CognitivaTech: Ingeniería de Software Inteligente y Sistemas Adaptativos. 2025; 2(1):6
doi: 10.63688/cognitivatech.v2.i1.6

ORIGINAL

Design principles in programming languages and their impact on software quality

Principios de diseño en lenguajes de programación y su impacto en la calidad del software

Segundo Alexander Rosado Martínez[1]

[1] Universidad Estatal de Milagro. Milagro, Ecuador.

Submitted: 06-01-2025 | Revised: 19-02-2025 | Accepted: 16-03-2025 | Published: 18-03-2025

Corresponding Author: Segundo Alexander Rosado Martínez

ABSTRACT

This study analyzes the fundamental principles that guide the design of modern programming languages, highlighting their impact on software quality. Based on a qualitative approach and a documentary analysis of scientific literature, three essential pillars were identified: correctness, maintainability, and performance. The results show that correctness is the primary objective, as it enables the development of programs that fulfill their intended purpose while minimizing errors from early stages through mechanisms such as strict type systems, compile-time validation, and clear abstractions. Regarding maintainability, it was found that principles such as modularity, low coupling, high cohesion, encapsulation, and separation of concerns facilitate the understanding, modification, and evolution of software, especially in dynamic development environments. On the other hand, performance, although secondary to the previous aspects, remains essential, as it ensures efficient program execution through techniques such as optimized compilation and predictable execution models. Furthermore, a direct relationship among these three principles was identified, showing that correctness enhances maintainability, which in turn contributes to improved performance. The analysis of the Wybe programming language served as an example of how these principles can be integrated into a concrete design proposal. In conclusion, the study demonstrates that a balanced approach among correctness, maintainability, and performance is key to developing reliable, adaptable, and efficient software.

Keywords: Programming languages; Software correctness; Maintainability; Performance.

RESUMEN

El presente estudio analiza los principios fundamentales que orientan el diseño de lenguajes de programación modernos, destacando su impacto en la calidad del software. A partir de un enfoque cualitativo y un análisis documental de literatura científica, se identificaron tres pilares esenciales: la correctitud, la mantenibilidad y el rendimiento. Los resultados evidencian que la correctitud constituye el objetivo prioritario, ya que permite desarrollar programas que cumplen con su propósito y minimizan errores desde etapas tempranas mediante mecanismos como sistemas de tipos estrictos, validaciones en compilación y abstracciones claras. En cuanto a la mantenibilidad, se determinó que principios como la modularidad, el bajo acoplamiento, la alta cohesión, el encapsulamiento y la separación de responsabilidades facilitan la comprensión, modificación y evolución del software, especialmente en entornos de desarrollo dinámicos. Por otro lado, el rendimiento, aunque subordinado a los aspectos anteriores, sigue siendo indispensable, ya que garantiza la eficiencia en la ejecución de los programas mediante técnicas como la compilación optimizada y modelos de ejecución predecibles. Asimismo, se identificó una relación directa entre estos tres principios, evidenciando que la correctitud favorece la mantenibilidad, y esta, a su vez, contribuye a optimizar el rendimiento. El análisis del lenguaje de programación Wybe permitió ejemplificar cómo estos principios pueden integrarse en una propuesta concreta de diseño. En conclusión, el estudio demuestra que un enfoque equilibrado entre correctitud, mantenibilidad y rendimiento es clave para el desarrollo de software confiable, adaptable y eficiente.

Palabras clave: Lenguajes de programación; Correctitud del software; Mantenibilidad; Rendimiento.

INTRODUCTION

In contemporary software development, the selection and design of a programming language are critical determinants of the quality of the resulting systems. A general-purpose language must be sufficiently flexible to accommodate a wide range of scenarios, from simple applications to complex solutions, while maintaining a balance among usability, reliability, and efficiency (Berrio Pacheco, Y., & Benitez Hernandez, R. M. 2024). The objective is not only to enable rapid code writing, but also to facilitate the creation of correct, clear, and sustainable programs over time (Nanz & Furia, 2015; Ray et al., 2014).

A key aspect of language design is software correctness, as a program may be highly efficient, but if it does not perform its intended function correctly, it loses all value. For this reason, modern languages incorporate mechanisms that allow programmers to precisely express their intentions, minimize errors, and ensure internal consistency (Romero Aguilar, J. F., & Freire Cobo, L. E. 2024). This is achieved through the use of well-defined abstractions, predictable behaviors, and structures that enhance code robustness against unforeseen situations (Hofmann, 2000; Meyer, 1988).

