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.

Recibido: 13-05-2025 | Revisado: 22-06-2025 | Aceptado: 28-07-2025 | Publicado: 29-07-2025

Autor de correspondencia: 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.

© 2026, Los autores. Este es un artículo de acceso abierto distribuido bajo los términos de la licencia Creative Commons Attribution (CC BY 4.0).

INTRODUCCION

El diseño y evolución de los lenguajes de programación ha sido un eje central en el desarrollo de la ingeniería de software, dado su impacto directo en la calidad, mantenibilidad y eficiencia de los sistemas informáticos(Alay, J. I. G., & Sevillano, R. P. C., 2022). A lo largo de las últimas décadas, la investigación en este campo ha evidenciado que la elección y construcción de un lenguaje no solo influye en la productividad del desarrollador, sino también en la robustez, corrección y escalabilidad del software resultante. En este contexto, estudios comparativos han analizado distintos lenguajes de programación y su relación con la calidad del código y su desempeño en entornos reales(Nanz & Furia, 2015; Ray et al., 2014). Asimismo, investigaciones más amplias han destacado la importancia estratégica de los lenguajes dentro de la ingeniería de software moderna(Gunter et al., 1996; Gupta et al., 2015).

Históricamente, los principios fundamentales del diseño de software han estado orientados a facilitar la construcción de programas correctos, comprensibles y fácilmente modificables. En este sentido, trabajos clásicos como los deParnas (1972)yStevens et al. (1974)sentaron las bases de conceptos clave como la modularidad y la descomposición estructurada. Estos principios, junto con el encapsulamiento de datos y la separación de responsabilidades, han sido ampliamente reconocidos como pilares para garantizar sistemas mantenibles y evolutivos. De acuerdo conMauwet al. (2004), estos enfoques no solo influyen en la arquitectura del software, sino que deben ser soportados directamente por los lenguajes de programación, los cuales actúan como el medio principal de abstracción entre el pensamiento del desarrollador y la ejecución computacional.

En este sentido, el diseño de lenguajes modernos ha evolucionado hacia la incorporación de mecanismos que promuevan la corrección desde etapas tempranas del desarrollo, reduciendo la probabilidad de errores y mejorando la confiabilidad del software (Ghaleb, T. A.,Aljasser, K., &Alturki, M. A. 2021). La inclusión de sistemas de tipos más rigurosos y modelos formales ha sido ampliamente estudiada en la literatura(Wadler, 1990; Hofmann, 2000). Asimismo, propuestas como las deWadler (1997)yHughes (1989)destacan el papel de las abstracciones expresivas en la mejora de la claridad y precisión del código. Estos avances responden a la necesidad de equilibrar productividad y precisión, evitando tanto la complejidad innecesaria como la permisividad excesiva que puede derivar en fallos difíciles de detectar.

Por otro lado, la mantenibilidad y la capacidad de evolución del software se han convertido en factores críticos, considerando que una parte significativa del costo total de los proyectos se destina a actividades de mantenimiento(Banker et al., 1998). En consecuencia, los lenguajes deben favorecer la creación de sistemas flexibles, donde los cambios puedan realizarse de manera localizada y controlada, sin afectar negativamente al resto del sistema. Este enfoque se relaciona con el uso de estructuras como tipos abstractos de datos(Naish et al., 2016)y técnicas de especialización y modularización en programación lógica(Puebla & Hermenegildo, 1995;Winsborough, 1992). Además, el uso de lenguajes declarativos y lógicos ha demostrado aportar ventajas en la organización del software y su mantenibilidad(Sterling &Yalçinalp, 1996; Gange et al., 2015).

Finalmente, el rendimiento continúa siendo un aspecto relevante en el diseño de lenguajes de programación, especialmente en aplicaciones que demandan alta eficiencia computacional. Aunque el desempeño está estrechamente ligado a la implementación de compiladores e intérpretes, investigaciones como las deLattner y Adve (2004)destacan el papel de los frameworks de compilación en la optimización del código. De igual manera, técnicas como la recolección de basura han sido ampliamente estudiadas para mejorar la gestión de memoria(Boehm & Weiser, 1988; Boehm et al., 1991). Estos avances muestran cómo las decisiones de diseño del lenguaje pueden facilitar u obstaculizar la optimización del código generado.

