PRISMA ODS
REVISTA MULTIDISCIPLINARIA SOBRE DESARROLLO SOSTENIBLE
ISSN: 3072-8452

PROPUESTA PARA MEDIR LAS BUENAS PRÁCTICAS DE PROGRAMACIÓN EN LA CALIDAD DEL SOFTWARE DESARROLLADO POR LOS ESTUDIANTES DE INGENIERÍA EN SISTEMAS COMPUTACIONALES

PROPOSAL FOR MEASURING PROGRAMMING BEST PRACTICES REGARDING THE QUALITY OF SOFTWARE DEVELOPED BY COMPUTER SYSTEMS ENGINEERING STUDENTS

AUTORES/AS

SINDYA YADIRA CASTILLO ORTIZ
TECNOLÓGICO NACIONAL DE MÉXICO / INSTITUTO TECNOLÓGICO DE IGUALA
MÉXICO

ENRIQUE MENA SALGADO
TECNOLÓGICO NACIONAL DE MÉXICO / INSTITUTO TECNOLÓGICO DE IGUALA
MÉXICO

LYDIA CUEVAS BRACAMONTES
TECNOLÓGICO NACIONAL DE MÉXICO / INSTITUTO TECNOLÓGICO DE IGUALA
MÉXICO

JOSÉ CARLOS DÍAZ GARCÍA
TECNOLÓGICO NACIONAL DE MÉXICO / INSTITUTO TECNOLÓGICO DE IGUALA
MÉXICO

MARÍA DEL CARMEN URIÓSTEGUI PERALTA
TECNOLÓGICO NACIONAL DE MÉXICO / INSTITUTO TECNOLÓGICO DE IGUALA
MÉXICO

doi 10.65011

Volumen 5 - Número 4
2026

Redilat
Red de Investigadores Latinoamericanos
PRISMA ODS
REVISTA MULTIDISCIPLINARIA SOBRE DESARROLLO SOSTENIBLE
LATINDES
INSTITUTO LATINOAMERICANO EN DESARROLLO SOSTENIBLE

Propuesta para Medir las Buenas Prácticas de Programación en la Calidad del Software Desarrollado por los Estudiantes de Ingeniería en Sistemas Computacionales

Proposal for Measuring Programming best Practices Regarding the Quality of Software Developed by Computer Systems Engineering Students

Sindya Yadira Castillo Ortiz
sindya.castillo@iguala.tecnm.mx
https://orcid.org/0009-0007-6065-6764
Tecnológico Nacional de México / Instituto Tecnológico de Iguala
México

Lydia Cuevas Bracamontes
lydia.cuevas@iguala.tecnm.mx
https://orcid.org/0000-0003-1624-7377
Tecnológico Nacional de México / Instituto Tecnológico de Iguala
México

Enrique Mena Salgado
enrique.mena@iguala.tecnm.mx
https://orcid.org/0000-0002-8862-7355
Tecnológico Nacional de México / Instituto Tecnológico de Iguala
México

José Carlos Díaz García
jcarlos.diaz@iguala.tecnm.mx
https://orcid.org/0009-0003-9363-0377
Tecnológico Nacional de México / Instituto Tecnológico de Iguala
México

María del Carmen Urióstegui Peralta
carmen.uriostegui@iguala.tecnm.mx
https://orcid.org/0009-0007-6661-9403
Tecnológico Nacional de México / Instituto Tecnológico de Iguala
México

Artículo recibido: 10/08/2026
Aceptado para publicación: 14/09/2026
Conflictos de Intereses: Ninguno que declarar

RESUMEN

En las aulas donde se enseña a programar sucede algo curioso: casi todos los planes de estudio insisten en que el estudiante debe "aplicar buenas prácticas de programación", pero muy pocos programas educativos dicen cómo se comprueba que eso realmente ocurrió. El docente termina evaluando lo que ve (si el programa corre, si entrega el resultado esperado), y deja fuera lo que no se ve a simple vista: si el código es legible, si está documentado, si otra persona podría darle mantenimiento sin sufrir. Este artículo de revisión bibliográfica reúne lo que la literatura especializada dice sobre la relación entre buenas prácticas de programación y calidad del software, con énfasis en el contexto de la formación de ingenieros en sistemas computacionales en instituciones como el Tecnológico Nacional de México (TecNM). A partir de esa revisión se plantea una propuesta de instrumento de medición, organizada en cinco dimensiones (legibilidad y estilo, modularidad y diseño, pruebas, documentación y control de versiones, y mantenibilidad medida con métricas objetivas), pensada para integrarse a la evaluación de proyectos en las asignaturas de programación y de ingeniería de software. La propuesta no busca sustituir el criterio del profesor, sino darle un lenguaje común y datos verificables para justificar sus decisiones de evaluación y para que el estudiante entienda, con evidencia, en qué está fallando.