Similarly, maintainability and the capacity for software evolution are essential factors, given that a significant portion of a system's lifecycle is devoted to its modification and improvement (Banker et al., 1998; Erlikh, 2000). In this context, principles such as modularity, separation of concerns, and encapsulation support the construction of more adaptable systems, where changes can be implemented without significantly affecting other parts of the program (Parnas, 1972; Stevens et al., 1974). These qualities are particularly important in agile methodologies and collaborative environments, where software evolves continuously (Espinosa et al., 2007; Herbsleb & Mockus, 2003).

Finally, although correctness and maintainability are priorities, performance remains a fundamental consideration. An effective programming language should enable the development of programs with adequate performance, utilizing techniques such as efficient compilation, code optimization, and comprehensible execution models (Lattner & Adve, 2004; Krishnan et al., 2000). Additionally, software complexity directly affects productivity and maintainability, reinforcing the need for balanced design (Benaroch & Lyytinen, 2023).

Collectively, these principles guide the design of modern programming languages such as Wybe, which aims to facilitate the creation of correct, maintainable, and efficient software. This perspective reflects a trend in software engineering toward developing tools that not only simplify programming but also promote best practices and high-quality outcomes, consistent with both classical and contemporary studies in the field.

METHODS

This research adopts a qualitative, descriptive–analytical approach to examine the essential principles guiding the design of modern programming languages, with particular emphasis on software correctness, maintainability, and performance. This approach enables the interpretation and integration of theoretical concepts from software engineering with current proposals such as the Wybe language.

The study employs a non-experimental design, as no variables are manipulated; instead, existing theoretical contributions are analyzed. Furthermore, it is based on a documentary design, supported by the review of relevant scientific literature, including academic articles, conference proceedings, and foundational works related to software engineering and programming languages.

Information was collected through a systematic literature review, prioritizing indexed academic sources such as scientific articles, conference proceedings, and specialized books. Documents were selected with a focus on:

principles of programming language design,
software quality (correctness, maintainability, and performance),
comparative studies on languages and development practices.

The selection criteria considered thematic relevance, the currency of the sources, and their theoretical contribution to the research problem.

The analysis was conducted using a content analysis approach, organizing the information into three main categories, consistent with the initial framework:

Software correctness: review of mechanisms that contribute to error prevention and consistency assurance.

Maintainability and evolution: identification of principles such as modularity, encapsulation, and separation of concerns.

Performance: study of strategies related to compilation, optimization, and efficient execution.

Subsequently, relationships were established among these categories to understand how they influence the design of current programming languages and the quality of the resulting software.

Additionally, a conceptual analysis of the Wybe programming language is included, using it as a reference to illustrate the practical application of the studied principles. This demonstrates how design decisions directly impact the ease of development, maintenance, and software efficiency.

The validity of the study is supported by the triangulation of theoretical sources, contrasting different authors and relevant approaches. Additionally, coherence is maintained among the objectives, introduction, and methodological development, ensuring that the analysis adequately addresses the research problem.

RESULTS

The documentary analysis enabled the identification of several relevant findings regarding the principles guiding the design of modern programming languages and their influence on software quality. First, it is evident that correctness is a central element in these languages. Languages that prioritize this aspect often incorporate mechanisms aimed at preventing errors from the earliest stages of development, such as more rigorous type systems, compile-time validations, and abstractions that reduce ambiguities. Furthermore, internal consistency and predictable language behavior facilitate developers' ability to anticipate code operation, thereby reducing the likelihood of failures. In this regard, the capacity to clearly express the programmer's intent is directly linked to greater software reliability.

Regarding maintainability, the results show that aspects such as modularity and proper separation of responsibilities are fundamental to facilitating software evolution. Languages that promote low coupling and high cohesion allow changes to be introduced without significantly affecting other parts of the system. Similarly, practices such as encapsulation and information hiding contribute to protecting system integrity and simplifying updates. These characteristics support the construction of more flexible software, especially in environments where requirements change frequently, as is the case in agile methodologies.

