CognitivaTech: Ingeniería de Software Inteligente y Sistemas Adaptativos. 2025; 2(1):22
doi: 10.63688/cognitivatech.v2.i1.22

ORIGINAL

Analysis of programming language design and its impact on software quality

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

Gustavo Adolfo Hernández Rosado[1]

[1] Universidad Espíritu Santo. Guayaquil, Ecuador.

Submitted: 13-05-2025 | Revised: 22-06-2025 | Accepted: 28-07-2025 | Published: 29-07-2025

Corresponding Author: Gustavo Adolfo Hernandez Rosado

ABSTRACT

This study analyzes the fundamental principles in programming language design and their impact on software quality, maintainability, and performance within the context of modern software engineering. Using a qualitative, descriptive analytical approach, a comprehensive review of relevant scientific and technical literature was conducted, complemented by a comparative analysis of different programming paradigms, including imperative, object-oriented, functional, and logic-based approaches. The results indicate that software correctness is the primary objective in language design, supported by robust type systems, compile time verification, and expressive abstractions that help reduce errors and improve code reliability. Furthermore, maintainability is strongly associated with principles such as modularity, low coupling, high cohesion, and encapsulation, which facilitate software evolution and reduce maintenance costs. Regarding performance, findings reveal a direct relationship with execution models and compilation strategies, highlighting that advances in optimization techniques have enabled high-level languages to achieve competitive performance without compromising developer productivity. Additionally, a clear trend toward multi-paradigm languages is observed, integrating diverse programming approaches to address the increasing complexity of modern systems. Overall, the findings emphasize the importance of balancing correctness, maintainability, and performance as a foundation for developing efficient and reliable software systems.

Keywords: programming languages; software quality; maintainability; performance.

RESUMEN

El presente estudio analiza los principios fundamentales en el diseño de lenguajes de programación y su impacto en la calidad del software, la mantenibilidad y el rendimiento, en el contexto de la ingeniería de software moderna. A través de un enfoque cualitativo de tipo descriptivo analítico, se realizó una revisión documental de literatura científica y técnica relevante, complementada con un análisis comparativo de distintos paradigmas de programación, incluyendo enfoques imperativos, orientados a objetos, funcionales y lógicos. Los resultados evidencian que la corrección del software constituye el eje prioritario en el diseño de lenguajes, promovida mediante sistemas de tipos robustos, verificación en tiempo de compilación y abstracciones expresivas que permiten reducir errores y mejorar la confiabilidad del código. Asimismo, se identificó que la mantenibilidad está fuertemente asociada a principios como la modularidad, el bajo acoplamiento, la alta cohesión y el encapsulamiento, los cuales facilitan la evolución del software y disminuyen los costos de mantenimiento. En cuanto al rendimiento, se observa una relación directa con el modelo de ejecución y las estrategias de compilación, destacando que los avances en optimización han permitido a lenguajes de alto nivel alcanzar desempeños competitivos sin sacrificar productividad. Adicionalmente, se evidencia una tendencia hacia lenguajes multiparadigma que integran diferentes enfoques para responder a la complejidad de los sistemas actuales. En conjunto, los hallazgos confirman la necesidad de equilibrar corrección, mantenibilidad y rendimiento como base para el desarrollo de software eficiente y confiable.

Palabras clave: lenguajes de programación; calidad del software; mantenibilidad; rendimiento.

INTRODUCTION

The design and evolution of programming languages have been central to the development of software engineering, given their direct impact on the quality, maintainability, and efficiency of computer systems (Alay, J. I. G., & Sevillano, R. P. C., 2022). Over recent decades, research in this field has demonstrated that the selection and construction of a language not only influence developer productivity but also affect the robustness, correctness, and scalability of the resulting software. In this context, comparative studies have analyzed various programming languages and their relationship with code quality and performance in real-world environments (Nanz & Furia, 2015; Ray et al., 2014). Additionally, broader research has highlighted the strategic importance of languages within modern software engineering (Gunter et al., 1996; Gupta et al., 2015).

Historically, the fundamental principles of software design have been oriented toward facilitating the construction of correct, understandable, and easily modifiable programs. In this regard, classical works such as those by Parnas (1972) and Stevens et al. (1974) established the foundations for key concepts such as modularity and structured decomposition. These principles, together with data encapsulation and separation of concerns, have been widely recognized as pillars for ensuring maintainable and evolvable systems. According to Mauw et al. (2004), these approaches not only influence software architecture but must also be directly supported by programming languages, which serve as the primary means of abstraction between the developer's intent and computational execution.

