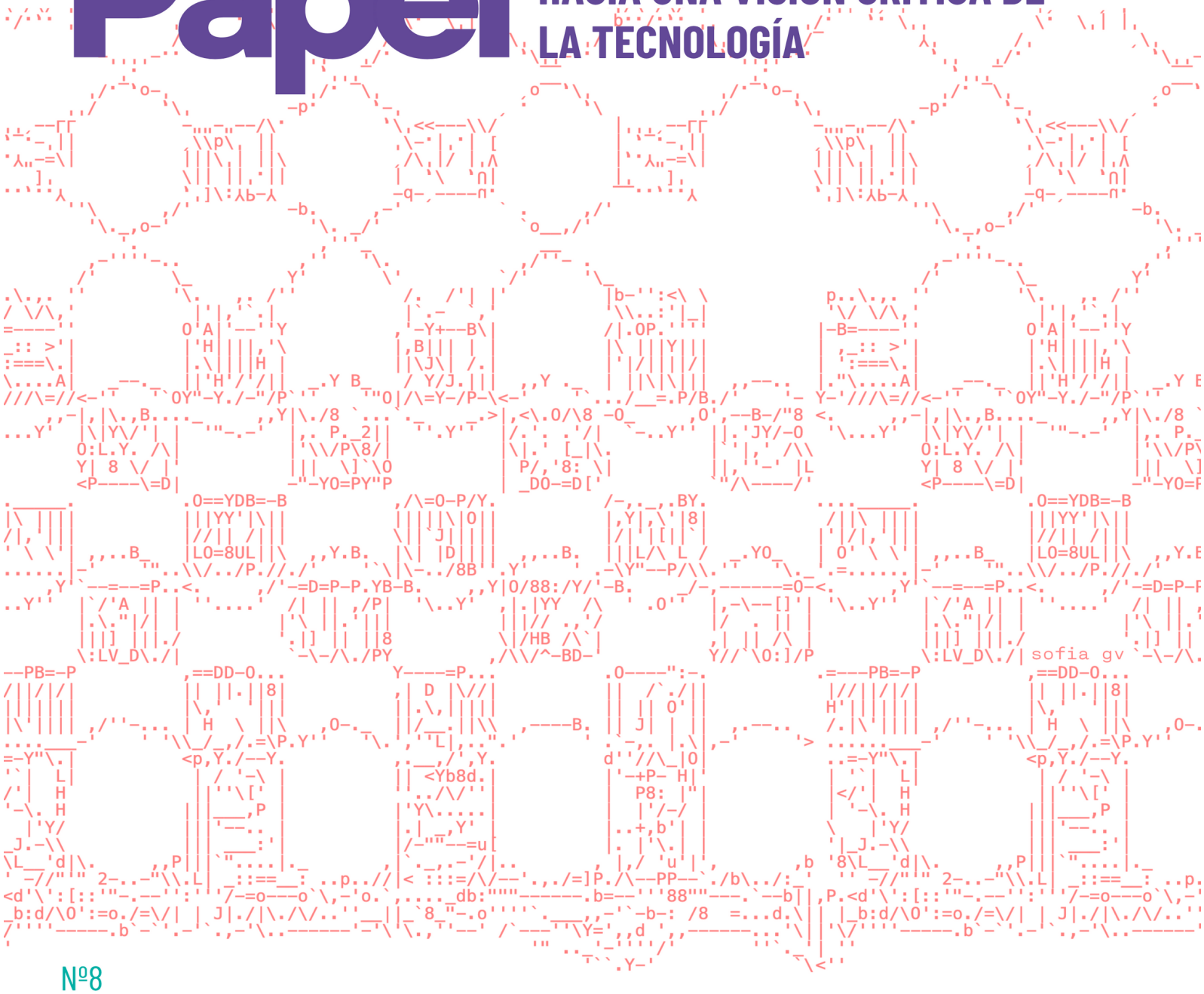
POLITÍCA, TECNOLOGÍA, EDUCACIÓN
TECNOLÓGICO DE COSTA RICA

ISSN 2953-7770

Del Zacate al Papel

HACIA UNA VISIÓN CRÍTICA DE LA TECNOLOGÍA



Nº8

MÁS ALLÁ DE LA MERCANCÍA: EL SOFTWARE COMO EXPRESIÓN ARTÍSTICA

Número 8, 2026

Del zacate al papel

Hacia una visión crítica de la tecnologías

Tabla de contenidos

Prefacio

Por Luis Ángel Meza Chavarría

Durante mucho tiempo, el software se ha entendido principalmente como un instrumento que genera utilidad: un conjunto de instrucciones diseñadas para automatizar tareas, optimizar procesos y resolver problemas prácticos con eficiencia y precisión. Desde esta perspectiva, la programación se presenta como una actividad puramente técnica, subordinada al rendimiento medible. Sin embargo, esta forma de entender el software es, cuando menos, incompleta. Más allá de su función instrumental, el software también puede concebirse como un medio de expresión creativa.

Esta idea no es del todo nueva. En su conocida conferencia *Computer Programming as an Art* (1974), Donald Knuth argumentó que la programación no debe reducirse a la mera corrección técnica, sino que puede aspirar a la belleza, el estilo y la elegancia. Su afirmación abrió la puerta a pensar en el código no solo como una herramienta funcional, sino también como una forma de creación artística. En este sentido, el software puede apreciarse estéticamente, tanto en su construcción interna como en las experiencias que posibilita. Un algoritmo puede admirarse por su elegancia, un programa por la claridad de su estructura y una obra digital por la originalidad de su diseño y su fuerza expresiva.

En las últimas décadas, esta dimensión artística del software se ha vuelto más visible. Los videojuegos, el arte generativo, las instalaciones digitales interactivas, el *live coding* y las imágenes y sonidos producidos algorítmicamente muestran que el software no es solo una herramienta para producir arte, sino también un medio de creación en sí mismo, igual que un lienzo o un libro. Como argumentan Cox y McLean (2013), la programación puede funcionar como una práctica estética y política, ya que el acto de hacer software nunca es neutral:

organiza posibilidades, impone estructuras y expresa maneras de ver y moldear el mundo.

Por otra parte, la formación en ingeniería suele enfatizar la resolución de problemas, la optimización y la precisión técnica. Si bien estas dimensiones son esenciales, no son suficientes por sí solas para ser un buen ingeniero. La ingeniería también requiere imaginación, creatividad en el diseño y la capacidad de pensar críticamente sobre el significado e impacto humano de lo que se construye. Por lo cual, reconocer el software como arte no significa negar su rigor, sino afirmar que el rigor y la creatividad pueden coexistir.

En este contexto, el octavo número de la Revista invita a reflexionar sobre las dimensiones estéticas y culturales del software. La intención es explorar al software no solo como un producto tecnológico, sino como una práctica creativa capaz de generar belleza, significado, y nuevas formas de experiencia cultural.

Referencias

Cox, G., & McLean, A. (2013). *Speaking code: Coding as aesthetic and political expression*. MIT Press.

Knuth, D. E. (1974). Computer programming as an art. *Communications of the ACM, 17*(12), 667-673. https://doi.org/10.1145/361604.361612

Maeda, J. (1999). *Design by numbers*. MIT Press.

McCullough, M. (1996). *Abstracting craft: The practiced digital hand*. MIT Press.

Paul, C. (2015). *Digital art* (3rd ed.). Thames & Hudson.

Artículos

La prosa del código: programar para ser leído

Por **Alejandro Umaña Miranda**
Santiago Valverde Álvarez

Cuando se empieza en la carrera de ingeniería en computación, no suele decirse que aprender a programar se parece, en muchos sentidos, a aprender a escribir. Se enseña sintaxis, estructuras de datos, algoritmos, paradigmas, complejidad. Se evalúa si el programa compila, si resuelve el problema correcto, si pasa los casos de prueba, si usa una solución suficientemente eficiente. La pregunta por la escritura casi nunca aparece de forma explícita. Rara vez se pregunta si el código tiene ritmo, si está organizado de una manera que permita entenderlo con naturalidad, si comunica con claridad la intención de quien lo escribió. Y, sin embargo, conforme se avanza en la disciplina, se vuelve evidente que es importante. Programar no es solamente dar instrucciones a una máquina. Es también producir un texto técnico que otros seres humanos tendrán que leer, interpretar, cuestionar, corregir y continuar.

Esta relación entre programación y literatura no debe entenderse como una simple comparación estética, ni como una metáfora elaborada para embellecer el oficio del programador. Se trata de una semejanza concreta, visible en la práctica cotidiana del desarrollo de software. Basta con enfrentarse a un sistema heredado, a una base de código sin documentación o a un conjunto de funciones cuyos nombres no explican nada para notar que el problema no es solo técnico. Es también un problema de escritura. Hay código que se deja leer como un texto bien construido, donde cada parte anticipa la siguiente, donde la estructura orienta, donde los nombres acompañan la lógica. Y hay código que se resiste como un texto torpe, lleno de repeticiones, ambigüedades y saltos bruscos. En ambos casos, lo que cambia no es únicamente el funcionamiento, sino la experiencia de lectura.

Donald Knuth (1992), en su propuesta de *Literate Programming*, sostiene que quien practica la programación literaria puede verse como un ensayista cuya preocupación central es la exposición de ideas y la excelencia del estilo. Su planteamiento no consiste simplemente en comentar más el código o en adornarlo con explicaciones innecesarias. Lo que propone es reorganizar la forma en que se piensa un programa, partiendo del hecho de que la comprensión humana no siempre sigue el mismo orden en que una máquina ejecuta instrucciones. Por eso, escribir bien un programa implica presentar sus conceptos en el orden que resulte más claro para el lector. Esta idea sigue siendo provocadora incluso hoy, porque obliga a desplazar el centro de la actividad: el programa no debe construirse primero para la máquina, sino para la persona que intentará entenderlo después.

Esa misma intuición aparece de manera aún más directa en Abelson y Sussman (1996), cuando afirman que un lenguaje de programación no es únicamente un medio para hacer que una computadora ejecute operaciones, sino también un medio formal para expresar ideas sobre metodología. En esa misma línea, señalan que los programas deben escribirse para que las personas los lean, y solo incidentalmente para que las máquinas los ejecuten. La frase se cita con frecuencia, quizá porque resume de manera precisa algo que muchos desarrolladores aprenden tarde: la legibilidad no es un detalle opcional ni una preocupación secundaria. Es parte de la calidad del software. Un programa puede producir la salida correcta y aun así estar mal escrito. Puede funcionar hoy y, al mismo tiempo, estar sembrando dificultades para mañana.

La semejanza entre programación y literatura puede observarse, en primer lugar, en la forma en que ambas construyen significado. No se trata simplemente de que las dos utilicen palabras o símbolos. La similitud es más profunda. Tanto en el texto literario como en el código, una unidad aislada rara vez significa mucho por sí sola. Su sentido depende del contexto en que aparece, de la red de relaciones que establece con otras unidades, del lugar que ocupa dentro de una estructura mayor. Un nombre de variable, una clase o una función adquieren

verdadero significado por su relación con el resto del sistema. De manera similar, una palabra en un poema, en una novela o en un ensayo no puede separarse del tono, del ritmo y de las asociaciones que la rodean. En ambos casos, el contexto no es un accesorio. Es parte esencial del significado.

También existe una similitud en la noción de estilo. En literatura, el estilo no es solamente una cuestión ornamental. Es una forma de ordenar la experiencia, de decidir qué se muestra primero, qué se omite, qué se enfatiza, qué voz se adopta. En programación ocurre algo parecido. Dos personas pueden resolver correctamente el mismo problema y producir programas profundamente distintos. Una puede escribir funciones cortas, nombres precisos y una secuencia lógica fácil de seguir. Otra puede concentrar demasiadas responsabilidades en un solo bloque, abusar de abreviaturas, repetir estructuras y obligar al lector a reconstruir mentalmente lo que el texto debió aclarar por sí mismo. En ambos casos hay estilo, pero no todo estilo comunica igual de bien. La cuestión no es eliminar la voz del programador, sino disciplinarla para que sea útil.

Esta observación es importante porque durante mucho tiempo la enseñanza de la programación ha privilegiado una visión instrumental del código. Se aprende a programar como si se tratara de ensamblar mecanismos: cada pieza debe cumplir una función, y el criterio final es que el sistema opere. Desde luego, esa dimensión es indispensable. Un programa que no funciona no puede defenderse por su belleza léxica. Pero reducir la programación a su ámbito funcional deja por fuera una parte decisiva del trabajo real. En la práctica profesional, programar no es solo producir soluciones desde cero. Es entrar en conversaciones técnicas ya empezadas. Es leer las decisiones de otros, descubrir supuestos que nadie documentó, interpretar convenciones, detectar contradicciones, corregir errores sin romper lo que ya existe. Es decir, gran parte del trabajo consiste en leer.

Ese hecho debería bastar para replantear la formación. Si el ingeniero pasa una porción significativa de su tiempo leyendo código ajeno, entonces la habilidad de escribir código legible no puede verse como un lujo estético. Es una condición de trabajo colaborativo. Un sistema bien escrito reduce el esfuerzo cognitivo de quienes participan en él. Disminuye el tiempo necesario para entender una parte del programa, facilita la detección de errores y hace menos costoso introducir cambios. Por el contrario, un código opaco incrementa la fricción en cada modificación. Obliga a interpretar demasiado, genera miedo a tocar ciertas partes y convierte tareas sencillas en procesos inciertos. Igual que ocurre con un texto mal redactado, la mala escritura en código no solo incomoda. Produce consecuencias concretas.

La conexión con la literatura también aparece en la idea de reescritura. Ningún escritor serio considera que la primera versión de un texto sea necesariamente la mejor. Escribir implica volver sobre lo escrito, cortar, reordenar, renombrar, simplificar, precisar. El trabajo no termina cuando la frase “funciona”, sino cuando comunica lo que debe comunicar de la mejor manera posible. En programación ocurre algo muy semejante. Refactorizar no es un acto accesorio ni una manía de perfeccionismo. Es una forma de revisión textual. Consiste en volver al código para mejorar su estructura, sin alterar su comportamiento observable. Renombrar una función para que exprese mejor su intención, dividir un bloque demasiado largo, mover responsabilidades a un módulo más adecuado, eliminar duplicaciones. Todo eso se parece menos a reparar una máquina que a editar un texto.

Visto así, el programador no solo resuelve problemas. También toma decisiones narrativas. Decide qué mostrar primero y qué después. Decide cuándo encapsular un detalle y cuándo exponerlo. Decide cuánto contexto necesita el lector antes de enfrentar una abstracción. Decide si una función cuenta una sola idea o varias mezcladas. Decide si el recorrido por el sistema será claro o accidentado. En cierto sentido, cada programa propone una lectura. Hay

programas que guían al lector con cuidado, como un ensayo bien construido. Otros lo arrojan a una secuencia de fragmentos inconexos, obligándolo a reconstruir una intención que debió estar más visible.

Incluso los comentarios de código pueden pensarse desde esta relación. Un mal comentario se parece a una mala nota al pie: repite lo obvio, interrumpe innecesariamente o explica mal aquello que el propio texto debería dejar claro. Un buen comentario, en cambio, no traduce línea por línea lo que ya se ve. Más bien aporta contexto, justifica una decisión, aclara una restricción de negocio, advierte sobre una consecuencia no evidente. En otras palabras, también aquí hay una dimensión de escritura responsable. No se trata de escribir más, sino de escribir mejor. Lo mismo vale para la documentación técnica: su propósito no es existir por obligación, sino reducir incertidumbre en quien llega después. No obstante, sí debería existir.

Escribir código ilegible no afecta solo a quien lo escribió. Afecta al equipo, al mantenimiento del sistema y, en última instancia, a las personas que dependen del software. Cuando una decisión mal explicada o una estructura innecesariamente compleja dificultan corregir un error, el costo puede trasladarse a usuarios, clientes, instituciones o servicios críticos. De ahí que la claridad no sea únicamente una virtud estética. Es una forma de responsabilidad profesional. Un ingeniero no trabaja en aislamiento absoluto. Produce artefactos que circulan, que se comparten, que se heredan. Escribir con cuidado es reconocer esa realidad.

La relación entre escritura y programación también ayuda a cuestionar cierta cultura académica que premia la solución rápida por encima de la solución comprensible. Muchas veces el estudiante aprende que lo importante es llegar a la respuesta correcta antes que los demás, aunque el proceso quede comprimido en nombres crípticos, bloques interminables o decisiones difíciles de justificar. El examen rara vez castiga la opacidad si el resultado final es correcto. Pero la práctica profesional sí lo hace, aunque no siempre de forma inmediata. La deuda

de legibilidad se acumula silenciosamente hasta que aparece como retraso, errores repetidos, dependencia excesiva de ciertas personas o miedo a tocar partes delicadas del sistema. Lo que en clase parecía una victoria, en un equipo real puede convertirse en una carga colectiva.

En este sentido, resulta valioso el hallazgo de Olivares-Rodríguez et al. (2023), quienes reportan una correlación positiva entre la producción de código limpio y las concepciones sobre la escritura en estudiantes de ingeniería en computación. Más allá del dato puntual, lo relevante es lo que sugiere: comprender la escritura como un proceso de comunicación mejora también la forma en que se programa. Esto tiene implicaciones pedagógicas claras. Tal vez enseñar programación no debería limitarse a enseñar estructuras correctas y soluciones eficientes, sino incluir también ejercicios de lectura, revisión y reescritura. Tal vez habría que preguntar no solo si el programa funciona, sino si se entiende. No solo si resuelve un problema, sino si lo expone con claridad.

También conviene notar que la comparación con la literatura no busca borrar las diferencias entre ambos campos. La programación no es literatura en sentido estricto, ni su valor depende de provocar emoción estética como lo hace una novela o un poema. El código está sometido a restricciones formales mucho más rígidas. Una coma fuera de lugar puede romper todo. La ambigüedad, que en literatura puede ser un factor expresivo, en programación suele ser un defecto. Sin embargo, esa diferencia no anula la comparación. Más bien la vuelve más interesante. Porque incluso dentro de un lenguaje altamente formal y restringido sigue existiendo espacio para la claridad, la estructura, la intención y el estilo. Precisamente ahí radica el reto. Escribir bien cuando hay tan poco margen para el error.

La literatura también enseña algo clave sobre la forma y el fondo. Por más que a veces se separen, en la práctica no hay contenido sin forma. Una misma idea, según cómo se cuente, puede sonar increíble o puede quedar

completamente confusa. En programación pasa lo mismo. Un algoritmo no existe por arte de magia fuera del código que lo escribe. La manera en que se divide una función, cómo la nombra, cómo se relaciona con el resto. Todo eso afecta si después alguien puede entenderla, modificarla o discutirla. Por eso la legibilidad no es un "detalle" que se agrega al final cuando ya se resolvió el problema. Es parte de la solución, desde el principio.

Desde esta perspectiva, resulta comprensible que algunos textos hayan querido pensar la programación también como una práctica cultural y expresiva, y no únicamente instrumental. Alonso Valero (2019), por ejemplo, sugiere que existen silimitudes más amplias entre los lenguajes de programación y la sensibilidad literaria. Incluso cuando no se trate de escribir "poesía en código", la observación sirve para recordar que los lenguajes técnicos también organizan formas de pensar, de mirar problemas y de construir sentido. No son recipientes neutros. Elegir una forma de escribir código es, en parte, elegir una forma de ordenar el mundo dentro de un sistema.

Del mismo modo, textos de divulgación como el de Pautassi (2016) insisten en esa cercanía entre el programador y el escritor, no para confundir sus oficios, sino para subrayar un rasgo común: ambos trabajan con lenguaje, estructura y revisión. Ambos dependen de una relación exigente entre precisión y claridad. Ambos producen algo que debe sostenerse más allá del instante de su creación. Y ambos saben, o deberían saber, que escribir mal tiene efectos. En un caso, sobre la lectura. En el otro, también sobre el funcionamiento, el mantenimiento y la confianza.

Por eso, la pregunta por la relación entre programación y literatura no es una extravagancia teórica ni una ocurrencia romántica. Es una pregunta práctica sobre la naturaleza del trabajo técnico. Programar bien no es solo producir instrucciones válidas y eficientes. Es construir textos legibles dentro de un lenguaje formal. Es ordenar ideas para que otros puedan habitarlas. Es escribir

con conciencia de lector. Tal vez por eso, conforme uno madura en la disciplina, empieza a sospechar que una parte importante del oficio no consiste en dominar cada vez más comandos, frameworks o paradigmas, sino en aprender a escribir mejor dentro de ellos.

Para finalizar, no se trata de convertir a los ingenieros en poetas ni de estetizar artificialmente la programación. Se trata de reconocer una verdad: escribir bien y programar bien comparten una raíz común, que es la responsabilidad hacia quien va a leer. Esa responsabilidad exige claridad, estructura, intención y cuidado. Exige pensar no solo en lo que el programa hace, sino en cómo lo dice. Y esa exigencia, lejos de ser un lujo humanista agregado desde afuera, forma parte de lo que distingue a un buen profesional. Porque al final, tanto en la literatura como en la programación, escribir bien es una manera de respetar el tiempo y la inteligencia del otro.