Palabras clave: buenas prácticas de programación, calidad del software, educación en ingeniería, métricas de código, ingeniería en sistemas computacionales

ABSTRACT

Something curious happens in programming classrooms: nearly every curriculum insists that students must "apply good programming practices," yet few academic programs specify how that claim can actually be verified. Instructors end up grading what is visible (whether the program runs, whether it produces the expected output), while leaving aside what is not immediately apparent: whether the code is readable, documented, or maintainable by someone else. This bibliographic review article gathers what the specialized literature says about the relationship between programming best practices and software quality, focusing on the training of computer systems engineers at institutions such as Mexico's National Technological Institute (TecNM). Based on that review, the article proposes a measurement instrument organized into five dimensions (readability and style, modularity and design, testing, documentation and version control, and maintainability measured through objective metrics), intended to be integrated into project evaluation in programming and software engineering courses. The proposal does not aim to replace the instructor's judgment; rather, it offers a shared vocabulary and verifiable data to justify grading decisions and to help students understand, with evidence, exactly where they are falling short.

Keywords: programming best practices, software quality, engineering education, code metrics, computer systems engineering

INTRODUCCIÓN

Cualquiera que haya evaluado un proyecto final de Programación Orientada a Objetos, o revisado el repositorio de un equipo en la asignatura de Ingeniería de Software, o cualquier otra asignatura que genere proyectos de algún sistema de software, reconoce el escenario: el programa "funciona" —cumple los casos de prueba que el propio estudiante decidió mostrar— pero por dentro es un amasijo de variables llamadas a, b, x1, funciones de doscientas líneas y ni un solo comentario documentando dichas funciones. El estudiante aprobó la asignatura, pero, la pregunta que casi nunca nos hacemos con suficiente rigor es si ese estudiante egresará sabiendo programar bien, o solamente sabiendo que su programa corrió el día de la entrega.

La literatura sobre ingeniería de software lleva más de cuatro décadas insistiendo en que la calidad de un producto de software no depende únicamente de que funcione, sino de un conjunto más amplio de atributos: qué tan fácil es mantenerlo, qué tan portable es, qué tan seguro, qué tan eficiente (Callejas Cuervo et al., 2017). La norma ISO/IEC 25010, heredera de la vieja ISO/IEC 9126, formaliza esto en ocho características de calidad de producto —entre ellas la mantenibilidad, la fiabilidad y la usabilidad— que sirven de referencia obligada para quien quiera hablar de calidad de software con algo más que impresiones (ISO/IEC, 2011). El problema es que estos modelos se diseñaron pensando en productos industriales, evaluados por equipos de aseguramiento de calidad con herramientas y tiempo que un aula de licenciatura simplemente no tiene.

Y sin embargo, la formación en buenas prácticas de programación no debería esperar hasta el trabajo profesional para empezar. Diversos estudios documentan que los hábitos de codificación que un estudiante adquiere durante la carrera —o que no adquiere, por falta de exigencia— se trasladan casi intactos a su ejercicio profesional (Niño Membrillo et al., 2020). Si en el aula nunca se exige nombrar variables de forma descriptiva, dividir el problema en funciones con una sola responsabilidad, o escribir al menos una prueba unitaria, es poco realista esperar que ese hábito aparezca espontáneamente en la empresa.

Si bien existen esfuerzos coordinados por la academia de sistemas computacionales en generar mejores prácticas en nuestros estudiantes en los procesos de desarrollo de software, a partir de líneas de código producido de calidad, es común encontrar al final de la formación a estudiantes con prácticas deficientes e inclusive con dificultades para escribir código que otra persona pueda leer.

A partir del presente artículo, se pretende el alcance de dos objetivos: en primera instancia el poder realizar una revisión de literatura sobre buenas prácticas de programación y su relación con la calidad del software, aterrizando con un análisis de prácticas del día a día en el aula, y en segundo lugar y aportación central, proponer un instrumento de medición que un docente pueda aplicar mediante herramientas gratuitas y sin necesitar un doctorado en ingeniería de software, que permitan evaluar, con evidencia y no solo con intuición, si un estudiante está aplicando buenas prácticas al construir software.

METODOLOGÍA

Este trabajo se desarrolló como una revisión bibliográfica de tipo narrativo con criterios de búsqueda sistemática, no como una revisión sistemática exhaustiva en el sentido estricto de PRISMA. Se consultaron bases de datos y repositorios de acceso abierto como Redalyc, Dialnet, SciELO, ResearchGate y Google Académico, utilizando combinaciones de los términos "buenas prácticas de programación", "calidad del software", "métricas de código", "enseñanza de la programación" y "modelos de calidad de software", tanto en español como sus equivalentes en inglés. Se privilegiaron artículos publicados entre 2011 y 2026, con la excepción deliberada del trabajo seminal de McCabe (1976) sobre complejidad ciclomática, que sigue siendo la base de buena parte de las métricas de mantenibilidad usadas hoy en día. Se dio prioridad a estudios realizados en contextos latinoamericanos y, particularmente, mexicanos, dado que el objeto de esta propuesta es un programa educativo del TecNM. De la revisión inicial se descartaron los textos centrados exclusivamente en calidad de software educativo (es decir, calidad de los programas usados para enseñar, no la calidad del código que producen los estudiantes), por no ser el foco de este trabajo, aunque se mencionan cuando aportan contraste.

