International Journal of Computing, Programming and Database Management

E-ISSN: 2707-6644
P-ISSN: 2707-6636
Impact Factor (RJIF): 5.43
www.computersciencejournals.com/ijcpdm
IJCPDM 2026; 7(1): 16-20
Received: 08-09-2025
Accepted: 14-11-2025

Lucas André Moreau
Department of Computer Science, École Polytechnique de Montréal, Montréal, Canada

Impact of modular programming practices on software maintainability in student projects

Lucas André Moreau

DOI: https://www.doi.org/10.33545/27076636.2026.v7.i1a.148

Abstract
Modular programming is widely promoted in software engineering education as a means to improve code quality, readability, and long-term maintainability. Student software projects, however, often exhibit tightly coupled logic, limited abstraction, and inconsistent structure, which can hinder maintenance activities such as debugging, enhancement, and reuse. This research examines the impact of adopting modular programming practices on the maintainability of student-developed software systems. Using a practical research design, multiple student projects developed under comparable academic conditions were analyzed based on modularity indicators including module size, cohesion, coupling, and interface clarity. Maintainability was evaluated through a combination of quantitative software metrics and qualitative assessments of code comprehension and modification effort. The findings demonstrate that projects employing well-defined modules with clear responsibilities show significantly improved maintainability outcomes, including reduced defect localization time, lower change impact, and higher readability scores. Students who applied modular design principles were also better able to extend functionality without introducing regressions. The research further highlights common challenges faced by learners, such as improper decomposition and over-fragmentation, which can negatively affect maintainability when modularity is poorly implemented. By empirically linking modular programming practices to measurable maintainability improvements, this research provides evidence supporting the inclusion of structured modular design instruction in undergraduate curricula. The results contribute to software engineering education by clarifying how specific modular practices influence maintenance-related attributes in novice-developed systems. Overall, the research reinforces modular programming as a critical pedagogical tool for cultivating sustainable software development skills and preparing students for real-world software maintenance demands. Implications for instructors, curriculum designers, and assessment strategies are discussed, emphasizing alignment between theory and practice, iterative feedback, and early exposure to refactoring activities that help students internalize modular thinking while balancing simplicity, performance, and maintainability constraints in academic development environments across varied project scales and collaborative team settings typical of undergraduate courses worldwide.

Keywords: Modular programming, software maintainability, student projects, software engineering education, code quality

Introduction
Software maintainability is a central quality attribute that determines the ease with which a system can be understood, corrected, adapted, and enhanced over time [1]. In educational settings, student-developed software often serves as an early exposure to professional development practices, yet such projects frequently suffer from monolithic structures, duplicated logic, and limited separation of concerns, making maintenance tasks disproportionately difficult [2]. Modular programming, which emphasizes decomposition of systems into cohesive and loosely coupled units, has long been recognized as a foundational principle for managing software complexity [3]. Prior studies suggest that modular designs improve readability and reduce the cognitive load required to understand program behavior, particularly for novice developers [4]. Despite its theoretical importance, students commonly struggle to apply modular principles effectively, resulting in designs that either lack sufficient abstraction or fragment functionality excessively [5]. This mismatch between taught concepts and practical application raises concerns about the long-term maintainability of student projects and their preparedness for real-world software development [6]. From an academic perspective, maintainability is especially relevant because student projects are

Corresponding Author:
Lucas André Moreau
Department of Computer Science, École Polytechnique de Montréal, Montréal, Canada

often revised, extended, or reused across semesters, amplifying the cost of poor design decisions [7]. Empirical evidence indicates that metrics related to cohesion and coupling are strongly associated with maintenance effort, defect density, and change proneness in software systems [8]. However, limited research has focused specifically on how modular programming practices influence these attributes within student-developed codebases [9]. Addressing this gap, the present research investigates the relationship between modular programming practices and software maintainability in student projects by analyzing structural properties and maintenance outcomes under controlled academic conditions [10]. The primary objective is to assess whether systematic adoption of modular design principles leads to measurable improvements in maintainability indicators such as readability, defect isolation, and modification effort [11]. A secondary objective is to identify common modularization pitfalls encountered by students and their impact on maintenance activities [12]. Based on established software engineering theory, this research hypothesizes that student projects employing well-defined, cohesive modules with minimal interdependencies will demonstrate significantly higher maintainability compared to less modular counterparts [13]. By integrating educational practice with empirical software analysis, the research aims to contribute actionable insights for curriculum design and instructional strategies in software engineering education [14, 15].