Referencias

Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and interpretation of computer programs (2nd ed.)*. MIT Press. https://web.mit.edu/6.001/6.037/sicp.pdf

Alonso Valero, E. (2019). Aproximación a la poesía escrita en lenguajes de programación (sobre Belén García Nieto). *Signa: Revista de la Asociación Española de Semiótica*, 28, 331–349. https://doi.org/10.5944/signa.vol28.2019.25055

Knuth, D. E. (1992). *Literate Programming.* CSLI Publications. https://www-cs-faculty.stanford.edu/~knuth/lp.html

Olivares-Rodríguez, C., Valdés-León, G., Vidal-Sepúlveda, M., & Oyarzún-Yañez, R. (2023). Escritura de texto y producción de código limpio: dos realidades de un mismo proceso en estudiantes de ingeniería. *Trabalhos em Linguística Aplicada.* https://doi.org/10.1590/1983-3652.2023.41439

Pautassi, M. A. (2016). Hemingway programador. *Semana.* https://www.semana.com/impresa/articulo/literatura-y-programacion-de-computadores/54206/

El costo de la propiedad: cómo la propiedad intelectual limita la creatividad en software

Por **Jose David Chaves Mena**
Jose Ignacio Madriz Ortiz
Sergio Andres Zuñiga Castillo

Las patentes se ingeniaron como mecanismo de protección a la creatividad e innovación de las personas, el carbón que impulsa el tren del progreso hacia adelante de una manera segura. Pero en los últimos años, este sistema se ha convertido en la arena entre los engranajes del sistema y en particular munición que las compañías grandes utilizan para ofuscar y acabar con la competencia antes de que puedan realizarse.

Como exponen Jaffe y Lerner (2004), el software es una de las áreas donde a las cortes que aprueban patentes se les dificulta aprobar y definir criterios que produzcan patentes de calidad. Gran parte de la fricción viene de la falta de especificidad que las oficinas de patentes requieren en las descripciones del software tal que alguien más con las suficientes habilidades pueda recrear el invento. El resultado de esto fue un entorno de patentes diluidas que no parecen introducir ninguna idea nueva, de patentes que reservan la idea de una implementación sin necesariamente tener realizada la solución. Debido a esto, el dueño de la patente tiene el derecho de excluir a otros de aplicar las mismas ideas aun el dueño no teniendo una implementación siquiera parcial de la idea.

Las oficinas de patentes, en particular para el software, deben aprobar soluciones no triviales, no obvias y lo suficientemente novedosas que requieran que el aplicante describa al software a suficiente detalle, tal que las patentes le pertenezcan a gente que en realidad haya creado algo en lugar de solo tener el concepto de una idea. Esto obstaculiza el desarrollo, pues los innovadores deben gastar recursos enormes en analizar patentes y defenderse, en lugar de en crear. Además, las patentes son costosas, todo el dinero se va en el registro, la

vigilancia legal y los litigios implican costos sociales y monetarios significativos que pueden beneficiar a grandes empresas con recursos para litigar en detrimento de desarrolladores independientes.

Debido a esta situación, hay gente que argumenta que el software no es algo patentable en el sentido tradicional y que no debería tratar de serlo: antes de que fuera patentable en los 80s la creación e innovación del software estaba en auge y además la naturaleza de su desarrollo es intrínsecamente iterativa, acumulativa y cooperativa. Estos dos argumentos demuestran la fuerte desconexión entre el software y el concepto de una patente, y de ahí nació un movimiento que impulsa la idea del software libre: noción que dicta que el software debería ser público y disponible a su uso y modificación de manera indiscriminada, con las esperanzas de que el software de ayer sean los bloques fundamentales con los cuales se desarrolla el software del mañana.

El uso de patentes no es utilizado a lo largo de toda la industria de producción de software. Existen áreas donde las patentes pueden no ser aceptadas o donde no es usual que se implementen patentes por distintas políticas. Sin embargo, es sencillo darse cuenta que algunos software y tecnologías esenciales o estándares para el funcionamiento de algunos dispositivos o programas, ya se encuentran patentados. Estas incluyen tecnologías como conexiones móviles, WiFi, compresión de video, interfaces estándar como USBs, entre otras, y entran en la categoría de SEPs (*Standard Essential Patents*). Este tipo de patente debe ser implementada bajo las condiciones FRAND (*Fair, Reasonable, and Non-Discriminatory*) de manera que licenciar productos que requieran SEPs sea una opción disponible para todos. A pesar de esto, tal como recalca Lloyd (2020), estos términos FRAND son algo ambiguos, además de que la entidad validadora no parece evaluar la “esencialidad” de la patente. Aparte de este tipo de patentes, las áreas más comunes de patentes surgen de los sectores donde hay una alta cantidad manejo

de datos: TIC y software empresarial, IA, blockchain, y áreas emergentes relacionadas a las anteriores.

La comunidad del software libre ha sido particularmente activa en señalar este tipo de problemas. Los defensores del software libre argumentan que el software no debería ser patentable, pues el progreso del sector siempre ha sido iterativo y cooperativo, basado en compartir conocimiento previo. Como explica Alevropoulos (2021) de la FSF, las patentes de software crean una "restricción inherente" a la libertad tecnológica al basarse en procesos matemáticos y algoritmos abstractos, rara vez representan "inventos" en el sentido tradicional.

En la práctica, los efectos restrictivos de las patentes afectan a todo tipo de software, no solo al libre, dado que suelen ser extremadamente amplias y vagas (Alevropoulos, 2021). La consecuencia es que el software puede seguir avanzando sin ellas, pero las patentes a menudo frustran activamente ese avance y, de hecho, existen pruebas sólidas de que la industria del software progresó durante décadas sin protección por patentes y que introducirlas masivamente puede frenar el crecimiento tecnológico como mencionan Alevropoulos (2021) y Fisher (2001).

Continuando los argumentos a favor del software libre como la manera correcta y más apropiada para el desarrollo del mismo, Fisher (2001) describe los efectos desafortunados de los patentes en la tecnología:

1. Son costosos de administrar: los sistemas de registro, los saldos de los empleados, los recursos necesarios para asegurar el cumplimiento de las patentes y abogados requeridos por los grupos interesados todos suman a que el mundo de la propiedad intelectual no sea accesible y un gran consumidor de recursos sociales.

2. La propiedad intelectual cada vez con más frecuencia es un obstáculo en el camino del desarrollo acumulativo. Puede ser que una patente encarezca, al tener que ser obtenida primero, o del todo impida si así lo desea el dueño

la creación de una nueva innovación que busca construir sobre algo preestablecido.

3. El sistema empodera al innovador a sobrecargar más del costo marginal para la replicación de sus productos, lo cual puede empujar a muchos consumidores finales fuera del mercado, lo cual resulta en una pérdida del exceso del consumidor que se pudo haber disfrutado en caso contrario.

Sin mencionar que la tensión existente entre las patentes y el software libre también ha generado conflictos legales a numerosas personas. Un caso es el de la demanda de las patentes de IP Innovation, la cual era una subsidiaria de Acacia Technologies (empresa española que se especializa en sistemas de gestión de almacenes (WMS/SGA)) contra los distribuidores de Linux Red Hat y Novell en 2010. En ese juicio, como menciona Tiller (2010), el jurado determinó que las patentes en cuestión eran inválidas y no habían sido infringidas. Es decir, falló a favor del software libre: patentes dobles y de cuestionable novedad fueron anuladas judicialmente. Este veredicto demostró que los tribunales pueden rechazar las “patentes malas” que simplemente reclaman ideas conocidas, y envió una señal a quienes usan las patentes para intimidar.

Otro punto crítico es que el sistema de patentes puede desalentar la adopción de tecnologías abiertas y estándares compartidos. El software moderno depende fuertemente de la interoperabilidad y la reutilización de componentes, pero la posibilidad de infringir una patente introduce incertidumbre a nivel legal. Según estudios recientes, incluso proyectos de código abierto, los cuales tradicionalmente han impulsado la innovación, se han convertido en objetivos frecuentes de problemas, precisamente por su visibilidad y adopción masiva (Tran, 2026)

En contraste, numerosos desarrolladores independientes han denunciado casos de litigios injustos que les impiden lanzar productos, mostrando que las demandas de patentes, incluso de conceptos simples, favorecen a grandes

corporaciones con mayor poder económico y experiencia legal. Por ejemplo, empresas tecnológicas han denunciado esquemas donde titulares de patentes presentan demandas débiles con el objetivo de forzar acuerdos extrajudiciales, más que defender una innovación real (Reuters, 2024).

Este fenómeno refuerza la idea de que el sistema no necesariamente protege la creatividad, sino que puede convertirse en una herramienta de coerción legal que afecta desproporcionadamente a actores con menos recursos y también evidencian una falla estructural en los criterios de patentabilidad, alineándose con lo expuesto por Jaffe y Lerner (2004) respecto a la baja calidad de muchas patentes de software.

Tristemente, desde una perspectiva crítica, estos ejemplos sugieren que el sistema de patentes ha evolucionado hacia un modelo que prioriza la protección legal sobre la innovación real. En lugar de incentivar la creación, muchas patentes funcionan como activos financieros utilizados para litigios o negociaciones, desvinculándose completamente del desarrollo tecnológico.

Otra parte negativa del mundo de las patentes es la existencia de ciertos NPEs (*non-practicing entities*), aquellos quienes poseen ciertas patentes, pero no generan nuevas tecnologías o más desarrollo con ellas. Algunas entidades como universidades obtienen patentes por el reconocimiento o prestigio que les da, sin embargo, hay un grupo más problemático. Los PAEs (Patent Assertion Entity) o conocidos vulgarmente como "*patent-trolls*" son aquellos quienes se centran de forma desproporcionada en las patentes, especialmente de software, y otros negocios similares. Estos se aprovechan de la ambigüedad o baja calidad de los estándares o las patentes y sus definiciones para crear litigios. Tal como lo mencionan Bessen et al. (2011) en su investigación, alrededor de 62% de las patentes NPE son en software y alrededor de 41% de los problemas legales alrededor de patentes de software vienen de algún NPE. Según este equipo de investigadores, el origen del crecimiento exponencial en las patentes de software

se debe a las aún presentes “fronteras difusas” que presentan estas patentes y permite a estas entidades encontrar fácilmente infracciones comunes.

Como se ha analizado a lo largo del texto, las patentes han causado avances y daños al mercado de producción de software, por lo que es importante conocer algunas alternativas para proteger y monetizar este tipo de productos. La más usual es el Copyright: según TRIPS (*Agreement on Trade-Related Aspects of Intellectual Property Rights*) y el derecho internacional, se reconoce la protección por derechos de autor a programas de ordenador como “obras literarias”. Lamentablemente, tal como notan Mujtaba y Zia-Ul-Haq (2024), la definición de *obra literaria* para un producto de software protege al código concreto, por lo que la misma idea puede ser implementada en un software distinto, por lo que el uso de Copyright es insuficiente para la protección del software. Una opción más moderna y centrada en software es la que fue mencionada en el caso de Alice v. CLS y analizada por Yi (2019) protección *sui generis*. Este es un sistema específico para software que busca combatir los “*patent-trolls*”, agilizar y crear un mercado de producción de software más sano. La protección sui generis corresponde a una protección tipo “*utility model*” de 10 años para software, centrada en aspectos técnicos de implementación y no solamente en funciones abstractas, además de que tiene requisitos de divulgación técnica más altos. Sui generis fue diseñada para cubrir el constante vacío legal, las fronteras difusas y casos poco específicos que presentaban otros métodos de protección de derechos de autor para soluciones de software.

Para concluir se quiere apelar al lector no solo como desarrollador sino también como consumidor y usuario porque cuando la innovación se ve limitada por barreras legales, no solo se afectan los desarrolladores, sino también los usuarios finales. Tecnologías potencialmente útiles pueden retrasarse o no llegar al mercado debido a conflictos legales, reduciendo el bienestar general y el acceso a nuevas soluciones. En este sentido, el problema de las patentes de

software trasciende lo técnico o económico, para ser un tema de interés público que afecta directamente el progreso tecnológico de la sociedad.

Referencias

Alevropoulos, P. (2021, 15 sept.). *La amenaza de las patentes de software persiste*. Free Software Foundation. https://www.fsf.org/es/blogs/la-amenaza-de-las-patentes-de-software-persiste

Bessen, J. E., Meurer, M. J., & Ford, J. L. (2011). The private and social costs of patent trolls. *SSRN Electronic Journal.* https://doi.org/10.2139/ssrn.1930272

Fisher, W. (2001). *Intellectual property and innovation: Theoretical, Empirical and Historical Perspectives. Berkman Klein Center For Internet & Society. https://share.google/FB5xBn2IsSBMvrcIh*

Jaffe, A. B., & Lerner, J. (2004). *Innovation and Its Discontents: How Our Broken Patent System is Endangering Innovation and Progress, and What to Do About It*. Princeton University Press. http://www.jstor.org/stable/j.ctt7t655

Li, Y. (2019). The current dilemma and future of software patenting. IIC; *International Review of Industrial Property and Copyright Law, 50*(7), 823–859. https://doi.org/10.1007/s40319-019-00841-w

Lloyd, I. J. (2020). 15. Software patents. En *Information Technology Law* (pp. 246–270). Oxford University Press. https://doi.org/10.1093/he/9780198830559.003.0015

Mujtaba, G., & Zia-Ul-Haq, M. (2024). The disclosure requirements of software patents: Suggestions for developing countries. *Kutafin Law Review, 11*(1), 96–123. https://doi.org/10.17803/2713-0533.2024.1.27.096-123

Reuters. (2024). *Databricks sues patent holders over alleged extortion scheme. https://www.reuters.com/legal/litigation/databricks-sues-patent-holders-over-alleged-extortion-scheme-2024-09-09*

Tiller, R. (2010, 3 mayo). *Total victory for open source software in a patent lawsuit.* Opensource.com. https://opensource.com/law/10/5/total-victory-patent-lawsuit-against-open-source-software

Tran, B. (2026). *The role of open source foundations in patent litigation*. PatentPC.https://patentpc.com/blog/the-role-of-open-source-foundations-in-patent-litigation

¿Puede el software realmente considerarse arte?

Por Víctor Hugo González González

La comparación entre arte y software puede resultar algo extraña e, indudablemente, llevaría a algunos artistas contemporáneos a arquear sus cejas con escepticismo, mientras un torrente incesante de contrargumentos puebla sus pensamientos ante la sola idea de similitud que la comparación evoca; sobre todo en una época donde la inteligencia artificial generativa pone constantemente a prueba el valor de la creatividad humana. Sin embargo, lo cierto es que la comparación podría ser más obvia que fútil, como comparar la relación existente entre las matemáticas y la ingeniería, o al menos así sería si la comparación se hubiese realizado hace 500 años.

Para contextualizar lo anterior es importante comprender que el concepto de arte, que deriva del latín *ars*, ha evolucionado con el pasar de los años (Kristeller, 1986, pp. 182-190). Incluso en épocas tan tardías como el Renacimiento —y etapas anteriores—, significaba destreza, una habilidad aplicada, por ejemplo, la necesitada para construir un objeto, una casa, una estatua, un armazón de una cama, un barco. Todas estas destrezas se denominaban artes: el arte del arquitecto, del escultor, del alfarero (Tatarkiewicz, 2001, pp. 39-40). Así, el concepto de reglas estaba estrechamente ligado al concepto de arte y su definición, lo que queda en evidencia con las definiciones ofrecidas por autores como Aristóteles (ca. 350 a. C./2016, libro VI, cap. III, pp. 136-137) y Tomás de Aquino (1947, I-II, q. 57, a. 3), que están fuertemente relacionadas al uso de la razón y la realización consciente del proceso. Bajo esta premisa, algo que fuera directamente producto de la improvisación o de la imaginación, no constituía arte para los antiguos o los escolásticos; si no la negación misma del arte.

Volviendo a la afirmación inicial, bajo esta conceptualización, no sería extraño denominar *el arte del ingeniero en software o en computación* al propio

software, que es definido como “All or part of the programs, procedures, rules, and associated documentation of an information processing system” (International Organization for Standardization [ISO] & International Electrotechnical Commission [IEC], 2015); es decir, la totalidad o una parte de los programas, procedimientos, reglas y documentación asociada a un sistema de procesamiento de información.

El concepto de arte sobrevivió hasta el Renacimiento, pero fue en esa época que comenzó una transformación que implicó la exclusión de los oficios y las ciencias del ámbito del arte y se incluyó la poesía. El consecuente proceso de tomar consciencia de que las artes restantes —las bellas artes — constituían por sí solas una clase coherente fue más complejo (Kristeller, 1986, pp. 213–216). Desde principios del siglo XX han aparecido una gran cantidad de definiciones del concepto de *arte*, sin que ninguna haya demostrado ser universalmente adecuada (Adajian, 2024, secs. 3-4).

Tatarkiewicz (2001) menciona que, si bien ninguna definición ha demostrado adecuación, “la categoría genérica a la que pertenece el arte no ha sido puesta nunca en duda: el arte es una actividad humana consciente” (p. 56). Entonces ¿qué diferencia al arte de otros tipos de actividad humana conscientes? ¿Podría esta diferencia excluir al software? Según el autor (pp. 56-61), se han propuesto diversos rasgos distintivos, todos los cuales resultan, en última instancia, incompletos:

1. **Producción de la belleza:** Tiene el defecto de que la “belleza” es una noción muy ambigua, la palabra más que un concepto es un signo de aprobación.
2. **Representación o reproducción de la realidad:** Con la imitación sucede lo mismo que con la belleza, tiene muchos significados diferentes.
3. **Creación de formas:** El significado de “forma” resulta ser demasiado extenso, no especifica el tipo de forma.

4. **Expresión:** El término “expresión”, incluso en su sentido más general, es demasiado restringido para abarcar todos los tipos de arte.
5. **Producción de experiencia estética:** El término “experiencia estética” no resulta ni más claro ni menos ambiguo que el de “belleza”.
6. **Producción de un choque:** Es la definición más reciente de las mencionadas. Es inaplicable a ciertos tipos de arte y difiere bastante de lo que generalmente se denomina arte clásico.

Es así que Tatarkiewicz (2001) opta por definir que “el arte es una actividad humana consciente capaz de reproducir cosas, construir formas, o expresar una experiencia, si el producto de esta reproducción, construcción, o expresión puede deleitar, emocionar o producir un choque”, señalando que una obra de arte es, en consecuencia, la realización material de ese proceso cuando logra “un tipo de experiencias que deleiten, emocionen o produzcan un choque” (p.67).

Con esta definición contemporánea clara es evidente que, al igual que la arquitectura; la música y el cine, ciertos productos de software tienen el potencial para ser considerados obras de arte, ya que la definición de arte muestra que debe ser una actividad humana consciente (cumple) capaz de reproducir cosas, construir formas o expresar una experiencia (cumple) y que el producto de la actividad tenga el efecto de deleitar, emocionar o producir un choque (tiene el potencial). Dada la naturaleza subjetiva de los efectos mencionados en la definición solo es posible aceptar que un producto de software se trata de arte si uno mismo, como sujeto, lo experimenta, pero no es; en este sentido, diferente a cualquier otra obra de arte.

Si bien la definición anterior es suficiente para nuestro objetivo, es importante mencionar que existen otras definiciones como la de George Dickie, Arthur Danto y Jerrold Levinson, que también gozan de reconocimiento (Zangwill, 2024). Sin embargo, estas son menos amplias y no abarcan todo el espectro del arte, o por el contrario resultan más amplias (o complementarias) a la definición

de Tatarkiewicz, para incluir el arte conceptual. Así, aunque se pueda reprochar a la definición de Tatarkiewicz no ser lo suficientemente amplia esta engloba potencialmente a los productos de software como obras de arte.

Quizá, aún en este punto, resulte disonante considerar que algunos productos de software son obras de arte como lo podrían ser las obras literarias, principalmente por el enfoque metódico y riguroso involucrado en la creación de los productos de software que suele ser bastante diferente del concepto de creatividad presente en el imaginario colectivo. Sin embargo, un enfoque metódico y riguroso no es contrario a la creatividad, incluso puede ser necesario en algunos procesos creativos. Kaufman y Sternberg (2010) citan entre las habilidades necesarias para la creatividad al conocimiento, las habilidades técnicas, habilidades especiales relacionadas con el área de trabajo, autodisciplina, perseverancia, no conformidad y evitar juicios prematuros (p. 150), todas estas habilidades pueden alcanzarse mediante un enfoque metódico y riguroso.

Como un ejemplo de un autor artístico literario que utilizaba enfoques metódicos en sus procesos creativos —como el que se seguiría, según el imaginario colectivo, en la creación de un nuevo producto de software—, tenemos al escritor y poeta estadounidense Edgar Allan Poe, quien incluso escribió *The Philosophy of Composition*, un ensayo en el que describe su proceso creativo utilizando el poema *El Cuervo* como ejemplo; en el mismo describe un proceso de creación lúcido, metódico y racional. En el mismo Poe busca conscientemente contradecir la idea de que una obra literaria debe ser necesariamente producto de la inspiración, y que una creación metódica puede resultar ventajosa (Poe, 2017, párr. 4). En contraste, Jorge Luis Borges expresaba una visión opuesta en el prólogo de *El informe de Brodie*:

> El ejercicio de las letras es misterioso; lo que opinamos es efímero y opto por la tesis platónica de la Musa y no por la de Poe, que razonó, o

> fingió razonar, que la escritura de un poema es una operación de la inteligencia. No deja de admirarme que los clásicos profesaran una tesis romántica, y un poeta romántico, una tesis clásica. (Borges, 1984, p. 1021)

Lo anterior expone que Borges claramente concebía la literatura como algo más espontáneo y no metódico, a diferencia de como sí lo hacía Poe. Sin embargo, más que un método correcto o incorrecto en la creación de obras literarias o un método real y otro imaginado, son dos métodos de un proceso creativo, ambos igualmente válidos y con resultados que han demostrado ser igualmente plausibles en la creación de obras de arte.