DESARROLLO

La calidad del software: qué medimos cuando decimos "calidad"

Antes de hablar de buenas prácticas conviene aclarar de qué estamos hablando cuando decimos "calidad de software", porque es un término que se usa con demasiada libertad. Callejas Cuervo et al. (2017) hacen un repaso útil de los distintos modelos que ha propuesto la disciplina desde los años setenta, y muestran que la mayoría comparte una idea de fondo: la calidad no es una propiedad única sino un conjunto de características que se pueden (y se deben) medir por

separado. Villalta y Carvallo (2015), en su revisión sistemática sobre modelos de calidad publicados entre 2007 y 2014, llegan a una conclusión parecida: existe una proliferación de modelos, pero casi todos terminan de una u otra forma anclados en las mismas categorías que ya proponía la norma ISO/IEC 9126 y que hoy continúa la ISO/IEC 25010.

Esta última norma es probablemente la referencia más citada en la actualidad. Define ocho características de calidad de producto: adecuación funcional, eficiencia de desempeño, compatibilidad, usabilidad, fiabilidad, seguridad, mantenibilidad y portabilidad (ISO/IEC, 2011). De estas ocho, la mantenibilidad es la que más directamente conecta con las buenas prácticas de programación, porque incluye subcaracterísticas como la modularidad, la reusabilidad, la analizabilidad (qué tan fácil es entender el código para diagnosticar un problema) y la capacidad de modificación. Un programa puede ser funcionalmente correcto —hace lo que debe hacer— y al mismo tiempo tener una mantenibilidad pésima, si nadie más que su autor puede entenderlo o cambiarlo sin romper algo.

Ese matiz es clave para este artículo. En un aula, lo que corresponde a las asignaturas donde se generan proyectos de programación, casi toda la evaluación tradicional se concentra en la adecuación funcional: ¿el programa hace lo que pedía el enunciado?, para ello los docentes recurren a una serie de casos de pruebas para poder evaluar si el programa funciona o no de una manera sistematizada, considerando además que se atienden grupos numerosos lo que hace que el número de programas a revisar se multiplique (39) y por consiguiente, el tiempo que se tiene destinado a la revisión se prolongue (11) lo que conlleva el riesgo de calificaciones inconsistentes, ya que los últimos programas que se revisen, es más probable que no se haga con la misma exhaustividad que los primeros y ya no habría, o tal vez sí, una baja calidad de la retroalimentación.

Adicional a lo ya mencionado, rara vez se evalúa con el mismo rigor la mantenibilidad, aunque sea la característica que más predice si ese código sobrevivirá seis meses después de entregado, ya que la mantenibilidad se refiere a qué tan bien se ha implementado el código, en términos de diseño y complejidad (Messer et al., 2024), para que el código sea fácil de mantener tanto para realizar correcciones como para modificaciones y que las funciones y procedimientos que se definan puedan ser reutilizados en contextos similares con el paso del tiempo.

Buenas prácticas de programación como determinante de calidad

Entendemos por buenas prácticas de programación el conjunto de convenciones, técnicas y hábitos de trabajo que, sin ser exigidos por ningún lenguaje de programación en particular, mejoran de forma consistente la legibilidad, mantenibilidad y confiabilidad del código que se produce, estas no son reglas de sintaxis; son reglas de oficio, generadas por la experiencia colectiva de la industria desarrolladora de software durante décadas.

Diferentes autores coinciden en algunos ejes recurrentes. En el primer eje está el estilo y las convenciones de codificación, por ejemplo: nombres de variables y de funciones que describen su propósito en lugar de abreviarlo hasta volverlo ilegible, la identación consistente, los límites razonables en la longitud de las líneas y de las funciones, y comentarios que expliquen el porqué de una decisión de diseño y no simplemente repitan lo que el código ya dice. Estas convenciones no son solo estética: facilitan que otra persona —un compañero de equipo, un revisor, incluso que el propio autor seis meses después— entienda el código con facilidad, sin tener que reconstruir mentalmente todo el razonamiento original.

En el segundo eje está el diseño modular: consiste en dividir un problema grande en partes más pequeñas, cada una con una función clara y única, en lugar de escribir funciones que hacen de todo un poco. Soraluz Soraluz et al. (2021), el estudio del desarrollo guiado por comportamiento (BDD) como práctica de calidad, documenta cómo organizar el trabajo alrededor de escenarios bien delimitados —y no con una lógica monolítica— reduciendo errores y facilitando las pruebas. Aunque el estudio se sitúa en un contexto de desarrollo ágil y profesional, el principio aplica igual de bien a un estudiante que está construyendo su primer sistema de gestión escolar: dividir el problema ayuda tanto a programar como a depurar.

En el tercer eje, quizás el menos considerado en la formación universitaria mexicana, es la práctica de las pruebas. Niño Membrillo et al. (2020), en un estudio donde se entrevistó a docentes de programación en distintas universidades del país, se documenta un panorama poco alentador: los estudiantes muestran dificultad para entender el problema antes de programarlo, desinterés por seguir una metodología que guíe su desarrollo, y una ausencia casi total de buenas prácticas, lo que deriva en programas que no cumplen su propósito o que lo cumplen solo de manera parcial. Diversos autores señalan que buena parte de ese problema no es de talento sino de hábito: a los estudiantes casi nunca se les exige realizar una prueba antes o

después de su código, y por lo tanto nunca desarrollan el reflejo de verificar su propio trabajo más allá de "correrlo y ver qué pasa".

