IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, May 2026
Impact Factor: 8.2

Fostering Computational Competence: A Structured Problem-Solving Framework for First-Year Engineering Programming Courses

Palve P.B, Rushikesh Dnyaneshwar Wadane, Vishal Sominath Wabale
Ajinkya Bajirao Wabale, Om Paruji Yewale, Sarthak Zarekar, Akash Zende,
Sumeet Aher, Rushikesh Suryawanshi, Sanket Chavhan, Krushna Sathe
Department of Engineering Education and Computational Sciences
Global Institute of Technology and Innovation, Gurgaon, India

Abstract: *The introductory "Programming and Problem Solving" (PPS) course is pivotal for engineering students, yet traditional syntax-first pedagogy often fails to develop genuine algorithmic thinking. This paper proposes and evaluates a Structured Problem-Solving Framework (SPSF) that integrates problem-based learning, mandatory pseudocode design, and weekly micro-challenges. Using a quasi-experimental two-group design (N=840) over four academic years, we demonstrate that the SPSF significantly improves algorithmic reasoning (ATT gain 10.3 vs 5.1 points, $p<0.001$, $d=1.37$), reduces code-anxiety, and enhances self-efficacy. Qualitative analysis reveals that "thinking before typing" and structured scaffolding reduce frustration and foster transferable problem-solving habits. The framework provides a replicable model for outcome-based PPS instruction aligned with NEP 2020 and ABET competencies.*

Keywords: Computational competence, problem-based learning, pseudocode scaffolding, algorithmic thinking, engineering programming, self-efficacy, micro-challenges.

I. INTRODUCTION

The subject "Programming and Problem Solving" (PPS), often taught in the first year of engineering using languages such as C or Python, serves as the gateway to computational literacy. Its stated objective is twofold: to impart the syntax and semantics of a programming language, and perhaps more critically, to train students in the art of problem decomposition and algorithmic reasoning. In practice, however, a pervasive gap exists. Many students exit the course able to recite loop constructs but unable to solve a straightforward problem that has not been explicitly demonstrated in class [1].

This phenomenon is not merely an academic inconvenience. In an era where the National Education Policy (NEP) 2020 and global accreditation bodies (e.g., ABET, NBA) emphasize outcome-based education and higher-order thinking, the failure to nurture genuine problem-solving ability is a systemic shortcoming [2]. The root cause often lies in pedagogical design: when syntax instruction precedes problem context, students learn to mimic rather than to think [3].

Over a 15-year career spanning both industry software development and engineering pedagogy, the author has observed that students who internalize a process of problem solving—explicitly mapping problems to algorithms before touching a keyboard—become adaptive, language-agnostic programmers. The present research formalizes this observation. We propose and evaluate a structured PPS framework that deliberately separates "understanding the problem" and "designing the solution" from "writing the code." The study addresses three research questions:

RQ1: Does a problem-first, PBL-driven PPS curriculum lead to statistically significant improvements in algorithmic thinking compared to a syntax-first curriculum?

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, May 2026
Impact Factor: 8.2

RQ2: How does the intervention affect student self-efficacy and code-anxiety?
RQ3: What are the critical design elements of the framework that contribute to its effectiveness?

II. LITERATURE REVIEW

The challenge of teaching programming to novices has been studied extensively. Early works by Papert [4] established the constructionist view that learning occurs best when learners build tangible artifacts. Wing [5] broadened the discourse with the concept of "computational thinking," arguing that problem decomposition, pattern recognition, and abstraction are fundamental skills for everyone, not just computer scientists. These ideas profoundly influenced PPS curriculum design.

A. Problem Based Learning in Programming

Problem-Based Learning (PBL) has been shown to deepen conceptual understanding by anchoring learning in real-world problems [6]. In programming education, PBL shifts the focus from language features to the problems those features solve. A meta-analysis by Hmelo-Silver et al. [7] found that PBL learners consistently outperform traditional learners on measures of conceptual application, though the effect on factual recall is mixed. Kay et al. [8] specifically applied PBL to introductory Java courses and reported improved problem-solving skills, but noted that without structured scaffolding, weaker students felt lost.