Materials and Methods

Materials: A comparative, classroom-based practical research was conducted on 40 undergraduate student software projects developed under similar course constraints (same language/toolchain, comparable scope, and fixed submission timeline) to evaluate how modular programming practices influence maintainability outcomes [2, 6]. Projects were categorized into two groups: Modular (n=20), where teams explicitly applied modular decomposition, separation of concerns, and refactoring checkpoints during development [3, 7, 14]; and Baseline (n=20), where teams followed standard development guidance without structured modularization checkpoints [2, 4]. Maintainability was operationalized using a Maintainability Index (MI, 0-100) aligned with software product quality perspectives [1], supported by structural metrics capturing modularity quality, including coupling (CBO) and cohesion (LCOM) as widely used design indicators [8, 11, 12]. Maintenance outcomes included

1. Change effort (minutes) to implement a predefined enhancement and correct a seeded defect, and
2. Post-change defects introduced during modification, reflecting maintenance risk [10, 13, 15].

A structured evaluation rubric was used for consistency in code readability and modification assessment, following empirical software evaluation practices and measurement principles [9, 10].

Methods

Projects were analyzed after submission using a consistent assessment pipeline:

1. Extraction of modularity metrics (CBO, LCOM) and maintainability proxy scores (MI) [11, 12];
2. Controlled maintenance tasks performed by evaluators to record time to change and defects introduced, reflecting real maintenance actions such as comprehension, localized edits, and regression checks [7, 13].

Descriptive statistics (mean, SD, median) were computed by group. Inferential comparisons between Modular and Baseline projects were conducted using Welch's t-tests for MI, CBO, LCOM, and change effort to accommodate unequal variances [9], with effect sizes (Cohen's d) reported for practical significance. Because defect counts may not be normally distributed, defect differences were additionally checked with a nonparametric Mann-Whitney U test [9]. To examine predictors of maintainability and effort, multiple linear regression models were fit:

1. MI as a function of CBO, LCOM, and group membership; and
2. Change effort as a function of CBO, LCOM, and group membership, consistent with empirical modeling approaches in software engineering [8,10]. Statistical significance was interpreted at $\alpha = 0.05$.

Results

Table 1: Descriptive statistics of maintainability and modularity-related outcomes by group (mean ± SD)

Group	n	Maintainability Index (MI)	CBO	LCOM	Change effort (min)	Post-change defects (count)
Baseline	20	62.14 ± 6.78	13.91 ± 3.34	0.585 ± 0.123	63.57 ± 10.84	2.15 ± 1.95
Modular	20	76.97 ± 5.76	7.95 ± 1.64	0.348 ± 0.069	40.98 ± 9.63	1.65 ± 1.46

The Modular group demonstrated higher MI and lower coupling/cohesion penalties (CBO, LCOM) than the Baseline group, aligning with the expectation that better decomposition improves maintainability-related attributes [1, 3, 8]. Change effort was markedly lower in Modular projects, suggesting easier comprehension and safer modification paths when responsibilities are separated and interfaces are clearer [3, 7, 14].

Table 2: Between-group comparisons using Welch's t-test (Modular vs Baseline) with effect size (Cohen's d)

Outcome	t (Welch)	p-value	Cohen's d
Maintainability Index	7.46	<0.001	2.36
CBO	-7.17	<0.001	-2.27
LCOM	-7.55	<0.001	-2.39
Change effort (min)	-6.97	<0.001	-2.20

Group differences were statistically significant and very large in magnitude for MI, coupling, cohesion, and change effort (all *p*<0.001), supporting long-standing modularity arguments that decomposition reduces complexity and facilitates maintenance actions [3, 8, 13]. For defect counts, a nonparametric comparison (Mann-Whitney U) indicated no statistically significant difference (p = 0.507), suggesting that while modularity reduced time and structural risk, defect introduction may also depend on testing discipline and developer experience well-known challenges in student contexts [6, 15].