En el cuarto eje está el control de versiones y la revisión de código entre pares. En estas revisiones de código —el ejercicio de que un compañero de equipo lea el trabajo de otro antes de integrarlo— no solo detectan errores; si no que también son, según la literatura, una de las formas más efectivas de transmitir conocimiento tácito dentro de un equipo. En la mayoría de los cursos de programación que se imparten en el TecNM, sin embargo, el uso de Git se limita a subir el proyecto final a un repositorio, sin historial de commits significativo y sin que nadie más que el propio autor haya revisado una sola línea.

En el quinto eje, y el más medible en términos cuantitativos, son las métricas objetivas de código. La complejidad ciclomática, propuesta originalmente por McCabe (1976), cuantifica el número de caminos independientes que puede tomar un programa y, por extensión, qué tan difícil será probarlo y mantenerlo exhaustivamente. Se considera generalmente aceptable un valor por debajo de diez para una función o método individual; tener valores más altos son señal de que conviene refactorizar (Auditoría de Código, 2023). A esta métrica se suman otras relacionadas con la llamada deuda técnica: como el costo acumulado, casi siempre invisible en el corto plazo, hacer decisiones de diseño apresuradas que se toman para entregas rápidas y que después hay que "pagar" en forma de tiempo de mantenimiento (Adictos al Trabajo, 2025).

Herramientas para hacer visible lo invisible

Si bien las métricas citadas, resultan de gran utilidad, en el aula resulta mucho más complicada la aplicación de las métricas, derivado de los altos volúmenes de trabajo cuando se deben revisar los códigos de hasta 30 estudiantes o múltiples proyectos finales complejos al cierre de semestre, representa un gran obstáculo el tiempo, siendo demasiado tedioso: medir complejidad ciclomática o cobertura de pruebas a mano, línea por línea. Aquí es donde entran las herramientas de análisis estático de código, que automatizan buena parte de ese trabajo.

En un análisis de herramientas, se encontró que SonarQube es probablemente la más conocida y accesible: la cual es una plataforma de código abierto que analiza más de veinte lenguajes de programación y da soporte para calcular automáticamente métricas como complejidad ciclomática, duplicación de código, cobertura de pruebas y densidad de "code smells" (patrones que suelen anticipar errores o dificultades de mantenimiento), el procesamiento de la

herramienta está reforzada por analizadores más específicos como Checkstyle, PMD o FindBugs según los diferentes lenguajes que soporta (Sentrio, 2024). De acuerdo a la investigación, ya existen experiencias documentadas de su uso en cursos universitarios de programación y pruebas de software en instituciones mexicanas, lo que sugiere que su adopción en el salón de clase no debiera estar fuera del alcance ni requiere infraestructura costosa de acuerdo con Studocu (2022). Otras herramientas equivalentes como Code Climate, ESLint con reglas de estilo, o los propios linters integrados en entornos como Visual Studio o PyCharm, estas últimas cumplen funciones similares y muchas son igualmente gratuitas para uso educativo, lo cual facilita su aplicación en las prácticas docentes de educación pública en México.

El poder sistematizar el análisis estático de acuerdo con AlOmar et al., (2023), ayuda a que los docentes pueden identificar de forma rápida y ágil, patrones recurrentes de deficiencia técnica y priorizar la corrección de defectos lo cual resulta fundamental para evitar desgastes mayores en procesos de alto costo de mantenimiento durante las etapas finales del desarrollo.

La aplicación de estas herramientas con informes detallados dirigidos a los estudiantes permite que aumenten su madurez en sus criterios de análisis y utilicen las recomendaciones de refactorización como un medio de aprendizaje interactivo, repercutiendo de forma significativa al medir el impacto real de sus ajustes en la calidad del producto final.

Este enfoque pedagógico transforma la evaluación en un ejercicio de mejora continua que trasciende más allá de la corrección funcional, donde los estudiantes desarrollan una comprensión de mayor impacto sobre los procesos necesarios de mantenimiento, soporte y la integridad técnica de sus sistemas (Horváth et al., 2025; Nikolić et al., 2024), todo los anterior, resulta prioritario para hacer cociente a los estudiantes de no olvidar en ningún momento al usuario final y objetivo de un código.

De forma adicional, el uso automatizado de estas herramientas permite al profesorado evaluar de manera simultánea la calidad de las entregas de código, accediendo de forma ágil a una revisión rápida que detecta de forma inmediata los errores comunes existentes en la mayoría de los códigos del grupo (Molnar et al., 2020), Facilitando en gran manera el ajuste de la práctica y la mejora progresiva de los procesos de aprendizaje y gestión de errores. Dicha visión agregada no solo optimiza la gestión del tiempo docente, de igual forma permite que a la par se realice la comparación y visibilización de la evolución del rigor técnico entre diferentes

semestres generando ciclos de mejora continua a partir de la aplicación de las estrategias de enseñanza a las necesidades observadas (Kaszab & Cserép, 2023), alcanzando un desarrollo de alto nivel al finalizar la formación profesional en áreas de desarrollo computacional, y no solamente como programadores.