Adicionalmente, el desarrollo de lenguajes específicos de dominio y herramientas de generación de lenguajes ha abierto nuevas perspectivas en el diseño de software, permitiendo una mayor adaptación a contextos particulares(Mernik et al., 2005; Mernik, 2005). Asimismo, enfoques como los propuestos porDobson et al. (2000)yGolden (1985)evidencian la importancia de adaptar los lenguajes a necesidades específicas de aplicación, reforzando su papel en la ingeniería de software.

En este contexto, surge la necesidad de analizar y proponer enfoques de diseño de lenguajes que integren de manera equilibrada la corrección, la mantenibilidad y el rendimiento, con el objetivo de satisfacer las demandas actuales del desarrollo de software en diversos dominios de aplicación, considerando además la evolución histórica y las tendencias actuales en el diseño de lenguajes(Dijkstra, 1963; Parr, 2013; Ghaleb,Aljasser, &Alturki, 2021).

METODOS

La presente investigación adopta un enfoque cualitativo de carácter aplicado, orientado a analizar los principios fundamentales en el diseño de lenguajes de programación y su relación con la calidad, mantenibilidad y rendimiento del software. Este enfoque permite comprender de manera integral cómo las decisiones de diseño influyen en la práctica del desarrollo de software, más allá de una simple evaluación cuantitativa.

El diseño metodológico es de tipo descriptivo–analítico. En una primera fase, se llevó a cabo una revisión documental exhaustiva de literatura científica, técnica e histórica relacionada con lenguajes de programación, paradigmas de desarrollo y principios de ingeniería de software. Esta revisión incluyó artículos de conferencias internacionales, publicaciones académicas, reportes técnicos y documentos fundamentales que han contribuido al desarrollo teórico del área. El análisis documental permitió identificar patrones, enfoques recurrentes y principios clave que han guiado la evolución de los lenguajes de programación.

En una segunda fase, se realizó un análisis comparativo de distintos enfoques de diseño de lenguajes, considerando aspectos como la corrección del software, la expresividad de las abstracciones, la modularidad, la mantenibilidad y el rendimiento. Para ello, se examinaron propuestas representativas de lenguajes imperativos, orientados a objetos, funcionales y lógicos, con el fin de establecer similitudes, diferencias y tendencias en su evolución. Este análisis permitió comprender cómo cada paradigma aborda los desafíos asociados al desarrollo de software moderno.

Posteriormente, se desarrolló una fase de síntesis conceptual, en la cual se integraron los hallazgos obtenidos en las etapas previas para estructurar un conjunto de principios de diseño orientados a mejorar la calidad del software. En esta fase, se priorizó la identificación de relaciones entre conceptos como modularidad, encapsulamiento, separación de responsabilidades y control de errores, destacando su impacto en la construcción de sistemas robustos y evolutivos.

Finalmente, se llevó a cabo una interpretación crítica de los resultados, orientada a evaluar la pertinencia de los principios identificados frente a las necesidades actuales del desarrollo de software. Esta etapa permitió reflexionar sobre las limitaciones de los enfoques existentes y la importancia de proponer soluciones que equilibren la productividad del desarrollador con la confiabilidad y eficiencia del sistema.

El proceso metodológico seguido garantiza una visión integral del problema de estudio, combinando análisis teórico y reflexión crítica, lo que permite aportar una base sólida para futuras investigaciones y propuestas en el diseño de lenguajes de programación.

RESULTADOS

Los resultados obtenidos se derivan del análisis documental y comparativo de los principales enfoques en el diseño de lenguajes de programación, en coherencia con los objetivos planteados en la introducción y el enfoque metodológico adoptado. A partir de la revisión de la literatura, se identificaron tres dimensiones clave que condicionan la efectividad de un lenguaje de programación: corrección del software, mantenibilidad/evolución y rendimiento. Estas dimensiones no actúan de forma aislada, sino que se encuentran estrechamente interrelacionadas.

