Enhancing Programming Skills Through an Innovative Interactive Learning Framework for Undergraduate Computer Science Students

Ms. Nivedita M H
Assistant Professor, Department of Computer Science
St Joesph's First Grade College, Mysuru
Email Id: niveditamh40@gmail.com

Abstract

Programming education requires students to develop logical reasoning, computational thinking, debugging ability, and independent problem-solving skills. However, traditional instructional approaches may not always provide sufficient opportunities for active participation and individualized feedback. This paper proposes an innovative interactive learning framework that combines micro-coding challenges, peer instruction, error-based learning, adaptive difficulty, and reflection-based assessment to improve programming education among undergraduate Computer Science students. The framework introduces a structured learning cycle in which students attempt programming problems, predict program outputs, identify errors, explain their reasoning, and independently apply the concepts to new problems. A classroom-based evaluation using pre-tests, post-tests, practical coding tasks, and student feedback is proposed. The framework aims to provide a low-cost, adaptable approach to programming instruction that can be implemented using existing computer laboratory facilities. Its effectiveness will be evaluated using actual classroom data.

Keywords: Programming Education, Interactive Learning, Computational Thinking, Error-Based Learning, Peer Instruction, Adaptive Learning, Undergraduate Computer Science.

1. Introduction

Programming is an essential component of undergraduate Computer Science education. It enables students to translate logical solutions into executable instructions while developing analytical thinking and computational problem-solving abilities.

Despite regular classroom instruction, students may encounter difficulties when applying theoretical concepts to practical programming tasks. Understanding syntax does not necessarily mean that a student can independently design an algorithm, trace program execution, identify logical errors, or develop a complete solution.

To address these challenges, this paper proposes an interactive learning framework that combines several instructional techniques into a single classroom process. Rather than relying exclusively on explanations and demonstrations, the framework encourages students to predict, practice, debug, collaborate, and reflect on their solutions.

The proposed approach is designed for undergraduate programming courses and does not require expensive software, specialized laboratory equipment, or advanced artificial intelligence systems.

2. Proposed Innovative Learning Framework

The proposed framework is named the **PRACTICE Framework**, representing seven connected stages of programming learning.

- **P — Predict:** Students predict the output of a program before executing it.
- **R — Resolve:** Students attempt a programming problem independently.
- **A — Analyze:** Students examine incorrect code and identify the source of errors.
- **C — Collaborate:** Students explain their approaches and compare solutions with peers.

- **T — Test:** Students execute and test their programs using different inputs.
- **I — Improve:** Students revise their solutions using feedback.
- **C — Create and reflect:** Students solve a related problem independently and reflect on what they learned.
- **E — Evaluate:** The instructor evaluates performance using a predefined rubric.

The stages form a repeatable learning cycle. The framework is a proposed instructional design in this paper, not an already validated educational model. Its usefulness must be established through classroom evaluation.

3. Innovative Teaching Strategies

3.1 Predict Before You Execute

Before executing a program, students write down the expected output and explain how they arrived at it.

For example, students may be shown a program containing a loop and asked to predict the output before running it.

This activity encourages students to trace program execution rather than depend entirely on the compiler. Differences between predicted and actual output can reveal misunderstandings about variables, conditions, and iteration.

Innovation: Record both the student's initial prediction and the final answer to examine changes in conceptual understanding.

3.2 Debugging Detective Challenge

The instructor presents a program containing one or more deliberate errors. Students must identify the error, explain why it occurs, and correct the program.

Students can be given different types of challenges:

- Syntax errors.
- Logical errors.
- Incorrect conditions.
- Loop boundary errors.
- Unexpected outputs.
- Incorrect handling of input values.

Innovation: Award learning points for correctly explaining an error, not merely producing the corrected code. This encourages deeper understanding of programming logic.

3.3 Peer Code Explanation

Students work in pairs, with one student explaining the algorithm and the other examining the logic, testing the code, and asking questions. Roles are then exchanged.

Innovation: Use a short explanation checklist that asks students to identify the purpose of the program, explain the algorithm, justify the conditions used, and describe how the program handles an unexpected input.

This activity makes students responsible for communicating and defending their reasoning.

3.4 Micro-Coding Challenges

Programming concepts are divided into short, achievable tasks that can be completed during a regular class.