Esta transición hacia indicadores objetivos no solo estandariza la evaluación académica, sino que permite integrar métricas de esfuerzo y calidad en el ciclo de vida de desarrollo de los estudiantes, facilitando un seguimiento detallado con un menor esfuerzo docente (Trinidad et al., 2012), lo cual es altamente apreciado en contextos de educación pública donde los recursos son limitados tanto de tiempo como de infraestructura con altas cargas de trabajo. Adicionalmente, es posible a partir de estas revisiones sistemáticas, la generación de análisis que induzcan a la identificación de soluciones similares, permitiendo a su vez, la emisión de retroalimentación colectiva y personalizada a los autores (Fernandez-Gauna et al., 2023), siendo igualmente útil en los proyectos integradores, los cuales son propios del modelo del TecNM. No obstante, es crucial considerar que la implementación de estos analizadores en entornos educativos debe gestionarse con cautela, dado que la alta incidencia de falsos positivos en los linters puede generar frustración en el alumnado y dificultar su proceso de aprendizaje si no se acompaña de una correcta interpretación de las reglas (AlOmar et al., 2023). Por tanto, resulta de gran importancia que los docentes contextualicen las alertas de calidad, integrando estos reportes dentro de una guía de buenas prácticas que fomente el pensamiento crítico y la autonomía técnica en el estudiante, en lugar de una mera corrección mecánica (Ferreira et al., 2024; Pérez et al., 2004), siendo importante contribuir de forma constante a la consolidación de buenas prácticas que perseveren en el quehacer del campo laboral. En este sentido, el análisis automatizado se convierte en una herramienta pedagógica que trasciende la simple revisión de sintaxis, fomentando una comprensión profunda de los estándares de ingeniería de software mediante la inspección directa del código fuente (Delgado-Pérez & Medina-Bulo, 2017; Gomes et al., 2017), evitando romper las grandes fallas de los primeros años de desarrollo de software donde un gran número de software, nunca fueron concluidos, derivado de una mala planeación.

Lo importante no es qué herramienta específica se adopte, sino el cambio de lógica que permite: pasar de “yo, como profesor, tengo la impresión de que este código está desordenado” a “este archivo tiene una complejidad ciclomática promedio de 14 y un 22% de líneas duplicadas”. La

segunda afirmación es discutible, mejorable, pero verificable. La primera depende enteramente del estado de ánimo y la paciencia del revisor ese día.

El contexto de la Ingeniería en Sistemas Computacionales en el TecNM

La formación de profesionales en Ingeniería en Sistemas Computacionales requiere que los egresados desarrollen competencias relacionadas con el diseño, desarrollo, implementación y evaluación de soluciones tecnológicas que respondan a criterios de calidad. En este sentido, el Tecnológico Nacional de México (TecNM) establece en el perfil de egreso de la carrera que los futuros ingenieros deben ser capaces de optimizar recursos y procesos mediante la aplicación de metodologías, estándares y buenas prácticas de ingeniería de software. Con ello, se busca favorecer la calidad de los productos desarrollados y el cumplimiento de normativas nacionales e internacionales (Tecnológico Nacional de México, 2023).

Este planteamiento es consistente con las recomendaciones internacionales relacionadas con la formación de profesionales en software y ciencias computacionales. Organizaciones como la Association for Computing Machinery (ACM) y la IEEE Computer Society consideran que la calidad del software es una competencia fundamental para el ejercicio profesional, debido a que el desarrollo de sistemas debe apoyarse en principios como la mantenibilidad, confiabilidad, seguridad, eficiencia y capacidad de evolución (ACM & IEEE Computer Society, 2020). De igual manera, el estándar ISO/IEC 25010 establece un modelo de calidad que permite evaluar características como la mantenibilidad, seguridad, compatibilidad, eficiencia del desempeño y usabilidad del software (ISO/IEC, 2011).

A pesar de lo anterior, en el contexto académico del TecNM todavía existe una brecha entre las competencias establecidas en el currículo y los mecanismos empleados para evaluarlas. En la práctica cotidiana de las academias de Ingeniería en Sistemas Computacionales, la valoración de los proyectos de programación suele enfocarse en aspectos como el cumplimiento de los requerimientos funcionales, la entrega en los tiempos establecidos y, en algunos casos, en criterios generales relacionados con el denominado "código limpio" o las "buenas prácticas de programación". Aunque estos aspectos son importantes, con frecuencia se valoran de manera subjetiva y no siempre se cuenta con indicadores específicos que permitan determinar objetivamente en qué medida se cumplen (Pressman & Maxim, 2020).

Esta situación plantea retos importantes para el proceso de enseñanza-aprendizaje. Desde la perspectiva del docente, puede ser relativamente fácil detectar un programa que presenta problemas de diseño, documentación o mantenibilidad; sin embargo, explicar y respaldar estas observaciones mediante evidencias objetivas puede ser más complicado cuando no se cuenta con instrumentos de evaluación claramente establecidos. Para el estudiante, la retroalimentación también puede perder parte de su utilidad cuando las observaciones no están sustentadas en criterios claros y medibles. Esto limita la posibilidad de identificar con precisión sus áreas de mejora y avanzar de manera gradual en el desarrollo de sus competencias profesionales (Sommerville, 2020).