Principios de diseño y su impacto en la calidad del software

El análisis evidenció que los lenguajes que priorizan mecanismos de control estricto, como sistemas de tipos robustos y verificación en tiempo de compilación, tienden a producir software con menor incidencia de errores. Asimismo, la claridad semántica y la previsibilidad del comportamiento del lenguaje contribuyen significativamente a la reducción de fallos lógicos.

En contraste, lenguajes con mayor flexibilidad sintáctica y menor restricción tienden a favorecer la rapidez de desarrollo inicial, pero pueden incrementar la probabilidad de errores en etapas posteriores del ciclo de vida del software.

Mantenibilidad y evolución del software

Los resultados muestran que la mantenibilidad está directamente relacionada con la forma en que un lenguaje permite estructurar el código. Principios como la modularidad, el bajo acoplamiento y la alta cohesión fueron identificados como factores determinantes para facilitar la evolución del software.

Tabla 1. Relación entre características del lenguaje y calidad del software.

Característica del lenguaje	Impacto en la calidad	Descripción
Tipado fuerte	Alto	Reduce errores en tiempo de ejecución
Tipado débil	Medio	Mayor flexibilidad, pero más propenso a errores
Abstracciones expresivas	Alto	Facilitan la representación clara de la lógica del programa
Baja previsibilidad	Bajo	Genera comportamientos inesperados
Control en compilación	Alto	Detecta errores antes de la ejecución

Fuente: Elaboración propia

Se observó que los lenguajes que promueven encapsulación y separación de responsabilidades permiten realizar modificaciones localizadas sin afectar otras partes del sistema, lo cual reduce el costo de mantenimiento y mejora la escalabilidad.

Tabla 2. Factores de diseño asociados a la mantenibilidad.

Factor	Nivel de impacto	Efecto observado
Modularidad	Alto	Facilita la reutilización y mantenimiento
Bajo acoplamiento	Alto	Reduce dependencia entre componentes
Alta cohesión	Alto	Mejora la claridad y organización del código
Encapsulamiento	Alto	Protege la integridad de los datos
Falta de separación de roles	Bajo	Dificulta modificaciones y mantenimiento

Fuente: Elaboración propia

Rendimiento y eficiencia computacional

En relación con el rendimiento, se identificó que los lenguajes diseñados con estructuras cercanas al hardware y compilación a código nativo presentan mejores resultados en eficiencia. Sin embargo, esta ventaja puede implicar un incremento en la complejidad del desarrollo.

Por otro lado, lenguajes con mayores niveles de abstracción tienden a sacrificar parte del rendimiento a cambio de una mayor productividad y facilidad de uso. No obstante, la inclusión de compiladores optimizadores y mejoras en los entornos de ejecución ha permitido reducir esta brecha en los últimos años.

Síntesis de resultados

A partir de la integración de los hallazgos, se identificó que no existe un único lenguaje de programación óptimo para todos los contextos, sino que su efectividad depende del equilibrio entre los tres ejes analizados. No obstante, los lenguajes más modernos tienden a converger hacia un modelo que combina:

- Alta seguridad y corrección mediante sistemas de tipos avanzados
- Facilidad de mantenimiento a través de modularidad y encapsulación
- Rendimiento aceptable mediante optimización y compilación eficiente

Tabla 3. Relación entre diseño del lenguaje y rendimiento

Característica	Impacto en rendimiento	Observación
Compilación a código nativo	Alto	Mayor eficiencia en ejecución
Interpretación	Medio	Menor rendimiento, mayor flexibilidad
Abstracción elevada	Medio	Reduce complejidad, pero puede afectar eficiencia
Optimización del compilador	Alto	Mejora significativa del desempeño
Modelo de ejecución complejo	Bajo	Dificulta predicción del rendimiento

Fuente: Elaboración propia

Tabla 4. Comparación general de paradigmas de programación

Paradigma	Corrección	Mantenibilidad	Rendimiento	Característica principal
Imperativo	Media	Media	Alta	Control directo del flujo de ejecución
Orientado a objetos	Alta	Alta	Media	Encapsulación y reutilización
Funcional	Alta	Alta	Media	Inmutabilidad y funciones puras
Lógico	Alta	Media	Baja	Declaratividad y razonamiento automático

Fuente: Elaboración propia

Los resultados confirman que el diseño de lenguajes de programación está fuertemente influenciado por la necesidad de equilibrar múltiples factores, donde la corrección del software emerge como prioridad fundamental, seguida de la mantenibilidad y, finalmente, el rendimiento. Este orden refleja una tendencia consistente en la literatura analizada, en la que se prioriza la confiabilidad del software sobre la optimización extrema.

