and explanation (RQ4)

In the third study phase (Phase 3: Year 3 or 4), we assessed how prior implementation experience influenced the students' ability to recognise and explain programming patterns. The students completed a questionnaire with Python snippets representing the four target patterns at three complexity levels – basic, intermediate and advanced – where complexity reflected syntactic structure and expression. They identified the pattern and explained its implementation.

Table 5

Proportion of incorrect responses in programming pattern recognition across difficulty levels

Programming Pattern	Task Difficulty	Incorrect
	B/I/A	%
Linear search	Basic	18
	Intermediate	35
	Advanced	30
Guarded search	Basic	18
	Intermediate	12
	Advanced	29
Counting	Basic	0
	Intermediate	6
	Advanced	6
Extreme values	Basic	6
	Intermediate	24
	Advanced	12

Recognition accuracy varied by pattern and difficulty level (Table 5). The counting pattern was recognised consistently well across all levels (94–100%), indicating strong conceptual understanding and robustness to syntactic complexity. In contrast, recognition accuracy declined for linear search (82% to 65%) and guarded search (88% to 71%) from basic to intermediate levels, suggesting that more complex representations hinder abstraction and generalisation.

We computed Pearson correlations between implementation scores in Phase 1 (including both correct and redundant solutions) and recognition accuracy. As shown in Table 6, a significant correlation was found for counting

at the intermediate level (r = .685, p = .002), indicating a strong connection between practical experience and conceptual recognition. Other patterns showed weak to moderate correlations, with no significant results except a borderline effect for basic-level linear search (r = .471, p = .056).

Table 6

Correlations between implementation success and pattern recognition accuracy across task levels

Pattern	Task Level	Stat. Sig.	*Pearson r*	*p-value*
	B/I/A	bor./yes/no		
Linear search	Basic	borderline	0.471	0.056
	Intermediate	no	-0.119	0.648
Guarded search	Advanced	no	-0.054	0.838
	Basic	no	-0.084	0.747
	Intermediate	no	0.345	0.175
Counting	Advanced	no	0.173	0.506
	Basic	—	nan	nan
	Intermediate	yes	0.685	0.002
Extreme values	Advanced	no	-0.091	0.728
	Basic	no	0.139	0.596
	Intermediate	no	-0.019	0.942
	Advanced	no	0.203	0.436

In order to test whether prior implementation predicted later recognition accuracy, we fitted a generalised linear mixed model (GLMM) with a binomial distribution and logit link. The dependent variable (RECOG_BIN) indicated correct pattern recognition (0 = incorrect, 1 = correct) and fixed effects included the number of successful prior implementations (IMP_TOTAL_OK), pattern type (PATTERN) and task difficulty (LEVEL) with a random intercept for each student (id). As shown in Table 7, the model was significant (F(6, 209) = 2.17, p = .047), indicating that the predictors contributed to recognition performance. However, implementation history did not significantly predict recognition (F(1, 209) = 1.27, p = .261). Only pattern type was a significant fixed effect (F(3, 209) = 3.11, p = .028), while task difficulty was not (p = .249). Nurollahian et al. (2024) similarly found that although CS2 students outperformed CS1 students overall, more than 30% still misidentified expert patterns, performing no better than CS1 students in these cases. This indicates that while experience improves implementation performance, significant difficulties in

pattern recognition persist, highlighting the need for instructional approaches that explicitly support recognition and explanation.

Table 7

GLMM with binomial distribution and logit link; random intercept for subject (id); dependent variable: RECOG_BIN

Predictor	*F*	*df*	*p-value*
IMP_TOTAL_OK	1.27	1.209	.261
PATTERN	3.11	3.209	.028
LEVEL	1.40	2.209	.249
Model overall	2.17	6.209	.047

In order to further explore the impact of prior implementation on recognition accuracy, a refined GLMM was constructed using a binomial distribution with a logit link and a random intercept for each student. Fixed effects included successful implementations of GLS_FOR (Phase 1 and 2), COUNT (Phase 1), EXTREM (Phase 1) and GLS_WHILE (Phase 1). These variables were selected based on theoretical relevance to the recognition tasks and empirical trends observed in earlier models. Variables related to LS and Year 2 implementations of COUNT, EXTREM and GLS_WHILE were excluded due to low variance, convergence issues or lack of contribution to model fit.

