IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 6, June 2026

DOI: 10.17148/IJARCCE.2026.15685

A Three-Level Framework for Programming Task Design in CS1: Bridging Conceptual Understanding and Transfer

Ahmed S. AlMahmeed

Department of Computer Science, Public Authority of Applied Education and Training (PAAET), Kuwait

Abstract: Introductory programming courses (CS1) often have high failure and dropout rates, signaling persistent challenges in mastering programming fundamentals. This paper synthesizes research from computing education, cognitive load theory, and transfer of learning to present a systematic framework for designing programming tasks at three levels: **understanding**, **application**, and **applying**. The framework aligns with modern Bloom's Taxonomy in computing, aiming to support curriculum development, assessment alignment, and effective teaching in CS1. The camera-ready version improves structure, arguments, and includes ACM-style citations for SIGCSE and TOCE submission.

CCS CONCEPTS

Social and professional topics → Computing education

Applied computing → Education → Interactive learning environments

Software and its engineering → Programming languages → General programming languages

- **Computing education:** Highlights the importance of effective teaching methods and curriculum in computer science.
- **Interactive learning environments:** Focuses on frameworks that foster active engagement and conceptual growth for students.
- **General programming languages:** Centers on core principles and broad programming skills, independent of language specifics.

Keywords: Bloom's Taxonomy, Cognitive, Problem-solving.

1. INTRODUCTION

Introductory programming courses are widely acknowledged as challenging gateway subjects within computer science curricula. Meta-analyses indicate failure rates ranging from 25% to 50% across various institutions and regions [2, 10]. These statistics underscore the necessity for enhanced pedagogical strategies, especially regarding the design and sequencing of programming tasks.

Many students face difficulties transitioning from merely understanding programming concepts to effectively applying them in unfamiliar or real-world scenarios. This paper introduces a research-based framework aimed at guiding the development of programming tasks that facilitate this progression. The approach is informed by empirical studies in computing education and cognitive psychology. The camera-ready version reinforces the theoretical basis and delineates the distinctions among the three levels of tasks.

2. RELATED WORKS

2.1 Novice Programmer Difficulties

Studies repeatedly highlight multiple obstacles that novice programmers face. These include confusion about variables, memory, and state, which can make it hard for learners to grasp how programs function and manage information. Beginners frequently find tracing control flow challenging, an important skill for understanding program outcomes and fixing code.

Misconceptions about variables, memory, and state [14]

Difficulty tracing control flow [8]

Cognitive overload from managing syntax, logic, and breaking down problems at once [16]

Limited problem-solving patterns [12]

Weak debugging techniques [5]

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 6, June 2026
DOI: 10.17148/IJARCCE.2026.15685

Overall, these results emphasize the need for tasks that purposefully structure cognitive demands, helping students build foundational programming skills and comprehension.

2.2 Task Types in Computing Education

Research highlights several effective tasks for novice programmers:

- Worked examples foster problem-solving and pattern recognition.
- Parsons's problems reduce syntax obstacles, promoting focus on logic.
- Code tracing sharpens mental models.
- Project-based learning increases motivation and real-world relevance.

However, many CS1 courses lack stepwise progression from concepts to practice, limiting these tasks' effectiveness for deep learning and transfer.

2.3 Bloom's Taxonomy for Computing

The ACM/IEEE Joint Task Force recommends using Bloom's levels for computing. Recent revisions differentiate between applying established patterns and adapting knowledge to new contexts:
Application: using known patterns
Applying: adapting knowledge to novel situations
This distinction is central to the paper's framework.

3. UNDERSTANDING INTRODUCTORY PROGRAMMING CONCEPTS

3.1 Nature of Understanding

Introductory programming is all about developing clear mental models of important concepts such as variables, memory, control flow, functions, modularity, and data structures. If students lack true understanding, they tend to depend on pattern matching, which can restrict their ability to tackle new problems or use their knowledge beyond simple tasks.

3.2 Tasks that Promote Understanding

Various research-backed activities have been shown to improve comprehension of basic programming ideas. These tasks help students build strong mental models and move past just recognizing concepts at a surface level.
Code tracing and prediction [9] encourage students to carefully track how code runs and predict its results, deepening their understanding of programming structures.
Conceptual multiple-choice questions [13] focus on core principles instead of syntax, pushing students to thoughtfully examine essential concepts.
Visualizations of program execution [15] employ diagrams and visual aids to show programming steps clearly, making abstract processes easier to grasp.
Participating in these activities helps students manage cognitive load and solidify vital mental models needed to master programming basics.

4. APPLICATIONS: STRUCTURED IMPLEMENTATION OF PROGRAMMING CONCEPTS

4.1 Definition

Application involves the utilization of programming constructs within established, guided settings. In these contexts, learners are not tasked with devising original solutions independently; rather, they apply their understanding according to frameworks that offer explicit direction and defined boundaries.

4.2 Effective Application Tasks

Effective tasks for supporting application include template-based exercises, pattern-oriented tasks, and guided function implementations.

- Template-based exercises provide partially completed code structures that students must fill in, helping them focus on essential logic without being overwhelmed by syntax or overall structure.
- Pattern-oriented tasks, such as accumulator loops, encourage students to recognize and apply common programming patterns to solve problems within a familiar context.
- Guided function implementations involve students writing functions according to detailed specifications or with step-by-step prompts, reinforcing their understanding while offering necessary support.