Table 1. Design principles of programming languages and their impact on software quality

Category	Identified principles	Description	Impact on software quality
Correctness	Predictable behavior	Language operations behave as expected	Reduces errors and increases reliability
	Expressive abstractions	Allows the programmer's intent to be clearly expressed	Improves accuracy and reduces ambiguities
	Enforced consistency	Validations that prevent execution of incorrect code	Prevents failures at early stages
	Robustness	Proper handling of errors and unexpected conditions	Prevents critical failures at runtime
Maintainability	Modularity	Division of the system into independent components	Facilitates changes and maintenance
	Low coupling and high cohesion	Minimizes dependencies and organizes responsibilities	Improves system understanding and evolution
	Data encapsulation	Hides internal implementation details	Maintains system integrity
	Separation of responsibilities	Distinction between interface and implementation	Enables modification of components without impacting others
Performance	Compilation to native code	Generation of efficient executable code	Enhances execution speed
	Compiler optimization	Automatic enhancement of generated code	Improves efficiency without altering design
	Predictable performance	Enables estimation of resource consumption	Facilitates development decision-making
	Incremental compilation	Enables rapid testing of modifications	Enhances developer productivity

Source: Own elaboration

With respect to performance, although it is not always the primary objective, it remains a key factor in ensuring the applicability of a language in different contexts. The results indicate that languages oriented toward efficiency enable the generation of optimized code through advanced compilation techniques and direct execution in machine code. Moreover, the predictability of performance is important, as it allows developers to estimate program behavior in terms of time and resource consumption, facilitating decision-making during development.

Finally, a close relationship among these three principles is identified. Correctness supports maintainability, as correct software is easier to modify and extend. In turn, good code organization contributes to improved performance. The analysis of the Wybe language exemplifies how these principles can be integrated into a concrete design proposal, highlighting the importance of considering software quality from the initial stages. Overall, the findings show that the design of modern programming languages is oriented toward creating tools that not only facilitate the development of functional software but also promote its reliability, adaptability, and efficiency.

DISCUSSION

The results obtained allow for reflection on the importance of design principles in programming languages and their direct relationship with software quality. First, it is confirmed that correctness must be the central axis in language development, since an incorrect program loses its value regardless of its performance or ease of modification. This finding is consistent with theoretical approaches in software engineering that prioritize error prevention from early stages through mechanisms such as compile-time validations and strict type systems.

Regarding maintainability, the results show that principles such as modularity, low coupling, and high cohesion not only facilitate code comprehension but also allow for its controlled evolution. This is especially relevant in current contexts where software is in constant change. The integration of practices such as encapsulation and separation of responsibilities reinforces the idea that good design not only responds to immediate needs but also anticipates future modifications.

On the other hand, performance, although placed at a third level of priority, remains an essential component. The discussion shows that a necessary balance exists between efficiency and clarity: optimizing code should not compromise the understanding or reliability of the system. In this sense, modern languages seek to offer tools that allow for adequate performance without transferring all the complexity to the programmer.

A relevant aspect identified is the interdependence among the three principles analyzed. Correctness supports maintainability, since well-structured and error-free code is easier to modify. In turn, good software organization allows optimizations to be implemented more efficiently, positively impacting performance. This relationship shows that the principles should not be addressed in isolation, but as an integrated set within language design.

The analysis of the Wybe language reinforces these approaches, showing how a focus on interface integrity and data flow control can significantly contribute to improving software quality. However, it is also recognized that implementing these principles may involve certain challenges, such as a possible decrease in initial development speed due to stricter constraints, which requires an appropriate balance between control and flexibility.

In summary, the discussion emphasizes that the design of modern programming languages is oriented toward providing tools that promote not only efficiency but also software reliability and sustainability. This reflects a shift in focus, where productivity is measured not only by coding speed but also by the ability to develop correct, maintainable, and efficient solutions over time.

CONCLUSIONS

The analysis conducted allows us to conclude that programming language design plays a fundamental role in software quality, as it directly influences how developers construct, maintain, and optimize their programs. In this context, it is evident that correctness, maintainability, and performance are essential pillars that must be considered in an integrated manner in the design of modern languages.

First, correctness is established as the primary objective, as it ensures that the software reliably fulfills its intended purpose. Mechanisms that promote consistency, clarity, and error prevention help reduce failures from early stages, resulting in safer and more robust systems.

Second, maintainability is confirmed as a key factor in the software lifecycle, given that a significant portion of development effort is devoted to the evolution and adaptation of systems. Principles such as modularity, encapsulation, and separation of concerns facilitate the understanding and modification of code, enabling efficient responses to changing requirements.

Finally, performance, although secondary to the previous two aspects, remains an indispensable element to ensure the viability of applications in various contexts. A well-designed language should provide tools that enable adequate levels of efficiency without compromising software clarity or reliability.

Taken together, these findings indicate that modern programming languages, such as Wybe, aim to go beyond mere code writing by promoting best practices that ensure the development of high-quality software. Therefore, it is concluded that a balanced approach among correctness, maintainability, and performance is essential to address current software engineering challenges and to contribute to the development of more reliable, adaptable, and efficient systems.