Asimismo, se evidencia una evolución hacia lenguajes que integran múltiples paradigmas, buscando aprovechar las ventajas de cada enfoque y mitigar sus limitaciones. Esta convergencia sugiere que el futuro del diseño de lenguajes estará orientado hacia modelos híbridos, más flexibles y adaptativos a distintos contextos de desarrollo.

DISCUSION

Los resultados obtenidos permiten establecer una relación consistente entre los principios de diseño de lenguajes de programación identificados y las tendencias reportadas en la literatura científica analizada. En particular, se observa una clara convergencia hacia la priorización de la corrección del software como eje central en el diseño de lenguajes modernos, lo cual coincide con diversos estudios que destacan la importancia de reducir errores desde etapas tempranas del desarrollo. Esta orientación se refleja en la adopción de sistemas de tipos más estrictos, mecanismos de verificación en compilación y abstracciones que favorecen un comportamiento predecible.

En relación con la calidad del software, los hallazgos de esta investigación son coherentes con estudios empíricos a gran escala que han demostrado que ciertas características de los lenguajes influyen significativamente en la incidencia de defectos y en la confiabilidad del código. En este sentido, los resultados obtenidos respaldan la idea de que lenguajes con tipado fuerte y estructuras más formales tienden a producir software más robusto, aunque en algunos casos puedan implicar una curva de aprendizaje más pronunciada o una menor velocidad inicial de desarrollo.

Por otro lado, los principios clásicos de la ingeniería de software, como la modularidad, el bajo acoplamiento y la alta cohesión, identificados en los resultados, mantienen plena vigencia y se alinean con los fundamentos teóricos establecidos en trabajos pioneros sobre diseño estructurado y descomposición de sistemas. La evidencia analizada confirma que estos principios no solo son relevantes a nivel arquitectónico, sino que también deben estar soportados directamente por los lenguajes de programación para facilitar su correcta aplicación. En este contexto, la encapsulación de datos y el ocultamiento de información emergen como mecanismos esenciales para mejorar la mantenibilidad y evolución del software.

Asimismo, la discusión de los resultados permite reforzar la relevancia del principio de separación de responsabilidades, ampliamente reconocido en la literatura como un factor clave para la organización efectiva del software. Los lenguajes que promueven interfaces claras y estructuras bien definidas facilitan la comprensión del código y reducen la complejidad cognitiva, lo cual es fundamental en proyectos de gran escala o en entornos colaborativos.

En cuanto al rendimiento, los resultados muestran una tensión inherente entre eficiencia y nivel de abstracción, lo cual ha sido ampliamente documentado en estudios sobre compiladores y modelos de ejecución. Si bien los lenguajes de bajo nivel o con compilación directa a código nativo ofrecen ventajas en términos de desempeño, los lenguajes de alto nivel han evolucionado significativamente gracias a técnicas avanzadas de optimización, reduciendo la brecha existente. Este hallazgo coincide con investigaciones que destacan el papel de infraestructuras como frameworks de compilación y sistemas de análisis intermedio en la mejora del rendimiento sin sacrificar la productividad.

Por otra parte, el análisis comparativo de paradigmas evidencia que no existe un enfoque único que domine en todos los escenarios, lo cual coincide con la diversidad de propuestas presentes en la literatura. Los paradigmas funcional y lógico, por ejemplo, destacan por su capacidad para garantizar propiedades formales y reducir efectos secundarios, mientras que los paradigmas imperativo y orientado a objetos continúan siendo ampliamente utilizados por su cercanía al modelo de ejecución y su aplicabilidad práctica. Esta diversidad refuerza la tendencia hacia lenguajes multiparadigma, capaces de integrar distintas formas de abstracción en un mismo entorno.