As shown in Table 8, the model approached statistical significance overall ($F(5, 198) = 2.11$, $p = .066$), suggesting that the predictors jointly contributed to recognition performance. Significant predictors included GLS_FOR in Phase 2 ($F = 5.58$, $p = .019$), COUNT in Phase 1 ($F = 5.30$, $p = .022$) and EXTREM in Phase 1 ($F = 5.86$, $p = .016$). GLS_FOR in Phase 1 showed a marginal effect ($p = .065$), while GLS_WHILE (Phase 1) was not significant ($p = .192$).

Table 8

Fixed effects estimates in GLMM for pattern recognition accuracy

Predictor	*F*	*df1*	*df2*	*p-value*
GLS_FOR_P1	3.46	1	198	0.065
GLS_FOR_P2	5.58	1	198	0.019
COUNT_P1	5.3	1	198	0.022
EXTREM_P1	5.86	1	198	0.016
GLS_WHILE_P1	1.71	1	198	0.192
Model overall	2.11	5	198	0.066

In order to profile the students and subsequently offer tailored support, we conducted k-means clustering (Ikotun et al., 2023; Kodinariya & Makwana, 2013; Omar et al., 2020) based on the quality of programming pattern implementation in both phases and the ability to recognise and explain patterns. We evaluated implementation quality using a three-level rubric for each programming task solution: correct – accurate implementation of the required pattern; redundant – functionally correct but inefficient or unnecessarily complex solution; incorrect – failed implementation of the pattern. For each student, we then calculated the overall proportion of correct implementations across all four target patterns (linear search, guarded search, counting, extreme values) in the first and second research phase. This aggregated implementation success indicator (normalised to a 0–1 scale) was used as the basis for clustering students with the k-means algorithm.

Figure 5

Performance profiles of three student clusters: implementation quality in Phase 1 and Phase 2, pattern recognition, explanation and recognition-explanation gaps

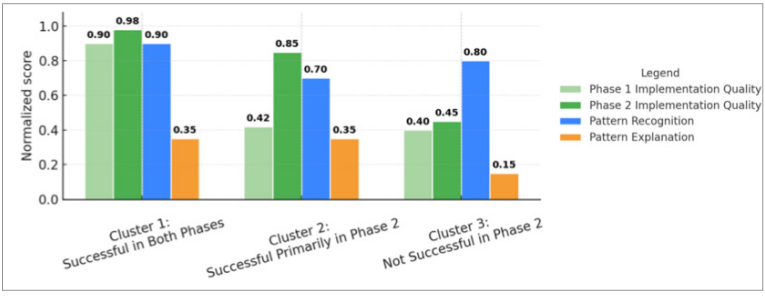


Figure 5 displays normalised average values (on a scale from 0 to 1) for different dimensions of programming pattern knowledge. The green bars indicate implementation quality in Phase 1 (light green) and Phase 2 (dark green). The blue bar (Pattern Recognition) shows the proportion of correctly recognised patterns in code analysis, assessed in Phase 3. The orange bar (Pattern Explanation) indicates the ability to verbally explain how patterns work, also assessed in Phase 3. Higher values indicate stronger performance across that dimension. The through k-means clustering. The consistent gap between recognition and explanation (blue vs. orange bars) across all three clusters further highlights the distinction between surface-level and conceptual understanding of programming patterns:

- Cluster 1 (Successful in Both Phases, $n = 8$): The students in this cluster achieved high implementation quality in both Phase 1 (0.90) and Phase

2 (0.98), and demonstrated strong pattern recognition ability (0.90, or 90%) but low explanation ability (0.35, or 35%). The normalised values represent the proportion of success on each dimension, scaled from 0 (0% success) to 1 (100% success). This indicates a substantial gap (0.55, or 55%) between what the students can recognise in code and what they can articulate in words.

- Cluster 2 (Successful Primarily in Phase 2, $n = 4$): These students showed lower implementation quality in Phase 1 (0.42) but improved substantially in Phase 2 (0.85). They demonstrated good pattern recognition ability (0.70, or 70%) and moderate explanation ability (0.35, or 35%), with the smallest gap between recognition and explanation (0.35, or 35%).
- Cluster 3 (Not Successful in Phase 2, $n = 5$): Despite weak implementation quality in Phase 1 (0.40) and limited improvement in Phase 2 (0.45), the students showed decent pattern recognition ability (0.80, or 80%) but very poor explanation ability (0.15, or 15%), resulting in the largest gap between recognition and explanation (0.65, or 65%).

