American Journal of Advanced Research, 2025, 9–1
July. 2025, pages 01-05
doi: 10.5281/zenodo.15870013
http://www.ibii-us.org/Journals/AJAR/
ISBN 2572-8849 (Online), 2572-8830 (Print)

A pedagogical framework for maximizing student learning outcomes: Readability and Reusability

Karuppasamy, Geetha[1] and Larson, Theodore[1, *]

[1]Department of Computer Science, Oklahoma State University

*Email: bjorn.larson@okstate.edu

Received on 04/04/2025; revised on 07/09/2025; published on 07/11/2025

Abstract

In an evolving educational landscape shaped by artificial intelligence, shifting student expectations, and industry-aligned goals, the role of the computer science educator is undergoing fundamental change. This paper introduces a pedagogical framework based on two core software development principles readability and reusability framed as foundational tools for teaching programming. The framework fosters clarity, modularity, and transferability, guiding students to evaluate each concept through these lenses. Implemented in an introductory Java course, the approach led to notable improvements in student confidence and comprehension, confirmed by self-assessments and faculty feedback. Beyond academic gains, the model promotes professional readiness and offers a transferable strategy for educators across disciplines.

Keywords: computer science, education, framework, pedagogy

1 Introduction

The relationship between professors and students has long been centered on a cooperative goal: equipping students with the skills, habits, and ways of thinking necessary for success in their chosen field. Two significant trends have emerged in recent years that complicate the educator's role: the accelerating adoption of AI in both academic learning (Hajkowicz et al., 2023) and industry (McElheran et al., 2024), and a shifting student mindset that views higher education as primarily a career pathway, rather than a space for broad intellectual development (Koseda et al., 2024).

These developments raise critical pedagogical questions. If students can access sophisticated AI tools to write or debug code, and if the perceived value of education is primarily employability, what then is the role of the educator? What value can an instructor bring to a learning experience increasingly supplemented or replaced by technology? And what, if anything, distinguishes the classroom from the abundance of free or low-cost online resources?

While there are multiple answers to these questions in literature, we propose that the educator's role now more than ever is to provide structure and a framework for learning. This role is not diminished by AI but made more urgent by it. Students now need guidance not only on what to learn but how to evaluate, refine, and transfer what they learn into different contexts. If education is shifting toward workplace readiness, then instructors must bring authentic industry practices into the classroom not as disconnected anecdotes, but as coherent pedagogical principles.

In this paper, we present such a principle: a pedagogical framework grounded in two enduring and industry-relevant metrics readability and reusability. We suggest that these concepts, while familiar in the context of software development, can serve as a foundation for teaching programming in a way that is both intellectually rigorous and practically aligned with professional standards.

Readability and reusability are not merely technical ideals; they are habits of mind that shape how students write, assess, and revise code. When introduced early and reinforced consistently, these principles help students see programming not as a series of disconnected assignments, but as a craft that values clarity, modularity, and long-term maintainability. This will provide the students with a wide amount of practice in the process, and a meta-narrative of how and why decisions made by expert practitioners are made in that way in a manner that is both pedagogically and practically sound.

In the sections that follow, we describe the development of this framework, its implementation in a first-year computer science course, and the feedback from students and instructors who engaged in it. We argue that such an approach not only improves student learning outcomes but also strengthens their ability to self-evaluate and adapt in an evolving technological landscape.

2 Frameworks and Models

As has been previously articulated (e.g., Larson & Crouse, 2022), a conceptual model is an attempt to provide a discretized and sensible model for a continuous and overly chaotic real phenomenon. It does not claim

that all frameworks are exact representations of natural or social phenomena, instead it provides a communicative framework. A well-designed model helps individuals focus their decision-making by narrowing the scope of possibilities and offering a framework for reasoning within constraints.

This communicative function is particularly relevant in education. Educators are not merely conveying content, they are conveying expertise, and expertise is, by its nature, complex a to the novice. Framing that expertise within a discretized model makes it more ideal. It transforms a domain's complexity into actionable insight. In this case, the distinction between a "model" and a "framework" is specious. However, for the purposes of this discussion, we use the term framework to emphasize its prescriptive intent. While a model may explain how things are, a framework helps shape what actions should be taken within a pedagogical setting.

An important part of using any framework, especially in education, is transparency about its limitations. Students should be made aware that frameworks are tools. They are designed to be helpful until they are no longer sufficient. As a bit of deliberate self-parody, we might say: a framework is like a simile—useful only until it's not.