Adicionalmente, los resultados obtenidos se alinean con propuestas más recientes en el diseño de lenguajes, donde se enfatiza la necesidad de lograr un equilibrio entre corrección, mantenibilidad y rendimiento, priorizando la confiabilidad del software sobre otros factores. Esta jerarquización responde a la creciente complejidad de los sistemas actuales y a la necesidad de minimizar riesgos en entornos críticos.

No obstante, es importante señalar que, aunque los resultados presentan una alta coherencia con la literatura, existen limitaciones inherentes al enfoque cualitativo adoptado. En particular, la ausencia de validación empírica directa sobre proyectos reales puede restringir la generalización de los hallazgos. Por ello, futuras investigaciones podrían complementar este enfoque mediante estudios experimentales o análisis cuantitativos que permitan medir de manera más precisa el impacto de cada característica del lenguaje en métricas específicas de calidad del software.

En síntesis, la discusión confirma que los principios identificados en esta investigación no solo están respaldados por la literatura existente, sino que también reflejan una evolución clara en el diseño de lenguajes de programación hacia modelos más seguros, mantenibles y equilibrados. Esta convergencia teórica y práctica refuerza la validez de los resultados y su relevancia para el desarrollo de software moderno.

CONCLUSION

La presente investigación permitió analizar de manera integral los principios fundamentales que orientan el diseño de lenguajes de programación y su impacto en la calidad del software, la mantenibilidad y el rendimiento. A partir de la revisión y el análisis comparativo realizados, se evidencia que el desarrollo de lenguajes ha evolucionado hacia un enfoque más equilibrado, donde la corrección del software se posiciona como el objetivo prioritario, seguido de la facilidad de mantenimiento y, finalmente, del desempeño computacional.

Los resultados confirman que la incorporación de mecanismos como sistemas de tipos robustos, verificación en tiempo de compilación y abstracciones bien definidas contribuye significativamente a la reducción de errores y a la construcción de software más confiable. Asimismo, se destaca que principios clásicos de la ingeniería de software, tales como la modularidad, el bajo acoplamiento, la alta cohesión y el encapsulamiento, continúan siendo determinantes en la capacidad de los sistemas para evolucionar de manera eficiente frente a nuevos requerimientos.

En relación con el rendimiento, se concluye que, si bien los lenguajes de bajo nivel mantienen ventajas en términos de eficiencia, los avances en técnicas de compilación y optimización han permitido que lenguajes de mayor nivel de abstracción alcancen niveles de desempeño aceptables sin comprometer la productividad del desarrollador. Este equilibrio resulta clave en el contexto actual, caracterizado por sistemas cada vez más complejos y demandantes.

Otro aspecto relevante es la consolidación de los enfoques multiparadigma, los cuales integran características de distintos modelos de programación con el fin de aprovechar sus fortalezas y mitigar sus limitaciones. Esta tendencia refleja una respuesta directa a la diversidad de necesidades del desarrollo moderno, donde no existe una solución única, sino herramientas adaptables a diferentes contextos y problemas.

No obstante, el estudio presenta limitaciones derivadas de su enfoque cualitativo y del carácter documental del análisis, lo que abre la posibilidad de futuras investigaciones que incorporen métodos empíricos y métricas cuantitativas para validar los hallazgos obtenidos. En particular, sería relevante evaluar el impacto de estos principios en proyectos reales y en distintos entornos de desarrollo.

En conclusión, el diseño de lenguajes de programación continúa siendo un campo dinámico y en constante evolución, en el que la búsqueda de equilibrio entre corrección, mantenibilidad y rendimiento constituye el principal desafío. Los aportes de esta investigación ofrecen una base conceptual sólida para comprender estas dinámicas y contribuyen al desarrollo de herramientas y enfoques más efectivos en la ingeniería de software contemporánea.

Referencias

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

Financiación

Los autores no recibieron financiamiento para el desarrollo de esta investigación.

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: (GAHR)
Curación de datos: (GAHR)
Análisis formal: (GAHR)
Investigación: (GAHR)
Metodología: (GAHR)
Administración del proyecto: (GAHR)
Software: (GAHR)
Supervisión: (GAHR)
Validación: (GAHR)
Visualización: (GAHR)
Redacción – Borrador original: (GAHR)
Redacción – Revisión y edición: (GAHR)