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.

Recibido: 06-01-2025 | Revisado: 19-02-2025 | Aceptado: 16-03-2025 | Publicado: 18-03-2025

Autor de correspondencia: 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.

INTRODUCCION

En el desarrollo de software actual, la selección y el diseño de un lenguaje de programación son determinantes para la calidad de los sistemas que se construyen. Un lenguaje de propósito general debe ser lo suficientemente flexible para adaptarse a distintos escenarios, desde aplicaciones sencillas hasta soluciones complejas, manteniendo un equilibrio entre facilidad de uso, confiabilidad y eficiencia(Berrio Pacheco, Y., & BenitezHernandez, R. M. 2024). No se trata únicamente de escribir código con rapidez, sino de facilitar la creación de programas correctos, claros y sostenibles en el tiempo(Nanz & Furia, 2015; Ray et al., 2014).

Un aspecto clave en el diseño de lenguajes es la correctitud del software, ya que un programa puede ser muy eficiente, pero si no cumple su función correctamente, pierde todo valor. Por ello, los lenguajes modernos incorporan mecanismos que permiten al programador expresar con precisión lo que desea lograr, minimizar errores y asegurar coherencia interna(Romero Aguilar, J. F., & Freire Cobo, L. E. 2024). Esto se consigue mediante el uso de abstracciones bien definidas, comportamientos predecibles y estructuras que fortalecen la robustez del código ante situaciones imprevistas(Hofmann, 2000; Meyer, 1988).

De igual manera, la mantenibilidad y la capacidad de evolución del software son factores esenciales, considerando que gran parte del tiempo de vida de un sistema se dedica a su modificación y mejora(Banker et al., 1998; Erlikh, 2000). En este sentido, principios como la modularidad, la separación de responsabilidades y el encapsulamiento favorecen la construcción de sistemas más adaptables, donde los cambios pueden realizarse sin afectar significativamente otras partes del programa(Parnas, 1972; Stevens et al., 1974). Estas cualidades son especialmente importantes en metodologías ágiles y entornos colaborativos, donde el software evoluciona de manera constante(Espinosa et al., 2007;Herbsleb& Mockus, 2003).

Por último, aunque la correctitud y la mantenibilidad son prioritarias, el rendimiento sigue siendo un elemento fundamental. Un buen lenguaje de programación debe permitir desarrollar programas con un desempeño adecuado, apoyándose en técnicas como la compilación eficiente, la optimización del código y modelos de ejecución comprensibles(Lattner & Adve, 2004; Krishnan et al., 2000). Asimismo, la complejidad del software influye directamente en la productividad y en la facilidad de mantenimiento, lo que refuerza la necesidad de un diseño equilibrado(Benaroch&Lyytinen, 2023).

En conjunto, estos principios guían el diseño de lenguajes de programación modernos como Wybe, cuyo enfoque se centra en facilitar la creación de software correcto, mantenible y eficiente. Esta perspectiva refleja una tendencia en la ingeniería de software orientada a desarrollar herramientas que no solo simplifiquen la programación, sino que también fomenten buenas prácticas y resultados de alta calidad, en línea con estudios clásicos y contemporáneos del área.

METODOS

La presente investigación se desarrolla bajo un enfoque cualitativo de carácter descriptivo–analítico, con el propósito de examinar los principios esenciales que orientan el diseño de los lenguajes de programación modernos, poniendo especial atención en la correctitud, la mantenibilidad y el rendimiento del software. Este enfoque facilita la interpretación y articulación de conceptos teóricos de la ingeniería de software con propuestas actuales como el lenguaje Wybe.

El estudio adopta un diseño no experimental, dado que no se intervienen ni manipulan variables, sino que se analizan aportes teóricos ya existentes. Además, se fundamenta en un diseño documental, apoyado en la revisión de literatura científica relevante, como artículos académicos, memorias de conferencias y obras clásicas relacionadas con la ingeniería de software y los lenguajes de programación.

La información se recopiló mediante una revisión bibliográfica sistemática, priorizando fuentes académicas indexadas, tales como artículos científicos, actas de congresos y libros especializados. Se seleccionaron documentos enfocados en:

principios de diseño de lenguajes de programación,
calidad del software (correctitud, mantenibilidad y rendimiento),
estudios comparativos sobre lenguajes y prácticas de desarrollo.

Los criterios de selección consideraron la pertinencia temática, la actualidad de las fuentes y su contribución teórica al problema investigado.

El análisis se llevó a cabo mediante un enfoque de análisis de contenido, organizando la información en tres categorías principales, coherentes con el planteamiento inicial:

Correctitud del software: revisión de mecanismos que contribuyen a prevenir errores y asegurar consistencia.

Mantenibilidad y evolución: identificación de principios como modularidad, encapsulamiento y separación de responsabilidades.

Rendimiento: estudio de estrategias relacionadas con la compilación, optimización y ejecución eficiente.

Posteriormente, se establecieron vínculos entre estas categorías para comprender cómo influyen en el diseño de los lenguajes de programación actuales y en la calidad del software generado.