Accordingly, the design of modern languages has evolved to incorporate mechanisms that promote correctness from the early stages of development, reducing the probability of errors and improving software reliability (Ghaleb, T. A., Aljasser, K., & Alturki, M. A., 2021). The inclusion of more rigorous type systems and formal models has been extensively studied in the literature (Wadler, 1990; Hofmann, 2000). Similarly, proposals such as those by Wadler (1997) and Hughes (1989) emphasize the role of expressive abstractions in enhancing code clarity and precision. These advances address the need to balance productivity and precision, avoiding both unnecessary complexity and excessive permissiveness that can lead to hard-to-detect failures.

Conversely, maintainability and the capacity for software evolution have become critical factors, considering that a significant portion of the total cost of projects is allocated to maintenance activities (Banker et al., 1998). Consequently, languages should facilitate the creation of flexible systems, where changes can be made in a localized and controlled manner without negatively affecting the rest of the system. This approach is related to the use of structures such as abstract data types (Naish et al., 2016) and techniques of specialization and modularization in logic programming (Puebla & Hermenegildo, 1995; Winsborough, 1992). Moreover, the use of declarative and logic languages has been shown to provide advantages in software organization and maintainability (Sterling & Yalçinalp, 1996; Gange et al., 2015).

Performance remains a relevant aspect in the design of programming languages, especially in applications that demand high computational efficiency. Although performance is closely linked to the implementation of compilers and interpreters, research such as that by Lattner and Adve (2004) highlights the role of compilation frameworks in code optimization. Similarly, techniques such as garbage collection have been extensively studied to improve memory management (Boehm & Weiser, 1988; Boehm et al., 1991). These advances demonstrate how language design decisions can facilitate or hinder the optimization of generated code.

Additionally, the development of domain-specific languages and language generation tools has opened new perspectives in software design, allowing for greater adaptation to particular contexts (Mernik et al., 2005; Mernik, 2005). Similarly, approaches such as those proposed by Dobson et al. (2000) and Golden (1985) demonstrate the importance of adapting languages to specific application needs, reinforcing their role in software engineering.

In this context, there is a need to analyze and propose language design approaches that integrate correctness, maintainability, and performance in a balanced manner, with the aim of meeting the current demands of software development in various application domains, while also considering the historical evolution and current trends in language design (Dijkstra, 1963; Parr, 2013; Ghaleb, Aljasser, & Alturki, 2021).

METHODS

This research adopts a qualitative applied approach, aimed at analyzing the fundamental principles in the design of programming languages and their relationship with software quality, maintainability, and performance. This approach enables a comprehensive understanding of how design decisions influence software development practice, beyond a simple quantitative evaluation.

The methodological design is descriptive–analytical. In the first phase, an exhaustive documentary review was conducted of scientific, technical, and historical literature related to programming languages, development paradigms, and software engineering principles. This review included international conference articles, academic publications, technical reports, and foundational documents that have contributed to the theoretical development of the field. The documentary analysis enabled the identification of patterns, recurring approaches, and key principles that have guided the evolution of programming languages.

In the second phase, a comparative analysis of different language design approaches was conducted, considering aspects such as software correctness, abstraction expressiveness, modularity, maintainability, and performance. Representative proposals from imperative, object-oriented, functional, and logic languages were examined to establish similarities, differences, and trends in their evolution. This analysis facilitated an understanding of how each paradigm addresses the challenges associated with modern software development.

Subsequently, a conceptual synthesis phase was carried out, in which the findings from the previous stages were integrated to structure a set of design principles aimed at improving software quality. In this phase, priority was given to identifying relationships between concepts such as modularity, encapsulation, separation of concerns, and error control, highlighting their impact on the construction of robust and evolvable systems.

Finally, a critical interpretation of the results was performed to evaluate the relevance of the identified principles in light of current software development needs. This stage enabled reflection on the limitations of existing approaches and the importance of proposing solutions that balance developer productivity with system reliability and efficiency.

The methodological process followed ensures a comprehensive view of the research problem by combining theoretical analysis and critical reflection, providing a solid foundation for future research and proposals in programming language design.

RESULTS

The results obtained derive from the documentary and comparative analysis of the main approaches in programming language design, in accordance with the objectives outlined in the introduction and the adopted methodological approach. From the literature review, three key dimensions were identified that influence the effectiveness of a programming language: software correctness, maintainability/evolution, and performance. These dimensions do not operate in isolation but are closely interrelated.