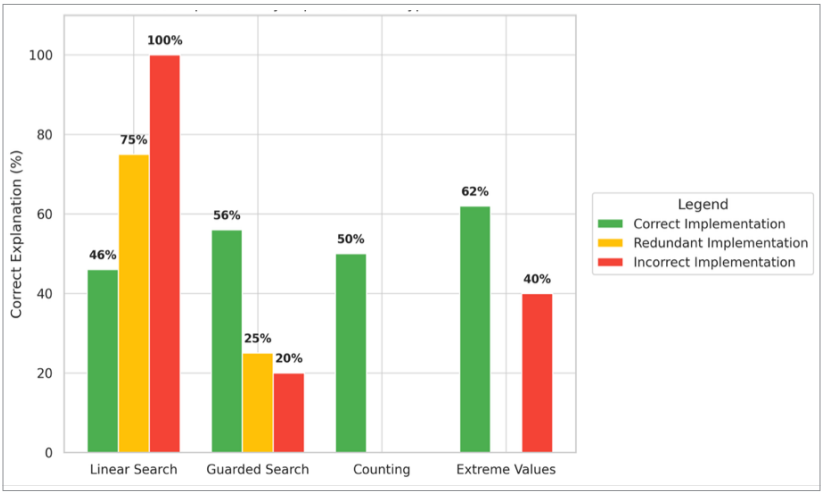
A key finding of our study is the universal presence of a gap between pattern recognition and explanation abilities across all of the student groups (Figure 5, difference between blue and orange bars). This suggests that implementation experience gained in Phases 1 and 2 alone is insufficient for developing the ability to articulate conceptual understanding. Even the students with excellent implementation skills in the earlier phases often lacked the corresponding abilities to explain the operation of these same patterns when assessed in Phase 3.

In order to examine the relationship between implementation and explanation ability, we computed Spearman rank correlations based on the students' combined implementation scores from Phase 1 (Introduction to Programming) and Phase 2 (Environments for CS Education). Implementations were rated as correct, redundant or incorrect; explanations (assessed in Phase 3) were scored from 0 to 1. Due to the ordinal and non-normal nature of the data, Spearman's ρ was used. The overall correlation was negligible ($\rho = 0.019$), indicating that code-writing proficiency and the ability to explain a solution function are largely independent skills, which is consistent with Lister et al. (2006).

As shown in Figure 6, each pattern displayed a unique implementation-explanation relationship; for example, in linear search, the students with flawed code provided better explanations than those with correct implementations. In contrast, only correct implementers explained the counting pattern successfully, although even here success was limited. Explanation accuracy rarely exceeded 60%, reinforcing the distinction between implementation accuracy and conceptual clarity.

Figure 6

Proportion of correct explanations by pattern type and implementation quality



These findings reveal a gap between procedural and conceptual knowledge: implementation competence does not necessarily translate into explanatory ability. While the students' code improved in Phase 2, explanation was assessed separately in Phase 3, so conceptual gains cannot be directly linked to implementation performance. As shown in Figure 6, explanation success varied by pattern but did not follow a consistent trend. In linear search, the students with redundant or even incorrect implementations explained the solution more successfully than those with correct code. In contrast, only the students with correct implementations explained the counting pattern successfully, although overall success remained modest even here. These results support earlier findings that novice programmers often struggle to move beyond surface-level understanding (Lister et al., 2006), thus highlighting the need for pedagogy that explicitly supports both coding accuracy and conceptual articulation.