Diversos estudios relacionados con la evaluación educativa señalan que los instrumentos utilizados deben ser válidos, confiables y transparentes, de manera que contribuyan al aprendizaje significativo y proporcionen mayor objetividad al proceso de evaluación (Biggs & Tang, 2011). Por esta razón, evaluar la calidad del código fuente implica ir más allá de las apreciaciones personales e incorporar indicadores que puedan verificarse. Entre estos pueden considerarse aspectos como la modularidad, complejidad, documentación, reutilización, legibilidad, manejo de errores y cumplimiento de estándares de codificación.

En este contexto, la presente propuesta busca atender un problema concreto dentro de la formación en Ingeniería en Sistemas Computacionales: la falta de un instrumento de evaluación estandarizado que permita valorar de manera objetiva la calidad del código desarrollado por los estudiantes. Más que insistir únicamente en la importancia de las buenas prácticas —un aspecto ampliamente reconocido en la literatura especializada—, se busca ofrecer una herramienta metodológica que permita convertir estos principios en criterios observables, medibles y aplicables de manera consistente. De esta forma, se espera fortalecer la objetividad de la evaluación, mejorar la retroalimentación que recibe el estudiante y generar evidencias que contribuyan al desarrollo de las competencias profesionales establecidas en el perfil de egreso del Tecnológico Nacional de México.

Propuesta de instrumento de medición

De acuerdo con lo anteriormente expuesto, se elaboró un instrumento como propuesta, el cual está estructurado en cinco dimensiones. En la siguiente tabla se describe cada dimensión propuesta, la cual va acompañada de herramientas de análisis estático, quienes permiten

métricas objetivas, evaluando de forma cualitativa, permitiendo que esta propuesta no dependa de una sola fuente de evidencia.

Tabla 1. Dimensiones

DIMENSIÓN	***QUÉ EVALÚA***	***TÉCNICA O HERRAMIENTA SUGERIDA***	***EVIDENCIA QUE GENERA***
Legibilidad y estilo	Nombres descriptivos, consistencia de formato, comentarios pertinentes	Linter automático (ESLint, Pylint, Checkstyle) + lista de cotejo docente	Reporte de advertencias de estilo; conteo de violaciones por cada 100 líneas
Modularidad y diseño	División en funciones/clases con responsabilidad única, ausencia de duplicación excesiva	SonarQube o Code Climate (duplicación, funciones largas)	Porcentaje de código duplicado; número de funciones que exceden el umbral de líneas definido
Pruebas	Existencia y cobertura de pruebas unitarias, casos límite considerados	Framework de pruebas del lenguaje (JUnit, pytest, etc.) + reporte de cobertura	Porcentaje de cobertura de líneas y ramas; número de pruebas por módulo
Documentación y control de versiones	Historial de commits significativo, README funcional, documentación mínima del proyecto	Revisión del repositorio Git (mensajes de commit, frecuencia, ramas)	Número de commits con mensajes descriptivos; presencia de documentación mínima viable
Mantenibilidad (métrica compuesta)	Complejidad ciclomática, índice de mantenibilidad general del proyecto	SonarQube (complejidad ciclomática, maintainability index)	Complejidad ciclomática promedio por función; calificación de mantenibilidad de la herramienta

Fuente: elaboración propia.

Cada rubro propuesto se califica con la siguiente escala: 1=insuficiente, 2=en desarrollo, 3=satisfactorio y 4=sobresaliente. El valor que se proporcione a cada dimensión y la valoración final será según el criterio de la academia. Queda como sugerencia que al menos en las materias de Programación Orientada a Objetos e Ingeniería de Software, la mantenibilidad y las pruebas tengan un peso comparable al de la funcionabilidad, y no al papel decorativo.

Es importante resaltar tres condiciones importantes para que el instrumento funcione de la mejor manera:

La automatización: de las cinco dimensiones, mínimo tres se pueden calcular con herramientas gratuitas, en las dimensiones de legibilidad, modularidad y mantenibilidad. Con dichas herramientas se consumirá poco tiempo en la valoración.

La gradualidad: es conveniente tomar en cuenta que no se puede ser riguroso en valorar las cinco dimensiones en proyectos de cursos introductorios, que en proyectos integradores de semestres avanzados. El instrumento está pensado para escalar su exigencia de acuerdo como el estudiante avanza en la retícula.

La retroalimentación formativa: el valor pedagógico del instrumento no está en la valoración final, sino que el estudiante reciba una retroalimentación con sugerencias y observaciones, con las que pueda analizar detenidamente, que parte de su programación tiene elevada complejidad o cual método o función deberá dividir.

DISCUSIÓN

Un instrumento de este tipo no resuelve por sí solo el problema que documentan Niño Membrillo et al. (2020): el desinterés de muchos estudiantes por seguir una disciplina de trabajo, o la dificultad para comprender el problema antes de programarlo. Ninguna herramienta de análisis estático mide comprensión de un problema; mide únicamente las huellas que esa comprensión (o su ausencia) deja en el código. En ese sentido, el instrumento propuesto es un complemento, no un sustituto, de estrategias didácticas más amplias como el aprendizaje basado en proyectos o la programación en pares, que la revisión de Jiménez-Toledo et al. (2019) sobre cursos introductorios de programación identifica como más efectivas para atender las dificultades de comprensión desde su origen.