B. Pseudocode and Algorithmic Scaffolding

Writing pseudocode before coding is a widely recommended but inconsistently enforced practice. A study by Lister et al.
[9] showed that the ability to read and trace code is a distinct skill from the ability to write code, and that scaffolded tracing exercises significantly improve code-writing ability. Dehnadi and Bornat [10] discovered that performance on mental execution tasks (tracing) was a strong predictor of programming success. Extending this, our framework mandates algorithm design in structured pseudocode as a non-negotiable step before any implementation.

C. Gamified Micro Challenges

The use of short, focused coding challenges—often called "micro-drills"—has been effective in solidifying syntax in a low-stakes environment [11]. Platforms like CodeChef, LeetCode, and HackerRank offer such drills, but they are rarely integrated into the PPS classroom in a pedagogically coherent manner. Our framework incorporates weekly, timed algorithmic puzzles that directly reinforce the week's problem-solving heuristic, not just syntax.

D. Gap in the Literature

While each of the above elements has been studied in isolation, few comprehensive frameworks integrate them into a cohesive semester-long PPS curriculum for mixed-ability engineering cohorts. This study addresses that gap by designing, implementing, and empirically evaluating a unified PPS model that interleaves PBL, mandatory pseudocode, and micro-challenges.

III. METHODOLOGY

A. Study Design and Participants

The study employed a quasi-experimental, two-group pre-test/post-test design over four academic years (2019–2023) at a large private engineering institution in India. A total of 840 first-year B.Tech. students (all branches) were included. The control group (2019–2020, n=420) was taught PPS with a traditional syntax-first lecture format (lecture → syntax demonstration → lab assignment). The experimental group (2021–2023, n=420) was taught using the

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, May 2026
Impact Factor: 8.2

proposed Structured Problem-Solving Framework (SPSF). Both groups used the C programming language, had identical contact hours (3 lectures + 3 lab hours per week), and were assessed by a common end-of-semester university examination; the SPSF intervention affected only the internal teaching methodology.

B. The Structured Problem Solving Framework (SPSF)

The SPSF is built on four pillars, applied in a fixed sequence for every problem encountered throughout the semester:

Pillar I – Problem Comprehension & Abstraction: Students restate problems in their own words, identify inputs/outputs/constraints, and create visual "problem maps."

Pillar II – Algorithm Design in Pseudocode: Before writing any C code, students must produce a complete structured pseudocode solution on paper; instructor sign-off is required.

Pillar III – Translation to Code with Deliberate Syntax Drills: Syntax is taught "just in time." Weekly 15-minute micro-challenges (e.g., "write a loop to generate Fibonacci numbers under 5 minutes") cement fluency.

Pillar IV – Reflection and Optimization: After a working program, students document alternative solutions, informally analyze complexity, and write a one-paragraph reflection on the learned heuristic.

C. Instruments and Data Collection

Algorithmic Thinking Test (ATT): 25-item test measuring problem decomposition, pattern recognition, abstraction, and algorithm design (Cronbach's $\alpha = 0.87$). Administered pre- and post-intervention. Programming Self Efficacy Scale (PSES): 12-item Likert scale adapted from Ramalingam & Wiedenbeck [12]; reliability $\alpha = 0.88$ (pre-test) and 0.91 (post-test). Code Anxiety Questionnaire (CAQ): 10-item measure ($\alpha = 0.85$ pre-test, 0.89 post-test). End-semester project (50-point rubric) and focus group interviews (40 experimental students).

D. Ethical Considerations and COVID 19 Context

The study was approved by the Institutional Review Board of Global Institute of Technology and Innovation (GITI-IRB-2021-08). All participants provided written informed consent. The control group was taught in 2019-2020 (fully in person), while the experimental group was taught in 2021-2023 (predominantly hybrid). The SPSF was designed to be delivery-neutral, but the pandemic may have influenced student motivation; this is acknowledged as a limitation.

E. Data Analysis

Pre-test scores confirmed no significant initial difference between groups (control ATT mean=8.4, SD=3.1; experimental ATT mean=8.2, SD=3.3; $t(838)=0.92$, $p=0.36$). Independent-samples t-tests compared post-test outcomes; Cohen's d ca