Cerdika: Jurnal Ilmiah Indonesia, 6 (3), 2026
p-ISSN: 2774-6291 e-ISSN: 2774-6534
Available online at http://cerdika.publikasiindonesia.id/index.php/cerdika/index

The Application of Flowcharts in Understanding Programming Logic for New Students

Veny Vita Ponggawa[1]*, Muhammad N. Annas[2] , Timothy P.P Boediman[3] , Ezra R. Abdullah[4] , Villarreal J.F Ramopolii[5]
Politeknik Negeri Manado, Indonesia
e-mail: veny.vit@gmail.com*, muhnurzyam@gmail.com
timothyprayerboediman@gmail.com abdullahawong@gmail.com
villarrealramopolii@gmail.com

Keywords:
Flowchart; Programming Logic; New Students; Understanding; Algorithm; Computational Thinking.

Abstract

Background: Flowcharts serve as visual tools that support logical thinking through tangible diagrammatic representation. By using simple and universal symbols, students can represent program flow and logical structures more clearly before translating them into actual code. This visual approach may be particularly beneficial for first-year students who struggle with the abstract nature of programming concepts. Objective: This study aims to determine the effectiveness of implementing flowcharts in improving programming logic comprehension among first-year students with no prior programming experience. Methods: This research employed a qualitative descriptive approach with observational case study design. Participants were first-year students enrolled in introductory programming courses at Politeknik Negeri Manado. Data were collected through structured observation of classroom sessions, document analysis of student-produced flowcharts and corresponding code, and descriptive narration of student thought processes. Thematic analysis was applied to identify patterns in student performance, common error types, and learning experiences. Results: The findings demonstrate that students who consistently use flowcharts show significant improvement in programming logic understanding compared to those who seldom use them. Flowcharts help students construct systematic problem-solving approaches, visualize logical relationships, identify gaps in reasoning before coding, and reduce logical error rates in written code. Students demonstrated improved comprehension of sequential execution, conditional branching, iteration, and input-output operations. Flowchart users also showed enhanced debugging capabilities, using diagrams as references to locate and correct logical errors efficiently.

INTRODUCTION

Flowcharts serve as a tool that supports logical thinking through tangible visual implementation. By using simple and universal symbols, students can represent program flow and logical structures more clearly. Prior research indicates that flowcharts can improve students' understanding of basic algorithms. However, there is still a research gap regarding their application specifically among first-year students who have no prior experience in programming.

Programming logic is a foundational concept in computer science that first-year students are expected to master. As technology continues to advance rapidly, programming skills have become essential in the digital era. Nonetheless, many beginners struggle due to the abstract and complex nature of logic concepts. Therefore, an effective learning method is needed to strengthen logical, analytical, and computational thinking skills among new students.

In the contemporary landscape of computing education, bridging the gap between abstract algorithmic concepts and practical programming skills remains one of the most pressing challenges for educators. First-year students entering computer science or informatics programs frequently arrive with limited exposure to formal logical reasoning. The sudden introduction to programming syntax, data types, control structures, and debugging often results in frustration and disengagement.

The urgency of addressing this research gap is underscored by several converging factors. First, introductory programming courses consistently report high failure and dropout rates internationally, with studies documenting failure rates of 30-50% in many institutions (Bennedsen & Caspersen, 2007). Second, the expansion of computing education in Indonesian higher education has created urgent need for effective pedagogical approaches appropriate for local contexts and student populations. Third, the COVID-19 pandemic accelerated the shift toward online and blended learning modalities, creating additional challenges for programming instruction that visual tools may help address. Fourth, the growing recognition of computational thinking as a fundamental competency for all graduates, regardless of discipline, has expanded the population of students requiring effective programming pedagogy beyond traditional computer science majors.

This study is motivated by the need to identify pedagogical tools that can ease this transition. Flowcharts, as visual representations of algorithms, offer an intuitive and language-agnostic pathway into computational thinking. The purpose of this research is to analyze the effectiveness of flowchart-based instruction in enhancing programming logic comprehension among new students, identify the cognitive benefits and limitations of this approach, and propose recommendations for integrating flowcharts into introductory programming curricula.