Tampoco es un instrumento neutral. Adoptar métricas de SonarQube como criterio de evaluación implica, de facto, enseñar a programar dentro de los supuestos y convenciones de esa herramienta, que en su mayoría provienen de la práctica profesional en la industria del software. Esto puede ser una virtud —acerca al estudiante a lo que se espera de él en un entorno laboral real— o un riesgo, si el docente termina “enseñando para la herramienta” en lugar de enseñar los principios que la herramienta simplemente automatiza. La recomendación, en ese sentido, es que el reporte de la herramienta se use siempre como punto de partida de una conversación pedagógica, y no como un veredicto automático e inapelable.

Queda pendiente, y se reconoce como una limitación de este trabajo, la validación empírica del instrumento: lo que aquí se presenta es una propuesta fundamentada en la literatura, no todavía una medición piloto en un grupo real de estudiantes del TecNM. Ese es, de hecho, el siguiente paso natural: aplicar el instrumento durante uno o dos semestres en alguna de las asignaturas de la academia, comparar los resultados con las calificaciones tradicionales, y ajustar los umbrales de cada dimensión según lo que la experiencia real muestre.

CONCLUSIONES

A lo largo de la experiencia como docentes y derivado de la investigación realizada se puede constatar que la calidad del software no es un atributo que aparezca por accidente al final de una carrera de ingeniería; se construye, o se deja de construir, práctica por práctica, semestre por semestre, en cada proyecto que un estudiante entrega y que un docente revisa. La literatura es clara en que existe una relación estrecha entre la aplicación sistemática de buenas prácticas de programación como estilo consistente, diseño modular, pruebas, control de versiones y atención a métricas objetivas como la complejidad ciclomática y la calidad final del producto de software, entendida en el sentido amplio que propone la norma ISO/IEC 25010.

El problema en la formación de ingenieros en sistemas computacionales no es, entonces, la falta de consciencia sobre esa relación, que los planes de estudio ya reconocen explícitamente en sus perfiles de egreso y es el fundamento de profesionistas del área, sino la falta de un mecanismo operativo para medirla dentro del aula. Este artículo propuso un instrumento de cinco dimensiones que incluyen legibilidad, modularidad, pruebas, documentación y mantenibilidad, apoyado en herramientas de análisis estático de acceso gratuito, con el propósito de que la evaluación de “buenas prácticas” deje de ser una impresión subjetiva del docente y se convierta en un ejercicio de evidencia compartida entre docente y estudiante.

Queda como tarea de la academia de Sistemas Computacionales, y de cualquier programa educativo interesado en esta propuesta, llevarla al terreno del aula, medir sus resultados y ajustarla con la evidencia que solo la práctica cotidiana puede ofrecer, para fortalecer nuestra actividad diaria en aula y laboratorios para consolidar un desarrollo de software con mayor éxito de calidad y buenas prácticas.

REFERENCIAS

ACM, & IEEE Computer Society. (2020). Computing Curricula 2020: Paradigms for global computing education. Association for Computing Machinery; IEEE Computer Society. https://doi.org/10.1145/3456287

AlOmar, E. A., AlOmar, S. A., & Mkaouer, M. W. (2023). On the use of static analysis to engage students with software quality improvement: An experience with PMD. En 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET) (pp. 179–191). IEEE. https://doi.org/10.1109/icse-seet58685.2023.00023

Biggs, J., & Tang, C. (2011). Teaching for quality learning at university (4.ª ed.). Open University Press.

Callejas Cuervo, M., Alarcón Aldana, A. C., & Álvarez-Carreño, A. M. (2017). Modelos de calidad del software, un estado del arte. Entramado, 13(1), 236–250. https://doi.org/10.18041/entramado.2017v13n1.25125

Delgado-Pérez, P., & Medina-Bulo, I. (2017). Análisis automático del código en prácticas de programación en orientación a objetos. En Actas del I Congreso Internacional de Innovación Docente e Investigación en Educación Superior (pp. 61–71). Adaya Press. https://doi.org/10.58909/ad17102678

Fernandez-Gauna, B., Rojo, N., & Graña, M. (2023). Automatic feedback and assessment of team-coding assignments in a DevOps context. International Journal of Educational Technology in Higher Education, 20(1), Artículo 64. https://doi.org/10.1186/s41239-023-00386-6

Ferreira, A., Brito, M. A., & Lima, J. de. (2024). Software quality in an automotive project: Continuous inspection. International Journal of Automotive Technology, 25, 1–12. https://doi.org/10.1007/s12239-024-00132-5

Gomes, P. H. de A., Garcia, R. E., Spadon, G., Eler, D. M., Olivete, C., & Correia, R. C. M. (2017). Teaching software quality via source code inspection tool. En 2017 IEEE Frontiers in Education Conference (FIE) (pp. 1–8). IEEE. https://doi.org/10.1109/fie.2017.8190658