3 Readability and Reusability

As a result, we specifically recommend that computer science students be introduced to the principles of readability and reusability from their very first programming course, and that these concepts be reinforced consistently throughout their academic journey. Rather than treating readability and reusability as peripheral skills or optional best practices, we argue that these two metrics should serve as the lens through which all code is evaluated. They should be embedded not merely as outcomes of instruction, but as the mechanisms by which learning objectives are effectively realized.

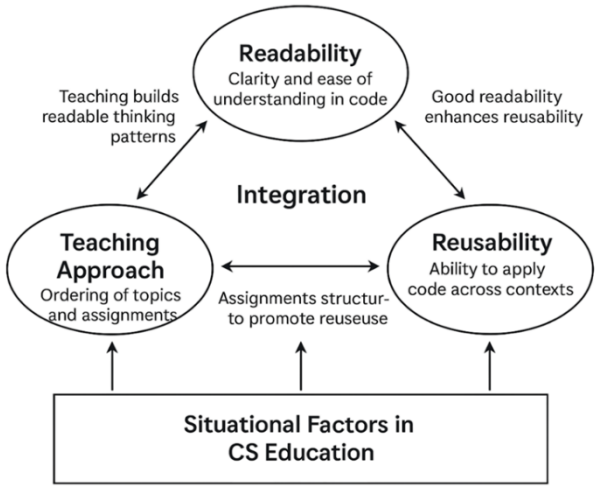

Fig. 1. A Holistic Model for Teaching Readable and Reusable Code in Context.

Figure 1 illustrates the integration of three interdependent elements Readability, Reusability, and Teaching Approach within the context of computer science education. The triangular structure emphasizes their reciprocal relationships, all informed by underlying situational factors, such as institutional goals, student backgrounds, and curriculum constraints.

- Readability refers to the clarity and ease of understanding in code. It is enhanced through teaching practices that foster readable thinking patterns and coding conventions.
- Reusability is defined as the ability to apply code across different contexts or assignments. It is supported by good readability and strengthened through assignment design that promotes modular, extensible solutions.
- Teaching Approach encompasses the sequencing of topics, structuring of assignments, and framing of instruction. It acts as the delivery mechanism that weaves readability and reusability into the student experience.

Each arrow reflects a bidirectional influence. For example, teaching fosters readability, while readable code improves a student's understanding of taught material. Similarly, readability enhances reusability, and the push for reusable design informs how topics are presented and practiced.

At the base of the framework lies Situational Factors in CS Education representing the constraints and opportunities that influence how instructors can integrate these principles into their courses.

This framework provides several critical benefits:

- Simplicity and Memorability: The terms are accessible and intuitive, making them easy for students to recall and apply consistently.
- Pedagogical Flexibility: The concepts are broad enough to apply across diverse programming topics from syntax and control structures to algorithms and software architecture yet focused enough to provide meaningful guidance.
- Instructional Interpretability: Instructors can interpret and apply the framework in ways that align with their own teaching styles and course objectives, while maintaining a shared vocabulary across sections or departments.
- Industry Alignment: Readability and reusability are well-established standards in professional software development. By internalizing these values, students gain not only academic proficiency but also professional readiness.

By framing all instructional content around these two principles, educators can help students develop durable coding habits that scale with complexity and remain relevant beyond the classroom.

3.1 Readability

Code readability refers to the degree to which source code can be easily understood by humans, a factor that directly impacts collaboration, maintainability, and long-term project success (Buse & Weimer, 2010). While readability is often subjective, it is influenced by a variety of structural and stylistic choices. These include naming conventions, code length, indentation, nesting levels, and even the presence or absence of whitespace (Rahman & Roy, 2018). Readability is not a one-size-fits-all metric, but rather a constellation of best practices that help reduce cognitive load when reviewing code.

We categorize code readability into three primary dimensions:

(1) Structural Elements: These include aspects of code layout and ogan ization, such as indentation, line length, and nesting depth. These features influence how easily the visual structure of the code can be parsed.

(2) Semantic Elements: This category focuses on meaning and naming elements like variable names, function signatures, and logical

organization of blocks. These impact on the reader's ability to understand what the code is doing.

(3) Supportive Elements: These are secondary but essential features like comments, spacing, and documentation. They provide additional context that enhances understanding, especially when revisiting or reusing code later.