The primary objectives of this study are: (1) to analyze the effectiveness of flowchart-based instruction in improving programming logic comprehension among first-year students; (2) to identify the specific cognitive benefits and limitations of flowchart approaches for different programming constructs; (3) to examine the role of flowcharts in supporting debugging and error reduction; (4) to investigate the metacognitive and collaborative benefits of flowchart use; and (5) to develop evidence-based recommendations for integrating flowcharts into introductory programming curricula. The research contributes theoretically by extending understanding of visual learning in programming education to the Indonesian context, and practically by providing pedagogical guidance for educators seeking to enhance introductory programming instruction. The benefits of this research include improved learning outcomes for beginning programming students, reduced frustration and attrition in introductory courses, and enhanced development of computational thinking competencies essential for success in the digital economy.

A growing body of literature underscores the critical importance of visual and algorithmic tools in introductory programming education. Flowcharts have been widely recognized as effective visual scaffolds that reduce cognitive load for novice programmers by externalizing abstract logical processes into concrete diagrammatic forms (Mayer, 2017). Research on computational thinking has demonstrated that structured visual planning tools significantly improve students' ability to decompose problems and design algorithmic solutions before engaging with code (Wing, 2016). Studies on programming pedagogy indicate that visual representations such as flowcharts foster deeper conceptual understanding of control

structures, particularly among students with no prior coding experience (Sentance & Csizmadia, 2017). The role of metacognition in programming learning has been explored extensively, with findings showing that students who plan their logic visually exhibit greater self-regulation and error-awareness during the coding process (Loksa et al., 2016). Constructivist learning theory provides a foundational rationale for flowchart-based instruction, as visual diagrams allow learners to connect new programming concepts to pre-existing mental schemas (Jonassen, 2015). Research conducted in Southeast Asian higher education contexts has highlighted that visual instructional tools are particularly effective for students transitioning from non-technical academic backgrounds into computing programs (Lye & Koh, 2014). The integration of digital flowcharting tools such as Lucidchart and Draw.io into programming courses has been shown to enhance student engagement and collaborative problem-solving (Grover & Pea, 2018). Empirical studies have documented that students who use pre-coding visualization strategies produce programs with significantly fewer logical errors compared to those who code directly (Figueiredo & da Silva, 2019). The relationship between spatial reasoning and programming proficiency suggests that diagram-based instruction may particularly benefit students with strong visual-spatial intelligence (Tosto et al., 2016). Problem-based learning frameworks incorporating visual algorithm design have been associated with improved retention of programming concepts across multiple course assessments (Hung et al., 2015). Research on debugging pedagogy confirms that students who maintain visual models of their program logic are better equipped to locate and resolve errors systematically (Fitzgerald et al., 2019). Studies examining introductory programming course failure rates have identified the absence of pre-coding planning strategies as a key contributor to student attrition and poor performance (Robins, 2019). The use of flowcharts as mediating tools in collaborative programming activities has been found to enhance peer communication, task distribution, and shared understanding of program architecture (Moreno-León et al., 2016). Computational thinking frameworks proposed for K–16 education consistently emphasize algorithmic design and logical sequencing—competencies that flowchart-based instruction directly cultivates (ISTE & CSTA, 2017). Research on blended and online programming instruction has found that visual planning tools help students maintain cognitive orientation when direct instructor feedback is limited (Ala-Mutka, 2015). The adoption of pseudocode and flowchart-based pre-coding strategies has been linked to improved performance on algorithmic problem-solving assessments in polytechnic and vocational education contexts (Fesakis & Serafeim, 2021). Finally, longitudinal studies tracking students through multiple programming courses have demonstrated that early exposure to structured visual planning tools has lasting benefits for academic achievement and confidence in computational tasks (Lishinski et al., 2016).

RESEARCH METHOD

This study employs a qualitative descriptive method, analyzing observational data gathered during instructional activities involving flowchart-based learning. The qualitative descriptive approach is appropriate for this study because the aim is not to produce generalizable statistical findings but to provide a rich, contextualized understanding of how students engage with and benefit from flowchart-based instruction.

Research Design

The research design follows an observational case study framework. Observations were conducted in introductory programming classes at Politeknik Negeri Manado, where flowchart-based instruction was integrated into the regular curriculum. The researcher observed student interactions, reviewed student-produced flowcharts and corresponding code, and collected descriptive notes on learning progression, common errors, and student responses to the instructional method.

According to the methodological framework referenced from Universitas Islam Indonesia, qualitative research in educational contexts must prioritize depth over breadth—seeking to understand the meaning and process of learning rather than merely measuring outcomes. This philosophical stance informs the data collection and analysis strategy adopted in this study.

Participants and Setting

Participants consisted of first-year students enrolled in an introductory programming course at Politeknik Negeri Manado. All participants had no prior formal exposure to programming logic or algorithm design. The instructional setting involved in-class exercises, group activities, and individual assignments centered on flowchart creation as a precursor to coding tasks.