Es conveniente mencionar que, si bien se ha señalado una posible causa de disonancia con base en el imaginario colectivo de que la creación de software implica un proceso metódico y riguroso, lo cierto es que esto no tiene por qué ser así. Bien podría ser cierto que buena parte del software se cree con un abordaje que se asemeja más a la tesis platónica de la Musa que menciona Borges. Aunque no es lo habitual, ni recomendado, algunos proyectos de software comienzan sin una visión clara de cómo terminarán y diferentes funciones se añaden producto de la inspiración durante el proceso de desarrollo.

Posiblemente el párrafo anterior sea un motivo de alarma para cualquier persona relacionada con el desarrollo de software, lo que es un indicativo de que un enfoque metódico como el de la tesis de Poe puede resultar más apto si se quiere un producto de software que en algún momento llegue a ser una obra de arte.

En relación con la aceptación de los productos de software como obras de arte, se pueden citar ejemplos ya aceptados como tal, que incluso se encuentran expuestos como los incluidos en The Museum of Modern Art (MoMA) que ha incluido videojuegos como *Pac-Man* (1980), *Journey* (2012) y la app *Biophilia* (2011) (The Museum of Modern Art, s. f.) Además, existen iniciativas como *Rhizome ArtBase*, que un archivo creado en 1999 con el fin de preservar y dar

acceso a obras de arte digital nacidas en internet (Rhizome, s. f.). Aunque puede argumentarse que los videojuegos, y otros productos de software, se consideran arte por la utilización de otros medios de expresión artística como la música, la literatura, las artes gráficas y el diseño; lo cierto es que lo mismo puede decirse del cine o el teatro. Lo que vuelve a los videojuegos y otros productos de software merecedores de una distinción, es la combinación única de esos medios y obras, sumado a los aportes propios del medio como tal.

La imagen de un laberinto en dos dimensiones inmóvil, una figura circular amarilla, la imagen de cuatro fantasmas de colores, una banda sonora simple y breve sin ninguna narración que acompañe al conjunto; podrían haber carecido de potencial para convertirse en una obra de arte. Fue la mecánica propia del juego, *Pac-Man*, y la interacción entre estos personajes lo que lo consolidó en la obra de arte que llegó a ser, gracias a la programación, gracias al software.

Referencias

Adajian, T. (2024). The definition of art. En E. N. Zalta & U. Nodelman (Eds.), The Stanford Encyclopedia of Philosophy (Fall 2024 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/fall2024/entries/art-definition/

Aquinas, T. (1947). Summa theologica (Fathers of the English Dominican Province, Trad.). Benziger Bros. https://www.ccel.org/ccel/a/aquinas/summa/cache/summa.pdf (Trabajo original publicado ca. 1265–1274)

Aristóteles. (2016). Ética a Nicómaco (P. de Azcárate, Trad.). Imprenta Nacional. https://www.imprentanacional.go.cr/editorialdigital/libros/literatura%20universal/etica_a_nicomaco_edincr.pdf (Trabajo original publicado ca. 350 a. C.)

Borges, J. L. (1984). Obras completas, 1923-1972 (14.ª ed. en offset). Emecé Editores. https://literaturaargentina1unrn.wordpress.com/wp-content/uploads/2012/04/borges-jorge-luis-obras-completas.pdf

International Organization for Standardization, & International Electrotechnical Commission. (2015). ISO/IEC 2382:2015, Information technology —Vocabulary. https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:-1:ed-3:v1:en

Kaufman, J. C., & Sternberg, R. J. (Eds.). (2010). The Cambridge handbook of creativity. Cambridge University Press. https://dl.icdst.org/pdfs/files3/5cfaa5529670419b12f7ecf6fc5f01a7.pdf

Kristeller, P. O. (1986). El sistema moderno de las artes. En El pensamiento renacentista y las artes: Colección de ensayos (B. Moreno Carrillo, Trad., pp. 179–240). Taurus. https://archive.org/details/kristeller-paul-oskar.-el-pensamiento-renacentista-y-las-artes-ocr-1986

Poe, E. A. (2017). The Raven, and The Philosophy of Composition. Project Gutenberg. https://www.gutenberg.org/cache/epub/55749/pg55749-images.html

Rhizome. (s. f.). About. Rhizome ArtBase. Recuperado el 3 de abril de 2026, de https://artbase.rhizome.org/wiki/About

Tatarkiewicz, W. (2001). Historia de seis ideas: Arte, belleza, forma, creatividad, mímesis, experiencia estética (F. Rodríguez Martín, trad.; 6.ª ed.). Tecnos. https://marisabelcontreras.wordpress.com/wp-content/uploads/2013/11/tatarkiewicz-historia-de-seis-ideas.pdf

The Museum of Modern Art. (s. f.). The Collection. Recuperado el 3 de abril de 2026, de https://www.moma.org/collection/works/

Zangwill, N. (2024). The definition of art. En E. N. Zalta & U. Nodelman (Eds.), The Stanford Encyclopedia of Philosophy (Fall 2024 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/entries/art-definition/

Más allá de la funcionalidad

Por **Natalia Orozco Delgado**
Daniel Méndez Romero
Evelyn Vanessa Ulate Obando

Así como los artistas, los programadores pasan por un proceso creativo y de inspiración directamente vinculado con su objetivo. Si bien tradicionalmente se separan por completo los conceptos de tecnología y arte, están estrechamente relacionados y, al tratarlos como ajenos, se minimizan los beneficios de establecer tal vínculo. Tratar la programación como un arte en sí misma brinda una perspectiva totalmente distinta a la apreciación y elaboración de código; no se trata de un proceso mecánico con parches para solucionar cualquier problema, por el contrario, es elegante y propio de artistas.

Como afirma Didier Verna (2018), el arte de la programación es algo bello en sí mismo, se comprende el valor en la elegancia de una solución y el cómo es digno de ser estudiado. De esta manera, no se enfoca únicamente en el aspecto funcional de un programa, sino que también integra un componente estético, de claridad, orden y mantenibilidad. Se podría pensar que, bajo estos ideales, los profesionales siempre buscan soluciones óptimas y elegantes. Sin embargo, esta visión artística de la programación suele entrar en conflicto con la realidad de la industria tecnológica contemporánea, donde predomina la presión por entregar resultados rápidamente, “apagar incendios” y priorizar la producción constante sobre la calidad del proceso creativo.

Un código es estético no solo por su aspecto visual, sino también por su coherencia, facilidad de comprensión y modularidad. Esto engloba la organización interna del proyecto, su documentación y elementos que mantengan su sencillez y funcionalidad. Cuando un programa está bien estructurado y explicado, se favorece la colaboración entre profesionales alrededor del mundo. De igual

manera, tales prácticas favorecen el sentido de comunidad en el área y contribuyen a la elaboración de material didáctico de utilidad para principiantes.

Precisamente porque el código puede poseer valor estético y comunicativo, resulta preocupante la creciente normalización de prácticas que priorizan únicamente la funcionalidad inmediata. El problema con la ideología del desecho y practicidad no se limita al trasfondo ético en aspectos laborales o académicos, sino que promulga una actitud de indiferencia hacia las creaciones actuales y futuras, no se busca crecer ni aprender de proyectos anteriores, inspirar o comunicar un mensaje. Sin embargo, no es culpa exclusiva de las personas programadoras o profesionales en *software*, sino que, la sociedad actual promulga una ideología que deposita el valor de un producto exclusivamente en su capacidad de producir capital. Por lo tanto, se evalúa arduamente la rapidez de elaboración y puesta en producción, reducir tiempo y recursos, bajo el único principio de generar tanto dinero como sea posible. Al calificar el trabajo bajo esos parámetros, se termina devaluando el trabajo hecho con dedicación y considerable inversión humana, limitando el potencial de los desarrolladores, reduciendo su oficio a una tarea mecánica y repetitiva, que concluye con la exclusión de su origen artístico.

Esta reducción del *software* a un producto meramente funcional no solo afecta la calidad técnica del código, sino también su capacidad de transmitir identidad y significado cultural. Tradicionalmente, se suele ver a la tecnología como un causante de la pérdida de cultura en nuevas generaciones. Según Rodríguez-Cadena (2019), las tecnologías digitales han propiciado el aislamiento de las personas, causando un desarraigo social y cultural. Sin embargo, esta superficialidad que se asigna a la tecnología es un atributo asignado, y no corresponde necesariamente al carácter implícito de esta misma.

El desarraigo cultural que expresa Rodríguez-Cadena responde ante el fenómeno del desarrollo tecnológico acelerado, que no ha permitido una

adaptación apropiada a las transformaciones que acarrea, tornando difícil visualizar que la tecnología trae consigo cambios culturales que deben ser abordados desde una perspectiva diferente, y que no necesariamente el cambio implica pérdida. Siendo así, resulta posible dar un giro a las tendencias, transformando la manera en la que se emplean estas nuevas herramientas, para propiciar un uso que promueva la integración de las personas con los espacios culturales y de reflexión.

La relación entre tecnología y cultura puede observarse claramente en el desarrollo de videojuegos con identidad local, donde el *software* deja de ser únicamente entretenimiento y se convierte en una herramienta de representación cultural. Un ejemplo claro de cómo un videojuego puede convertirse en obra cultural, es el caso de *Don Memo*, un videojuego costarricense en desarrollo, concebido como un *pixel platformer 2D*. La historia sigue a un valiente toro que recorre Costa Rica para rescatar a su nieta y enfrentar al *Pisuicas*, mientras combate criaturas inspiradas en leyendas y elementos propios del imaginario nacional. Más allá de ser un producto de entretenimiento, el juego se construye como una reinterpretación digital del folclore costarricense, trasladando mitos, símbolos y referencias locales al lenguaje interactivo del *software*.

La decisión de desarrollar el juego en formato 2D con estética *pixel art* no se debe a una limitación técnica o tecnológica, sino a una elección segura sobre una expresión artística. El pixel art surge desde los orígenes de los videojuegos, trayendo una memoria colectiva digital, que al mismo tiempo funciona como lienzo para reinterpretar elementos culturales propios. En este caso, la tecnología no sustituye la cultura, sino que la traduce, y reinterpreta. Cada *sprite* diseñado, cada escenario ambientado en paisajes nacionales y cada enemigo inspirado en leyendas populares implican un proceso creativo que deja atrás la simple programación, y da pie a la observación del entorno.

Por consiguiente, el componente artístico visible y la narrativa del videojuego depende completamente de una estructura técnica subyacente: el código. Aunque invisible para el jugador, este constituye el medio que permite materializar la experiencia cultural y estética propuesta por los desarrolladores, pues es el código el que define la física de los saltos, las colisiones, las capacidades de los enemigos, y de qué manera va fluyendo la historia.

Es aquí donde se puede retomar la pregunta central: si consideramos proyectos como *Don Memo* únicamente como productos de consumo digital, su valor termina cuando deja de ser lucrativo. Pero si los entendemos como obras culturales, entonces su preservación, incluido su código fuente, se vuelve relevante, aunque el público sea de nicho. No se trata solo de conservar un videojuego, sino de cuidar una manifestación del pensamiento creativo nacional en formato digital. En esa transición de producto a patrimonio es donde cambia nuestra relación con la tecnología, dejamos de consumir pasivamente y comenzamos a reconocerla como parte de nuestra identidad cultural.

En este punto, es importante vincular el desarrollo de videojuegos como *Don Memo* con el concepto de preservación digital del patrimonio cultural inmaterial. Isa et al. (2018) sostienen que las tecnologías digitales pueden funcionar como herramientas clave para documentar, transmitir y revitalizar expresiones culturales que, de otro modo, podrían perderse con el paso del tiempo. Si bien tradicionalmente la preservación cultural se asociaba con archivos físicos o registros escritos, hoy los entornos interactivos permiten algo más profundo, experimentar la cultura. Un videojuego ambientado en Costa Rica, que incorpora leyendas como el *Pisuicas*, gastronomía típica y símbolos identitarios, no sólo representa cultura, sino que la convierte en una experiencia participativa. En este sentido, el código deja de ser únicamente una estructura funcional y se transforma en un medio de conservación cultural; cada línea programada sostiene una narrativa, una memoria colectiva y una reinterpretación contemporánea del

folclore nacional. Así, el acto de programar adquiere una dimensión patrimonial y artística.

Por otro lado, cabe destacar que buscar un enfoque más cultural o artístico no implica sacrificar la eficiencia o funcionalidad del código, y tampoco busca promover la supremacía de la estética. Proponer una visión artística de la tecnología pretende que el enfoque sea primordialmente el proceso de creación, más que el resultado esperado; significa que el código debe hacerse y rehacerse las veces que sea necesario, aunque en términos de funcionalidad cumpla con las expectativas, porque el arte surge de esos pequeños detalles que, quizá no hacían falta, pero son esas sutilezas que generan una conexión auténtica con el observador, y dotan de identidad propia al artista.

Asimismo, apostar por el arte significa dejar de lado la competitividad. Hoy en día, se generan una infinidad de productos, tecnologías y código con el único propósito de ser mejor que otro. Muchas veces, incluso se margina la originalidad, para solo copiar la fórmula de éxito que a alguien más se le ocurrió, y de esa manera tener una parte del pastel para sí. Desarrollar cualquier cosa que tenga como fin primario superar a los demás para beneficio propio, se despoja por sí mismo del valor desinteresado del arte y la cultura. Y, desafortunadamente, esta es la máxima que rige en el mundo moderno.

Asimismo, comprender la programación como una práctica artística también implica reconocer su dimensión colectiva. A diferencia de la visión individualista predominante en la industria tecnológica, movimientos como el código abierto fomentan espacios de colaboración donde el conocimiento, las ideas y las soluciones pueden compartirse y perfeccionarse de manera comunitaria, permitiendo una buena integración social.

En cuanto a esta integración social, estos espacios permiten formar una comunidad invaluable para muchas personas, permitiendo adquirir nuevas experiencias, pues estas comunidades logran atraer tanto a personas

experimentadas, como a principiantes, de diferentes países e idiosincrasias. Justamente, el código abierto permite que este tipo de interacciones contengan también un carácter de formativo a nivel técnico y reflexivo, pues se suele compartir material para aprendizaje, iniciar foros de discusión y consultar directamente a personas con mayor experiencia, promoviendo una mezcla enriquecedora de perspectivas, ideas, soluciones y aprendizaje.

Sin embargo, este ideal colaborativo suele verse limitado por dinámicas laborales que priorizan productividad constante y competencia acelerada. En muchos casos, las largas jornadas y la presión empresarial terminan sofocando cualquier posibilidad de experimentación creativa o desarrollo artístico dentro del *software*. Es por este principio que se obliga a los trabajadores a trabajar largas jornadas, que acaban ahogando cualquier pizca de entusiasmo o creatividad, y que culmina reduciéndolos a simples números u objetos desechables. Y es entonces cuando se deja de vivir, y ahora solo se puede sobrevivir. Sin embargo, ¿cómo se podría hacer arte, si ya no se está vivo? Por ende, un enfoque artístico implicaría sacrificar el avance acelerado, y moverse con paciencia.

Así, replantear la programación como una forma de arte no significa abandonar la funcionalidad o la eficiencia, sino reconocer que el *software* también puede contener valor cultural, humano y creativo. Esta perspectiva permite comprender el código no solo como una herramienta de producción, sino como una manifestación contemporánea de identidad y memoria colectiva.

Particularmente, resulta importante comprender que la tecnología no necesariamente aleja al individuo de la cultura, como suele verse desde ciertas perspectivas críticas. Por el contrario, cuando el desarrollo tecnológico incorpora intención creativa y sensibilidad humana, puede convertirse en un medio para preservar identidades, transmitir tradiciones y fomentar nuevas formas de expresión cultural. Desde esta visión, el código deja de entenderse únicamente

como una herramienta técnica y pasa a concebirse también como un lenguaje con propósito, capaz de comunicar ideas, valores y experiencias.

En consecuencia, reconocer la programación como una forma de arte implica aceptar que detrás de cada sistema existe una intención que trasciende la simple ejecución de tareas. La funcionalidad y la eficiencia continúan siendo importantes, pero pueden coexistir con la creatividad, la estética y el componente cultural. Solo así es posible construir una relación más humana con la tecnología, donde el desarrollo de *software* no represente únicamente producción, sino también expresión y memoria colectiva.

Referencias

Isa, W., Zin, N., Rosdi, F., & Sarim, H. (2018). Digital Preservation of Intangible Cultural Heritage. *Indonesian Journal of Electrical Engineering and Computer Science*. https://doi.org/10.11591/ijeecs.v12.i3.pp1373-1379.

Rodriguez-Cadena, R. (2019). Tecnología digital y afectaciones a la cultura de aprendizaje de sujeto social. *Revista Venezolana de Gerencia*, 2, 502-514. https://www.redalyc.org/journal/290/29063446027/29063446027.pdf

Verna, D. (2018). Lisp, Jazz, Aikido--Three Expressions of a Single Essence. *The Art, Science, and Engineering of Programming*, 2 (3), 10-50. https://doi.org/10.22152/programming-journal.org/2018/2/10

Uso de la ingeniería como acto rebelde

Por Valeria Marín Barquero
Beatriz Rebeca Diaz Gomez

Actualmente, las sociedades contemporáneas —especialmente aquellas influenciadas por modelos económicos occidentales— se encuentran enfocadas en la producción. Esto se refleja en una cultura que valora a los individuos, principalmente, a través de sus logros, su capacidad de generar resultados y su aporte económico. Esta realidad es inseparable del sistema educativo predominante en estas regiones, el cual suele incentivar dicha mentalidad desde la infancia. Lo anterior resulta fundamental, pues de acuerdo con Castillo Sánchez y Gamboa Araya (2012), a la educación se le ha encomendado la labor de plasmar en los estudiantes, desde edades tempranas, los ideales, las expectativas y la visión del mundo mediante las cuales se forma al ciudadano bajo ciertos paradigmas. No obstante, este enfoque no es universal; mientras que modelos como el finlandés priorizan el bienestar socioemocional y ciertos sistemas asiáticos se centran en el deber ético y la cohesión colectiva, en el contexto aquí analizado, la formación profesional se convierte en un engranaje más dentro de dinámicas orientadas a la productividad. Así, se relegan todos aquellos aspectos del desarrollo humano que no contribuyen directamente a la acumulación de capital o al crecimiento material acelerado.

La ingeniería es descrita por Kuimova et al. (2017) como una profesión donde es imperativo visualizar la realidad de formas imaginativas. Según estos autores, el éxito en la carrera de ingeniería contemporánea requiere la capacidad de pensar “fuera de la caja” para generar ideas innovadoras ante problemas técnicos complejos. Sin embargo, incluso estas capacidades creativas e innovadoras pueden quedar subordinadas a intereses económicos que condicionan las decisiones técnicas y éticas. Si se valora más la eficiencia económica a corto plazo, es probable que los proyectos de ingeniería se centren

menos en la sostenibilidad ambiental o la equidad social. En este sentido, pese a que en la mayoría de las naciones existen códigos de ética que exigen poner el bienestar público por encima de todo, autores como Derek Baldwin y Michael Dack (citados en Ilić-Krstić & Ilić-Petković, 2015) advierten que los ingenieros siguen siendo subordinados y la ética tiene poca influencia frente a lo económico. Dack además señala que el empleador posee el "poder de vida o muerte económica" sobre el trabajador, lo que convierte la obediencia al código de ética en un acto de valentía poco común.

Bajo esa perspectiva, ver a la ingeniería como un mecanismo para alcanzar la paz es, sin duda, un acto rebelde. Esta visión parte de entender la paz no solo como la ausencia de conflictos armados, sino como un estado de justicia social donde la reducción de las brechas de desigualdad es condición indispensable para la estabilidad y la armonía comunitaria. En este sentido, Alexei Ochoa, en una entrevista realizada por Sanabria Rodríguez (2025a), destaca la importancia de una ingeniería que no se limite al servicio de grandes corporaciones o al desarrollo militar, sino que se enfoque en la vida, el bienestar y la armonía, tanto humana como animal y ambiental. Estas ideas suelen catalogarse como "soñadoras", pero ese es precisamente otro síntoma del mismo problema: cabe preguntarse por qué el uso de la ingeniería para combatir las desigualdades se percibe hoy como una utopía irrealizable y no como una meta prioritaria.

Esta perspectiva de la ingeniería como subordinada al poder económico no es actual, sino una evolución de sus orígenes. La razón del auge de la ingeniería tiene parte de sus orígenes en la abolición de la esclavitud, en países como Estados Unidos, la industria dependía de la mano de obra esclava, así que la incentivación de innovaciones tecnológicas fue y ha sido para enriquecer a algunos pocos y abaratar la mano de obra (Jung & Yoon, 2025). Es decir, la ingeniería no nació únicamente como una herramienta para el desarrollo de la especie humana sino como un mecanismo para mantener la productividad bajo esquemas de control vertical. Esta herencia explica por qué el diseño tecnológico

actual sigue priorizando la eficiencia del capital sobre la dignidad de todos los seres que comparten este planeta.