Recognition-explanation disparity in programming pattern pedagogy (RQ5)

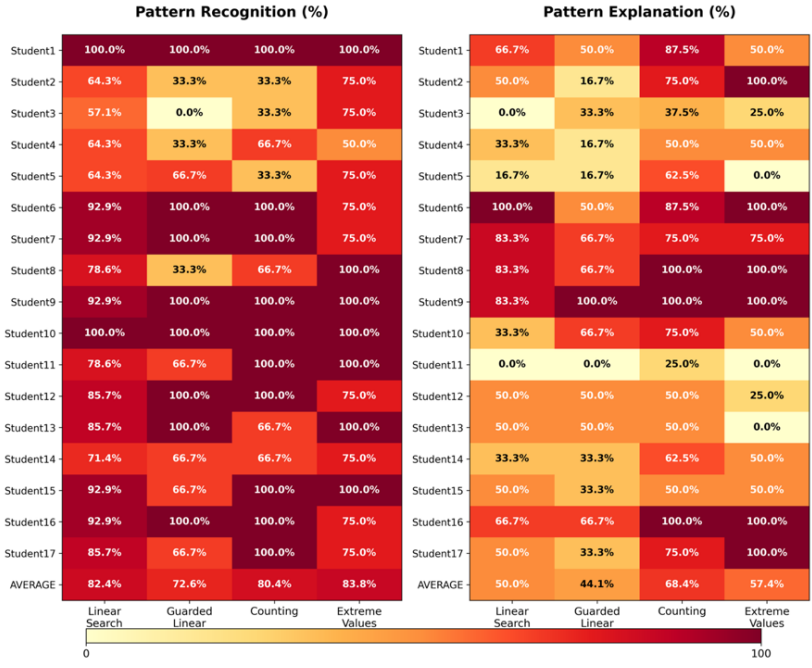
This section examines the relationship between the students' recognition and explanation of programming patterns, and its pedagogical implications in introductory CS education. Figure 7 displays shows recognition (left) and explanation (right) performance for 17 students across four programming patterns. The scores were computed as follows:

$$Recognition\_accuracy = \frac{Num\_of\_corr\_recognized\_patterns}{Num\_of\_tasks\_where\_that\_pattern\_was\_expected}$$

$$Explanation\_performance = \frac{(Num\_of\_corr\_explained\_patterns) + 0{,}5 * (Num\_of\_partially\_corr\_explained\_patterns)}{Num\_of\_tasks\_where\_that\_patt\_was\_expected}$$

Figure 7

Heatmap of programming pattern recognition accuracy and explanation performance among pre-service CS teachers



Our analysis reveals a consistent gap between the students' recognition and explanation abilities. Recognition ranged from 72.6% to 83.8%, while explanation lagged behind at 44.1% to 68.4%, indicating a 25–30% gap. This finding aligns with cognitive theory distinguishing recognition as a lower-level process and explanation as conceptually demanding (Fuller et al., 2007).

Linear search showed the largest disparity (82.4% vs. 50.0%), which is notable given its central role in novice instruction. Guarded search followed (28.4%), suggesting that error-handling increases conceptual demands. Extreme values (26.5%) similarly revealed challenges in explaining updating logic. By contrast, counting exhibited the smallest gap (12.1%), indicating its potential

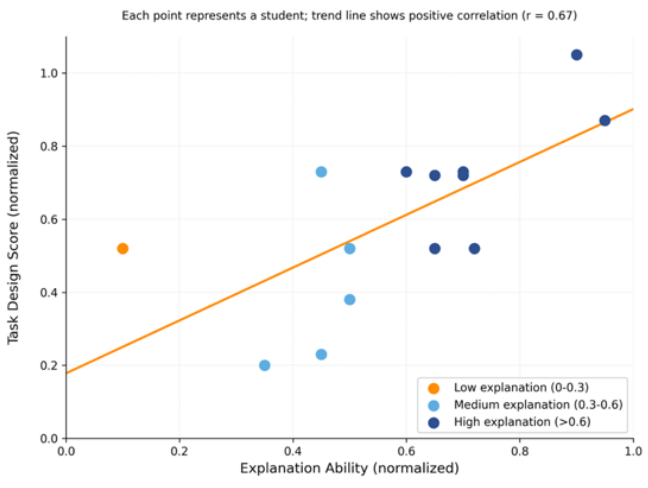
as a pedagogical entry point. This recognition-explanation gap aligns with Lister et al., 2006, who found that novice programmers typically provide multi-structural responses (line-by-line descriptions), while experts offer relational responses that capture the code's overall purpose. Our findings extend this phenomenon to programming patterns, where students can identify structures but struggle to explain their conceptual foundations, which is a critical limitation for future educators. These results highlight the importance of explicitly developing explanation skills in CS teacher education. As highlighted in a study by Soloway (1986), the "explanations" component of programming knowledge appears considerably more difficult to develop than the "mechanisms" component. Our findings empirically demonstrate that novices may recognise patterns but struggle to articulate their underlying mechanisms. Bridging this gap is essential for preparing teachers who can communicate not only what a pattern does but also how and why.