Design principles and their impact on software quality

The analysis showed that languages prioritizing strict control mechanisms, such as robust type systems and compile-time verification, tend to produce software with a lower incidence of errors. Similarly, semantic clarity and predictability of language behavior contribute significantly to the reduction of logical faults.

Table 1. Relationship between language features and software quality.

Language feature	Impact on quality	Description
Strong typing	High	Reduces runtime errors
Weak typing	Medium	Provides greater flexibility but is more prone to errors
Expressive abstractions	High	Facilitates clear representation of program logic
Low predictability	Low	Leads to unexpected behaviors
Compile-time checking	High	Detects errors before execution

Source: Own elaboration

In contrast, languages with greater syntactic flexibility and fewer restrictions tend to facilitate rapid initial development but may increase the likelihood of errors in later stages of the software life cycle.

Maintainability and evolution of software

The results indicate that maintainability is directly related to how a language allows code to be structured. Principles such as modularity, low coupling, and high cohesion were identified as key factors in facilitating software evolution.

It was observed that languages promoting encapsulation and separation of concerns enable localized modifications without affecting other parts of the system, thereby reducing maintenance costs and improving scalability.

Table 2. Design factors associated with maintainability.

Factor	Level of impact	Observed effect
Modularity	High	Facilitates reuse and maintenance
Low coupling	High	Reduces inter-component dependencies
High cohesion	High	Enhances code clarity and organization
Encapsulation	High	Ensures data integrity
Lack of role separation	Low	Impairs modification and maintenance

Source: Own elaboration

Performance and computational efficiency

With regard to performance, it was found that languages designed with structures close to hardware and compilation to native code demonstrate better efficiency. However, this advantage may involve increased development complexity.

Conversely, languages with higher levels of abstraction tend to sacrifice some performance in exchange for greater productivity and ease of use. Nevertheless, the inclusion of optimizing compilers and improvements in runtime environments has helped to reduce this gap in recent years.

Synthesis of results

Based on the integration of the findings, it was identified that there is no single optimal programming language for all contexts; rather, effectiveness depends on the balance among the three analyzed axes. Nonetheless, more modern languages tend to converge toward a model that combines:

- High safety and correctness through advanced type systems
- Ease of maintenance through modularity and encapsulation
- Acceptable performance through optimization and efficient compilation

Table 3. Relationship between language design and performance.

Feature	Impact on performance	Observation
Compilation to native code	High	Increases execution efficiency
Interpretation	Medium	Lower performance, increased flexibility
High-level abstraction	Medium	Reduces complexity but may decrease efficiency
Compiler optimization	High	Substantial performance improvement
Complex execution model	Low	Complicates performance prediction

Source: Own elaboration

The results confirm that the design of programming languages is strongly influenced by the need to balance multiple factors, with software correctness emerging as a fundamental priority, followed by maintainability and, lastly, performance. This order reflects a consistent trend in the analyzed literature, where software reliability is prioritized over extreme optimization.

Table 4. General comparison of programming paradigms.

Paradigm	Correctness	Maintainability	Performance	Main feature
Imperative	Medium	Medium	High	Direct control over execution flow
Object-oriented	High	High	Medium	Encapsulation and reusability
Functional	High	High	Medium	Immutability and pure functions
Logic	High	Medium	Low	Declarativity and automated reasoning

Source: Own elaboration

Similarly, there is evidence of an evolution toward languages that integrate multiple paradigms, aiming to leverage the advantages of each approach and mitigate their limitations. This convergence suggests that the future of language design will be oriented toward hybrid models that are more flexible and adaptive to different development contexts.

DISCUSSION

The results obtained establish a consistent relationship between the identified programming language design principles and the trends reported in the analyzed scientific literature. In particular, there is a clear convergence toward prioritizing software correctness as the central focus in the design of modern languages, which aligns with various studies highlighting the importance of reducing errors from the early stages of development. This orientation is reflected in the adoption of stricter type systems, compile-time verification mechanisms, and abstractions that promote predictable behavior.

Regarding software quality, the findings of this research are consistent with large-scale empirical studies demonstrating that certain language features significantly influence the incidence of defects and code reliability. In this regard, the results support the idea that languages with strong typing and more formal structures tend to produce more robust software, although in some cases they may involve a steeper learning curve or lower initial development speed.

Furthermore, the classical principles of software engineering, such as modularity, low coupling, and high cohesion, identified in the results, remain fully valid and are aligned with the theoretical foundations established in pioneering works on structured design and system decomposition. The analyzed evidence confirms that these principles are not only relevant at the architectural level, but must also be directly supported by programming languages to facilitate their correct application. In this context, data encapsulation and information hiding emerge as essential mechanisms to improve software maintainability and evolution.