En ese sentido, la reflexión de Diego Munguía, en una entrevista realizada por Sanabria Rodríguez (2025b), permite ampliar y reforzar el argumento central de este artículo, porque muestra que la tecnología y la computación tampoco escapan a las estructuras de poder que atraviesan a la ingeniería en general. Su planteamiento cuestiona la idea de que los sistemas tecnológicos son neutrales, universales o válidos de la misma manera en cualquier contexto sin arrastrar intereses, sesgos o visiones del mundo. Por el contrario, recuerda que toda tecnología surge desde condiciones históricas concretas, responde a intereses determinados y termina reproduciendo, muchas veces, las prioridades del mundo que la produce. Esta observación resulta clave para la discusión aquí planteada, porque permite entender que los sistemas tecnológicos no surgen al margen de la historia, sino dentro de estructuras económicas, culturales y políticas concretas. Del mismo modo, la idea se relaciona de forma directa con lo expuesto anteriormente: si la educación forma profesionales bajo ciertos paradigmas, y si la ingeniería ha sido absorbida por una lógica que privilegia la productividad, la obediencia y la rentabilidad, entonces no resulta extraño que gran parte del desarrollo tecnológico actual continúe favoreciendo esos mismos fines antes que la justicia social, el equilibrio ambiental o el bienestar colectivo.

Visto así, la ingeniería no solo puede ser utilizada para sostener los mecanismos del poder económico, sino que con frecuencia ha sido moldeada precisamente para ello. Esto ayuda a comprender mejor por qué una propuesta de ingeniería orientada a la paz, al cuidado y a la vida suele parecer ingenua o poco realista dentro del imaginario dominante. No se trata de que sea inviable, sino de que contradice una tradición histórica en la que el conocimiento técnico ha sido valorado sobre todo por su capacidad de aumentar la producción, optimizar recursos, reducir costos y mantener ciertas jerarquías sociales. En otras palabras, aquello que hoy suele presentarse como sentido común en la práctica profesional

de la ingeniería no es natural, sino el resultado de un proceso histórico donde lo técnico fue colocado al servicio de intereses económicos específicos.

Desde esa perspectiva, incluso el software puede entenderse como una expresión de esa misma tensión. Como advierte Munguía (Sanabria Rodríguez, 2025b), la computación no es neutral, sino que está atravesada por relaciones de poder e intereses históricos. En una línea similar, Delanty y Harris (2021) plantean que la tecnología debe analizarse como una práctica social situada, mientras que Miller (2021) cuestiona explícitamente la idea de que los artefactos tecnológicos sean moral y políticamente neutrales. Cuando se pregunta qué hace que un software tenga un estilo propio, comparable al de una pintura o una película, la respuesta no se limita a su apariencia visual o a la elegancia de su código. Su estilo también se encuentra en la lógica que organiza, en las acciones que promueve, en las decisiones que automatiza y en los límites que impone. Un sistema puede estar diseñado para vigilar, clasificar, acelerar el consumo o maximizar ganancias, pero también puede orientarse a la accesibilidad, la colaboración, la inclusión y el fortalecimiento de comunidades. Por ello, el estilo de un software no es superficial: expresa una visión del mundo. Del mismo modo que una obra artística refleja los valores, tensiones y sensibilidades de su contexto, la tecnología también deja ver qué intereses la originan y qué tipo de realidad busca consolidar.

Esta visión del software como un reflejo de valores se fundamenta en que el acto de programar es un acto de creación. Como señala Graham (2004), existe una idea errónea de que el "hacking" es una actividad fría y metódica, mientras que la pintura es una expresión impulsiva. Para Graham en realidad, los programadores/hackers y los pintores son de los tipos de personas más parecidas, pues la búsqueda de la excelencia técnica y estética sitúa al desarrollador en el mismo plano que el artista: ambos utilizan sus herramientas para dar forma a algo que antes no existía, dotándolo de una identidad y un propósito que trascienden la mera funcionalidad comercial. Al entender que el

código nace de la misma forma creativa que un lienzo, es entonces que la frontera entre lo técnico y lo artístico se desvanece.

Esto permite enlazar también la programación con la literatura. Programar no es únicamente escribir órdenes para una máquina, sino construir sentido a través del lenguaje. Tal como ocurre en la literatura, en la programación se eligen estructuras, se jerarquizan elementos, se da forma a una intención y se produce una determinada manera de representar el mundo. El código, aunque deba ser ejecutado por una computadora, también es leído, interpretado y modificado por otras personas; por eso, no es una escritura vacía, sino una forma de organización simbólica. En este punto, Buse y Weimer (2010) resultan relevantes al demostrar que la legibilidad del código es una cualidad medible y significativa dentro del desarrollo de software, lo que refuerza la idea de que programar también implica escribir de forma comprensible para otros. En una línea cercana, Graham (2004) compara el trabajo de los programadores con el de los artistas, destacando que ambos participan en procesos de creación donde la forma, la intención y la claridad importan tanto como la técnica. Tanto la literatura como la programación construyen mundos posibles: una los narra y la otra los vuelve operativos. Ambas ordenan la realidad desde ciertos criterios y, precisamente por eso, nunca son completamente neutrales.

A la luz de todo lo anterior, la propuesta de pensar la ingeniería como un acto rebelde adquiere aún más fuerza. Si la formación profesional, la tecnología y los procesos de innovación han sido históricamente absorbidos por modelos centrados en la acumulación, entonces ejercer la ingeniería desde la ética, la paz, la inclusión y el cuidado de la vida representa una ruptura real con ese mandato. Es una forma de negarse a aceptar que el único destino posible del conocimiento técnico sea servir al mercado, a la competencia o a la dominación. En cambio, supone recuperar la posibilidad de imaginar una ingeniería distinta: una que no mida su valor solo por lo que produce o por cuánto abarata costos, sino también por su capacidad de dignificar la existencia, reducir desigualdades y proteger el

mundo compartido. Bajo esa mirada, usar la ingeniería a favor de la vida deja de ser una fantasía ingenua y se convierte, precisamente, en el gesto más consciente y necesario de resistencia.

Esta resistencia es evidente cuando cuestionamos si el desarrollo de software debe estar condicionado por muros de propiedad que limiten su alcance. Como expone Stallman (2019), imponer restricciones al uso del software es equivalente a colocar peajes en cada esquina de nuestras carreteras: aunque se puede considerar que la razón de ese cobro es para el bien común —recaudar fondos—, su existencia dificulta la utilidad de la vía, creando obstáculos que entorpecen el flujo de la sociedad. Si un programa ya existe, restringir su acceso no genera un valor real, sino que empobrece el ecosistema colectivo. Por tanto, una ingeniería ética y rebelde es aquella que rechazaría convertir el código en una serie de impuestos digitales y apuesta por modelos de financiación que no obstruyan el conocimiento con el objetivo de enriquecer aún más a unos pocos. Al final del día, construir una "carretera libre" (software accesible para todos) no es solo técnicamente más eficiente, sino que es la única forma de garantizar que la tecnología sea un motor de equidad y no una herramienta de exclusión (Stallman, 2019, pp. 124-125).

Como postura final, sostenemos que la ingeniería sí debe asumirse como un acto rebelde. A nuestro juicio, no tiene sentido seguir formando profesionales técnicamente competentes si ese conocimiento termina puesto, casi de manera automática, al servicio de la rentabilidad, la obediencia y la producción sin cuestionamientos. Creemos que la ingeniería no debería evaluarse solo por su eficiencia ni por cuánto abarata costos, sino por su capacidad de construir y transformar el mundo. Precisamente por eso, nos parece indispensable defender una práctica ingenieril comprometida con la dignidad, la justicia social, la sostenibilidad y el bienestar colectivo. Si bien hoy esa postura puede parecer utópica, es únicamente porque se ha normalizado la subordinación del ingenio al poder económico. Resistirse a esa lógica y orientar el conocimiento técnico hacia

la vida, la inclusión y el acceso compartido al conocimiento no es una exageración ideológica, sino una responsabilidad ética. En ese sentido, la verdadera rebeldía de la ingeniería consiste en negarse a ser únicamente un instrumento del mercado y recuperar su capacidad de servir al bien común.

Referencias

Buse, R. P. L., & Weimer, W. (2010). *Learning a metric for code readability. IEEE Transactions on Software Engineering, 36*(4), 546–558. https://doi.org/10.1109/TSE.2009.70

Castillo Sánchez, M., & Gamboa Araya, R. (2012). Desafíos de la educación en la sociedad actual. *Diálogos Educativos*, *12*(24), 55–69. https://dialnet.unirioja.es/descarga/articulo/4156179.pdf

Delanty, G., & Harris, N. (2021). *Critical theory and the question of technology: The Frankfurt School revisited. Thesis Eleven, 166*(1), 88–108. https://doi.org/10.1177/07255136211002055

Graham, P. (2004). *Hackers & Painters: Big Ideas from the Computer Age* (1st ed.). O'Reilly Media. https://books.google.com/books?id=IezOirt2n-gC

Ilic-Krstic, I., & Ilic-Petkovic, A. (2015). Engineering ethics and sustainable development. *Safety Engineering*, *5*(2). https://doi.org/10.7562/se2015.5.02.08

Jung, Y., & Yoon, C. (2025). The shadow of slavery on industrial innovation: Evidence from the US South. *Journal of Comparative Economics*, *53*(2), 511–533. https://doi.org/10.1016/j.jce.2025.03.003

Kuimova, M., Burleigh, D. D., & Rodionov, D. A. (2017). Creativity in engineering education. PONTE International Scientific Researchs Journal, 73(2). https://doi.org/10.21506/j.ponte.2017.2.6

Miller, B. (2020). *Is technology value-neutral? Science, Technology, & Human Values, 46*(1), 53–80. https://doi.org/10.1177/0162243919900965

Sanabria Rodríguez, A. [Profe Aurelio Sanabria Rodríguez]. (2025a, 21 de octubre). *Alexei Ochoa - Ingeniería para la paz | Entrevista* [Video]. YouTube. https://www.youtube.com/watch?v=IdjesBgfmRo

Sanabria Rodríguez, A. [Profe Aurelio Sanabria Rodríguez]. (2025b, 12 de agosto). *Diego Munguía Molina - Computación y colonialidad | Entrevista* [Video]. YouTube. https://www.youtube.com/watch?v=biSYdzT5sYg

Stallman, R. (2019). *Free software, free society: Selected essays of Richard M. Stallman*. https://dspace.itsjapon.edu.ec/jspui/bitstream/123456789/449/1/softlibre-enriquecido.pdf (Obra original publicada en 2002).

Paralelismos entre el software y el arte, el programador como artista

Por Adrián Apuy Garro
Brandon Ojeda Rivas

Al pensar en el software, nuestro primer pensamiento puede ser imaginar múltiples líneas de código destinadas a ejecutar una función específica en una computadora, todo desarrollado por un programador. Pero, si se analizan a fondo diversos softwares, se puede observar que cada uno está estructurado de una forma distinta o con un lenguaje de programación diferente, incluso cuando cumplen la misma función.

Esta variabilidad hace que un software posea un estilo propio, comparable a la expresión artística, siendo un reflejo de la vida, el amor y el tiempo que el desarrollador, como artista, le dedica. Esta idea coincide con lo expuesto en el artículo *Coding as a form of art: Why learning to code should be creative* (Algocademy, s.f.), donde se plantea que programar puede considerarse una forma de arte debido a la combinación entre creatividad, lógica y expresión personal. En efecto, la programación constituye una forma de expresarse libremente, ya que en este mundo digital hay muchas formas de lograr un mismo objetivo o de realizar un proyecto; lo que lo diferencia es quién lo crea, ya que este decide qué técnica utilizar o cuál le parece la mejor opción para cumplir su objetivo. Esto queda a la percepción y experiencia de quien desarrolla. Ese ingenio y creatividad es lo que distingue el trabajo del programador y desarrollador, ya que este tiene libertad de expresión para decidir la manera en que realizará su trabajo. Esto es un paralelismo con el arte, pues lo único que limita esto es la creatividad del ser humano.

Por ejemplo, en la industria de los videojuegos, los juegos más vendidos de la historia surgieron efectivamente por medio de la creatividad de los desarrolladores. Bien es el ejemplo de muchos juegos, que no se mencionarán por

cuestiones de derechos de autor, que fueron inspirados por experiencias propias de los creadores, como memorias de la niñez, sueños o simplemente una idea que surgió en algún punto de sus vidas.

Un punto a destacar son las limitantes que enfrentan los programadores y desarrolladores para el desarrollo de sus proyectos, como bien puede ser la época en la que los quieren realizar o las limitaciones de la tecnología, ya que muchas veces la tecnología de la época en la que nace quien tiene la idea es insuficiente o no tiene la capacidad necesaria para llevarla a cabo. Pero esto, además de ser una limitante, puede servir como una forma de explotar su creatividad y de usar los recursos que tiene a disposición de manera ingeniosa para alcanzar su objetivo. Muchas veces, estas limitaciones terminan siendo una forma de impulsar la creatividad, ya que se explota la capacidad de los individuos de utilizar y sacar el mayor provecho a las herramientas que tienen a su alrededor. Esto tiene un paralelismo con la metodología "cincel y martillo", la cual nos fuerza a sacar el mayor provecho posible a las herramientas que tenemos y a no depender únicamente de estas, sino a aprender a utilizar el ingenio y la creatividad, y tener esa visión de que siempre hay algo que no hemos visto y que podemos hacer diferente.

En el artículo "*Creatividad*", María Luisa Vecina Jiménez señala que la creatividad puede ser definida como una de las características más importantes del ser humano, ya que esta tiene que ver con la ejecución de las personas en diversos contextos (Vecina Jiménez, 2006). Esto indica que el proceso creativo es distinto para cada ser humano, una cualidad que ninguna máquina es capaz de poseer, ya que mientras la máquina replica, el humano innova, y he ahí el epicentro de esta temática: la expresión creativa tiene infinitas posibilidades; la única limitación real es nuestra imaginación, ya que, si lo puedes imaginar y si lo puedes soñar, lo puedes lograr.

En efecto, gran parte de las conquistas de la humanidad surgieron de visiones audaces: desde aquel que soñó con volar por los cielos y pensó en la idea que daría paso a los aviones, hasta la persona que vio que un computador podía ser utilizado para algo más que hacer cálculos. Asimismo, en este tiempo de inteligencia artificial, no debemos dejar morir nuestra creatividad ni nuestro ingenio, aquello que nos caracteriza como humanos, ese don que se nos fue otorgado. Así, el programador se puede entender como el artista que esculpe con sus manos los proyectos que nacieron de sus ideas, de sus sueños y de lo más profundo de su ser.

Pero ¿cómo se asimila el programador como artista? Pues, para el desarrollo de código, un programador no solo debe enfrentarse a constantes problemas y resolverlos a medida que aparecen, sino también crear soluciones elegantes, eficientes y originales. Al respecto, Cody Hulcher menciona en *Coding si creative* que la programación es un proceso inherentemente creativo, ya que distintos desarrolladores pueden construir soluciones completamente diferentes para un mismo problema (Hulcher, 2016). Así como un artista busca la belleza en su obra, un programador busca la simplicidad, claridad y armonía en su código.

En el libro *El arte de la programación informática*, Donald Knuth (2021) enfatiza que un programa no solo debe funcionar, sino también estar bien estructurado y ser comprensible. En sí, se puede destacar que un programador debe organizar sus ideas y explicar bien su lógica computacional, asimilándose a un artista escribiendo una historia coherente. Además, al igual que en el arte, existen múltiples formas de resolver un mismo problema, pero algunas destacan por su creatividad, claridad y elegancia. El programador, en este sentido, toma decisiones que reflejan su estilo propio, combinando lógica y creatividad para construir soluciones únicas. En sí, se puede afirmar que un programador no solo ejecuta instrucciones, sino que organiza ideas, comunica una lógica y da forma a una “obra” digital, asimilándose a un artista que escribe una historia con coherencia, intención y sentido estético.

Para profundizar en este concepto, es preciso examinar la creatividad digital. Según lo expuesto por Lee y Chen (2015) en el artículo "Digital creativity: Research themes and framework", publicado en la revista *Computers in Human Behavior*, esta se define como aquella manifestación presente en todas las formas impulsadas por tecnologías digitales. Lo anterior evidencia que la tecnología constituye un medio de expresión artística mediante el uso de herramientas creativas; es decir, recursos inmersivos que inspiran al público y brindan soluciones a las problemáticas actuales y futuras a través de la implementación de medios tecnológicos.

Un ejemplo de esto es cómo un lenguaje de programación es comparable con el lenguaje artístico. Todos los lenguajes de programación se especializan en áreas de la informática, como el desarrollo de aplicaciones, programas para solucionar problemas, sitios web o videojuegos. Además, poseen métodos distintos para su creación y herramientas múltiples únicas. Asimismo, las distintas áreas del arte —visuales, escénicas, musicales o literarias— utilizan métodos y herramientas diferentes. De esta manera, cada lenguaje de programación ofrece estructuras, estilos y posibilidades distintas. Esto refuerza la idea de que programar no es solo una actividad técnica, sino una forma de expresión.

Finalmente, un tema que se debe abarcar debido a la problemática y los prejuicios que genera es el papel que toma la inteligencia artificial en todo esto. Como bien se sabe, la inteligencia artificial llegó para cambiar muchos aspectos de nuestra vida. Según Bernal (2024), esta tecnología ha permitido automatizar procesos y mejorar la eficiencia en el desarrollo de software, aunque también ha generado preocupación por el posible reemplazo de ciertas tareas humanas. Esto tiene mucho que ver con esta temática, ya que muchas personas se sienten desanimadas a la hora de querer aprender sobre el mundo de la tecnología debido a la gran amenaza que representa la inteligencia artificial en un nuevo mundo donde cualquier cosa puede ser reemplazada con su uso.

Pero, como se mencionó anteriormente, algo que la IA jamás podrá reemplazar es el ingenio humano y la creatividad. A la hora de trabajar con software, siempre debemos hacernos esa pregunta: en la era de la inteligencia artificial, ¿cuál es mi aporte?, ¿qué es lo que me distingue? Efectivamente, la respuesta a esta pregunta es la creatividad, ya que las máquinas no entienden conceptos ambiguos ni comprenden los sentimientos. Eso es algo característico de los seres vivos y, en especial, de los seres humanos, quienes tenemos una capacidad de entendimiento y percepción realmente increíble. Podemos expresarnos muchas veces sin necesidad de palabras; podemos expresarnos mediante ideas, expresiones, representaciones y mediante la materialización de esas ideas que nacen de nuestros corazones.

Referencias

Bernal, J. (2024, enero 25). *Ventajas e Inconvenientes del Uso de la Inteligencia Artificial en el Desarrollo de Software*. BLMovil. https://www.blmovil.com/ventajas-e-inconvenientes-del-uso-de-la-inteligencia-artificial-en-el-desarrollo-de-software/

Coding as a form of art: Why learning to code should be creative. (s/f). Algocademy.com. https://algocademy.com/blog/coding-as-a-form-of-art-why-learning-to-code-should-be-creative/

Hulcher, C. (2016, abril 9). *Coding is creative*. Medium. https://medium.com/@chulcher/coding-is-creative-c85b7566a41b

Knuth, D. E. (2021). *The Art of Computer Programming*. E-bookshelf.de. https://content.e-bookshelf.de/media/reading/L-16039326-72e9d20f61.pdf

Lee, M. R., & Chen, T. T. (2015). Digital creativity: Research themes and framework. Computers in Human Behavior, 42, 12–19. https://doi.org/10.1016/j.chb.2014.04.001

Vecina Jiménez, M. L. (2006). Creatividad. *Papeles del psicólogo*, *27*(1), 31–39. https://www.redalyc.org/articulo.oa?id=77827105

Más allá de la optimización: El impacto humano de tratar el arte del software como propiedad privada

Por Carlos Andrés Abarca Mora
Joshua Corrales
Dilan Hernandez

Este artículo de opinión examina cómo el software pierde su sentido más humano, creativo y orientado al bien común cuando queda atravesado por intereses privados. Comenzamos analizando el caso de la inteligencia artificial, una tecnología que nació con la promesa de ampliar el conocimiento, facilitar tareas y aportar soluciones útiles, pero que también puede terminar respondiendo más a la rentabilidad que al beneficio social.

Luego, exploramos el papel de la interfaz y el diseño digital, mostrando cómo lo que antes podía entenderse como una forma de acercar la máquina a las personas hoy muchas veces se usa para dirigir conductas, captar atención y provocar mayor interacción. Finalmente, nos enfocamos en las redes sociales como el ejemplo más visible de esta transformación. Aunque surgieron con la idea de conectar personas, compartir intereses y abrir espacios de expresión, con el tiempo pasaron a funcionar bajo una lógica comercial que busca retener usuarios, aumentar el consumo y convertir la atención en ganancia.