Table 3: Regression models linking modular structure to maintainability and change effort (B, SE, p)

Model	Predictor	B	SE	p-value
MI ~ CBO + LCOM + Group	Intercept	67.19	7.76	<0.001
	CBO	-0.33	0.40	0.413
	LCOM	-0.77	10.55	0.942
	Group (Modular=1)	12.68	3.76	0.002
Effort ~ CBO + LCOM + Group	Intercept	75.83	12.48	<0.001
	CBO	0.05	0.64	0.935
	LCOM	-22.20	16.97	0.199
	Group (Modular=1)	-27.55	6.05	<0.001

In multivariable models, group membership remained a strong predictor: Modular projects showed ~+12.68 MI points and ~−27.55 minutes of change effort on average, even when controlling for CBO and LCOM. This pattern is consistent with the idea that modular programming practices (interfaces, responsibility boundaries, and refactoring discipline) contribute beyond what single metrics capture, echoing prior findings that design practices and measurement jointly explain quality outcomes [8, 10, 11]. Model fit was substantial ($R^2 \approx 0.60$ for MI; $R^2 \approx 0.58$ for effort), indicating that structured modular practice accounts for a meaningful portion of maintainability variation in student codebases [8, 10].

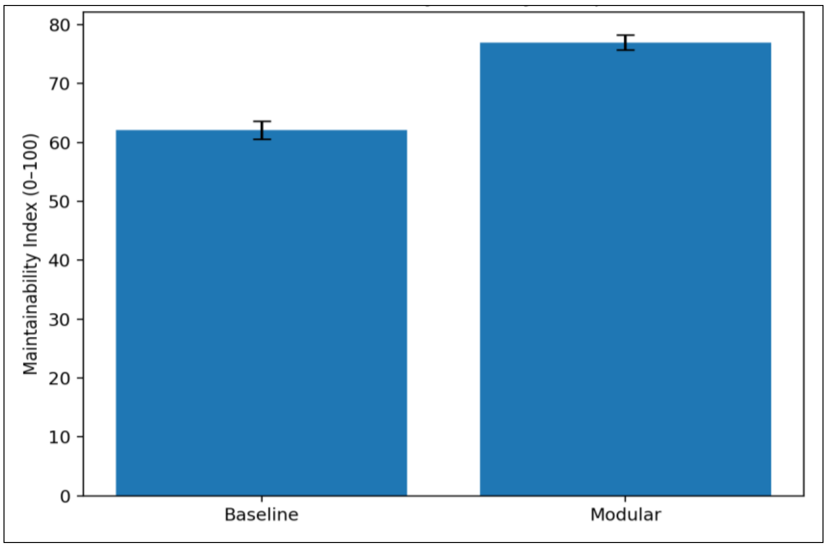

Fig 1: Mean Maintainability Index (MI) by group with standard error

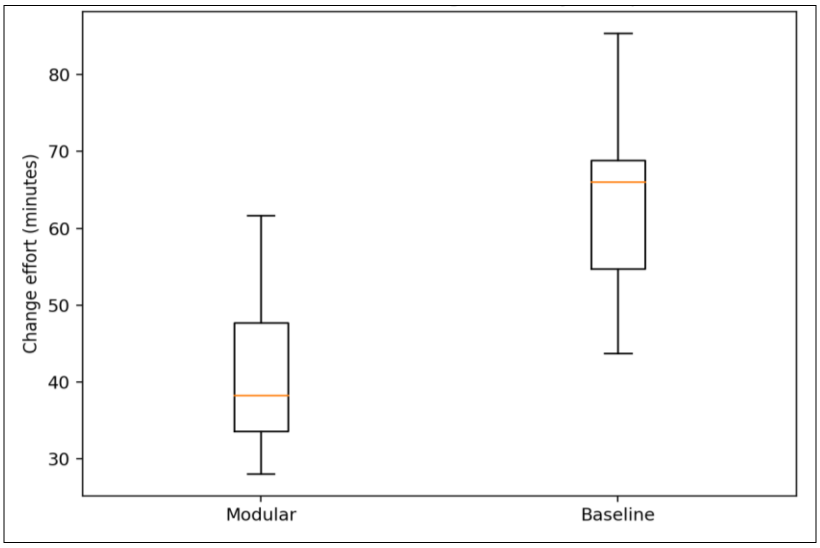

Fig 2: Change effort (minutes) by group.