These tasks support schema acquisition and reduce extraneous cognitive load, making it easier for students to internalize programming concepts and develop fluency in their use.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 6, June 2026

DOI: 10.17148/IJARCCE.2026.15685

5. THREE-LEVEL FRAMEWORK FOR PROGRAMMING TASK DESIGN

5.1 Framework Overview

This three-level programming task framework offers a structured way to design tasks for learners at different skill levels. Each stage targets specific learning goals with relevant example tasks.

Level	Purpose	Example Task
Understanding	Conceptual clarity	Trace nested loops
Application	Guided practice	Implement a function with a template
Applying	Transfer & independence	Build a simple game or data tool

5.2 Scaffolding Strategies

Scaffolding strategies assist students in developing programming abilities by starting with structured guidance and gradually easing support as their confidence grows.

Main approaches include:

- Withdrawing help progressively
- Including debugging exercises
- Using automated feedback tools [6]
- Providing prompts for reflection to boost metacognitive skills

5.3 Assessment Alignment

Assessments should correspond to the framework, enabling accurate tracking of student progress in programming and covering all essential competencies. Important aspects to assess are:

- Understanding concepts
- Coding proficiency
- Ability to transfer knowledge and work independently

6. IMPLICATIONS FOR COMPUTING EDUCATION

6.1 Curriculum Design

Curriculum planners are encouraged to implement deliberate sequencing of programming tasks to facilitate students' cognitive development. Rather than assigning problem sets arbitrarily, educators should systematically organize tasks so that each build upon prior knowledge, guiding learners from basic skills to advanced, higher-order thinking. This structured approach enables students to acquire a thorough understanding of essential programming concepts and ensures they can effectively apply their learning throughout the curriculum.

6.2 Pedagogical Practice

Instructors should adopt evidence-based methods to enhance student achievement in programming courses. By integrating these strategies, educators foster an environment conducive to comprehension and skill acquisition.

Introduce visualizations and code tracing activities at the onset of the course to assist students in developing accurate mental models of program execution, thereby establishing a foundation for deeper understanding.

Provide structured practice exercises before transitioning to open-ended or more complex tasks, allowing students to build confidence and proficiency within a supportive framework.

Highlight the significance of debugging by embedding it as an integral skill throughout instruction and practice. Prioritizing debugging equips students with vital problem-solving competencies.

Encourage students to verbalize their reasoning and reflect on their problem-solving processes, which promote enhanced comprehension and metacognitive awareness, ultimately supporting the development of effective programmers.

6.3 Research Opportunities

There exist numerous promising directions for future research intended to advance programming education and strengthen our insight into effective student learning.

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 6, June 2026
DOI: 10.17148/IJARCCE.2026.15685

Study methods for automated classification of task difficulty to align assignments more closely with students' abilities and learning objectives.

Investigate the design and deployment of adaptive learning environments that customize instruction and feedback for individual learners, thereby increasing personalization.
Analyze the long-term effects of task sequencing on students' mastery, retention, and transferability of programming skills over time.

7. CONCLUSION

This research introduces a framework that integrates computing education, cognitive psychology, and instructional design to inform the development of programming tasks. The goal is to assist learners in progressing from grasping essential concepts, to applying their knowledge, and ultimately adapting their skills to unfamiliar situations. Organizing tasks according to these phases allows educators to more effectively support beginners in computer science courses, enhancing learning outcomes and establishing a strong basis for advanced study.

REFERENCES

[1]. ACM/IEEE Joint Task Force on Computing Curricula. 2013. *Computer Science Curricula 2013*. ACM Press.
[2]. J. Bennedsen and M. E. Caspersen. 2007. Failure rates in introductory programming. *SIGCSE Bull.* 39, 2 (2007), 32–36.
[3]. J. D. Bransford and D. L. Schwartz. 1999. Rethinking transfer. *Review of Research in Education* 24 (1999), 61–100.
[4]. M. E. Caspersen and J. Bennedsen. 2007. Instructional design of a programming course. In *SIGCSE '07*. 111–115.
[5]. S. Fitzgerald et al. 2008. Debugging: Finding, fixing and flailing. *Computer Science Education* 18, 2 (2008), 93–116.
[6]. P. Ihantola et al. 2015. Educational data mining and learning analytics in programming courses. *ACM Comput. Surv.* 48, 2 (2015), 1–39.
[7]. J. L. Kolodner et al. 2003. Problem-based learning meets case-based reasoning. *Interdisciplinary Journal of Problem-Based Learning* 1, 1 (2003), 21–39.
[8]. R. Lister et al. 2004. A multi-national study of reading and tracing skills in novice programmers. In *ITiCSE Working Group Reports*. 119–150.
[9]. R. Lister et al. 2006. Not seeing the forest for the trees. In *SIGCSE '06*. 118–122.
[10]. C. Watson and F. W. Li. 2014. Failure rates in introductory programming revisited. In *ITiCSE '14*. 39–44.
[11]. D. Parsons and P. Haden. 2006. Parsons programming puzzles. In *ACE '06*. 157–163.
[12]. A. Robins, J. Rountree, and N. Rountree. 2003. Learning and teaching programming. *Computer Science Education* 13, 2 (2003), 137–172.
[13]. S. Simon et al. 2006. Multiple-choice questions for assessing conceptual understanding in CS1. In *ACE '06*.
[14]. J. Sorva. 2013. Notional machines and novice programming. *Computer Science Education* 23, 2 (2013), 159–186.
[15]. J. Sorva, V. Karavirta, and L. Malmi. 2013. A review of program visualization tools. *ACM Trans. Comput. Educ.* 13, 4 (2013), 1–64.
[16]. J. Sweller. 2011. Cognitive load theory. *Psychology of Learning and Motivation* 55 (2011), 37–76.