Pattern explanation ability as a predictor of task design quality (RQ6)

We tested whether explanation ability was correlated with the students' task design performance using correlation analysis. The results revealed a strong positive relationship ($r = 0.71$; Figure 8), supporting the role of conceptual clarity in pedagogical competence.

Figure 8

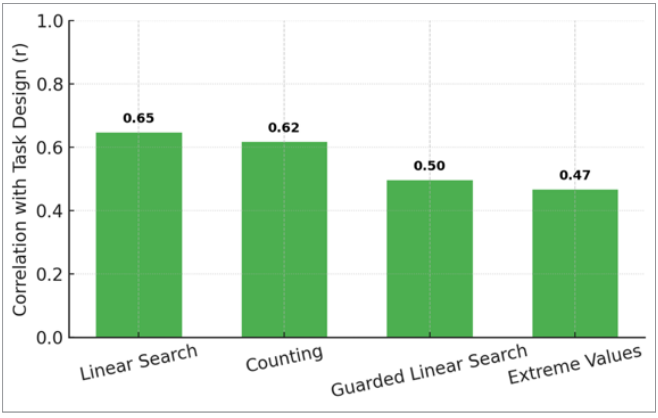
Relationship between explanation ability and task design quality



Pattern-specific correlations (Figure 9) show that the ability to explain **linear search** (r = 0.65) and **counting** (r = 0.62) most strongly correlates with task design quality, while **guarded search** (r = 0.50) and **extreme values** (r = 0.47) have moderate effects. This suggests that explanatory mastery of foundational patterns supports broader instructional skill more than explanations of complex cases.

Figure 9

Pattern-specific correlations (r) between explanation and task design performance



The results underscore the importance of fostering not only procedural fluency in implementing patterns but also the ability to explain them, as such explanatory clarity translates directly into higher-quality pedagogical task design. The pattern-specific correlations suggest that the ability to articulate how and why foundational patterns work is particularly important for pedagogical competence. This aligns with cognitive theory distinguishing recognition as a lower-level process and explanation as conceptually demanding (Lister et al., 2006; Venables et al., 2009), underscoring the need to prioritise explanation skills in CS teacher education (Nurollahian et al., 2025; Simon & Snowdon, 2011).

Discussion

The results of our longitudinal study highlight the multifaceted development of conceptual knowledge of programming patterns among pre-service computer science teachers. Our first research question examined how successfully students implement different programming patterns and whether this

success improves with experience. The analysis showed that the proportion of correct implementations increased from 64% in the first phase to 79% in the second, indicating progress, albeit unevenly across patterns. Students encountered the fewest difficulties with counting, confirming the findings of Lahtinen et al. (2005) that additive patterns are more accessible and serve as an entry point in developing conceptual knowledge. In contrast, patterns requiring initialisation and continuous updating of variables and handling of boundary conditions – especially guarded search and extreme values – proved most problematic (Astrachan & Wallingford, 1998; Fisler, 2014). Our study adds a new contribution by showing that these challenges are also evident among pre-service teachers, a group not previously examined in detail.

The second research question investigated how the choice of loop construct (for versus while) influences pattern implementation. The results indicate that for loops supported early understanding and reduced errors in simpler patterns such as linear search, whereas while loops were initially associated with more errors. However, in more complex tasks such as guarded search, the greatest improvement was observed in while loops (63% to 81% correct solutions in Phase 2). This supports the findings of Bonar and Soloway (1983), Soloway (1986) and Soloway et al. (1983), all of whom noted that for loops provide structured support, whereas while loops require more conceptual control but ultimately foster deeper understanding. Pedagogically, this highlights the importance of exposing students to both loop types in varied contexts, as each type contributes differently to developing conceptual flexibility in algorithmic problem solving.