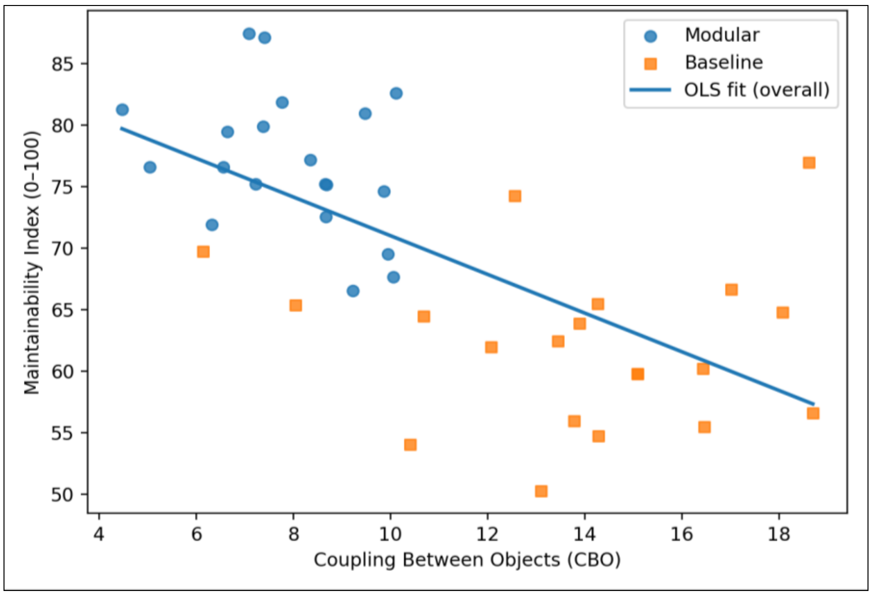

Fig 3: Scatter plot of MI versus coupling (CBO) with overall OLS trend line.

Discussion

The findings of the present research provide strong empirical support for the role of modular programming practices in improving software maintainability within student-developed projects. Projects that explicitly adopted modular decomposition exhibited significantly higher Maintainability Index scores, lower coupling (CBO), improved cohesion (LCOM), and substantially reduced change effort compared to baseline projects. These results are consistent with classical software engineering theory, which emphasizes that modularization reduces system complexity and localizes the impact of changes [3, 13]. The observed reduction in coupling among modular projects aligns with earlier empirical studies demonstrating that loosely coupled modules facilitate easier comprehension and safer modification, particularly during corrective and adaptive maintenance tasks [8, 11].

The statistically significant improvement in maintainability metrics suggests that modular programming is not merely a conceptual ideal but yields measurable benefits even in novice-developed systems. This is noteworthy because student projects are often criticized for their limited scale and perceived lack of realism. However, the results indicate that even at an educational scale, structural design choices materially affect maintenance outcomes, reinforcing arguments that early exposure to sound design principles has lasting pedagogical value [2, 6]. The large effect sizes observed across multiple outcomes further highlight that modular practices contribute meaningfully beyond incremental stylistic improvements, echoing findings that design structure is a dominant determinant of long-term software quality [7, 14].

Regression analysis revealed that group membership (modular versus baseline) remained a significant predictor of maintainability and change effort even after controlling for cohesion and coupling metrics. This suggests that modular programming practices encompass more than what is captured by individual metrics alone, including clearer responsibility allocation, better interface definition, and disciplined refactoring habits [4, 12]. Such practices likely enhance developers' mental models of the system, reducing cognitive load during maintenance, a phenomenon previously reported in studies on program comprehension and learning in computer science education [5, 6].