A partir de estos tres casos, sostenemos que el software no es neutral; sino que refleja las decisiones y prioridades de quienes lo desarrollan. En ese proceso, se debilita su dimensión artística, libre y genuina, y es reemplazada por dinámicas marcadas por la productividad, la permanencia y el lucro. Así, la tecnología deja de servir plenamente a las personas y comienza, en parte, a servirse de ellas. En conjunto, cuestionamos la idea de progreso tecnológico cuando este avance no prioriza la sensibilidad, la comunidad ni la expresión auténtica.

Este análisis surge desde una inquietud personal: ser un cliente más de los productos que encontramos en el mercado, los cuales comenzaron como

una excelente herramienta que facilitaba tareas de la vida diaria y poco a poco tomaron una dirección orientada a la productividad y la explotación de interacciones de los clientes. No se trata solo de un análisis técnico, sino de una postura crítica ante un mercado que ha convertido nuestra experiencia humana en su principal mercancía.

El software en un inicio comenzó como una idea creativa, como una solución a problemas o incluso como una forma de expresión artística, pero en la práctica es diferente, es tratado como propiedad privada de una empresa o de un desarrollador. La transición de “bien común” a un producto privado no solamente afecta el quién contra el código fuente, sino que también afecta el cómo se diseña y para qué es utilizado. Muchas veces prioriza la eficiencia, los datos y los ingresos por encima del impacto social y humano que este pueda llegar a tener.

Cuando el software se convierte en propiedad privada, las empresas ejercen presión para que este se optimice por encima de todo, beneficiando a la empresa a costa del bienestar ambiental, social y psicológico de las personas. Por ejemplo, las redes sociales han pasado de ser plataformas diseñadas para conectar a las personas a ser espacios digitales publicitarios con algoritmos creados para maximizar el tiempo de uso y atención de dichas plataformas por medio de patrones considerados adictivos. Lo mismo sucede con la experiencia de usuario (UX): lo que en sus orígenes percibimos como un arte para hacer las interfaces amables y comprensibles, parece haber mutado hacia lo que podríamos llamar un “motor de dopamina”. Desde nuestra perspectiva, esta evolución prioriza la cantidad de clics, la interacción y la inmersión por encima de la comodidad o la autonomía del usuario.

En el presente artículo sostenemos que el software debe de ser tratado como un bien común y no como un producto de propiedad privada. Argumentamos que cuando el software es encerrado en licencias restrictivas y sólo se buscan tendencias que beneficien el comercio, se pierde su potencial de

este como herramienta artística, colaborativa, transparente y al servicio de las comunidades y se termina reforzando las dinámicas de explotación, adicción y desigualdad.

La inteligencia artificial logra representar de manera acertada esta tensión entre el software como un bien común y como propiedad privada. En su origen, la inteligencia artificial surge con un propósito creativo y científico: la idea de construir un sistema capaz de simular e imitar el pensamiento humano, siendo capaz de resolver problemas complejos y lograr ampliar el conocimiento colectivo. Su misión, en teoría, era la de beneficiar a la sociedad, facilitando tareas, mejorando procesos complejos y haciendo que el acceso a la información sea más sencillo. No obstante, cuando esta tecnología es desarrollada en un modelo de propiedad privada, su propósito se disipa y se comienza a mover a objetivos más comerciales, donde la eficiencia, la productividad y la rentabilidad se convierten en las prioridades centrales.

Aunque reconocemos los aportes de la IA en salud o educación, es imperativo visibilizar sus amenazas. Como advierte la UNESCO, “estos rápidos cambios también plantean profundos dilemas éticos, ya que los sistemas de inteligencia artificial pueden reproducir prejuicios y generar discriminación, además de afectar derechos fundamentales si no se diseñan y utilizan de manera responsable” (UNESCO, 2024). Frente a esto, consideramos que la IA carece de neutralidad: materializa los sesgos de quienes la programan. Al subordinar esta tecnología a intereses corporativos privados, denunciamos el peligro crítico de que la rentabilidad económica se imponga sobre el bien común, automatizando y profundizando las desigualdades.

Por otro lado, una parte del software actual es la UI, que se enfoca en la estética, el arte, la estructura y la apariencia del mismo. Los sistemas operativos pasaron de ser terminales para el uso meramente funcional a ser más orgánicas, humanas y se abría un nuevo concepto para el usuario. Mark Asher (2017) afirma

que “la interfaz de usuario se dirige hacia formas orgánicas de interactividad que son nativas de nuestra biología” (traducción propia). Podríamos interpretarlo como un intento de adaptar la máquina como si fuera otra extensión de nosotros, una especie de simbiosis. Este modelo ya sea seguido por personas o empresas, es notorio en los principios de la creación de las ideas, productos o programas, especialmente, en el momento en que el cliente no ha mostrado patrones que estimulen las interacciones del usuario.

No obstante, la relación máquina-humana se ha invertido. La interfaz de usuario (UI) abandonó su esencia artística para subordinarse al beneficio corporativo, donde el diseño persigue “crear interfaces que respeten el tiempo de los usuarios, reduzcan la frustración y, sí, los hagan hacer clic” (Raikwar, 2025, traducción propia). Así, la UI se instrumentalizó para hiperestimular y monetizar nuestra atención. Como alternativa a este modelo extractivista, resulta esencial reivindicar el software libre como un verdadero bien común. A diferencia del software propietario, que diseña interfaces para manipular a un consumidor pasivo, el ecosistema libre se basa en la transparencia y la colaboración comunitaria. Este enfoque devuelve a la tecnología su propósito emancipador, priorizando la autonomía humana y las necesidades colectivas por encima de la rentabilidad publicitaria.

Bajo la misma línea del estímulo mental negativo y adictivo, resulta crítico observar cómo las redes sociales representan una degeneración del propósito original del software, el cual ha sido cooptado por intereses privados hasta despojarlo de su esencia artística y conectiva. Si bien estas herramientas nacieron con la promesa utópica de eliminar barreras geográficas y democratizar los espacios de encuentro, nuestro análisis sugiere que hoy operan en una dirección diametralmente opuesta al bien común. Lejos de ser plataformas al servicio de la interacción, estas han adoptado lo que Zuboff (2019) define como la lógica del capitalismo de vigilancia. Al operar bajo este modelo, sostenemos que el verdadero propósito del software ya no es conectar a los usuarios, sino

expropiar su experiencia humana, su tiempo y atención para convertirla en materia prima. En consecuencia, el diseño adictivo que observamos hoy no es un fallo accidental, sino un requisito estructural de un negocio que prioriza la extracción de datos conductuales por encima del bienestar social.

Las redes se han convertido en el ejemplo más claro de un software de carácter comercial con fines de lucro, al usar algoritmos diseñados de forma analítica y estructurada. Estos "responden a una lógica comercial propia de la economía de la atención, en la cual captar, retener y monetizar el tiempo del usuario constituye el objetivo central de las plataformas" (Logatt Grabner & Parra Bolaños, 2025, p. 135). Elementos como los videos cortos, las publicaciones y las historias son altamente efectivos para captar la atención del consumidor y reforzar su intención de adquirir. Detrás de esto existe un diseño que personaliza dichos elementos, explotando vulnerabilidades emocionales que pueden llevar a la adicción. De manera silenciosa, convierten al propio consumidor en un agente activo dentro del proceso publicitario, dejando en evidencia la pérdida de la esencia artística inicial de las redes, cada vez más orientadas al rendimiento comercial que a la expresión genuina.

En suma, la evolución del software en áreas críticas como la inteligencia artificial, el diseño de interfaces y las redes sociales revela una trayectoria de desplazamiento desde ideales humanistas hacia imperativos estrictamente comerciales. La transición de la IA de herramienta de conocimiento colectivo a un modelo de rentabilidad privada subraya la pérdida de neutralidad tecnológica, donde el sesgo y la eficiencia económica suelen prevalecer sobre la ética social. De igual manera, la metamorfosis de la interfaz de usuario —que ha pasado de buscar una simbiosis orgánica con la biología humana a convertirse en un mecanismo de extracción de atención— evidencia cómo la estética y la creatividad se han subordinado a la productividad del clic.

Finalmente, el fenómeno de las redes sociales representa la culminación de esta tensión, donde la promesa original de conectividad global ha sido absorbida por la economía de la atención. Al explotar vulnerabilidades emocionales mediante algoritmos diseñados para la retención perpetua, el software bajo el régimen de propiedad privada deja de ser un espacio de expresión genuina para transformarse en una infraestructura de consumo. En última instancia, recuperar la esencia del software como un bien común requiere no solo un rediseño técnico, sino un cambio de paradigma que priorice la autonomía del usuario y el impacto social por encima de la lógica de la monetización sistemática.

Berners-Lee (1999) sostiene que "la web es más una creación social que técnica. La diseñé para un objetivo social, ayudar a la gente a trabajar junta y no como un juguete técnico. El beneficio último de la tecnología es servir a la humanidad" (p. 123). Tal como la web, consideramos que todo software tiene que servir como una herramienta que extienda las capacidades del ser humano, en conocimiento y técnica. Para no perder el horizonte de la tecnología necesitamos enfocarnos en las necesidades cotidianas, aprender a pensar de manera no productiva y salir de la caja de la productividad para poder crear soluciones, y código "bello" que explore lo que la máquina privada-productiva no alcanza.

Referencias

Asher, M. (18 de julio de 2017). *A brief history of UI and what's coming*. Adobe Blog. https://blog.adobe.com/en/publish/2017/07/18/a-brief-history-of-ui-and-whats-coming

Berners-Lee, T. (1999). *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. HarperSanFrancisco.

Comisión Europea. (s. f.). *Inteligencia artificial: riesgos y oportunidades.* https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/europe-fit-digital-age/excellence-and-trust-artificial-intelligence_es

Logatt Grabner, C., & Parra-Bolaños, N. (2025). Adicción a las redes sociales, marketing digital y diseño algorítmico: Un análisis multidimensional de un fenómeno global con implicancias en la salud mental. *Lexenlace: Revista Latinoamericana de Ciencias Sociales, Educación Comercial y Derecho, 2*(2), 132-139. https://revistalexenlace.com/index.php/ojs/article/view/13/36

Raikwar, H. (23 de enero de 2025). *Psychology tricks to boost clicks: How top designers use cognitive science in UX.* UX Planet. https://uxplanet.org/psychology-tricks-to-boost-clicks-how-top-designers-use-cognitive-science-in-ux-c79d442b12ec

UNESCO. (2024). *Ética de la inteligencia artificial*. https://www.unesco.org/es/artificial-intelligence/recommendation-ethics

Zuboff, S. (2019). *La era del capitalismo de la vigilancia: La lucha por un futuro humano frente al nuevo poder corporativo.* Paidós.

Del producto descartable al patrimonio cultural: por qué el software debe preservarse como memoria técnica y social

Por Mauricio González Prendas

Durante mucho tiempo el software ha sido evaluado casi exclusivamente por su rendimiento práctico. Un programa parece valioso mientras funciona, produce resultados y puede reemplazarse por una versión más nueva cuando cambian las necesidades técnicas o comerciales. Esa lógica ha favorecido una cultura de actualización permanente que suele presentar la obsolescencia como algo natural, cuando en realidad también produce pérdida histórica. Si una parte importante de la vida contemporánea transcurre dentro de programas, plataformas, motores de juego, sistemas de gestión y entornos virtuales, entonces la desaparición de esos programas no es solo un asunto técnico, sino también cultural.

Esta postura encuentra respaldo en la investigación sobre preservación de software, ya que Matthews et al. (2010) demostraron que preservar software no equivale a guardar archivos aislados, porque el comportamiento de un programa depende de componentes, contextos y propiedades que deben seguir siendo inteligibles en el tiempo. En una línea semejante, Haux et al. (2021) plantean que el patrimonio digital obliga a pensar la memoria de la era informática desde marcos culturales más amplios, lo cual refuerza la idea de que el software no debe entenderse únicamente como una herramienta funcional, sino también como una forma de registro cultural. Desde esa perspectiva, mi tesis es que el software debe preservarse como memoria técnica y social porque en él se inscriben formas de trabajo, juego, aprendizaje, administración y creación que ninguna captura puramente documental logra sustituir por completo.

El software como memoria cultural

Considerar el software como patrimonio no significa afirmar que todo programa posea el mismo valor histórico, sino reconocer que muchos sistemas condensan experiencias colectivas irrepetibles. Esto puede verse con claridad en los videojuegos y otros medios interactivos. Guttenbrunner et al. (2010) sostienen que los videojuegos forman parte del patrimonio cultural digital y muestran que su preservación exige mucho más que almacenar un soporte físico o una copia binaria, ya que el juego depende de hardware, reglas, interfaz, documentación y condiciones materiales de uso. Lo que se intenta conservar no es una cosa inmóvil, sino una experiencia configurada por tecnologías concretas.

El mismo problema aparece en los entornos virtuales complejos, como muestran McDonough y Olendorf (2011) al estudiar Second Life, pues archivar un mundo multiusuario supone enfrentar no solo dificultades técnicas, sino también restricciones contractuales, dependencia de servidores y cambios continuos en la interacción social. Ese caso ayuda a entender por qué el software no puede reducirse a una secuencia de instrucciones desligada de su contexto, ya que un programa importante deja huellas en hábitos, comunidades, lenguajes de trabajo y prácticas culturales, de modo que preservarlo implica resguardar también las condiciones que hicieron posible su significado.

Preservar software no es guardar archivos, sino conservar contextos

Uno de los errores más comunes en este debate consiste en imaginar que la preservación termina cuando se deposita una copia en un repositorio, aunque la literatura especializada muestra que esa idea es insuficiente. Woods y Brown (2010) explican que la emulación puede ser una estrategia eficaz para recuperar ejecutables heredados, pero advierten que su viabilidad depende de disponer de suficiente información sobre los entornos de origen, mientras que Hsu y Brown (2011) agregan que el análisis de dependencias es indispensable cuando se pretende recrear materiales digitales antiguos, porque sin bibliotecas,

componentes auxiliares y relaciones de compatibilidad, un programa puede sobrevivir como archivo y desaparecer como experiencia ejecutable.

A esta dimensión técnica se suma una dimensión documental, ya que Jones et al. (2017) defienden la identificación persistente y la citación del software como requisitos fundamentales para describir, rastrear y reutilizar programas a lo largo del tiempo. Su planteamiento es decisivo porque muestra que la preservación también depende de que el software pueda ser reconocido como objeto intelectual y referenciado de manera estable, algo que se relaciona con lo señalado por Molina Salinas (2023), quien plantea que la preservación del patrimonio digital requiere metadatos consistentes, normalización e interoperabilidad. En otras palabras, conservar software exige mantener la relación entre código, contexto, descripción, entorno y acceso, ya que la pérdida de cualquiera de esas capas debilita la memoria que el sistema podía transmitir.

Obsolescencia, propiedad intelectual y bien común

La obsolescencia del software no es un simple efecto colateral del progreso, sino que también funciona como una forma de amnesia institucional y cultural, porque cuando un formato se abandona, una plataforma cierra o un sistema deja de ejecutarse en el hardware actual, la sociedad pierde acceso a parte de su experiencia registrada. Esa pérdida se vuelve más grave cuando la preservación choca con regímenes rígidos de propiedad intelectual, como muestran McDonough y Olendorf (2011) al señalar que, en mundos virtuales comerciales, los límites contractuales pueden impedir acciones necesarias para el archivo. Por eso, la pregunta no es si la propiedad privada existe, sino hasta qué punto puede bloquear por completo la continuidad de la memoria digital.

A mi juicio, el software debe pensarse como un bien con dimensión patrimonial incluso cuando tenga propietarios legítimos, una idea que no elimina los derechos de autor, pero sí cuestiona que esos derechos agoten la discusión pública. González-Barahona et al. (2023), al estudiar el conjunto de licencias

preservadas en Software Heritage, muestran la importancia documental que tiene registrar el marco jurídico de los programas dentro de los esfuerzos de conservación. Ese dato permite defender una posición equilibrada: la creatividad y la innovación necesitan incentivos, pero una cultura completamente cerrada termina volviendo inaccesibles las herramientas con las que una sociedad produjo conocimiento, ocio, administración y memoria.

Qué debería preservarse

No todo software puede preservarse con el mismo nivel de detalle, pero eso no invalida la tarea, sino que obliga a establecer criterios históricos y culturales para priorizar aquellos programas que transformaron prácticas sociales, educativas, científicas o artísticas, así como los sistemas que representan comunidades, instituciones o momentos técnicos significativos. En algunos casos bastará con conservar código fuente, documentación y capturas de uso, mientras que en otros será necesario mantener ejecutables, dependencias, manuales, registros de interacción y entornos emulados, por lo que la decisión correcta dependerá del tipo de software y del valor que se pretenda transmitir a futuro.

Por eso la preservación no debe orientarse por la acumulación indiscriminada, sino por una lógica de memoria pública, ya que preservar software también significa decidir qué queremos seguir entendiendo de nuestra propia historia digital. Allí reside la diferencia entre ver el código como residuo reemplazable o como patrimonio cultural, porque cuando se asume esta segunda perspectiva, cambia también nuestra relación con la tecnología: dejamos de pensarla solo como consumo inmediato y empezamos a verla como parte del archivo vivo de la vida contemporánea.

El software merece ser preservado porque articula memoria técnica, prácticas sociales y experiencia cultural, y la investigación revisada muestra de manera consistente que la conservación de programas no puede reducirse al

almacenamiento de archivos, ya que su supervivencia depende de contextos de ejecución, dependencias, metadatos, identificación persistente y marcos de acceso. Además, también evidencia que la obsolescencia no es neutral, porque borra huellas de trabajo, creación y sociabilidad que una sociedad digitalizada no debería perder sin reflexión crítica.

Por esta razón, preservar software implica cambiar la forma en que entendemos la tecnología, pues el código no debería verse únicamente como un producto descartable, sino también como una obra cultural capaz de conservar parte de la memoria social de una época. Aunque el software pueda estar protegido por derechos de propiedad intelectual, esa condición no elimina su dimensión pública ni su valor patrimonial, de modo que las tensiones entre creatividad, innovación y propiedad intelectual no justifican el abandono de estos materiales, sino que exigen diseñar mecanismos que permitan conservarlos con criterios históricos, técnicos y públicos de largo plazo.

Referencias

González-Barahona, J. M., Montes-Leon, S., Robles, G., & Zacchiroli, S. (2023). The software heritage license dataset (2022 edition). *Empirical Software Engineering, 28*(6), 147. https://doi.org/10.1007/s10664-023-10377-w

Guttenbrunner, M., Becker, C., & Rauber, A. (2010). Keeping the game alive: Evaluating strategies for the preservation of console video games. *International Journal of Digital Curation, 5*(1), 64–90. https://doi.org/10.2218/ijdc.v5i1.144

Haux, D. H., Maget Dominicé, A., & Raspotnig, J. A. (2021). A cultural memory of the digital age? *International Journal for the Semiotics of Law, 34*(3), 769–782. https://doi.org/10.1007/s11196-020-09778-7

Hsu, A., & Brown, G. (2011). Dependency analysis of legacy digital materials to support emulation based preservation. *International Journal of Digital Curation, 6*(1), 99–110. https://doi.org/10.2218/ijdc.v6i1.175

Jones, C. M., Matthews, B., Gent, I., Griffin, T., & Tedds, J. (2017). Persistent identification and citation of software. *International Journal of Digital Curation, 11*(2), 104–114. https://doi.org/10.2218/ijdc.v11i2.422

Matthews, B., Shaon, A., Bicarregui, J., & Jones, C. (2010). A framework for software preservation. *International Journal of Digital Curation, 5*(1), 91–105. https://doi.org/10.2218/ijdc.v5i1.145

McDonough, J., & Olendorf, R. (2011). Saving Second Life: Issues in archiving a complex, multi-user virtual world. *International Journal of Digital Curation, 6*(2), 89–108. https://doi.org/10.2218/ijdc.v6i2.192

Molina Salinas, C. (2023). Desafíos de la preservación digital del patrimonio cultural en México: el caso de Mexicana. *Cuadernos.Info*, (55), 211–232. https://doi.org/10.7764/cdi.55.48751

Woods, K., & Brown, G. (2010). Assisted emulation for legacy executables. *International Journal of Digital Curation, 5*(1), 160–171. https://doi.org/10.2218/

¿Debe el software que utilizan los Estados ser libre o propietario?

Por Dominic Jiménez Alfaro

El software libre se puede definir como aquel que respeta la libertad de las personas usuarias, ya que permite usarlo, copiarlo, estudiarlo, modificarlo y también redistribuirlo sin mayores restricciones (Programa de las Naciones Unidas para el Desarrollo y Universidad Nacional de Costa Rica [PNUD y UNA], 2013, p. 103). A partir de esto, surge una interrogante fundamental: ¿es realmente necesario que las instituciones públicas prioricen el software libre sobre el propietario, o se trata de una medida prescindible?

Más allá de ser una herramienta funcional o un recurso económico, el software libre debe entenderse como un auténtico patrimonio digital y una forma de expresión cultural colectiva. Al permitir que el código sea visto, transformado y distribuido, estamos ante una construcción de conocimiento similar a la lengua, la literatura o la pintura: un bien común que no pertenece a una corporación, sino a la humanidad. En este sentido, optar por tecnologías abiertas en el Estado no es solo una decisión financiera, sino un acto de resistencia política y cultural. Es defender el derecho de una sociedad a comprender las herramientas que dan forma a su realidad cotidiana, transformando al ciudadano de un simple consumidor de “cajas negras” en un participante activo de su propia cultura digital.