The third research question addressed whether guarded search is systematically more difficult than linear search due to its greater cognitive complexity. The results confirmed that students made more errors in guarded search, which is consistent with Lister et al. (2004). However, correctness improved with experience, even without explicit instructional support, suggesting that more complex patterns can become manageable over time. In contrast, linear search showed an increase in redundant solutions in Phase 2. This suggests that even in simpler patterns, students may persist in overly cautious or non-optimised strategies, either due to local line-by-line reasoning (Robins et al., 2003) or unsuccessful attempts to apply more advanced constructs. This finding is consistent with Xie et al. (2019), who emphasised that tracing, recognising templates and writing code with them are distinct skills acquired incrementally. Our contribution here is to show that while correctness increases with experience, optimisation may decline, revealing a need for instructional support specifically targeted at recognising and reducing redundancy in linear search.

The fourth research question examined whether prior implementation experience predicts later success in recognising and explaining patterns. Our analyses showed that this link was weak: GLMM results indicated that the type of pattern, rather than the history of successful implementations, was the main predictor. Similar to the findings of Nurollahian et al. (2024), our study showed that even students with more programming experience could still misidentify expert patterns, indicating persistent conceptual gaps. Cluster analyses confirmed that this disconnect between implementation experience and later understanding was evident across all of the student groups, including those who had performed well in earlier phases. This is particularly important for pre-service computer science teachers, who must be experts in both computing and pedagogy. Our study shows that successful implementation alone does not guarantee conceptual understanding, underscoring the need for deliberate instructional support in this area.

The fifth research question examined the relationship between recognition and explanation of programming patterns. The results showed a consistent gap: recognition success was relatively high (73–84%), while explanation lagged by 25–30 percentage points. The largest disparity occurred in linear search and the smallest in counting, indicating that more complex patterns place greater demands on conceptual understanding. This "recognition-explanation gap" supports the findings of Lister et al. (2006), who observed that novices often remain at a multistructural level of understanding, and Soloway (1986), who noted that explanatory knowledge develops much more slowly than procedural mechanisms. Our study extends these insights by focusing on pre-service teachers, for whom explanation ability is a critical professional competence. The findings show that recognition ability alone does not ensure explanatory ability. For teacher education, this implies that curricula must systematically include tasks that develop explanatory skills, as these are a core teaching competence. Research such as Weinman et al. (2021) further demonstrates that structured activities like faded Parson's problems can strengthen recognition and transfer without requiring major curricular reform, suggesting a promising approach for bridging this gap in teacher preparation.

The sixth research question explored the relationship between explanation ability and the quality of pedagogical task design. The results showed a strong positive correlation ($r = 0.71$), confirming that conceptual clarity directly supports pedagogical competence. The strongest associations were found for explanations of linear search and counting, suggesting that mastery of foundational patterns contributes more to high-quality task design than explanations of more complex cases. This aligns with research identifying explanation

as a more cognitively demanding process than recognition (Lister et al., 2006; Venables et al., 2009). Our study builds on this by demonstrating its direct significance for pre-service teachers: a successful teacher must not only have procedural skills in implementation but also be able to clearly and meaningfully explain how patterns work and, based on this, design effective learning tasks (Nurollahian et al., 2025; Simon & Snowdon, 2011).

Overall, our findings demonstrate that procedural knowledge and conceptual understanding of programming patterns do not develop in parallel, and one does not necessarily guarantee the other. Although the participating students improved in implementation with experience, gaps remained in explanation, optimisation and the transfer of knowledge to pedagogical contexts. Our contribution is to show that pre-service teacher education must deliberately foster both implementation and explanatory skills, as only their combination enables high-quality task design and effective teaching of programming.

Conclusion

This study examined how pre-service computer science teachers develop conceptual understanding of core programming patterns over time. Across multiple phases, our findings showed that implementation accuracy improved with experience, but not all patterns progressed equally: additive patterns such as counting were more accessible, while state-tracking and conditional patterns (guarded search, extreme values) remained challenging. Differences between for and while loops confirmed that both constructs support learning in distinct ways, with while loops ultimately fostering deeper understanding in more complex tasks. At the same time, linear search revealed persistent redundancy, underscoring the need to address not only correctness but also optimisation in instruction.

Importantly, our results highlight the fact that prior implementation success does not reliably predict later recognition or explanation of patterns. A consistent gap was observed between the participating students' ability to recognise patterns and their ability to explain them, and explanation ability was strongly correlated with the quality of pedagogical task design. These findings point to a critical implication for teacher education: pre-service teachers must not only be able to write and recognise correct code but also to articulate how and why patterns work, so that they can design effective learning tasks and clearly explain programming concepts to their students, competencies that are central to becoming successful computer science teachers.