A novel aspect of our pedagogical framework is the intentional framing of new concepts through the lens of readability. Rather than treating code structure as a peripheral concern, we introduce programming constructs by explicitly connecting them to the goal of writing clearer code. For example:

(1) Loops: Improve readability by reducing repetition and condensing loginto a more compact and digestible format, allowing students to focus on conditions rather than scrolling through repeated blocks.

(2) Comments: Offer supplemental context that explains why certain decisions were made, especially when the logic is non-obvious. Well-placed comments act as a silent guide for the reader.

(3) Recursion: While potentially more abstract, recursion provides mathematically elegant solutions that can be easier to reason about, especially when solving problems like factorials or traversing trees. Teaching students to evaluate recursion based on clarity helps them decide when it aids or hinders readability. Figure 2 illustrates how key programming constructs contribute to improved code readability. The horizontal flow connects three essential elements

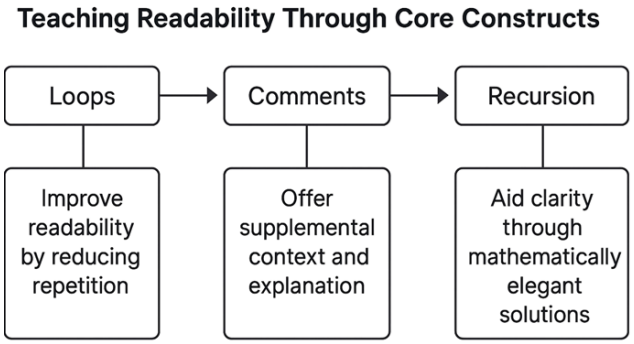

Fig. 1. Teaching Readability in Computer science education

Students should be encouraged to actively evaluate both their own code and that of their peers through the lens of readability. In the early stages of learning, students often produce code that, while functionally correct, may be verbose, repetitive, or poorly structured. As they are introduced to readability-enhancing constructs such as loops, modular functions, and meaningful naming conventions, their code begins to evolve. By comparing these later efforts with their earlier work, students can observe concrete improvements: shorter and more organized scripts, clearer logic flow, and better use of abstraction.

This reflective process allows students to internalize what makes code more readable not just theoretically, but experientially. It also aligns with common practices in professional software environments, where teams rely on coding standards and peer reviews to ensure that code remains maintainable and comprehensible over time. By mirroring these practices in the classroom, we help students not only improve their programming skills but also prepare for collaborative, real-world development settings.

3.2 Reusability

Code reusability refers to the ability to apply existing code to new contexts with little or no modification (Mejba et al., 2023). While the definition may require some adaptation in academic settings where learning objectives often take precedence over real-world applicability it remains one of the most critical drivers of efficiency in professional software development. In industry, reusability supports long-term project continuity, minimizes redundant effort, and reduces onboarding and testing time (Sojer & Henkel, 2010). It also ensures that knowledge embedded in code outlives individual contributors, supporting sustainable development practices across teams and organizations.

The students should be encouraged to review their own work for the potential to reuse it in future assignments. Rather than requiring that each assignment be approached from scratch, instructors can promote the reuse of previously written logic, functions, or structures especially when those solutions have proven to be clear, effective, and well-documented. Though creativity and doing original work is valued in academia, forcing students to write brand-new code from scratch for every single assignment might unintentionally teach the wrong lesson. In the real world especially in professional software development reusing, modularizing, and adapting existing code is not just acceptable, it's considered best practice to reduce development and testing times.

A key contribution of this framework is the intentional framing of reusability as a pedagogical tool. Every new programming construct can be introduced by asking: How does this improve the potential to reuse code? This question shifts the learning process from isolated problem-solving to a mindset of designing for adaptability and scalability. For example:

(1) Methods: Encapsulate logic into discrete units that can be reused across multiple programs or scenarios, reducing duplication and improving maintainability.

(2) Object-Oriented Principles: Abstractions like classes and inheritance allow for code to be extended or repurposed in new projects with minimal changes.

(3) Comments: Serve as documentation that enhances reusability by making the code understandable to future users (including the original author), independent of its original context.