En este sentido, el software libre tiene ventajas claras en seguridad. Al ser de código abierto, una comunidad global puede auditar el sistema constantemente para encontrar fallos o proponer mejoras. Como señala Giraldo Gallego (s.f.):

> El software libre al ofrecer su disponibilidad del código fuente permite obtener mejores niveles de servicio en todo el ámbito local, además

> los *bugs* y las vulnerabilidades son encontradas más rápidamente ya que la misma comunidad se encarga de ello al estar en constante trabajo y actualización. (p. 2)

Con el software propietario, por el contrario, el acceso es cerrado y la institución pública o el Estado depende exclusivamente de que la empresa proveedora decida solucionar las vulnerabilidades, sin tener la posibilidad de revisar internamente cómo funciona la herramienta. Peor aún, existe incertidumbre sobre la continuidad del servicio o el riesgo de enfrentar incrementos arbitrarios en los precios.

Por otro lado, esta apertura garantiza la transparencia y la autonomía del Estado. El hecho de poder modificar el código permite que los gobiernos ajusten las herramientas exactamente a sus necesidades operativas, en lugar de adaptar sus procesos a soluciones rígidas impuestas por empresas extranjeras. Esto elimina la dependencia técnica y asegura que el control de las plataformas públicas permanezca en manos de la institución y no de terceros.

También es importante resaltar el factor económico. A diferencia del software privado, que suele acarrear costos elevados por una licencia permanente o de tiempo limitado, el software libre muchas veces es gratuito o tiene costos mucho más bajos. Esto puede significar un ahorro bastante grande para los Estados. Por ejemplo, una licencia de Microsoft 365 actualmente puede costar aproximadamente $99.99, mientras que alternativas como LibreOffice se pueden usar sin costo en sistemas basados en Linux, consolidándose como una excelente alternativa al paquete de Microsoft Office.

No obstante, más allá de la parte técnica, hay algo que es incluso más preocupante y peligroso. Cuando los gobiernos usan software de empresas extranjeras, se genera una dependencia tecnológica. Esto significa que aspectos claves como las actualizaciones, el acceso o incluso el funcionamiento del sistema dependen de decisiones externas. Incluso puede haber recopilación de

datos gubernamentales sin consentimiento, lo que vulnera la soberanía de un país y sirve de antesala para la filtración de información estatal. En cambio, el software libre da más control sobre lo que realmente hace el sistema y brinda total transparencia a las personas. La libertad de crear sus propios artefactos tecnológicos es fundamental para la autonomía de un país; después de todo, el software libre trata sobre el intercambio de conocimientos y la transferencia de tecnología (Gomes da Silva Lisboa y Zazula Beatriz, 2022, p. 34).

Ahora bien, tampoco todo es perfecto en el software libre. Hay algunas limitaciones. Por ejemplo, en ciertos casos puede quedarse un poco atrás en diseño o en algunas funciones si se compara con el software propietario por dar un simple ejemplo a las personas prefieren mucho más el diseño que puede tener un software como Adobe Photoshop a las posibles opciones que existen en el software libre. Esto se debe en parte a la gran brecha y desigualdad que existen entre las comunidades que desarrollan software libre no siempre tienen los mismos recursos o financiamiento que las grandes empresas, esto desarrollando a una limitación y no al desarrollo para el bien común de todos los usuarios. Deberían las personas o ciudadanos de tener la capacidad de auditar las herramientas que usa su Estado para juzgar por sí mismos si están introduciendo sus datos en un programa seguro, no obstante, esto no siempre sucede, por lo cual el usuario debe confiarles su información a empresas externas.

Otro elemento fundamental para mencionar lo dice el Programa de las Naciones Unidas para el Desarrollo y la Universidad Nacional de Costa Rica (2013), es la gestión de los riesgos asociados:

> Un proyecto de implementación de software libre es la gestión de los riesgos asociados. Gran parte de estos riesgos provienen de la resistencia al cambio o la complejidad de los proyectos de migración. En este sentido, y como las mismas empresas TIC participantes en este estudio

> recomiendan, es necesario revisar cuidadosamente la viabilidad del proyecto para valorar sus posibilidades de éxito (PNUD et al., 2013, p. 103).

Esto deja claro que la implementación del software libre no es un proceso tan simple como parece, sino que requiere planificación, análisis y una buena estrategia para que realmente funcione dentro de las instituciones públicas. Esto cobra mayor relevancia cuando se han utilizado los mismos programas durante décadas, lo cual conlleva largos procesos de capacitación o, peor aún, la resistencia al cambio y la persistencia en la dependencia de programas de empresas extranjeras.

Si logramos superar este miedo al cambio, el beneficio para la sociedad sería enorme. Tratar el software como un bien común hace que el dinero que pagamos con impuestos no se vaya fuera del país para alquilar licencias, sino que se quede aquí para crear empleos técnicos locales. Al final, lo que buscamos es que las herramientas del Estado sean puentes y no muros. Un sistema público debería ser tan abierto como una plaza o un parque, donde no haya barreras invisibles que nos impidan saber cómo funcionan las cosas. Esto genera confianza en la gente, porque cuando el gobierno es transparente con la tecnología que usa, le está diciendo al ciudadano que no tiene nada que ocultar. Es una forma de asegurar que el control de nuestra información siempre responda a lo que el pueblo necesita y no a los intereses de negocios de terceros que solo buscan vender una suscripción más.

Más allá de la dificultad que existe para implementar el software libre en empresas, existen casos en donde los gobiernos han logrado su adopción. Por ejemplo, en diversos países se han realizado esfuerzos destacados en el sector público. En Brasil, durante el gobierno de Lula da Silva, se impulsó desde el 2004 la idea de implementar software libre en toda la administración pública, con la intención de ahorrar recursos y alcanzar mayor independencia tecnológica, como parte de una estrategia más amplia de inclusión digital (Birkinbine, 2016). Aunque,

con los años —sobre todo después del 2016—, ese impulso disminuyó y las políticas gubernamentales tendieron nuevamente al uso de software privado.

Ahora bien, en el caso de Costa Rica, se han presentado iniciativas en la Asamblea Legislativa, tales como el proyecto de ley contenido en el expediente N.º 16.912, el cual buscaba promover el uso de software libre en las instituciones públicas. Sin embargo, dicha iniciativa fue archivada y no alcanzó a convertirse en ley, por lo cual no se generó el impacto esperado ni se concretó la migración tecnológica que se planteaba para las instituciones públicas del país (Asamblea Legislativa de la República de Costa Rica, 2008).

Para poner esto en perspectiva, podemos mirar lo que ocurrió en 2023 con la CCSS (Caja Costarricense de Seguro Social). Bajo la licitación de ofimática N.º 2023LY-000009-0001101150, la institución destinó cerca de 6.5 millones de dólares anuales al pago de licencias de Microsoft Office 365 hasta 2027. La Caja Costarricense de Seguro Social (2024) señala que dentro de esta contratación se adquirieron más de 11 mil licencias perpetuas de Microsoft 365 con un costo unitario de 412,49 USD, evidenciando el enorme gasto económico que representa para una institución pública depender de software privado. Es inevitable pensar que, de haber elegido el camino del software libre, ese monto millonario no solo hubiera se habría ahorrado año tras año y poder destinar esos fondos a mejorar la estructura hospitalaria del país. Por supuesto, un cambio así requiere invertir primero en capacitar al personal, pero a largo plazo, el beneficio económico y la independencia tecnológica para el país serían mucho más profundos.

En conclusión, la pregunta de si el software que utiliza el Estado debe ser propiedad privada o un bien común nos invita a repensar qué tipo de instituciones queremos construir. Si bien es cierto que el software privado en algunos casos ofrece soluciones rápidas y más estéticas, el software libre nos abre la puerta a una verdadera independencia tecnológica y a un manejo más responsable de los fondos públicos. Al tratar las herramientas digitales como un bien común, no solo

estamos buscando ahorrar millones en licencias como las de la CCSS, sino que estamos eligiendo la transparencia y el derecho de cada ciudadano a saber cómo se gestionan sus datos. El camino hacia esta soberanía digital requiere, por supuesto, un esfuerzo sostenido en capacitación y un cambio de mentalidad; sin embargo, el resultado final —un Estado más autónomo, seguro y eficiente— constituye una inversión que vale la pena para el bienestar de todos.

Referencias

Asamblea Legislativa de la República de Costa Rica. (2008). Expediente N.º 16.912. Ley de Fortalecimiento de la Gestión Pública mediante el Uso de Software Libre. https://www.asamblea.go.cr/sd/SiteAssets/Lists/Consultas%20Biblioteca/EditForm/Tramitaci%C3%B3n%20del%20Proyecto%20de%20Ley%2016.912%20.pdf

Birkinbine, B. J. (2016). *Free Software as Public Service in Brazil: An Assesment of Activism, Policy and Technology. International Journal of Communication 10, 3893–3908*.

Caja Costarricense de Seguro Social. (2024). *Licenciamiento Ofimática y Productividad Institucional (Licitación No. 2023LY-000009-0001101150)* [Documento de licitación]. Sistema Integrado de Compras Públicas (SICOP).https://www.sicop.go.cr/moduloBid/open/bid/EP_OPJ_EXA222.jsp?biddocUnikey=D20240201154823139717068241032560&releaseYn=N&cartelNo=20231110231&cartelSeq=00

Giraldo Gallego, L. (s. f.). Uso del software libre y casos de éxito [Archivo PDF]. Punto GPL; Universidad de Costa Rica.https://migracion.ucr.ac.cr/wp-content/uploads/2017/03/CE-Colombia-Uso_del_software_libre_y_casos_de_exito.pdf

Gomes da Silva Lisboa, F. y Zazula Beatriz, M. (2022). *La efectividad de las iniciativas del gobierno brasileño para software libre y código abierto. Revista Hipertextos, 10 (17), pp. 31-50* https://doi.org/10.24215/23143924e047

Programa de las Naciones Unidas para el Desarrollo & Universidad Nacional de Costa Rica. (2013). *Retos y oportunidades del software libre en la administración pública en Costa Rica* (F. Mata Chavarría, Dir.). Impresión Gráfica del Este.

El arte de programar con identidad

Por José Julián Monge Brenes
Carlos Ávalos Mendieta

El software es presentado comúnmente como una herramienta que sirve para automatizar tareas, ordenar datos o resolver problemas de manera eficiente. Esa forma de verlo no está mal, pero nos parece incompleta, pues, reducir el software a su utilidad es dejar por fuera una parte importante de lo que realmente es. Actualmente el software no solo resuelve problemas, sino que también organiza experiencias, moldea la forma en que trabajamos, crea ambientes visuales y define cómo se puede interactuar con el mundo digital.

Esta idea coincide con lo planteado por Manovich (2013), quien afirma que el software se ha convertido en una interfaz con el mundo, con los demás, con la memoria y con la imaginación (pp. 2–3), lo cual deja claro que ya no estamos hablando sólo de una herramienta escondida detrás de una pantalla, sino de un medio cultural de primer orden. En consonancia, Fuller (2008) sostiene que el software estructura y hace posible gran parte del mundo contemporáneo, por lo que no debería verse únicamente como una herramienta técnica (pp. 1–4). Por eso, bajo esta premisa, es válido pensar que además de ser un producto técnico, el software también puede ser una forma de expresión. Reconocer que la programación puede generar belleza, significado y nuevas experiencias culturales y que con reconocer eso no le resta valor técnico, sino que muestra que creatividad y precisión pueden convivir, lo que nos lleva a cuestionar una idea arraigada: programar no debería verse como lo opuesto al arte.

Donald Knuth (1974) ya defendía esta idea cuando explicó por qué la palabra *art* sí era adecuada para hablar de programación. Su punto no era negar la parte científica, sino recordar que en programar también hay criterio,

sensibilidad y búsqueda de calidad, no solo obediencia a reglas (pp. 667–669). Esa idea es importante porque muchas veces, sobre todo en carreras técnicas, se nos enseña a pensar que si algo funciona entonces ya está bien. Pero un software puede funcionar y aun así sentirse feo, frío, confuso o genérico. También puede funcionar y al mismo tiempo, sentirse claro, elegante y hasta percibirse identificado con la persona o el equipo que lo creó. Ahí es donde empieza a aparecer cierto estilo.

Cuando se habla de estilo propio en software, no nos referimos solo a que el código se vea "bonito", sino, que se refiere a que un programa puede traslucir una forma de pensar. Fuller (2008) nos dice que el estilo no es simplemente un adorno o una acumulación de trucos, sino una manera de acceder a distintos universos de referencia; además, cuando habla de elegancia en software, la presenta como algo que no se puede demostrar de forma absoluta, sino como una práctica que se aprende y se ejercita, casi como un arte (pp. 89–90), lo que nos dice que el estilo aparece cuando el software trasciende una suma de funcionalidades y empieza a mostrar una identidad propia.

Dentro de esta discusión sobre el estilo propio del software, el software libre aporta un ejemplo importante, porque muestra que la identidad de un programa no solo puede estar en su apariencia o funcionamiento, sino también en los valores que defiende. Como señala Fuller (2008), "Source code can be considered to have aesthetic properties; it can be displayed and viewed" [El código fuente puede considerarse dotado de propiedades estéticas; puede mostrarse y observarse] (p. 240). Añadiendo a esto, existe el caso de *Ubuntu*, que es un sistema operativo de uso libre, el cual plantea que el software debería estar disponible sin costo, en el idioma local y con libertad para adaptarlo según las necesidades de las personas.

Lo anterior resulta importante porque muestra que el estilo del software no es solo una cuestión visual o técnica, sino también puede ser ético y político,

llegando a decir que un software puede tener identidad no sólo por cómo se ve o cómo funciona, sino por la visión de mundo que sostiene, colaboración, acceso, modificación y comunidad.

Incluso el código fuente, es decir, el conjunto de instrucciones escritas por las personas programadoras para que un software funcione, puede pensarse de forma estética.

No se trata solo de una parte técnica escondida, sino de un espacio donde también se nota una forma de ordenar ideas, resolver problemas y tomar decisiones. Dicho código puede verse como una obra en sí misma, y que escribir un programa puede ser una experiencia parecida a hacer arte por uno mismo. Esta idea nos parece poderosa porque devuelve la atención a la programación como acto creativo. No todo software revela esa dimensión, hay mucho código repetitivo o pensado solo para salir del paso. Pero eso no quita la posibilidad de que exista una programación con voz propia. Más bien la puede llegar a volver más visible, cuando aparece se nota en la claridad, en la manera en que una solución parece simple sin ser pobre y en cómo la forma técnica logra transmitir una visión o una misión.

Ahora bien, hablar de identidad en software se vuelve más complejo en un contexto donde cada vez hay más desarrollo en equipos y más uso de inteligencia artificial. Estas herramientas ya no solo corrigen errores o completan palabras, sino que pueden sugerir fragmentos de código, estructuras y posibles soluciones. De hecho, Peng et al. (2023) muestran que herramientas como *GitHub Copilot* pueden acelerar algunas tareas de programación, pero eso no significa que el estilo propio desaparezca. Más bien, cambia el lugar donde aparece la identidad del programador. Barke et al. (2023) explican que los programadores usan estos modelos tanto para acelerar tareas cuando ya tienen una idea clara, como para explorar opciones cuando todavía están buscando una solución. Para nosotros, eso demuestra que la IA puede participar en la ejecución, pero no reemplaza completamente el criterio humano. Todavía hay decisiones que siguen

dependiendo de la persona o del equipo: qué problema vale la pena resolver, qué solución aceptar, qué estructura mantener, qué experiencia se quiere construir y qué tipo de software se desea entregar. En ese sentido, la identidad no está en hacer todo desde cero por orgullo, sino en dirigir, seleccionar y dar forma al resultado final con criterio propio. La herramienta puede colaborar, pero el estilo aparece en la dirección y diseño, no solo en la ejecución.

Creemos que un software puede tener un estilo propio. Su estilo nace de elecciones técnicas, visuales, éticas y de interacción, de cómo ordena posibilidades, de cómo guía a quien lo usa y de qué visión del mundo deja ver. Pensar así el software no lo vuelve menos serio ni menos práctico, más bien al contrario, obliga a tomarlo más en serio. Si aceptamos que el software también expresa, entonces ya no basta con preguntar si funciona. También hay que preguntarse qué dice, cómo lo dice y desde dónde lo dice.

Entonces, si el software también expresa, lo primero que habría que aceptar es que el estilo no es un adorno superficial ni una capa de pintura sobre una estructura ya terminada. El estilo, cuando es auténtico, nace desde dentro del proceso de creación, igual que la pincelada de un pintor no se añade al final, sino que es la forma misma en que la pintura encuentra su camino sobre el lienzo. Al programar, el estilo se nota en decisiones tan pequeñas como nombrar una variable con cuidado o tan grandes como elegir una arquitectura completa. No es algo que se pueda separar de la función, más bien, es la función hecha carne, hecha lenguaje, hecha gesto. Por eso nos parece que hablar de estilo en software no es minimizar la técnica, todo lo contrario, es reconocer que la técnica, cuando está viva, deja marcas.

Paul Fishwick (2005) lo planteó de modo certero: la computación estética no es aplicar reglas universales de claridad o eficiencia, sino en permitir que las representaciones del software capturen formas personalizadas y estilísticas (p. 139). Dicho de otro modo, un programa puede tener la misma huella que un dibujo

hecho a mano. No porque sea torpe o descuidado, sino porque las decisiones que se toman, desde el flujo de interfaz hasta el tono de sus mensajes de error delatan una forma particular de entender el mundo. Cuando usamos un software y sentimos que "tiene personalidad" no estamos imaginando algo, estamos percibiendo el rastro de quien lo construyó, igual que reconocemos a un amigo por su forma de caminar antes de verle la cara.

McCullough (1996), por su parte, argumenta que todo medio tiene una textura. La madera tiene vetas, la arcilla tiene plasticidad, el lenguaje tiene ritmo. Y el software, aunque parezca etéreo, también tiene su propio "grano". Para él, un artesano es alguien que aprende a sentir las posibilidades y los límites de su medio, y que trabaja con ellos, no en su contra (p. 198). Programar con identidad es exactamente eso, aprender a sentir qué pide el código, cuándo forzar una solución y cuándo soltar, cuándo añadir una línea y cuándo callar, eso no se enseña en un manual de buenas prácticas, se aprende haciendo, probando, equivocándose y, sobre todo, prestando atención a lo que uno mismo siente mientras escribe.

Es pertinente acudir a una voz especializada en otro campo: Terry Hyland, quien ha estudiado la educación vocacional y el trabajo artesanal, sostiene que la separación entre trabajo intelectual y trabajo manual es un error histórico que hemos heredado sin cuestionar (2017, p. 315). Hyland sugiere que en el trabajo manual el pensamiento y la acción no están separados, sino que ocurren de forma integrada. Nos parece que esa idea es clave para entender por qué el software puede tener estilo; si aceptamos que programar no es solo aplicar lógica fría, sino un acto integrado donde la mente y la mano convergen, entonces el código deja de ser un producto anónimo y se convierte en una extensión de quien lo escribe.

Desde nuestra experiencia, el estilo se nota cuando el programador decide qué hacer en situaciones donde no hay reglas fijas. Un formulario puede tener un mensaje de error que, en lugar de decir "dato inválido", dirá "esto parece que no

va a funcionar, pero gracias por intentarlo". Una aplicación puede tener un botón que late suavemente antes de que lo toques, como tentándonos a tocarlo. Esos pequeños gestos no mejoran la eficiencia medida en milisegundos, pero transforman la relación entre la máquina y la persona. Esto nos parece que esto es puro estilo, no se trata de hacer software bonito para vender más, sino que se trata de hacer software que tenga memoria de sí mismo, que no trate a quien lo usa como un extraño.

También creemos que el estilo puede no ser evidente, existe software que parece no tener ninguna intención estética, cuando es funcional, plano, casi invisible, pero incluso esa invisibilidad es una decisión. Un programa que nunca pregunta, que nunca duda, que nunca se toma una pausa, también está diciendo algo sobre su creador. Quizá dice que la eficiencia lo es todo, quizá dice que la emoción estorba o quizá dice que quien lo escribió no quería dejar huella, solo resolver un problema y desaparecer. Nos parece que eso también es un estilo, aunque sea el estilo de la ausencia. Como ocurre en el arte minimalista, donde la reducción no es carencia de ideas, sino una declaración precisa de “menos es más” y de que todo lo que sobra estorba.

En conclusión, un software con identidad no es un lujo reservado para artistas excéntricos o proyectos con presupuesto ilimitado; es, simplemente, un programa hecho por quien se toma en serio su oficio y entiende que cada línea de código es una frase capaz de ser dicha con una entonación distinta. Pensar el software como expresión no lo vuelve menos útil; al contrario, lo humaniza. Como recordaba Manovich (2013), el software es nuestra interfaz con el mundo, con los demás y con nuestra propia memoria (p. 2). Por ello, cuando un programa posee estilo y deja ver la huella de su creador, deja de ser una mera herramienta para convertirse en un mediador cultural que nos invita a reconocernos en él; es, en última instancia, un modo de encontrarnos con el otro o, incluso, con nosotros mismos.