For example, a lesson on loops may include the following challenges:

1. Print numbers from 1 to 10.
2. Display only even numbers.
3. Calculate the sum of the first ten natural numbers.
4. Count the digits in an integer.
5. Modify a program to handle an additional condition.

Innovation: Arrange the tasks from basic to more challenging, allowing students to progress after demonstrating understanding of the preceding task.

The purpose is to provide immediate practice without making students wait until the end of the lesson to apply a concept.

3.5 Adaptive Difficulty Ladder

Students are provided with problems at three levels:

- **Level 1 — Foundation:** Understand and complete a partially written program.
- **Level 2 — Application:** Write a program using a specified concept.
- **Level 3 — Challenge:** Solve a new problem that combines multiple concepts.

Students move to a higher level when they meet a predefined performance criterion.

Innovation: Maintain a simple progress record showing the concepts each student has mastered and the concepts requiring additional practice.

This can be implemented using a spreadsheet rather than specialised adaptive-learning software.

3.6 Real-World Programming Problems

Abstract programming exercises are connected to familiar situations.

Examples include:

- Calculating student grades using conditional statements.
- Preparing a shopping bill using arithmetic operations.
- Managing attendance records using arrays or lists.
- Calculating electricity charges using decision structures.
- Generating a simple student-result report using functions.

Innovation: Ask students to explain how the programming concept relates to the real-world problem and to propose at least one additional feature.

This approach helps students connect programming logic with practical applications.

3.7 Reflection-Based Error Journal

After each programming exercise, students record three items:

1. The error or difficulty they encountered.

2. The reason they believe it occurred.
3. The correction or strategy they used to solve it.

Innovation: Review these journals periodically to identify recurring class-wide difficulties and adapt subsequent lessons accordingly.

The journal can also provide qualitative evidence about students' debugging strategies and learning experiences.

3.8 Optional AI-Assisted Debugging

Where institutionally permitted, students may use a generative AI tool to request an explanation of a programming error. They must then verify the explanation by testing the code and describing the correction in their own words.

The instructor can compare students' independent debugging performance with their performance on supported tasks.

Innovation: Evaluate whether students can transfer what they learned from AI-assisted debugging to a new programming problem without AI assistance.

AI use should be disclosed in the research protocol, and student data should not be entered into external tools without appropriate privacy safeguards.

4. Research Objectives

The study aims to:

1. Evaluate changes in programming performance following implementation of the PRACTICE Framework.
2. Examine changes in students' ability to trace programs and identify programming errors.
3. Investigate whether structured peer explanations support students' conceptual understanding.
4. Assess the usefulness of progressive programming challenges in supporting learners with different skill levels.
5. Examine students' perceptions of confidence, participation, and independent problem-solving.
6. Identify implementation challenges and opportunities for improving the proposed framework.

5. Research Methodology

5.1 Research Design

A classroom-based intervention using pre-tests, post-tests, practical programming tasks, and student feedback is proposed.

Where feasible, the study may include a comparison group receiving the usual teaching approach. Both groups should cover comparable syllabus content and complete assessments of similar difficulty.

If only one class is available, a one-group pre-test and post-test design can be used, with the limitation that observed improvements cannot be attributed exclusively to the intervention.

5.2 Participants

The participants may be undergraduate BCA or Computer Science students enrolled in an introductory programming course.

The final paper should report the actual institution, number of participants, programming language, duration of the intervention, sampling procedure, and relevant student characteristics.

5.3 Duration

A four-week intervention is proposed, subject to the course timetable.

Week	Activities	Evidence collected
1	Baseline test, output prediction, basic coding challenges	Pre-test scores
2	Debugging detective tasks and peer code explanations	Error identification and explanation scores
3	Adaptive difficulty ladder and real-world problems	Task completion and coding rubric scores
4	Independent transfer problems, reflection, and post-test	Post-test scores and feedback

5.4 Assessment Measures

Programming performance can be evaluated using a common rubric covering:

- Algorithm design.
- Code correctness.
- Program tracing.
- Error identification and correction.
- Independent problem-solving.
- Explanation of programming logic.

The assessments should be designed so that the post-test measures application to new problems rather than simple memorisation of previously practised answers.

5.5 Data Analysis

The study may compare pre-test and post-test scores using appropriate statistical methods. If the design and assumptions permit, a paired-samples t-test may be used for the same students' scores; an appropriate alternative may be used when its assumptions are not met.