Although limited by a small, context-specific sample, our study provides novel insights by focusing on pre-service teachers, a population rarely studied

in this context. This limitation highlights the need for replication with larger and more diverse cohorts, as well as for longitudinal studies that follow future teachers into classroom practice. Further research should also explore targeted interventions, such as explanation-focused training or the use of structured pattern-based activities, in order to determine whether they can reduce the recognition-explanation gap and strengthen pedagogical competence.

Ethical statement

The study was approved by the Ethics Committee of the Faculty of Education, University of Ljubljana.

Disclosure statement

The authors have no conflict of interest to declare.

References

Abesadze, S., & Nozadze, D. (2020). Make 21st century education: The importance of teaching programming in schools. *International Journal of Learning and Teaching*, 158–163. https://doi.org/10.18178/ijlt.6.3.158-163

Astrachan, O., Berry, G., Cox, L., & Mitchener, G. (1998). Design patterns: An essential component of CS curricula. In J. Lewis, J. Prey, D. Joyce, & J. Impagliazzo (Eds.), *SIGCSE '98: Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education* (pp. 153–160). Association for Computing Machinery. https://doi.org/https://doi.org/10.1145/273133.273182

Astrachan, O., & Wallingford, E. (1998, August). *Loop patterns*. Users.cs.duke.edu. http://www.cs.duke.edu/~ola/patterns/plopd/loops.html

Bonar, J., & Soloway, E. (1983). Uncovering principles of novice programming. In J. R. White, L. Landweber, A. Demers, & T. Teitelbaum (Eds.), *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '83* (pp. 10–13). Association for Computing Machinery. https://doi.org/10.1145/567067.567069

Bosco, A., Santiveri, N., & Tesconi, S. (2019). Digital making in educational projects. *Center for Educational Policy Studies Journal*, *9*(3), 51–73. https://doi.org/10.26529/cepsj.629

Fisler, K. (2014). The recurring rainfall problem. In Q. Cutts, B. Simon, & B. Dorn (Eds.), *Proceedings of the Tenth Annual Conference on International Computing Education Research* (pp. 35–42). Association for Computing Machinery. https://doi.org/10.1145/2632320.2632346

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T. L., Thompson, D. M., Riedesel, C., & Thompson, E. (2007). Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bulletin*, *39*(4), 152–170.

https://doi.org/10.1145/1345375.1345438

Ikotun, A. M., Ezugwu, A. E., Abualigah, L., Abuhaija, B., & Heming, J. (2023). K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data. *Information Sciences*, *622*, 178–210. https://doi.org/10.1016/j.ins.2022.11.139

Kiesler, N. (2022). Reviewing constructivist theories to help foster creativity in programming education. In *2022 IEEE Frontiers in Education Conference (FIE)* (pp. 1–5). IEEE. https://doi.org/10.1109/FIE56618.2022.9962699

Kodinariya, T. M., & Makwana, P. R. (2013). Review on determining number of cluster in K-means clustering. *International Journal of Advance Research in Computer Science and Management Studies*, *1*(6), 90–95. http://ijarcsms.com/docs/paper/volume1/issue6/V1I6-0015.pdf

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

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, *36*(4), 119–150. https://doi.org/10.1145/1041624.1041673

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees. In R. Davoli, M. Goldweber, & P. Salomoni (Eds.), *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 118–122). Association for Computing Machinery. https://doi.org/10.1145/1140124.1140157

Nurollahian, S., Keuning, H., & Wiese, E. (2025, February 16). Teaching well-structured code: A literature review of instructional approaches [Unpublished paper]. *Accepted to the 2025 IEEE/ACM 37th International Conference on Software Engineering Education and Training (CSEE&T)*.