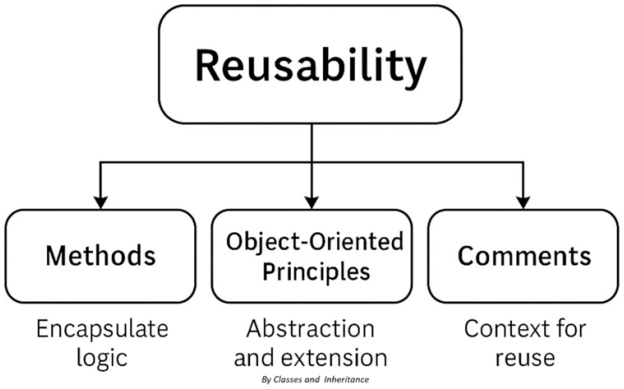

Fig. 1. Teaching Reusability in Computer Science Education

Figure 3 illustrates how core programming constructs methods, object-oriented principles and comments contribute to code reusability.

3.3 Recommendations

As a result, the mantra of this framework is, every concept in a computer science course should be evaluated in terms of how it improves readability, reusability, or both. These twin goals provide students with an ongoing structure for self-evaluation and refinement. Anecdotally, we observe that novice students tend to focus on only one dimension usually readability but with practice and guided instruction, they begin to recognize that the most powerful programming constructs improve both simultaneously. This dual focus not only enhances academic understanding but also cultivates habits aligned with industrial readiness, where maintainability and adaptability are essential in professional coding environments. Figure 4 illustrate the interplay between Readability and Reusability as foundational principles in computer science education.

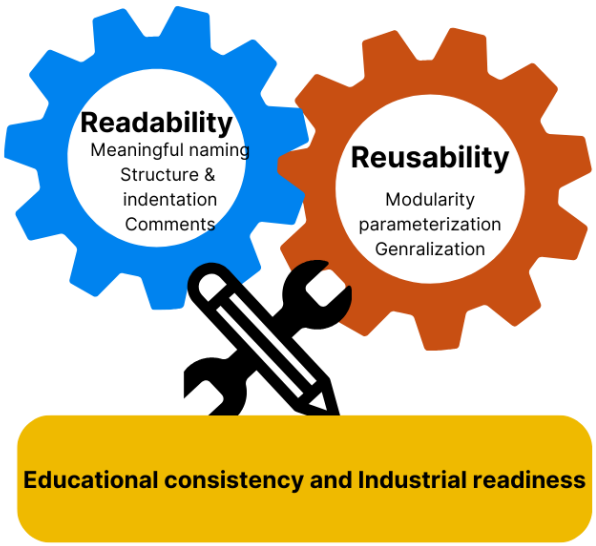

Fig. 1. Readability and Reusability as Interlocking Foundations

This observation opens the door to future empirical research: How do students develop the capacity to evaluate code across both dimensions? At what point does their perspective shift from seeing readability and reusability as separate concerns to understanding their interdependence?

4 Data and Observations

In an average-sized introductory computer science course (Java Programming I) at an R1 university targeted at technically inclined students, but not necessarily computer science majors, students were given a pre- and post-course self-evaluation. They were informed that the purpose of this exercise was not for grading, but solely for self-reflection and to guide future course improvements after final grades were submitted. Students also participated in a standard university course evaluation process.

Pre/Post-Test Self-Evaluation Questions:

(1) I feel confident in my ability to read and write code in at least one formal programming language.
(2) I feel confident in my ability to read and write code in Java.
(3) I feel confident in my ability to solve problems.
(4) I feel confident in my ability to solve problems using structured, iterative steps.
(5) I feel confident in my ability to solve problems in a way that is understandable by a computer.
(6) I have been exposed to Linux before.
(7) I know what a GUI is and can create one.
(8) I know what compilation is and how it fits into the programming process.
(9) I have a framework for learning a new programming language on my own, if necessary.
(10) I could teach someone how to write and run a "Hello, World" program and answer basic questions about it.

On average, students rated themselves 4.8/10 on the pre-test and 9.6/10 on the post-test, a substantial improvement in self-perceived competency. While these numbers are not statistically rigorous due to the small, non-randomized sample and non-empirical nature of the instrument, they nonetheless indicate a positive trend. A more formal study with larger cohorts and controlled variables would be needed to draw generalizable conclusions about the framework's effectiveness.

In addition, 100% of the qualitative comments in the official course evaluation mentioned increased confidence and understanding. Faculty peer reviewers noted the novelty and coherence of framing programming concepts around readability and reusability, describing it as perceptively beneficial for student comprehension and retention.