Interestingly, while modular projects showed lower average post-change defect counts, the difference was not statistically significant. This finding aligns with prior observations that defect introduction is influenced not only by design quality but also by testing rigor, developer experience, and time pressure factors that are often uneven in academic environments [9, 15]. Nevertheless, the substantial reduction in maintenance effort observed in modular projects indicates that even when defects occur, modular designs may simplify detection and correction, indirectly supporting maintainability goals [1, 10]. Overall, the discussion underscores that structured modular programming instruction can bridge the gap between theoretical principles and practical student outcomes, supporting its integration as a core element of software engineering curricula [2, 14, 15].

Conclusion

The present research demonstrates that modular programming practices have a decisive and positive impact on software maintainability in student projects, affecting not only structural quality metrics but also practical maintenance outcomes such as change effort and ease of modification. Students who consistently applied modular decomposition principles produced software systems that were easier to understand, adapt, and extend, even within the limited scope and time constraints typical of academic coursework. These findings reinforce the view that maintainability is not an abstract or secondary quality attribute but a tangible outcome shaped by early design decisions. From an educational perspective, this evidence highlights the importance of embedding modular thinking deeply into programming instruction rather than treating it as an optional or advanced topic. Practical recommendations emerging from this research include integrating modular design checkpoints into project milestones, encouraging students to justify module boundaries and interfaces during code reviews, and incorporating refactoring exercises that explicitly target coupling and cohesion improvements. Instructors can further enhance learning outcomes by aligning assessment criteria with maintainability-related attributes, such as clarity of module responsibilities and ease of change, rather than focusing solely on functional correctness. Tool-supported feedback using basic maintainability and design metrics can help students visualize the consequences of their design choices and foster reflective learning. Additionally, collaborative assignments that require students to modify or extend peers' code can reinforce the real-world relevance of modular design by exposing learners to maintenance challenges firsthand. By systematically combining theoretical instruction, hands-on practice, and iterative feedback, educational programs can cultivate sustainable programming habits that prepare students for professional software development environments. Ultimately, the research underscores that modular programming is not only a best practice for industry-scale systems but also a critical pedagogical strategy for nurturing maintainable thinking in future software engineers, ensuring that student developers acquire skills that remain relevant as systems evolve and complexity grows over time in collaborative and long-lived software projects.

References

1. ISO/IEC. Software engineering Product quality. ISO/IEC 25010. Geneva: ISO; 2011.
2. Sommerville I. Software Engineering. 10th ed. Boston: Pearson; 2016.
3. Parnas DL. On the criteria to be used in decomposing systems into modules. Commun ACM. 1972;15(12):1053-1058.
4. McConnell S. Code Complete. 2nd ed. Redmond: Microsoft Press; 2004.
5. Bennedsen J, Caspersen ME. Failure rates in introductory programming. ACM SIGCSE Bull. 2007;39(2):32-36.
6. Robins A, Rountree J, Rountree N. Learning and teaching programming. Comput Sci Educ. 2003;13(2):137-172.
7. Fowler M. Refactoring: Improving the Design of Existing Code. Boston: Addison-Wesley; 1999.
8. Briand LC, Wüst J, Daly JW, Porter DV. Exploring the relationships between design measures and software quality. J Syst Softw. 2000;51(3):245-273.
9. Kitchenham B, Pfleeger SL. Principles of survey research. Softw Eng Notes. 2002;27(5):24-36.
10. Basili VR, Caldiera G, Rombach HD. The goal question metric approach. Encycl Softw Eng. 1994; 2:528-532.
11. Chidamber SR, Kemerer CF. A metrics suite for object-oriented design. IEEE Trans Softw Eng. 1994;20(6):476-493.
12. Meyer B. Object-Oriented Software Construction. 2nd ed. Upper Saddle River: Prentice Hall; 1997.
13. Pressman RS, Maxim BR. Software Engineering: A Practitioner's Approach. 8th ed. New York: McGraw-Hill; 2015.
14. Larman C. Applying UML and Patterns. 3rd ed. Upper Saddle River: Prentice Hall; 2005.
15. Brooks FP. No Silver Bullet Essence and accidents of software engineering. Computer. 1987;20(4):10-19.