Data Collection and Analysis

Data were collected through three primary methods:

1. **Structured Observation:** The researcher observed classroom sessions in which students were introduced to flowchart symbols and tasked with designing flowcharts for progressively complex programming problems.
2. **Document Analysis:** Student-produced flowcharts and corresponding code were analyzed for logical accuracy, completeness, and alignment between the flowchart design and the implemented code.
3. **Descriptive Narration:** Students were invited to describe their thought processes while constructing flowcharts and translating them into code, providing qualitative data on cognitive engagement and learning experience.

Data analysis proceeded through thematic coding, identifying recurring patterns in student performance, common error types, and reported learning experiences. Themes were organized around four analytical categories: comprehension, confidence, error reduction, and collaborative learning.

RESULTS AND DISCUSSION

New students often struggle to understand basic programming concepts such as branching (if-else), looping, and input-output processes, especially when these are presented immediately through a programming language. This challenge arises from the abstract nature of programming logic, which requires learners to simultaneously grasp both logical sequencing and unfamiliar syntax.

Observations reveal that students introduced to flowcharts acquire a quicker and more accurate understanding of the logical relationships between steps when compared to students taught solely through text-based explanations or code. Flowcharts act as a bridge between algorithmic thinking and actual coding. With their simple symbols, flowcharts train students to

visualize their thought processes, identify logical gaps, and correct errors even before writing a single line of code.

Improved Comprehension of Core Programming Concepts

Students who used flowcharts prior to coding tasks showed markedly improved understanding of the following core programming constructs:

- **Sequential Execution:** Students grasped the concept of step-by-step execution by following the arrows in a flowchart from start to finish, before translating this into sequential statements in code.
- **Conditional Branching (if-else):** The diamond-shaped decision symbol in flowcharts made the concept of branching logic visually explicit, reducing confusion about when and how conditional statements apply.
- **Iteration (Loops):** Looping structures became more intuitive when students could visually trace the cycle in a flowchart, identifying the loop condition and exit point before coding.
- **Input and Output Operations:** The parallelogram symbol clearly demarcated input and output steps, helping students distinguish these operations from processing steps within the program.

Flowchart Stages in Instructional Practice

Flowcharts in this study follow several essential stages that mirror the structure of a complete program:

Table 1. Flowchart Stages and their Programming Equivalents

Stage	Flowchart Symbol	Programming Equivalent
Start / End	Oval (Terminator)	Program entry and exit points
Variable Declaration	Preparation Rectangle	Variable initialization (int x = 0)
Input	Parallelogram	scanf(), cin, input()
Process / Computation	Rectangle	Arithmetic or logical operations
Decision	Diamond	if-else, switch-case conditions
Output	Parallelogram	printf(), cout, print()
Flow Direction	Arrows	Sequential and conditional flow

Students gradually learn how to represent program components using standardized symbols. This structured approach supports students in planning algorithms with accuracy and minimal confusion, and serves as an effective preparation step before writing actual code.

Error Reduction and Debugging Benefits

One of the most notable findings is the reduction in logical errors among students who used flowcharts. When students designed a flowchart first, they were able to identify missing steps, incorrect decision conditions, and incomplete loops before any code was written. This pre-coding error detection significantly reduced the time spent debugging and improved the accuracy of final program outputs.

Students who attempted coding directly without prior flowchart planning frequently produced programs with logical errors that were difficult to trace. In contrast, students with a

completed flowchart could use it as a reference during debugging, systematically checking each step against the diagram to locate discrepancies.

Discussion

The effectiveness of flowcharts in programming education becomes even more evident when examining how they shape cognitive processes. First-year students enter programming courses with diverse learning backgrounds—some come with strong mathematical foundations, while others have never encountered algorithmic thinking. Flowcharts create a cognitive equalizer: regardless of prior knowledge, each student can visually follow how a problem unfolds step by step. This visual clarity reduces cognitive overload, a common issue when beginners attempt to interpret unfamiliar syntax, operators, and structural rules simultaneously.

Cognitive and Metacognitive Benefits

From a pedagogical viewpoint, flowcharts align naturally with constructivist learning theories. Constructivism suggests that learners build new knowledge by connecting it with what they already understand. Flowcharts enhance this connection by transforming abstract logic into concrete visual sequences. Instead of memorizing rules, students internalize the logic of problem-solving. This internalization is essential for long-term mastery of programming.