Como complemento, se incorpora un análisis conceptual del lenguaje de programación Wybe, utilizándolo como referencia para ilustrar la aplicación práctica de los principios estudiados. Esto permite evidenciar cómo las decisiones de diseño impactan directamente en la facilidad de desarrollo, el mantenimiento y la eficiencia del software.

La validez del estudio se respalda mediante la triangulación de fuentes teóricas, contrastando distintos autores y enfoques relevantes. Asimismo, se asegura la coherencia entre los objetivos, la introducción y el desarrollo metodológico, garantizando que el análisis responda de manera adecuada al problema de investigación.

RESULTADOS

El análisis documental permitió identificar diversos hallazgos relevantes sobre los principios que orientan el diseño de los lenguajes de programación modernos y su influencia en la calidad del software.

Tabla 1. Principios de diseño de lenguajes de programación y su impacto en la calidad del software

Categoría	Principios identificados	Descripción	Impacto en la calidad del software
Correctitud	Comportamiento predecible	Las operaciones del lenguaje funcionan de forma esperada	Reduce errores y aumenta la confiabilidad
	Abstracciones expresivas	Permite expresar claramente la intención del programador	Mejora la precisión y disminuye ambigüedades
	Consistencia forzada	Validaciones que evitan ejecutar código incorrecto	Previene fallos en etapas tempranas
	Robustez	Manejo adecuado de errores y condiciones inesperadas	Evita fallos críticos en ejecución
Mantenibilidad	Modularidad	División del sistema en componentes independientes	Facilita cambios y mantenimiento
	Bajo acoplamiento y alta cohesión	Minimiza dependencias y organiza responsabilidades	Mejora la comprensión y evolución del sistema
	Encapsulamiento de datos	Oculta detalles internos de implementación	Protege la integridad del sistema
	Separación de responsabilidades	Diferencia entre interfaz e implementación	Permite modificar partes sin afectar otras
Rendimiento	Compilación a código nativo	Generación de código ejecutable eficiente	Mejora la velocidad de ejecución
	Optimización del compilador	Mejora automática del código generado	Incrementa eficiencia sin afectar diseño
	Rendimiento predecible	Permite estimar consumo de recursos	Facilita toma de decisiones en desarrollo
	Compilación incremental	Permite probar cambios rápidamente	Aumenta productividad del desarrollador

Fuente: Elaboración propia

En primer lugar, se evidencia que la correctitud es un elemento central en estos lenguajes. Aquellos que priorizan este aspecto suelen incorporar mecanismos destinados a prevenir errores desde las primeras etapas del desarrollo, como sistemas de tipos más rigurosos, validaciones en tiempo de compilación y abstracciones que reducen ambigüedades.

Además, se observa que la consistencia interna y el comportamiento predecible del lenguaje facilitan que los desarrolladores anticipen el funcionamiento del código, lo que disminuye la probabilidad de fallos. En este sentido, la capacidad de expresar con claridad la intención del programador se vincula directamente con una mayor confiabilidad del software.

Por otro lado, en relación con la mantenibilidad, los resultados muestran que aspectos como la modularidad y la adecuada separación de responsabilidades son fundamentales para facilitar la evolución del software. Los lenguajes que promueven bajo acoplamiento y alta cohesión permiten introducir cambios sin afectar significativamente otras partes del sistema. Asimismo, prácticas como el encapsulamiento y el ocultamiento de la información contribuyen a proteger la integridad del sistema y a simplificar su actualización. Estas características favorecen la construcción de software más flexible, especialmente en entornos donde los requisitos cambian con frecuencia, como ocurre en metodologías ágiles.

En cuanto al rendimiento, aunque no siempre constituye el objetivo principal, sigue siendo un factor clave para garantizar la aplicabilidad de un lenguaje en distintos contextos. Los resultados indican que los lenguajes orientados a la eficiencia permiten generar código optimizado mediante técnicas avanzadas de compilación y ejecución directa en código máquina. Además, la predictibilidad del rendimiento resulta importante, ya que permite a los desarrolladores estimar el comportamiento del programa en términos de tiempo y consumo de recursos, facilitando la toma de decisiones durante el desarrollo.

Finalmente, se identifica una relación estrecha entre estos tres principios. La correctitud favorece la mantenibilidad, ya que un software correcto resulta más sencillo de modificar y ampliar. A su vez, una buena organización del código contribuye a mejorar el rendimiento. El análisis del lenguaje Wybe permitió ejemplificar cómo estos principios pueden integrarse en una propuesta concreta de diseño, resaltando la importancia de considerar la calidad del software desde las etapas iniciales. En conjunto, los hallazgos evidencian que el diseño de lenguajes de programación modernos se orienta hacia la creación de herramientas que no solo faciliten el desarrollo de software funcional, sino que también promuevan su confiabilidad, adaptabilidad y eficiencia.

DISCUSION

Los resultados obtenidos permiten reflexionar sobre la importancia de los principios de diseño en los lenguajes de programación y su relación directa con la calidad del software. En primer lugar, se confirma que la correctitud debe ser el eje central en el desarrollo de lenguajes, ya que un programa incorrecto pierde su valor independientemente de su rendimiento o facilidad de modificación. Este hallazgo coincide con enfoques teóricos de la ingeniería de software que priorizan la prevención de errores desde etapas tempranas mediante mecanismos como validaciones en tiempo de compilación y sistemas de tipos estrictos.