Of course, correlation is not causation. It is possible that the positive student outcomes observed were driven less by the framework itself and more by the intentionality and learner-centric design implemented by the instructor. This leads us to a broader recommendation.

5 Broader Applications to Other Fields

The authors suggest that while this specific framework of readability and reusability was developed within the context of computer science education, the underlying pedagogical principle is frequently based on principle and not the specific subject. pedagogy as a result, the specific call to action, even in other fields it to take some subset of industry common terminology or metric and use it as the rubric when introducing new concepts.

Instructors across disciplines might consider adopting similarly intuitive and discipline-relevant frameworks. The idea is not to chase perfect fidelity to theory but to offer learners a navigational tool even if whimsical or imperfect that structures thinking and promotes transferability. A few illustrative, fictional examples:

- Alchemy: Represent all processes as either distillation (isolating core ideas) or combination (synthesizing multiple elements). New tools or techniques are introduced in terms of how they enhance one or both actions.
- Metaphysics: Frame learning in terms of constraints tightening and loosening them. Students evaluate ideas by how they move toward or away from an idealized, unconstrained conceptual space.
- Holophone: Imagine knowledge acquisition as the pursuit of harmony. Each new topic or skill either adds to a resonant whole or introduces dissonance that must be resolved through understanding.

These analogies are intentionally lighthearted and exaggerated "farcical," even yet that simplicity makes them memorable. They provide a cognitive hook, a shared language between instructor and student, and a structure for organizing knowledge that otherwise risks being fragmented.

6 Conclusion

As programming education adapts to new technological and pedagogical realities, instructors must not only transfer knowledge but also equip students with durable, transferable practices that reflect professional expectations. This paper presents readability and reusability as guiding principles through which students can develop strong, communicative, and maintainable coding habits. By embedding these principles consistently across assignments and instructional strategies, educators can provide students with a clear view for both evaluating and improving their work.

While the observed outcomes are promising, the results should be interpreted as a starting point for further research, rather than conclusive evidence. Larger-scale studies and cross-institutional comparisons could help refine the model and assess its broader applicability. Furthermore, the broader philosophical insight—that teaching is most impactful when it uses familiar and meaningful frameworks to shape unfamiliar concepts extends beyond computer science. Whether through readability, distillation, or harmony, students benefit when learning is organized around an intuitive, consistent narrative. In this spirit, we invite instructors in all disciplines to explore how core values of their respective professions might be reframed as pedagogical anchors, enriching both instruction and student understanding.

Acknowledgements

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

Funding

This work has been supported by the …..

Conflict of Interest: none declared.

References

Buse, R. & Weimer, W. (2010). Learning a Metric for Code Readability. Software Engineering, IEEE Transactions on. 36. 546-558. 10.1109/TSE.2009.70.

Hajkowicz, S., Sanderson, C., Karimi, S., Bratanova, A., & Naughtin, C. (2023). Artificial intelligence adoption in the physical sciences, natural sciences, life sciences, social sciences and the arts and humanities: A bibliometric analysis of research publications from 1960-2021. arXiv preprint arXiv:2306.09145.

Koseda, E., Cohen, I., Cooper, J., & Mcintosh, B. (2024). Embedding employability into curriculum design: The impact of education 4.0. Policy Futures in Education. 10.1177/14782103241282121.

Larson, T & Crouse, G. (2022). Proposal for Ray's Multivector: An objective metric for culling strategic plans. Journal of Management Science and Business Intelligence, 7(2).

McElheran, K., & Brynjolfsson, E. (2024). AI adoption in America: Who, what, and where. Journal of Economics & Management Strategy, 33(1), 5-29. DOI: 10.1177/14782103241282121

Mejba, R., & Miazi, S., & Palash, A., & Sobuz, T., & Ranasinghe, R. (2023). The Evolution and Impact of Code Reuse: A Deep Dive into Challenges, Reuse Strategies and Security. 6. 10.5281/zenodo.10141558.

Rahman, M. & Roy, C. (2018). On the Use of Context in Recommending Exception Handling Code Examples. 10.48550/arXiv.1807.02261.

Sojer, M. & Henkel, J. (2010). Code Reuse in Open-Source Software Development: Quantitative Evidence, Drivers, and Impediments. J. AIS. 11. 10.17705/1jais.00248.