Moreover, the discussion of the results reinforces the relevance of the principle of separation of concerns, widely recognized in the literature as a key factor for effective software organization. Languages that promote clear interfaces and well-defined structures facilitate code comprehension and reduce cognitive complexity, which is fundamental in large-scale projects or collaborative environments.

With respect to performance, the results indicate an inherent tension between efficiency and the level of abstraction, which has been widely documented in studies on compilers and execution models. While low-level languages or those that compile directly to native code offer advantages in terms of performance, high-level languages have evolved significantly due to advanced optimization techniques, narrowing the existing gap. This finding is consistent with research highlighting the role of infrastructures such as compilation frameworks and intermediate analysis systems in improving performance without sacrificing productivity.

Additionally, the comparative analysis of paradigms shows that no single approach dominates in all scenarios, which is consistent with the diversity of proposals present in the literature. The functional and logic paradigms, for example, are notable for their ability to guarantee formal properties and reduce side effects, while the imperative and object-oriented paradigms continue to be widely used due to their closeness to the execution model and practical applicability. This diversity reinforces the trend toward multiparadigm languages capable of integrating different forms of abstraction within the same environment.

Furthermore, the results obtained are aligned with more recent proposals in language design, which emphasize the need to achieve a balance between correctness, maintainability, and performance, prioritizing software reliability over other factors. This hierarchy responds to the increasing complexity of current systems and the need to minimize risks in critical environments.

Nevertheless, it is important to note that, although the results show high coherence with the literature, there are inherent limitations to the qualitative approach adopted. In particular, the absence of direct empirical validation on real projects may restrict the generalizability of the findings. Therefore, future research could complement this approach through experimental studies or quantitative analyses that allow for a more precise measurement of the impact of each language feature on specific software quality metrics.

In summary, the discussion confirms that the principles identified in this research are not only supported by the existing literature, but also reflect a clear evolution in programming language design toward safer, more maintainable, and balanced models. This theoretical and practical convergence reinforces the validity of the results and their relevance for modern software development.

CONCLUSIONS

This research enabled a comprehensive analysis of the fundamental principles guiding programming language design and their impact on software quality, maintainability, and performance. Based on the review and comparative analysis conducted, it is evident that language development has evolved toward a more balanced approach, prioritizing software correctness as the primary objective, followed by ease of maintenance and, finally, computational performance.

The results confirm that the incorporation of mechanisms such as robust type systems, compile-time verification, and well-defined abstractions significantly contributes to error reduction and the development of more reliable software. Furthermore, classical software engineering principles—such as modularity, low coupling, high cohesion, and encapsulation—remain decisive for enabling systems to efficiently evolve in response to new requirements.

Regarding performance, it is concluded that although low-level languages retain advantages in terms of efficiency, advances in compilation and optimization techniques have enabled higher-level languages to achieve acceptable performance levels without compromising developer productivity. This balance is crucial in the current context, which is characterized by increasingly complex and demanding systems.

Another relevant aspect is the consolidation of multiparadigm approaches, which integrate features from different programming models to leverage their strengths and mitigate their limitations. This trend reflects a direct response to the diversity of needs in modern development, where there is no single solution but rather tools adaptable to different contexts and problems.

Nevertheless, this study has limitations due to its qualitative approach and the documentary nature of the analysis, which opens the possibility for future research incorporating empirical methods and quantitative metrics to validate the findings. In particular, it would be relevant to assess the impact of these principles in real projects and across different development environments.

In conclusion, programming language design remains a dynamic and constantly evolving field, in which the pursuit of balance among correctness, maintainability, and performance constitutes the main challenge. The contributions of this research provide a solid conceptual foundation for understanding these dynamics and support the development of more effective tools and approaches in contemporary software engineering.