En relación con la mantenibilidad, los resultados evidencian que principios como la modularidad, el bajo acoplamiento y la alta cohesión no solo facilitan la comprensión del código, sino que también permiten su evolución de manera controlada. Esto es especialmente relevante en contextos actuales donde el software está en constante cambio. La integración de prácticas como el encapsulamiento y la separación de responsabilidades refuerza la idea de que un buen diseño no solo responde a necesidades inmediatas, sino que también anticipa futuras modificaciones.

Por otra parte, el rendimiento, aunque ubicado en un tercer nivel de prioridad, sigue siendo un componente esencial. La discusión muestra que existe un equilibrio necesario entre eficiencia y claridad: optimizar el código no debe comprometer la comprensión ni la confiabilidad del sistema. En este sentido, los lenguajes modernos buscan ofrecer herramientas que permitan alcanzar un rendimiento adecuado sin trasladar toda la complejidad al programador.

Un aspecto relevante identificado es la interdependencia entre los tres principios analizados. La correctitud favorece la mantenibilidad, ya que un código bien estructurado y libre de errores es más fácil de modificar. A su vez, una buena organización del software permite implementar optimizaciones de manera más eficiente, impactando positivamente en el rendimiento. Esta relación evidencia que los principios no deben abordarse de forma aislada, sino como un conjunto integrado dentro del diseño del lenguaje.

El análisis del lenguaje Wybe refuerza estos planteamientos, mostrando cómo un enfoque centrado en la integridad de las interfaces y en el control del flujo de datos puede contribuir significativamente a mejorar la calidad del software.

Sin embargo, también se reconoce que la implementación de estos principios puede implicar ciertos desafíos, como una posible disminución en la rapidez inicial de desarrollo debido a restricciones más estrictas, lo que exige un equilibrio adecuado entre control y flexibilidad.

En síntesis, la discusión pone de manifiesto que el diseño de lenguajes de programación modernos está orientado a proporcionar herramientas que promuevan no solo la eficiencia, sino también la confiabilidad y sostenibilidad del software. Esto representa un cambio de enfoque, donde la productividad no se mide únicamente por la rapidez de codificación, sino por la capacidad de desarrollar soluciones correctas, mantenibles y eficientes a lo largo del tiempo.

CONCLUSION

El análisis realizado permite concluir que el diseño de lenguajes de programación desempeña un papel fundamental en la calidad del software, ya que influye directamente en la forma en que los desarrolladores construyen, mantienen y optimizan sus programas. En este contexto, se evidencia que la correctitud, la mantenibilidad y el rendimiento constituyen pilares esenciales que deben ser considerados de manera integrada en el diseño de lenguajes modernos.

En primer lugar, la correctitud se posiciona como el objetivo prioritario, debido a que garantiza que el software cumpla con su propósito de manera confiable. Los mecanismos que promueven la consistencia, la claridad y la prevención de errores permiten reducir fallos desde etapas tempranas, lo que se traduce en sistemas más seguros y robustos.

En segundo lugar, la mantenibilidad se confirma como un factor clave en el ciclo de vida del software, dado que gran parte del esfuerzo de desarrollo se centra en la evolución y adaptación de los sistemas. Principios como la modularidad, el encapsulamiento y la separación de responsabilidades facilitan la comprensión y modificación del código, permitiendo responder de manera eficiente a cambios en los requisitos.

Finalmente, el rendimiento, aunque subordinado a los dos aspectos anteriores, sigue siendo un elemento indispensable para garantizar la viabilidad de las aplicaciones en distintos contextos. Un lenguaje bien diseñado debe ofrecer herramientas que permitan alcanzar niveles adecuados de eficiencia sin comprometer la claridad ni la confiabilidad del software.

En conjunto, estos hallazgos reflejan que los lenguajes de programación modernos, como Wybe, buscan ir más allá de la simple escritura de código, promoviendo buenas prácticas que aseguren el desarrollo de software de alta calidad. Por lo tanto, se concluye que un enfoque equilibrado entre correctitud, mantenibilidad y rendimiento es esencial para enfrentar los desafíos actuales de la ingeniería de software y contribuir al desarrollo de sistemas más confiables, adaptables y eficientes.

Referencias

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

Financiación

Ninguna

Conflictos de interés

Los autores afirman que no existen conflictos de intereses en este estudio y que se han seguido éticamente los procesos establecidos por esta revista. Además, aseguran que este trabajo no ha sido publicado parcial ni totalmente en ninguna otra revista.

Contribución de autoría

Conceptualización: SARM
Curación de datos: SARM
Análisis formal: SARM
Investigación: SARM
Metodología: SARM
Administración del proyecto: SARM
Recursos: SARM
Software: SARM
Supervisión: SARM
Validación: SARM
Visualización: SARM
Redacción – Borrador original: SARM
Redacción – Revisión y edición: SARM