Referencias

Barke, S., James, M. B., & Polikarpova, N. (2023). Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages, 7*(OOPSLA1), 85–111. https://dl.acm.org/doi/epdf/10.1145/3586030

Fishwick, P. A. (Ed.). (2005). Perspectives on Aesthetic computing. https://scispace.com/pdf/perspectives-on-aesthetic-computing-1puqgta3gd.pdf

Fuller, M. (Ed.). (2008). Software studies: A lexicon. MIT Press. https://monoskop.org/images/a/a1/Fuller_Matthew_ed_Software_Studies_A_Lexicon.pdf

Hyland, T. (2017) Craft Working and the "Hard Problem" of Vocational Education and Training. Open Journal of Social Sciences, 5, 304-325. https://doi.org/10.4236/jss.2017.59021

Knuth, D. E. (1974). Computer programming as an art. Communications of the ACM, 17(12), 667-673. https://doi.org/10.1145/361604.361612

Manovich, L. (2013). Software takes command. Bloomsbury Academic. https://monoskop.org/images/9/90/Manovich_Lev_Software_Takes_Command_2013.pdf

McCullough, M. (1996). *Abstracting craft: The practiced digital hand.* MIT Press. https://creativetech.mat.ucsb.edu/readings/abstracting_craft_medium.pdf

Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). *The impact of AI on developer productivity: Evidence from GitHub Copilot* [Preprint]. arXiv. https://arxiv.org/pdf/2302.06590

Ingeniería para la Paz: Más allá del código, un puente hacia la inclusión humana

Por **Keylor Solano Vega**
Osmar Ariel Sánchez Torres

Tradicionalmente, en las aulas de ingeniería se enseña que el quehacer profesional es puramente instrumental; una práctica técnica, libre de valores y moralmente neutra (Páez Michel, 2020). Sin embargo, esta visión resulta miope. La ingeniería es, en su esencia, un acto humano, social y político con el poder de transformar profundamente el entorno y la convivencia. La verdadera "Ingeniería para la Paz" surge cuando dejamos de entender el desarrollo de software como la fabricación de productos descartables y empezamos a cuestionar a qué intereses sirve realmente la tecnología que construimos.

Nuestra experiencia como asistentes en el laboratorio INCLUTEC del Tecnológico de Costa Rica, específicamente en el proyecto de la Plataforma Internacional de Edición de Lengua de Señas (PIELS), nos ha permitido reconfigurar nuestra perspectiva y visibilizar entornos sociales que a menudo son ignorados por el personal técnico. Mientras la carrera tecnológica frenética actual genera estrés e incertidumbre sobre el futuro profesional, las brechas entre los grupos sociales mayoritarios y las personas con discapacidad se ensanchan. Al implementar un avatar virtual 3D que facilita la comunicación efectiva de las personas sordas, comprendimos que la ingeniería no debe ser solo una herramienta para el mercado. Como señala Ochoa en una entrevista realizada por Sanabria (2025), esta disciplina debe ser un motor de "paz positiva", definida "no como la ausencia de conflicto, sino como la búsqueda de estructuras que eliminen la desigualdad y dignifiquen la vida".

Este compromiso social nos conduce a una postura crítica frente a la formación académica tradicional. A menudo, la teoría prioriza estándares de calidad y modelos de madurez técnica que solo son viables en entornos

corporativos de alta rentabilidad. No obstante, en la ingeniería para la paz, surge la necesidad de transitar desde una rigidez teórica estricta hacia una praxis pragmática que priorice el impacto social. No se trata de entregar productos mediocres, sino de reconocer que la ética nos exige ofrecer soluciones funcionales que atiendan necesidades humanas urgentes, incluso si esto implica desafiar la percepción idealizada de los estándares de la industria.

En este sentido, el presente artículo propone que repensar el software como una obra cultural, y no como un objeto de consumo desechable, transforma tanto nuestra relación con la tecnología como nuestra responsabilidad ética. Para sustentar esta tesis, exploramos el software como una forma de expresión, el papel del software libre en la reducción de brechas de acceso y las implicaciones de construir tecnología con una auténtica vocación social.

La calidad técnica y la IA como pilares de la seguridad social

En el ejercicio de la ingeniería, la seguridad y la calidad no son solo requisitos técnicos, sino compromisos éticos que generan paz. Un software mal diseñado o inseguro puede causar estrés o incluso daños sociales, mientras que una herramienta bien construida dignifica la vida. En nuestro trabajo con el proyecto PIELS, hemos comprobado que la calidad se traduce directamente en inclusión; si el avatar 3D no representa fielmente la lengua de señas, la brecha comunicativa persiste y la exclusión se profundiza. En este escenario, la Inteligencia Artificial (IA) actúa como un aliado estratégico que potencia nuestra formación técnica. Como sostienen Zambrana Copaja et al. (2025), la integración de estas tecnologías en la educación superior permite “adaptar contenidos educativos, automatizar tutorías y anticipar posibles dificultades académicas, favoreciendo así un acceso más equitativo” (p. 1), lo cual nos educa para crear productos más robustos que reduzcan las desigualdades estructurales.

Sin embargo, surge un dilema ético sobre la autoría en estos entornos de co-creación: ¿quién es el autor de una herramienta como PIELS cuando

intervienen algoritmos generativos y múltiples colaboradores de diversas disciplinas? Desde nuestra perspectiva, la autoría debe evolucionar de un concepto de propiedad privada a uno de "obra cultural preservable". Visualizar el código de PIELS como patrimonio cultural, y no como un objeto de consumo desechable, eleva el estándar de calidad; ya no escribimos líneas de código para cumplir con una entrega académica, sino para legar una infraestructura que sostenga la comunicación de una comunidad históricamente invisibilizada.

No obstante, esta búsqueda de excelencia técnica enfrenta un reto pragmático en el marco de la ingeniería para la paz. A menudo, la teoría exige estándares de madurez que resultan viables únicamente en entornos con abundantes recursos. En proyectos de impacto social urgente, surge la necesidad de transitar desde la rigidez teórica hacia una agilidad ética que permita ofrecer soluciones que ayuden en el presente. No se trata de comprometer la seguridad o la calidad, los cuales permanecen como pilares fundamentales, sino de comprender que la paz también se construye mediante la prontitud del servicio. En este equilibrio, la IA se convierte en una herramienta de eficiencia social, permitiéndonos alcanzar niveles de robustez aceptables en tiempos reducidos, para no postergar la respuesta a quienes enfrentan barreras de comunicación.

El código como expresión cultural: Redefiniendo la autoría y la estética del software

Si dejamos de ver el código como un producto desechable y lo empezamos a ver como una obra cultural, el concepto de autoría cambia por completo. En la ingeniería tradicional, el autor es quien posee la licencia o quién escribió la línea de código para una empresa. Pero en la ingeniería para la paz, la autoría tiene más que ver con la intención y el impacto social. Sostenemos que el software es una manifestación del esfuerzo humano por transformar su entorno; detrás de cada arquitectura hay un proceso creativo donde se empatiza con un problema real para construir una solución con audiencia y propósito. En el proyecto PIELS,

esto es evidente: desarrollar un avatar virtual no es un simple ejercicio técnico, es un acto de reconocimiento hacia la comunidad sorda.

Aquí surge una duda necesaria: ¿hay verdadera autoría cuando usamos IA o trabajamos en proyectos colaborativos? Nuestra postura es que la autoría no se pierde, sino que evoluciona. La IA es una herramienta que asiste en la construcción, pero la visión ética y la decisión de usar esa tecnología para cerrar una brecha social sigue siendo humana. Como menciona Ochoa (2025), la ingeniería debe permitir que quien la ejerce se sienta pleno con lo que hace, transformando la realidad y no siendo un simple "robot" que sigue instrucciones. Por lo tanto, el autor es quien dirige la herramienta hacia la paz positiva. Si el código es cultura, entonces merece ser cuidado y preservado, porque una arquitectura de software ordenada y una interfaz funcional son decisiones creativas, pensadas y analizadas para que generen confianza, seguridad, dignidad, comodidad y que ayude a tener una mejor vida a el usuario, algo que en algunos software que son tratados como propiedad privada y su único objetivo es ganar dinero; nuestra opinión sobre el tema es que mucho software que solo busque lucrar, también puede ayudar implícitamente a la sociedad, ya que, es software que todos utilizamos en el día a día y sacamos provecho de ello.

Software libre y soberanía digital: El reto de democratizar el bien común

El dilema de tratar el software como propiedad privada o como bien común define qué tanto nos interesa realmente la paz social. Creemos que la democratización del software es uno de los caminos más viables para que la tecnología llegue a quienes más la necesitan. Cuando el código se cierra bajo licencias privadas, se crean barreras económicas que ensanchan las brechas de desigualdad. Al respecto, Bodero et al. (2020) señalan que el software libre es uno de los mayores bienes comunes de la sociedad actual, pero no ha sido aprovechado del todo, esto último para nosotros se da gracias a la naturaleza del software libre, dado que, en muchas ocasiones este modelo de desarrollo no tiene

publicidad, alcance o recursos suficientes para tener un impacto real. Por otro lado, para poblaciones vulnerables, acceder a herramientas de forma libre no es un lujo, es un paso hacia su soberanía digital; les permite apropiarse de la tecnología sin depender de que una empresa decida si su inclusión es rentable o no (Guerschberg, 2025).

Sin embargo, hay que ser realistas, no todo software puede mantenerse libre sin un plan; un riesgo real es la falta de recursos que puede degradar la calidad y la seguridad del producto. Opinamos que, para que el software libre sea una verdadera herramienta de paz, debe pasar por procesos de revisión de calidad tan exhaustivos como los de la industria privada, en medida de lo posible. La paz no se construye con software mediocre o inseguro, sino con productos de alta calidad que sean sostenibles mediante donaciones o apoyo de organizaciones. La ingeniería para la paz nos exige encontrar ese equilibrio: democratizar el acceso para reducir la injusticia, pero garantizando que la herramienta sea lo suficientemente robusta para no fallarle a la comunidad que confía en ella.

En conclusión, la ingeniería para la paz no es un ideal lejano, sino una necesidad urgente que debe integrarse en la formación técnica y ética de los futuros profesionales. Nuestra experiencia en el laboratorio INCLUTEC nos ha demostrado que el código no es un objeto inanimado ni neutral; es una herramienta con carga social que tiene el poder de generar armonía o profundizar la exclusión. Al dejar de ver el software como un producto desechable y empezar a tratarlo como una obra cultural y un bien común, no solo elevamos los estándares de seguridad y calidad, sino que también le damos la humanidad a una disciplina que por mucho tiempo se ha creído puramente instrumental.

Este cambio de paradigma exige ingenieros capaces de utilizar la Inteligencia Artificial no para sustituir el criterio propio, sino para potenciar la creación de soluciones más robustas y accesibles. Como sostiene Ochoa en la

entrevista realizada por Sanabria (2025), el profesional alcanza una verdadera plenitud cuando deja de ser un robot que sigue instrucciones externas y empieza a aplicar un pensamiento crítico para transformar su realidad. La soberanía digital y la democratización del acceso no son solo términos teóricos, son actos de justicia que permiten que comunidades como la sorda se apropien de la tecnología para dignificar su vida. Al final, nuestra responsabilidad como ingenieros e ingenieras no termina cuando el código compila, sino cuando nos aseguramos de que nuestra obra cultural sea sostenible, segura y, ante todo, un puente hacia una sociedad más equitativa.

Referencias

Bodero, E. M., Villacrés, E. P., Radicelli, C. D., y Pomboza, M. R. (2020). El conocimiento y el software libre como un bien común. Revista Espacios, 41(30), Art. 29. https://revistaespacios.com/a20v41n30/a20v41n30p29.pdf.

Guerschberg, L. (2025). Impacto de la soberanía digital en poblaciones vulnerables: El rol del software libre en educación para cerrar la brecha digital. *Sapiens Discoveries International Journal, 3*(1), 1-22. https://doi.org/10.71068/cjtzqc95

Páez Michel, A. L. (2020). La enseñanza de la ética profesional a ingenieros: un caso de estudio. *Innovación Educativa, 20*(84), 125-146. https://www.scielo.org.mx/scielo.php?pid=S1665-26732020000300125&script=sci_arttext

Sanabria Rodríguez, A. [Profe Aurelio Sanabria Rodríguez]. (21 de octubre de 2025). Alexei Ochoa - Ingeniería para la paz | Entrevista [Video]. YouTube.https://www.youtube.com/watch?v=IdjesBgfmRo

Zambrana Copaja, R., Salinas Montemayor, A. D., Macías García, F. A., y Escobar, E. E. (2025). Inteligencia artificial en la educación superior para promover un aprendizaje personalizado e inclusivo: una revisión sistemática. *Revista Invecom, 6*(2), e602051. https://ve.scielo.org/scielo.php?pid=S2739-00632026000202051&script=sci_arttext.

El código como lienzo: La importancia de la estética del software en la era de la mercancía

Por **Justin Vasquez Tellez**
Bryan Morales Calderón

En la formación académica de la ingeniería, se nos condiciona a pensar en el software bajo una lógica principalmente utilitaria. Se nos enseña que un buen programa es aquel que optimiza recursos, reduce tiempos de ejecución, resuelve un problema con alta precisión y que carece de bugs. Bajo esta premisa, el código se percibe como una herramienta invisible, un medio para un fin comercial o funcional que pierde su valor una vez completada su tarea, ya que ese es su único propósito. Sin embargo, esta visión es profundamente reduccionista y omite una dimensión fundamental de nuestra labor: la capacidad del software para ser un medio para la expresión creativa y una obra cultural en sí misma.

Si entendemos el arte como una manifestación que organiza la experiencia humana y genera significado, es imposible ignorar que, hasta cierto punto, tiene relación con la programación. Al programar, no solo estamos dando instrucciones a una máquina, sino que estamos construyendo estructuras lógicas, diseñando según nuestras diferentes formas de pensamiento y a la vez moldeando la forma en que las personas entienden e interactúan con la realidad digital. El software deja de ser una simple mercancía para convertirse en una extensión de la literatura y la arquitectura moderna, donde la elegancia o simpleza de un algoritmo funcional puede llegar a conmover como la métrica de un poema.

La elegancia como criterio de calidad: El legado de Donald Knuth

Para mantenernos en la idea del software como arte, es importante recurrir a la figura de Donald Knuth. En su histórica conferencia *Computer Programming as an Art* presentada originalmente en 1974, Knuth (1974/2007) cuestionó la visión de la programación como una actividad puramente técnica. Según su

planteamiento, la programación aspira a la belleza a través del estilo y la claridad. Knuth sostiene que un programador que se preocupa por la estética de su código no solo produce un software más robusto, sino que, al crearlo, experimenta una satisfacción creativa similar a la de un músico componiendo una sinfonía.

Desde nuestra perspectiva, la belleza en el software se manifiesta en la legibilidad y la armonía que se encuentra dentro del sistema. Hay algo, por así decirlo, artístico en un código que, siendo complejo en su función, logra ser simple en su lectura. Cuando nos enfrentamos a un proyecto donde las funciones están conectadas de manera lógica y fluida, estamos ante una obra de artesanía digital, en comparación a cuando el código es un desastre desordenado e inentendible (aunque funcional). La utilidad de mercado suele premiar la rapidez del lanzamiento, pero la calidad artística del software se sostiene en aquello que no siempre es visible para el usuario: la arquitectura del código. Un programa "bello" es aquel que respeta a ambos, tanto al lector humano como a la máquina que lo realiza.

El software como arquitectura de la experiencia: Videojuegos y arte generativo

Para comprender el software como una obra cultural, no podemos ignorar partes en donde la funcionalidad y la estética son inseparables. Un ejemplo que en la actualidad y en el futuro va a seguir siendo importante es el desarrollo de videojuegos independientes (*indies*), donde el código no solo sostiene las mecánicas, sino que tiene un papel importante en la parte emocional del creador. En proyectos como estos, el programador actúa como un arquitecto de mundos, y como estos videojuegos son creados por un grupo limitado de personas, es algo muy personal, en comparación a un trabajo hecho por una gran empresa. Como señala Christiane Paul (2023) en su análisis sobre el arte digital, el software permite crear instalaciones interactivas donde el espectador ya no es un sujeto pasivo, sino un co-creador de la experiencia a través de los algoritmos de respuesta.

Desde nuestro punto de vista, esto demuestra que el código tiene un *estilo* propio que se puede comparar al de una película. Así como un director de cine elige un ángulo de cámara para transmitir una emoción, un programador elige un algoritmo para transmitir su idea, ya sea de forma simple o compleja. La belleza no está solo en el producto final, sino en la lógica o procedimiento que lo sustenta.

El diseño por números: La estética de la limitación

Podríamos considerar también la obra de John Maeda (1999), quien en *Design by Numbers* explora cómo la programación permite a los diseñadores romper las barreras de las herramientas comerciales tradicionales. Al crear nuestras propias herramientas mediante el código, recuperamos esa parte artística.

En lugar de usar un software cerrado que impone sus propias reglas, se crea el software para imponer las nuestras. Consideramos que esta libertad es lo que vuelve a la programación tan interesante.

La analogía literaria: El código como narrativa

Otra forma de expandir nuestra visión del software es compararlo con la literatura. Existe una relación entre la figura del escritor y la del programador: escribimos para ser leídos por otros humanos. Un sistema de software bien documentado y estructurado cuenta una historia sobre cómo se resolvió un problema, cuáles fueron las decisiones del autor y qué prioridades tenía en ese momento histórico.

Cuando dejamos de ver el código como algo que se hace solo para cumplir su tarea, se siente desechable, pero si lo tratamos como una obra que merece ser preservada, estamos otorgándole un valor personal. El software de los sistemas operativos pioneros hoy se estudia como piezas históricas no solo por lo que hicieron, sino por cómo fueron escritos bajo limitaciones extremas. Eso demuestra

la creatividad que tuvieron los programadores al conseguir crear el código a pesar de tener herramientas tan básicas; esa es la verdadera importancia: dejar una huella de pensamiento humano en un medio digital que, de otro modo, sería puramente comercial.

El código como discurso de expresión social

Avanzando en esta reflexión, el software no solo es una obra estética, sino también un acto político. Como argumentan Cox y McLean (2013) en su obra *Speaking Code*, la programación es una práctica que organiza posibilidades e impone estructuras en el mundo. El acto de escribir código nunca es neutral; cada decisión técnica conlleva una carga política sobre cómo debería funcionar la sociedad, ya que define qué es posible, qué optimizar, qué modelar y en última instancia, quiénes son los sujetos habilitados a ser el público meta para interactuar con el programa.

No solo la importancia recae en que un código funcione; también importa la accesibilidad, la privacidad o la facilidad para el usuario. Poniendo de ejemplo a arquitectos que logran diseñar una ciudad o lugar hermoso, en el caso de que no tenga accesibilidad o no tomara en cuenta las necesidades de los usuarios con necesidades especiales, solo ciertas personas podrían disfrutarla, ya que, por ejemplo, si hay muchas escaleras, no hay rampas, un edificio enormemente alto sin ascensor, los usuarios no tienen cómo transportarse libremente y, por ende, no tienen cómo disfrutar esa obra realmente.

Con base en esta información, podemos decir que el movimiento del software libre se presenta como una forma de resistencia cultural. Cuando el código se libera, deja de ser una propiedad privada. Aquí, la autoría se vuelve colaborativa y el estilo de un proyecto se define por la comunidad que lo mantiene. Ver el código como una obra digna de ser preservada cambia nuestra relación con la tecnología: dejamos de ser simples consumidores para ser partícipes de una cultura tecnológica duradera.

La tensión con la inteligencia artificial y la nueva autoría

Uno de los debates más urgentes en la actualidad es la intervención de la inteligencia artificial en la creación de software. ¿Puede algo como un código generado por un modelo de lenguaje ser considerado arte? Nuestra opinión es que, si bien la IA puede generar fragmentos de código funcionales, basarse completamente en su ayuda para hacer el código completo carece de la intención que define a la obra artística y, además, ¿qué diferenciaría a un programador que se basa completamente en la ayuda de la IA para su código de una persona común haciendo lo mismo? El arte requiere de una visión humana que busque comunicar algo.

La IA es una herramienta técnica avanzada, pero ¿se puede considerar que la autoría se mantiene en quien forma el concepto y le dice a alguien más que lo haga? Además, dado que los creadores o dueños de las IA tienen influencia en el contenido que se genera, en este caso, si la herramienta no permite hacer todo lo que usted quiere crear, ¿realmente se tiene la misma libertad creativa si alguien más limita lo que se puede hacer? En este sentido, dejar el trabajo de la creación total a la IA es renunciar a nuestra voz, creatividad personal y expresión artística, y reducir el software a una simple mercancía automatizada y sin alma.

La ingeniería con imaginación

En la formación de un ingeniero es común escuchar que su producto o trabajo tiene que ser lo más eficiente posible; aunque eso es cierto y siempre se debe tender a mejorar, no debe reducirse solo a eso. La verdadera innovación o diferencia entre un ingeniero y otro está en su toque personal. En si su producto está pensado y hecho con cariño, en vez de simplemente realizarlo para entregar o para “hacer lo mínimo aceptable” como un producto comercial, eso es lo que lo diferencia. Como sugiere Malcolm McCullough (1996), la capacidad de pensar críticamente sobre el impacto humano de lo que construimos es lo que separa a un técnico de un verdadero ingeniero con visión artística.