Nurollahian, S., Rafferty, A. N., Brown, N., & Wiese, E. (2024). Growth in knowledge of programming patterns: A comparison study of CS1 vs. CS2 students. In B. Stephenson, J. A. Stone, L. Battestilli, S. A. Rebelsky, & L. Shoop (Eds.), *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (pp. 979–985). Association for Computing Machinery. https://doi.org/10.1145/3626252.3630865

Nurollahian, S., Rafferty, A. N., & Wiese, E. (2023). Improving assessment of programming pattern knowledge through code editing and revision. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (pp. 58–69). IEEE. https://doi.org/10.1109/ICSE-SEET58685.2023.00012

Omar, T., Alzahrani, A., & Zohdy, M. (2020). Clustering approach for analyzing the student's efficiency and performance based on data. *Journal of Data Analysis and Information Processing*, *08*(03), 171–182. https://doi.org/10.4236/jdaip.2020.83010

Proulx, V. K. (2000). Programming patterns and design patterns in the introductory computer science course. In S. Haller (Ed.), *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education* (pp. 80–84). Association for Computing Machinery. https://doi.org/10.1145/330908.331819

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

Robins, A. V. (2019). Novice programmers and introductory programming. In S. Fincher, & A. V. Robins (Eds.), *The Cambridge handbook of computing education research* (pp. 327–376). Cambridge University Press. https://doi.org/10.1017/9781108654555.013

Simon, & Snowdon, S. (2011). Explaining program code. In K. Sanders, M. E. Caspersen, & A. Clear (Eds.), *Proceedings of the Seventh International Workshop on Computing Education Research* (pp. 93–100). Association for Computing Machinery. https://doi.org/10.1145/2016911.2016931

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

Venables, A., Tan, G., & Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. In M. Clancy, M. Caspersen, & R. Lister (Eds.), *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (pp. 117–128). Association for Computing Machinery. https://doi.org/10.1145/1584322.1584336

Vivitsou, M. (2019). Digitalisation in education, allusions and references. *Center for Educational Policy Studies Journal*, *9*(3), 117–136. https://doi.org/10.26529/cepsj.706

Webb, M., Davis, N., Bell, T., Katz, Y. J., Reynolds, N., Chambers, D. P., & Sysło, M. M. (2017). Computer science in K-12 school curricula of the 2lst century: Why, what and when? *Education and Information Technologies*, *22*(2), 445–468. https://doi.org/10.1007/s10639-016-9493-x

Weinman, N., Fox, A., & Hearst, M. A. (2021). Improving instruction of programming patterns with Faded Parsons Problems. In Y. Kitamura, A. Quigley, K. Isbister, T. Igarashi, L. Bjørn, & S. Drucker (Eds.), *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (pp. 1–4). Association for Computing Machinery. https://doi.org/10.1145/3411764.3445228

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35. https://doi.org/10.1145/1118178.1118215

Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., Tan, A. H., Hwa, L., Li, M., & Ko, A. J. (2019). A theory of instruction for introductory programming skills. *Computer Science Education*, *29*(2–3), 205–253. https://doi.org/10.1080/08993408.2019.1565235

Yadav, A., Good, J., Voogt, J., & Fisser, P. (2017). Computational thinking as an emerging competence domain. In M. Mulder (Ed.), *Competence-based Vocational and Professional Education. Technical and Vocational Education and Training: Issues, Concerns and Prospects, vol 23* (pp. 1051–1067). Springer. https://doi.org/10.1007/978-3-319-41713-4_49

Zapušek, M. (2022). *Domenska ontologija programskih vzorcev pri uvodnem programiranju* [Domain ontology of programming patterns in introductory programming] [Doctoral dissertation, University of Ljubljana]. RUL. https://repozitorij.uni-lj.si/IzpisGradiva.php?id=136467

Zeng, Y., Yang, W., & Bautista, A. (2023). Teaching programming and computational thinking in early childhood education: A case study of content knowledge and pedagogical knowledge. *Frontiers in Psychology*, *14*. https://doi.org/10.3389/fpsyg.2023.1252718

Biographical note

Matej Zapušek, PhD, is an assistant professor in the field of computer science education at the Faculty of Education, University of Ljubljana, Slovenia. His research focuses on the didactics of introductory programming, with particular interest in programming patterns, formalizations of knowledge and domain ontologies that support deeper conceptual understanding. He also investigates the role of serious games and game design-based learning in education, as well as emerging opportunities and challenges related to the use of generative artificial intelligence in teaching and learning.

Irena Nančovska Šerbec, PhD, is an assistant professor in computer science education at the Faculty of Education, University of Ljubljana, Slovenia. Her work focuses on teaching introductory programming and fostering the development of computational thinking in early and primary education.

Her research interests include modelling in education, text-mining and learning analytics, and the development of teachers' digital and AI literacy, with particular attention to how data-driven approaches can support more effective teaching and learning.