Additional analysis may include effect sizes, confidence intervals, task-level performance, and questionnaire summaries.

Student reflection journals may be analysed thematically to identify recurring difficulties and perceived benefits.

All numerical results must come from actual collected data.

6. Experimental Evaluation and Illustrative Results

To examine the potential effectiveness of the proposed PRACTICE Framework, a classroom-based evaluation is planned using pre-test and post-test assessments. The evaluation will focus on programming performance, program tracing, debugging ability, and independent problem-solving among undergraduate Computer Science students.

For illustration, a hypothetical comparison involving 60 students divided into an experimental group and a conventional-teaching group is presented in Table 1. The scores are assumed examples intended to demonstrate the proposed analysis procedure; they are not results obtained from actual participants.

Table 1. Illustrative comparison of programming assessment scores

Group	Students	Pre-test mean	Post-test mean	Improvement
PRACTICE Framework	30	48.2	76.4	28.2
Conventional teaching	30	47.6	62.1	14.5

The illustrative comparison shows how changes in assessment scores can be examined across the two instructional approaches. In this example, the PRACTICE group demonstrates a larger numerical increase in mean performance. However, this difference cannot be interpreted as evidence of effectiveness because the figures have not been obtained from an actual classroom investigation.

In the completed study, the collected scores will be analysed using appropriate statistical methods, including descriptive statistics, suitable significance tests, and effect-size estimates where applicable. Student feedback and reflection journals will also be examined to understand their experiences with coding challenges, collaborative learning, and debugging activities.

The final results will be reported using actual participant data. This evaluation is intended to determine whether the framework supports measurable improvements in programming competence and whether students can apply their learning independently to unfamiliar problems.

6. Expected Contributions

The proposed framework is intended to contribute to programming education in four ways.

Integrated learning cycle: It brings prediction, coding, debugging, collaboration, testing, and reflection into one repeatable process.

Low-cost implementation: It can be implemented using regular computer laboratories, printed exercises, and simple progress-tracking spreadsheets.

Process-oriented assessment: It evaluates students' reasoning and debugging approaches in addition to whether the final program is correct.

Transfer of learning: It includes independent problems designed to assess whether students can apply concepts to unfamiliar tasks.

These are proposed contributions. Their practical value should be assessed through the study rather than assumed in advance.

7. Conclusion

This paper proposes the PRACTICE Framework as a structured approach to interactive programming education for undergraduate Computer Science students. By combining output prediction, micro-coding challenges, error analysis, peer explanations, progressive problem-solving, and reflective learning, the framework seeks to provide students with more opportunities to practice and understand programming concepts.

The framework is designed to be affordable and adaptable to regular undergraduate teaching environments. A systematic classroom evaluation is necessary to determine whether it improves programming performance, debugging ability, student engagement, and independent problem-solving. Future research may examine its effectiveness across different institutions, programming languages, student backgrounds, and course durations.

References

1. Calderon, I., Silva, W., & Feitosa, E. (2024). Active learning methodologies for teaching programming in undergraduate courses: A systematic mapping study. *Informatics in Education, 23*(2), 279–322. https://doi.org/10.15388/infedu.2024.11

2. Bowman, N. A., Jarratt, L., Culver, K. C., & Segre, A. M. (2021). The impact of pair programming on college students' interest, perceptions, and achievement in computer science. *ACM Transactions on Computing Education, 21*(3), Article 3. https://doi.org/10.1145/3440759

3. Sentance, S., Waite, J., & Kallia, M. (2019). Teachers' experiences of using PRIMM to teach programming in school. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (pp. 476–482). Association for Computing Machinery. https://doi.org/10.1145/3287324.3287477

4. Sentance, S., Waite, J., & Kallia, M. (2019). Teaching computer programming with PRIMM: A sociocultural perspective. *Computer Science Education, 29*(2–3), 136–176. https://doi.org/10.1080/08993408.2019.1608781

5. Yang, S., Baird, M., O'Rourke, E., Brennan, K., & Schneider, B. (2024). Decoding debugging instruction: A systematic literature review of debugging interventions. *ACM Transactions on Computing Education, 24*(4), Article 45. https://doi.org/10.1145/3690652

6. Chen, C.-Y. (2025). Effects of worked examples with explanation types and learner motivation on cognitive load and programming problem-solving performance. *ACM Transactions on Computing Education, 25*(2), Article 23.