REFERENCES

1. Alay, J. I. G., & Sevillano, R. P. C. (2022). Evolución de los sistemas de lenguaje de programación a lo largo de la historia. E-IDEA Journal of Engineering Science, 4(10), 14-26. https://revista.estudioidea.org/ojs/index.php/esci/article/view/237
2. Boehm, H. J., & Weiser, M. (1988). Garbage collection in an uncooperative environment. Software: Practice and Experience, 18(9), 807–820. https://doi.org/10.1002/spe.4380180902
3. Boehm, H. J., Demers, A. J., & Shenker, S. (1991). Mostly parallel garbage collection. In Proceedings of PLDI. https://doi.org/10.1145/113445.113449
4. Dobson, S., Nixon, P., Wade, V., Terzis, S., & Fuller, J. (2000). Vanilla: An open language framework. In Generative and Component-Based Software Engineering (pp. 91–104). Springer. https://doi.org/10.1007/3-540-40048-6_8
5. Dijkstra, E. W. (1963). On the design of machine independent programming languages. Annual Review in Automatic Programming, 3, 27–42. https://doi.org/10.1016/S0066-4138(63)80003-8
6. Ghaleb, T. A., Aljasser, K., & Alturki, M. A. (2021). An extensible compiler for implementing software design patterns as concise language constructs. International Journal of Software Engineering and Knowledge Engineering, 31(7), 1043–1067. https://doi.org/10.1142/S0218194021500327
7. Golden, D. G. (1985). Software engineering considerations for the design of simulation languages. Simulation, 45(4), 173–180. https://doi.org/10.1177/003754978504500403
8. Gunter, C., Mitchell, J., & Notkin, D. (1996). Strategic directions in software engineering and programming languages. ACM Computing Surveys, 28(4), 727–737. https://doi.org/10.1145/242223.242283
9. Hughes, J. (1989). Why functional programming matters. The Computer Journal, 32(2), 98–107. https://doi.org/10.1093/comjnl/32.2.98
10. Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of CGO. https://doi.org/10.1109/CGO.2004.1281665
11. Mauw, S., Wiersma, W. T., & Willemse, T. A. C. (2004). Language-driven system design. International Journal of Software Engineering and Knowledge Engineering, 14(6), 625–663. https://doi.org/10.1142/S0218194004001828
12. Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. ACM Computing Surveys, 37(4), 316–344. https://doi.org/10.1145/1118890.1118892
13. Mernik, M. (2005). Incremental programming language development. Computer Languages, Systems & Structures, 31(1), 1–16. https://doi.org/10.1016/j.cl.2004.02.001
14. Nanz, S., & Furia, C. A. (2015). A comparative study of programming languages in Rosetta Code. In Proceedings of ICSE. https://doi.org/10.1109/ICSE.2015.90
15. Parr, T. (2013). The definitive ANTLR 4 reference. Pragmatic Bookshelf. https://doi.org/10.5555/2501720
16. 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
17. Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2014). A large-scale study of programming languages and code quality in GitHub. In Proceedings of FSE. https://doi.org/10.1145/2635868.2635922
18. Sterling, L., & Yalçinalp, Ü. (1996). Logic programming and software engineering: Implications for software design. The Knowledge Engineering Review, 11(4), 333–345. https://doi.org/10.1017/S026988890000802X
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
20. Wadler, P. (1990). Linear types can change the world! In Programming Concepts and Methods. https://doi.org/10.1007/978-1-4471-3744-9_10
21. Wadler, P. (1997). How to declare an imperative. ACM Computing Surveys, 29(3), 240–263. https://doi.org/10.1145/262009.262011
22. Winsborough, W. (1992). Multiple specialization using minimal-function graph semantics. Journal of Logic Programming, 13(2–3), 259–290. https://doi.org/10.1016/0743-1066(92)90012-O
23. Puebla, G., & Hermenegildo, M. (1995). Implementation of multiple specialization in logic programs. In PEPM. https://doi.org/10.1145/215465.215529
24. Naish, L., Schachte, P., & MacNally, A. (2016). Lightweight efficient safe polymorphic algebraic data types for C. Software: Practice and Experience, 46(12), 1685–1703. https://doi.org/10.1002/spe.2415
25. Hofmann, M. (2000). A type system for bounded space and functional in-place update. In Programming Languages and Systems. https://doi.org/10.1007/3-540-46425-5_11
26. Gange, G., Navas, J. A., Schachte, P., Søndergaard, H., & Stuckey, P. J. (2015). Horn clauses as an intermediate representation for program analysis. Theory and Practice of Logic Programming, 15(4–5), 526–542. https://doi.org/10.1017/S1471068415000211
27. Gupta, G., Pontelli, E., & others. (2015). Language-based software engineering. Science of Computer Programming, 97, 37–40. https://doi.org/10.1016/j.scico.2014.02.010
28. Veloza Peñuela, I. A. (2023) Diseño de un prototipo de modelo con parámetros aplicables al aseguramiento de la calidad en el desarrollo de sistemas expertos como rama de la inteligencia artificial apoyado en arquitectura de tecnología de la información y de la solución. https://repository.unad.edu.co/handle/10596/55396

FUNDING

The authors received no funding for the development of this research.

CONFLICT OF INTEREST

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

AUTHORSHIP CONTRIBUTIONS

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