Horváth, M., Pietriková, E., & Gurbáľ, F. (2025). Personalized learning analytics through static code analysis in Computer Science Education. Acta Informatica Pragensia, 15(1), 54–71. https://doi.org/10.18267/j.aip.283

International Organization for Standardization. (2011). Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models (ISO/IEC Estándar N.º 25010:2011). https://www.iso.org/standard/35733.html

Jiménez-Toledo, J. A., Collazos, C., & Revelo-Sánchez, O. (2019). Consideraciones en los procesos de enseñanza-aprendizaje para un primer curso de programación de computadores: una revisión sistemática de la literatura. TecnoLógicas, 22(edición especial), 83–117. https://doi.org/10.22430/22565337.1520

Kaszab, P., & Cserép, M. (2023). Detecting programming flaws in student submissions with static source code analysis. Studia Universitatis Babeș-Bolyai Informatica, 68(1), 37–54. https://doi.org/10.24193/subbi.2023.1.03

Krusche, S., von Frankenberg, N., Reimer, L. M., & Bruegge, B. (2020). An interactive learning method to engage students in modeling. En Proceedings of the 42nd International Conference on Software Engineering: Software Engineering Education and Training (pp. 12–22). Association for Computing Machinery. https://doi.org/10.1145/3377814.3381702

McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, SE-2(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Messer, M., Brown, N. C. C., Kölling, M., & Shi, M. (2024). Automated grading and feedback tools for programming education: A systematic review. ACM Transactions on Computing Education, 24(1), 1–35. https://doi.org/10.1145/3636515

Molnar, A.-J., Motogna, S., & Cristina, V. (2020). Using static analysis tools to assist student project evaluation. En Proceedings of the 13th International Conference on Computer Supported Education (pp. 7–12). Association for Computing Machinery. https://doi.org/10.1145/3412453.3423195

Nikolić, D., Stefanović, D., Nikolić, M., Dakić, D., Stefanović, M., & Koprivica, S. (2024). Uncovering determinants of code quality in education via static code analysis. IEEE Access, 12, 102450–102462. https://doi.org/10.1109/access.2024.3426299

Niño Membrillo, Y. E., Morales Salgado, M. G. R., Vázquez Reyes, S., & Sánchez Rinza, B. (2020). Mejores prácticas y criterios de calidad en el proceso de desarrollo de código en los cursos de programación en la enseñanza superior. Revista Tecnología, Ciencia y Educación, (17), 97–126.

Pérez, J. R. P., Fernández, D. R., & González-Rodríguez, M. (2004). ¿Es posible la eliminación de los errores de los programas? En Informática educativa: Nuevos retos (p. 99). Universidad de Oviedo. https://dialnet.unirioja.es/servlet/articulo?codigo=6795510

Pressman, R. S., & Maxim, B. R. (2020). Software engineering: A practitioner's approach (9.ª ed.). McGraw-Hill Education.

Sommerville, I. (2020). Engineering software products: An introduction to modern software engineering. Pearson.

Soraluz Soraluz, A. E., Valles Coral, M. Á., & Lévano Rodríguez, D. (2021). Desarrollo guiado por comportamiento: Buenas prácticas para la calidad de software. Ingeniería y Desarrollo, 39(1), 190–204. https://doi.org/10.14482/inde.39.1.620.101

Tecnológico Nacional de México. (2023). Programa educativo de Ingeniería en Sistemas Computacionales: Perfil de egreso y competencias específicas. Tecnológico Nacional de México.

Trinidad, P., Resinas, M., Segura, S., & Ruiz-Cortés, A. (2012). Evaluación y seguimiento de trabajos en equipo de desarrollo de software a través de la calidad del código fuente. UPCommons Institutional Repository (Universitat Politècnica de Catalunya). https://hdl.handle.net/2099/15038

Villalta, A., & Carvallo, J. P. (2015). Modelos de calidad de software: Una revisión sistemática de la literatura. Maskana, 6(1), 107–117. https://doi.org/10.18537/mskn.06.01.09

Williams, J., Kane, D., & Cappuccini-Ansfield, G. (2008). Student satisfaction surveys: The value of taking a historical perspective. Quality in Higher Education, 14(2), 135–155. https://doi.org/10.1080/13538320802278347

© Los autores. Este artículo se publica en Prisma ODS bajo la Licencia Creative Commons Atribución 4.0 Internacional (CC BY 4.0). Esto permite el uso, distribución y reproducción en cualquier medio, incluidos fines comerciales, siempre que se otorgue la atribución adecuada a los autores y a la fuente original.



: https://doi.org/10.65011/prismaods.v5.i4.339

Cómo citar este artículo (APA 7ª edición):
Castillo Ortiz, S. Y., Cuevas Bracamontes, L., Mena Salgado, E., Díaz García, J. C., & Urióstegui Peralta, M. del C. (2026). Propuesta para Medir las Buenas Prácticas de Programación en la Calidad del Software Desarrollado por los Estudiantes de Ingeniería en Sistemas Computacionales. *Prisma ODS: Revista Multidisciplinaria Sobre Desarrollo Sostenible*, *5*(4), 357-375. https://doi.org/10.65011/prismaods.v5.i4.339