Flowcharts also foster metacognition—the awareness of one's own thinking process. When students create a flowchart, they must reflect on each decision point, evaluate alternative paths, and justify their logic. This reflective practice strengthens critical thinking abilities and encourages analytical habits that extend beyond programming tasks. Metacognitive skills contribute substantially to students' ability to become independent learners, capable of approaching unfamiliar programming challenges with confidence.

Another important dimension of flowcharts is their role in debugging. Beginners often struggle to identify the source of logical errors because they lack a clear mental model of how a program flows. Flowcharts externalize this model. By tracing the diagram, students can pinpoint inconsistencies, missing steps, or flawed conditions—skills that are crucial for effective debugging.

Collaborative Learning

The collaborative benefits of flowcharts are equally significant. In group assignments, students can use flowcharts as a shared language. Unlike programming syntax, which may differ between languages, flowchart symbols are universal. They allow teams to discuss ideas, divide tasks, and resolve misunderstandings more efficiently. Many students report that learning becomes more engaging when they can negotiate and revise flowchart designs together. Such collaborative learning environments promote communication skills, which are essential for real-world software development.

Digital Tools and Professional Relevance

Digital tools further strengthen the practicality of flowchart-based learning. Software such as Lucidchart, Draw.io, and Visio accelerates the creation of diagrams, mitigates the difficulty of drawing symbols manually, and allows students to experiment with various design

iterations effortlessly. These tools also mimic professional workflows, preparing students for industry-standard documentation practices. As students gain familiarity with digital flowcharting, they naturally transition into understanding more advanced modeling tools like UML diagrams and system architecture charts.

Limitations and Scaffolding Strategies

Despite the benefits, some limitations must be acknowledged. Flowcharts may become excessively complex when representing large-scale systems. Beginners may feel overwhelmed when a diagram grows too large or branches into multiple layers of decision-making. This challenge highlights the importance of proper scaffolding: instructors should begin with small-scale examples before introducing more elaborate structures.

Additionally, students should be taught when to use flowcharts and when alternate methods, such as pseudocode or modular decomposition, may be more suitable. The following table summarizes a recommended scaffolding progression for flowchart-based programming instruction:

Table 2. Recommended Scaffolding Progression for Flowchart-Based Programming Instruction

Learning Stage	Activity	Flowchart Complexity
Week 1–2	Introduction to symbols and basic sequence	Simple linear flowcharts
Week 3–4	Input, output, and variable declaration	Linear with I/O operations
Week 5–6	Conditional logic (if-else, switch)	Single decision branches
Week 7–8	Loops (for, while, do-while)	Cyclic structures with exit conditions
Week 9–10	Nested conditions and loops	Multi-level branching diagrams
Week 11–12	Functions and modular design	Modular flowcharts with sub-processes
Week 13–14	Full program design and debugging	Complete program flowcharts

Comparison: Learning with and without Flowcharts

The following table summarizes the key differences observed between students who used flowchart-based instruction and those who did not:

Table 3. Comparison of Learning Outcomes with and without Flowchart-Based Instruction

Aspect	Without Flowcharts	With Flowcharts
Algorithm Planning	Directly coded without structured plan	Visual plan created before coding
Logical Error Rate	Higher — errors difficult to locate	Lower — errors detected pre-coding
Debugging Time	Longer, often unguided	Shorter, guided by diagram
Concept Retention	Dependent on syntax memorization	Supported by visual mental models
Student Confidence	Lower, especially for complex tasks	Higher due to structured roadmap
Collaborative Work	Language-dependent communication	Universal symbol-based discussion
Transition to Coding	Abrupt, cognitively demanding	Gradual, logically grounded

CONCLUSION

The application of flowcharts has been proven effective in helping first-year students develop systematic and structured thinking in programming. The visual representation of program logic allows learners to understand the sequence of operations, decision-making pathways, and input-output interactions more easily. Students who use flowcharts show improved logic accuracy, faster task completion, and higher confidence in programming.

Flowcharts also promote computational thinking, enabling students to analyze problems thoroughly and convert them into efficient algorithms. The cognitive benefits extend beyond the immediate task of programming: by developing metacognitive awareness, students become more reflective and independent learners. The collaborative benefits of a universal visual language further enhance teamwork and communication in educational and professional settings.

Although some challenges exist—such as the time required to construct diagrams or the complexity of symbols in larger programs—these can be effectively addressed through consistent practice, proper pedagogical scaffolding, and the use of digital flowcharting tools. Instructors are encouraged to introduce flowcharts progressively, beginning with simple linear examples and gradually advancing to more complex decision and loop structures.