REFERENCES

1. Banker, R. D., Davis, G. B., & Slaughter, S. A. (1998). Software development practices, software complexity, and software maintenance performance: A field study. Management Science, 44(4), 433–450. https://doi.org/10.1287/mnsc.44.4.433
2. Benaroch, M., & Lyytinen, K. (2023). How much does software complexity matter for maintenance productivity? IEEE Transactions on Software Engineering, 49(5), 2459–2475. https://doi.org/10.1109/TSE.2022.3153703
3. Berrio Pacheco, Y., & Benitez Hernandez, R. M. (2024). Estudio de incidencia de lenguajes de programación y entornos de desarrollo en el mercado laboral. https://repositorio.unicordoba.edu.co/server/api/core/bitstreams/1286aa8f-6124-418c-928e-39afca743206/content
4. Brooks, F. P. (1975). The mythical man-month. ACM SIGPLAN Notices, 10(6), 193–200. https://doi.org/10.1145/390016.808319
5. Erlikh, L. (2000). Leveraging legacy system dollars for e-business. IT Professional, 2(3), 17–23. https://doi.org/10.1109/6294.848309
6. Espinosa, J. A., Slaughter, S. A., Kraut, R. E., & Herbsleb, J. D. (2007). Team knowledge and coordination in geographically distributed software development. Journal of Management Information Systems, 24(1), 135–169. https://doi.org/10.2753/MIS0742-1222240104
7. Gosling, J., & McGilton, H. (1995). The Java language environment: A white paper. Sun Microsystems. https://www.oracle.com/java/technologies/javase/whitepaper.html
8. Herbsleb, J. D., & Mockus, A. (2003). An empirical study of speed and communication in globally distributed software development. IEEE Transactions on Software Engineering, 29(6), 481–494. https://doi.org/10.1109/TSE.2003.1205177
9. Hofmann, M. (2000). A type system for bounded space and functional in-place update. En Programming Languages and Systems (pp. 165–179). Springer. https://doi.org/10.1007/3-540-46425-5_11
10. Krishnan, M. S., Kriebel, C. H., Kekre, S., & Mukhopadhyay, T. (2000). An empirical analysis of productivity and quality in software products. Management Science, 46(6), 745–759. https://doi.org/10.1287/mnsc.46.6.745.12035
11. Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. En Proceedings of the International Symposium on Code Generation and Optimization. https://doi.org/10.1109/CGO.2004.1281665
12. Lehman, M. M. (1996). Laws of software evolution revisited. En European Workshop on Software Process Technology (pp. 108–124). Springer. https://doi.org/10.1007/BFb0017026
13. Meyer, B. (1988). Object-oriented software construction. Prentice Hall. https://books.google.com.ec/books/about/Object_oriented_Software_Construction.html?id=D6i-QgAACAAJ&redir_esc=y
14. Nanz, S., & Furia, C. A. (2015). A comparative study of programming languages in Rosetta Code. En Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (pp. 778–788). IEEE. https://doi.org/10.1109/ICSE.2015.90
15. Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), 1053–1058. https://doi.org/10.1145/361598.361623
16. Ramasubbu, N., & Balan, R. K. (2007). Globally distributed software development project performance: An empirical analysis. En Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (pp. 125–134). https://doi.org/10.1145/1287624.1287643
17. Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2014). A large scale study of programming languages and code quality in GitHub. En Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 155–165). ACM. https://doi.org/10.1145/2635868.2635922
18. Romero Aguilar, J. F., & Freire Cobo, L. E. (2024). Diseño de diagramas de componentes de software, utilizando un gran modelo de lenguaje y aplicando técnicas de optimización para lograr resultados contextualmente relevante (Doctoral dissertation, ESPOL. FIEC). https://www.dspace.espol.edu.ec/handle/123456789/67425
19. Stevens, W. P., Myers, G. J., & Constantine, L. L. (1974). Structured design. IBM Systems Journal, 13(2), 115–139. https://doi.org/10.1147/sj.132.0115

FUNDING

No financing

CONFLICT OF INTEREST

The authors declare that there are no conflicts of interest in this study and that the ethical procedures established by this journal have been followed. Furthermore, they confirm that this work has not been published, either partially or in full, in any other journal.

AUTHORSHIP CONTRIBUTIONS

Conceptualization: SARM
Data curation: SARM
Formal analysis: SARM
Investigation: SARM
Methodology: SARM
Project administration: SARM
Resources: SARM
Software: SARM
Supervision: SARM
Validation: SARM
Visualization: SARM
Writing – original draft: SARM
Writing – review and editing: SARM