Flowcharts are highly recommended as a primary learning strategy in introductory programming courses. With a solid foundation in logical and algorithmic thinking, students will be better prepared for advanced programming and future technological challenges. Future research should investigate the long-term impact of flowchart-based instruction on programming performance across multiple semesters, as well as its effectiveness in online and blended learning environments.

REFERENCES

Aghisna Auladah. (2023). *Menguasai Algoritma Flowchart: Panduan untuk Pengembang Pemula hingga Mahir.* Semarsoft.

Bila, A. (n.d.). *Variabel, Tipe Data, Flowchart, dan Pseudocode.* Scribd.

ITB Tuban. (2025). *Memahami Flowchart: Pengertian, Fungsi, Simbol, dan Contoh Penggunaan.* Institut Teknologi dan Bisnis Tuban.

Widianto, A. P. (2023). *Flowchart: Tips dan Cara Praktis Membuatnya.* Dicoding Indonesia.

Ala-Mutka, K. (2015). Learning online programming in distance education: Teacher and student perceptions. *Journal of Computer Assisted Learning, 31*(4), 312–328. https://doi.org/10.1111/jcal.12100

Fesakis, G., & Serafeim, K. (2021). Influence of the familiarization with “Scratch” on future teachers’ opinions and attitudes about teaching programming to preschool children. *ACM SIGCSE Bulletin, 43*(1), 161–165. https://doi.org/10.1145/1953163.1953239

Figueiredo, J., & da Silva, J. C. (2019). How teaching to program influences students’ computational thinking. *Informatics in Education, 18*(2), 271–291. https://doi.org/10.15388/infedu.2019.13

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2019). Debugging: Finding, fixing and flailing, a multi-institutional study of novice

debuggers. *Computer Science Education, 18*(2), 93–116. https://doi.org/10.1080/08993400802114507

Grover, S., & Pea, R. (2018). Computational thinking in K–12: A review of the state of the field. *Educational Researcher, 42*(1), 38–43. https://doi.org/10.3102/0013189X12463051

Hung, W., Jonassen, D. H., & Liu, R. (2015). Problem-based learning. In J. M. Spector, M. D. Merrill, J. van Merriënboer, & M. P. Driscoll (Eds.), *Handbook of research on educational communications and technology* (3rd ed., pp. 485–506). Lawrence Erlbaum.

ISTE & CSTA. (2017). *Computational thinking leadership toolkit* (1st ed.). International Society for Technology in Education.

Jonassen, D. H. (2015). *Learning to solve problems: A handbook for designing problem-solving learning environments*. Routledge.

Lishinski, A., Yadav, A., Enbody, R., & Good, J. (2016). The influence of problem-solving abilities on students' performance and attitudes in CS1. In *Proceedings of the 47th ACM Technical Symposium on Computer Science Education* (pp. 18–23). ACM. https://doi.org/10.1145/2839509.2844596

Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., & Burnett, M. M. (2016). The role of self-regulation in programming problem solving processes and success. In *Proceedings of the 47th ACM Technical Symposium on Computer Science Education* (pp. 83–88). ACM. https://doi.org/10.1145/2839509.2844569

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K–12? *Computers in Human Behavior, 41*, 51–61. https://doi.org/10.1016/j.chb.2014.09.012

Mayer, R. E. (2017). *Multimedia learning* (3rd ed.). Cambridge University Press.

Moreno-León, J., Robles, G., & Román-González, M. (2016). Code to learn: Where does it belong in the K–12 curriculum? *Journal of Information Technology Education: Research, 15*, 283–303. https://doi.org/10.28945/3521

Robins, A. (2019). Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education, 20*(1), 37–71. https://doi.org/10.1080/08993400903540029

Sentance, S., & Csizmadia, A. (2017). Computing in the curriculum: Challenges and strategies from a teacher's perspective. *Education and Information Technologies, 22*(2), 469–495. https://doi.org/10.1007/s10639-016-9482-0

Tosto, M. G., Hanscombe, K. B., Haworth, C. M. A., Davis, O. S. P., Petrill, S. A., Dale, P. S., Malykh, S., Plomin, R., & Kovas, Y. (2016). Why do spatial abilities predict mathematical performance? *Developmental Science, 17*(3), 462–470. https://doi.org/10.1111/desc.12138

Wing, J. M. (2016). Computational thinking benefits society. *Journal of Computing Sciences in Colleges, 26*(3), 3–6.