El rigor y la creatividad no son opuestos; se necesita de ambos para crear sistemas o productos que no solo sean resolver un problema o terminar un trabajo; también deben tener un significado profundo, que se note el cariño con el que está hecho y el empeño que ha tomado hacerlo.

En conclusión, reconocer el software como una forma de arte es un acto de rebeldía contra la mercantilización absoluta de la tecnología. Al valorar el código por su estilo, su elegancia y su capacidad expresiva, estamos humanizando la ingeniería. Como futuros profesionales, tenemos la responsabilidad de no ver nuestras creaciones simplemente como productos con fecha de caducidad, sino como piezas de un rompecabezas cultural que define nuestra época. El software es un espejo de nuestra persona, sociedad y, como tal, merece ser tratado con el respeto y la dedicación que se le otorga a cualquier obra maestra de la humanidad.

Referencias

Cox, G., & McLean, A. (2012). *Speaking code: Coding as aesthetic and political expression*. MIT Press. https://doi.org/10.7551/mitpress/8193.001.0001

Knuth, D. (2007). *Computer programming as an art. AMC.* (Obra original publicada en 1974). https://doi.org/10.1145/1283920.1283929

Maeda, J. (1999). *Design by numbers*. MIT Press. https://archive.org/details/designbynumbers0000maed/page/8/mode/2up

McCullough, M. (1996). *Abstracting craft: The practiced digital hand*. MIT Press. https://archive.org/details/abstractingcraft0000malc

Paul, C. (2023). *Digital art* (4th ed.). Thames & Hudson. https://openlibrary.org/books/OL27504948M/Digital_Art

El arte detrás de los algoritmos

Por **Ximena Molina Portilla**
Susana Feng Liu

Por definición, ¿podría el software ser arte? El concepto de arte suele variar dependiendo de la fuente de la cual se obtenga su definición. Para efectos de este artículo, tomaremos como referencia la propuesta por Tolstói (2024) quien dice que “el arte no es una alegría, ni un placer, ni una diversión; el arte es una gran cosa. Se trata de un órgano vital de la humanidad que transporta al dominio del sentimiento las concepciones de la razón” (p. 140). Lo que distingue al arte, bajo esta perspectiva, no es la motivación interna de quien crea, sino la capacidad de la obra para transmitir emociones o ideas del mundo a quienes la experimentan. En este sentido, el software podría considerarse arte porque logra comunicar valores, emociones o intenciones a través de su diseño, funcionamiento e impacto en sus usuarios, no simplemente porque el programador lo haya creado con sentimiento o buena intención

El software debe tener aspectos con los cuales evaluar su calidad, de la misma manera que se evaluaría una obra musical o literaria. No se podrían utilizar los modelos de calidad existentes que ayudan a realizar esta evaluación porque, desde un punto de vista artístico, estos no tienen sentido. Además, hay que tomar en cuenta el tipo de software que se esté evaluando. Por ejemplo, un lenguaje de programación puede ser evaluado por su propósito, su originalidad, su nivel de ambigüedad, entre otros criterios. Con esto podemos confirmar que existen criterios aplicables al software, solo hay que moldearlos o crearlos de acuerdo al tipo del mismo.

Tomando el ejemplo anterior de los lenguajes de programación, podemos observar cómo este caso se asemeja a la literatura, ya que muchos de estos criterios son igualmente aplicables en ella. Al igual que en la literatura, el código es algo que se lee, puede confundir al lector, frustrarlo y generarle muchas otras

emociones. Incluso más allá de su lectura, cierto software puede transmitir valores culturales concretos: un sistema desarrollado en una lengua indígena, por ejemplo, no solo cumple una función técnica, sino que visibiliza comunidades históricamente marginadas y evidencia tensiones relacionadas con la inclusión tecnológica. Esto nos inclinaría aún más hacia la idea de considerar el software como arte, pero cabe destacar un área específica que últimamente ha dado mucho de qué hablar y donde quizás empecemos a dudar de nuestra postura: la inteligencia artificial.

¿Podría un producto desarrollado por IA ser considerado arte, aunque esta no posea sentimientos y, sobre todo, un alma?

En los últimos años, se ha generado un gran debate con respecto a la inteligencia artificial, específicamente la generativa. Si entendemos el arte como aquello que logra transmitir emociones e ideas, habría que preguntarse si un producto generado por IA es capaz de provocar ese efecto en quien lo recibe. Sin embargo, hay que recordar que la IA no se genera sola: debe haber un ser humano que construya un prompt que dispare la generación de una imagen, video, texto o audio. En este sentido, se ha pensado que la IA constituye una capa adicional de abstracción dentro del proceso creativo, de manera análoga a otras herramientas como compiladores o frameworks. La intención y los valores del desarrollador no desaparecen, sino que se expresan a través del uso de estas tecnologías, por tanto, lo que se comunica a través del resultado podría, en principio, seguir siendo de origen humano.

No obstante, hay un punto importante que, a pesar de no estar incluido en la definición de arte, hace la diferencia: la originalidad. En mi opinión, este aspecto es el punto de quiebre de la discusión. Hay que recordar que la manera en que funciona la inteligencia artificial “es esencialmente estadístico: aprende patrones, distribuciones y relaciones de palabras, imágenes o sonidos para generar resultados coherentes a nuevas entradas” (Aguilar, 2025). Todo lo que genera la

inteligencia artificial es un conjunto de elementos que “aprendió” que, en otras palabras, significa que extrajo de algún elemento de una base de datos. De acuerdo con Osorio Umaña (2025), un objeto solo puede considerarse original si refleja la personalidad de su autor a través de elecciones libres y creativas. En este sentido, cuando el proceso de creación está condicionado por exigencias técnicas, reglas u otras limitaciones que anulan dicha libertad, la jurisprudencia del Tribunal de Justicia de la Unión Europea concluye que el objeto carece de la originalidad necesaria para ser protegido como obra (p. 6). Bajo este criterio, la inteligencia artificial no puede ser considerada original.

Ahora bien, vale la pena cuestionarse si otros tipos de software enfrentan el mismo problema. La respuesta es no. Por ejemplo, los videojuegos, especialmente aquellos desarrollados por un equipo pequeño de personas. Existen muchos juegos cuyo propósito es concientizar sobre algún tema específico, y esto se suele lograr desarrollando mecánicas distintivas que los hagan únicos y memorables. Crear un juego con una idea original es totalmente necesario en este ámbito. A diferencia de la IA, estos softwares no toman decisiones creativas por sí mismos, sino que ejecutan directamente las instrucciones del programador.

¿De quién es el arte cuando el código es de todos?

El tema de la originalidad trae a la mesa un aspecto importante del arte: este suele tener dueño. Un pintor firma su pintura una vez que la termina para mostrar que fue él quien la creó, una canción siempre trae una sección de créditos para rendir homenaje a las personas que la crearon, pero ¿y el software? Es cierto que se puede mostrar al público ejecutables con sus respectivos créditos, sin embargo, ¿qué pasa con el software de código libre? ¿Acaso este representa una amenaza al arte?

El hecho de que un código sea libre, significa que, cualquier persona pueda modificarlo o utilizarlo, no implica que carezca de autoría. En el software de código abierto existen diversas formas de reconocer la contribución de quienes

participan en su desarrollo. Una de las más comunes es a través del historial de *commits* en plataformas como GitHub o GitLab, donde cada cambio queda registrado con el nombre del autor. Además, según la guía en línea de *Open Source Guide*, se recomienda que en los proyectos se incluyan archivos de documentación como *CONTRIBUTORS.md* o secciones de agradecimientos en el *README.md* (GitHub, s.f.), que funcionan como espacios formales para dar crédito a los programadores y colaboradores. De esta manera, aunque el código sea compartido libremente, la autoría y las aportaciones individuales permanecen visibles y reconocidas.

Ahora bien, bajo el entendimiento de que el código puede llegar a ser arte cuando existe intención, propósito, sentimiento y una estructura cuidadosamente construida, en ese caso no podríamos necesariamente clasificar al código abierto como una amenaza para el arte en sí mismo. Por el contrario, podríamos decir que esta forma de desarrollar software llega a modificar la forma en la que entendemos la autoría y la creación artística.

Por ejemplo, si se compara con la música, un músico puede crear una pista considerada completamente propia a partir de demos, samples o melodías ya existentes. Aunque parte de elementos previos, la obra final sigue siendo suya porque la transforma, la reorganiza y le imprime su propio estilo. Del mismo modo, el código abierto parte de una base existente, pero cada desarrollador aporta algo de sí mismo: una solución, una mejora, una forma distinta de organizar el sistema o incluso una visión diferente del proyecto. Por supuesto, siempre y cuando se realice bajo los lineamientos con los cuales fue creado el proyecto de software libre. Puesto que no necesariamente todos admiten contribuidores (Open Source Guide, s.f.), que en ese caso la modificación del código fuente ya no se consideraría una contribución y podría llegar incluso a considerarse una copia más.

Bajo esta premisa, el código abierto puede entenderse como una forma de arte de tipo colaborativa. A diferencia de otras disciplinas donde la participación de varias personas puede diluir la autoría, en el software libre esta se conserva y se documenta. Como habíamos mencionado anteriormente, cada contribuidor recibe reconocimiento mediante *commits*, historiales de versiones o documentos dentro de plataformas como GitLab. De esta manera, el proyecto sigue teniendo autores, pero estos no son una sola persona, sino una comunidad de individuos que enriquecen constantemente la obra original. De manera que la nueva versión es una nueva obra original.

Además, esta colaboración suele surgir de una motivación distinta a la puramente económica. Tal como describe González Barahona et al. (2008), "el modelo de desarrollo en proyectos de software libre suele ser más informal, debido a que gran parte del equipo de desarrollo realiza esas tareas de manera voluntaria y sin recompensa económica" (pp. 111-112). Es decir, los contribuidores no participan porque esperan una ganancia inmediata o porque desean convertir el proyecto en un producto para vender, sino porque buscan mejorar una herramienta que consideran útil y ponerla al alcance de más personas. Existe una intención genuina de aportar, perfeccionar el código y permitir que cualquiera pueda utilizarlo, adaptarlo o incluso continuarlo.

Esta idea la podemos relacionar con ciertas formas de arte que adquieren valor precisamente cuando dejan de estar limitadas a unos pocos. El arte callejero, por ejemplo, existe en espacios públicos y puede ser visto por cualquier persona, sin importar su condición económica o social. Básicamente se toma el "espacio público como un lugar donde la interacción se basa en la posibilidad de expresar las diferencias, la pluralidad de discursos entendiendo lo "público" como un espacio común a todos, compartido por todos." (Levit y Pérez Roig, 2024, p. 75). Entonces, de la misma forma en la que el arte callejero se legitima en un espacio público accesible por todos, el software libre bajo esta perspectiva también obtiene valor cuando puede llegar a la mayor cantidad de personas.

Por otro lado, bajo este contexto, algunas personas podrían argumentar que abrir el código le resta valor, ya que cualquiera puede acceder a él, modificarlo o reutilizarlo. Sin embargo, esa crítica parte de una concepción del valor basada en la exclusividad. El principio mismo del código abierto consiste en que el conocimiento sea accesible para todos y en que la tecnología pueda evolucionar gracias a la participación colectiva. Por ello, también es válido pensar que el código abierto adquiere un valor mayor conforme llega a más personas, del mismo modo en que una obra artística puede volverse más significativa cuando logra inspirar, ayudar o transformar a una comunidad entera.

Referencias

Aguilar, O. (1 de julio, 2025). *Inteligencia artificial generativa vs. el potencial del cerebro humano: ¿Qué nos hace únicos?* Sanfer. https://sanfer.com.mx/blog/inteligencia-artificial-generativa-vs-el-potencial-del-cerebro-humano-que-nos-hace-unicos

GitHub. (s.f.). *Open Source Guide*. https://opensource.guide/

González Barahona, J., Seoane Pascual, J., y Robles, G. (2008). Introducción al software libre, febrero 2008. Universitat Oberta de Catalunya. https://openaccess.uoc.edu/server/api/core/bitstreams/f0f98d43-f6cf-45e6-abb6-1b58e68825c3/content

Levit, E. L., y Pérez Roig, P. (2024). Arte callejero: una herramienta de transformación social y del entorno. En M. Jalo (Coord.), Reconstruir la memoria pedagógica: experiencias docentes del departamento de Lenguas Modernas del Colegio Nacional (pp. 75-84). Editorial de la Universidad Nacional de La Plata (EDULP). http://sedici.unlp.edu.ar/handle/10915/170095

Osorio Umaña, F. (2025). El concepto de originalidad del Tribunal de Justicia de la Unión Europea y la inteligencia artificial. Revista Justicia & Derecho, 8(1), 1–11. https://doi.org/10.32457/rjyd.v8it.2873

Tolstoi, L. (2024). *¿Qué es el arte?* Ediciones Clío. https://doi.org/10.5281/zenodo.13958192

Umaña, F. O. (2025). El concepto de originalidad del Tribunal de Justicia de la Unión Europea y la inteligencia artificial. *Revista Justicia y Derecho*, *8*(1), 1. https://revistas.uautonoma.cl/index.php/rjyd/article/download/2873/2102

Autoría y propiedad del software que construimos entre todos: ¿a quién pertenece el código?

Por Kaleb Coto Elizondo
Tomás André Medina Vargas

A primera vista, si vos tecleás el código, sos el dueño. Sin embargo, cualquiera que programe hoy sabe que esa noción se complica cuando se examina con detenimiento. En los proyectos de código abierto contribuyen miles de personas; las herramientas de inteligencia artificial generan bloques enteros de código a partir de una sola instrucción y la práctica habitual incluye la reutilización de soluciones publicadas en plataformas como GitHub. Encajar esta forma de trabajo colectivo en las definiciones tradicionales de autoría individual y propiedad privada requiere, cuando menos, una revisión crítica de dichos conceptos.

No es solo una pregunta teórica; tiene consecuencias concretas: quién puede usar un programa, quién puede modificarlo si contiene un error, quién puede auditarlo cuando maneja datos sensibles. Estas preguntas adquieren mayor urgencia ahora que el software está presente en la salud, la educación y los procesos electorales. Por eso vale la pena detenerse a analizar, más allá del debate abstracto sobre autoría, qué modelo de propiedad resulta más adecuado para nuestra sociedad y qué implica cada opción.

La autoría en tiempos de colaboración e inteligencia artificial

Antes, programar era una actividad individual o de equipos pequeños y cerrados. El modelo de código abierto transformó radicalmente esa imagen. Proyectos como Linux, Python o el núcleo de Android existen gracias a contribuciones distribuidas de personas en todo el mundo. Claro, Linus Torvalds inició el desarrollo del kernel en 1991, pero atribuirle la autoría completa de lo que existe hoy resulta impreciso: el sistema actual incorpora el trabajo de miles de desarrolladores que nunca tuvieron contacto directo entre sí. Como explica Paul

(2015), el arte digital y el software superaron los marcos de la creación individual al introducir formas de colaboración que esos marcos simplemente no contemplaban.

La expansión de herramientas de inteligencia artificial que generan código agrega otra capa a este problema. Franganillo (2023) analiza cómo estos sistemas están modificando las bases tradicionales de la creatividad al producir contenido a partir de instrucciones textuales, sin que exista un acto deliberado de escritura por parte del programador. Cuando vos le pedís a una IA que escriba una función completa y la integra en su proyecto, el proceso no sigue el modelo clásico de composición, aunque la decisión de usarla, adaptarla y asumir sus consecuencias sí recae sobre vos.

En ese sentido, la IA no es fundamentalmente distinta de otras herramientas que amplían la capacidad del programador sin reemplazar su juicio. Funciona de forma análoga a un compilador: transforma instrucciones en un resultado utilizable, pero a nadie se le ocurriría considerarlo coautor del programa. Si el código generado produce un error, la responsabilidad recae sobre quien lo incorporó sin revisarlo. Si funciona correctamente, el mérito corresponde a quien supo plantear el problema y evaluar la solución. Al final, escribir a mano cada línea importa bastante menos que entender qué hace ese código y por qué decidiste usarlo.

Por esa misma razón, reutilizar código publicado en plataformas como GitHub no equivale a una apropiación indebida: es parte de cómo funciona la cultura compartida del desarrollo de software. Lo que distingue a un profesional honesto no es el origen del código, sino la claridad con que reconoce las fuentes y asume la responsabilidad sobre el producto final.

Propiedad intelectual: ¿un modelo que ya no encaja?

El modelo de propiedad intelectual que regula el software fue diseñado originalmente para obras atribuibles a un creador identificable: un libro, una

composición musical, una pintura. Su premisa fundamental es que existe un autor singular cuyo esfuerzo creativo merece protección jurídica durante un periodo determinado. El software contemporáneo rara vez cumple esa condición: es iterativo, se construye sobre versiones anteriores, incorpora dependencias externas y, en muchos casos, resulta del trabajo distribuido de cientos de personas a lo largo de años. Aplicar ese marco a un objeto que es por naturaleza colectivo genera tensiones que se vuelven evidentes en la práctica: las licencias de código abierto, las disputas sobre dependencias y los límites de la propiedad sobre algoritmos son problemas que el derecho de autor clásico no estaba diseñado para resolver.

El desarrollo de software opera, en gran medida, bajo una lógica de conocimiento compartido y acumulado. Los programadores adaptan soluciones que otros publicaron, contribuyen a proyectos ajenos y construyen sus herramientas sobre código que alguien más escribió antes. Cox y McLean (2013) señalan que el software nunca es neutral: define qué operaciones son posibles y bajo qué condiciones, e impone estructuras que organizan las posibilidades de quienes lo usan. En ese sentido, decidir quién puede ver, modificar o distribuir el código no es solo una decisión económica, sino una decisión sobre quién tiene acceso y control sobre esas estructuras.

Esto no implica que el modelo propietario carezca de legitimidad. Desarrollar software requiere una inversión significativa de tiempo y recursos económicos, y es razonable que quienes lo producen busquen condiciones para recuperar esa inversión. Desde esta perspectiva, la propiedad intelectual cumple una función económica comprensible. El problema no es la existencia del modelo privado, sino la tendencia al monopolio que puede generar cuando no coexiste con alternativas reales y accesibles. Cuando no hay opción, el usuario no está eligiendo: está aceptando lo único disponible, y eso tiene implicaciones que van bastante más allá del precio del software.

Stallman (2020) define el software libre en términos de libertad, no de precio: la posibilidad de ejecutar, estudiar, modificar y redistribuir un programa convierte al usuario en alguien que comprende y controla las herramientas que usa. Esta distinción resulta especialmente relevante cuando el software en cuestión gestiona datos médicos, procesos electorales o comunicaciones privadas. En esos contextos, no poder revisar el código no es solo una limitación técnica, sino un problema de confianza institucional. Como señalaba McCullough (1996), las herramientas digitales moldean la manera en que pensamos y actuamos; si esas herramientas operan bajo control exclusivo y sin alternativas accesibles, el margen de autonomía tanto técnica como ciudadana se reduce de forma concreta.

Lo que resulta preocupante, en definitiva, no es el software privado como tal, sino la concentración de poder que puede derivarse de un ecosistema donde solo existe ese modelo. Un entorno tecnológico con diversidad de opciones —libre y privado— puede servir mejor a distintas necesidades y contextos, sin que ninguno de los dos modelos deba imponerse como el único válido.

El software como expresión: entre el código y la cultura

Hay una dimensión de esta discusión que suele quedar fuera del debate jurídico y económico: si el software puede ser arte, la pregunta sobre a quién pertenece adquiere una capa adicional de complejidad. Knuth (1974) argumentaba que la programación no debería reducirse a la corrección técnica, sino que puede aspirar a la belleza, el estilo y la elegancia. Un algoritmo bien escrito tiene algo comparable a la claridad de un buen párrafo o a la coherencia estructural de una composición musical.

Maeda (1999) va en una dirección similar cuando reflexiona sobre cómo las herramientas digitales transforman la relación entre quien crea y lo que crea: la herramienta pasa a ser parte del resultado. Esta perspectiva complica la pregunta de la autoría, pero también la enriquece. Si el software puede entenderse como

una forma de expresión cultural, entonces las decisiones sobre quién puede verlo, copiarlo o modificarlo no son solo decisiones económicas: son decisiones sobre quién tiene acceso a esa expresión y quién queda excluido de ella.

Vale reconocer que esta analogía entre software y arte tiene límites. No todo programa aspira a la elegancia estética; muchos son desarrollados bajo criterios de eficiencia y funcionalidad sin ninguna pretensión artística. Sin embargo, incluso en esos casos, el hecho de que el software se construya de forma colectiva y acumulativa hace que la pregunta sobre la autoría individual siga siendo compleja. La dimensión expresiva simplemente la hace más visible.

Esta idea tiene implicaciones profundas para la formación en ingeniería. Si el código puede ser expresión, aprender a programar no es solo aprender a resolver problemas: es aprender a comunicarse con precisión, a construir cosas que otros puedan entender y mejorar, y a participar en una conversación técnica y cultural que lleva décadas en marcha.

Una reflexión desde la ingeniería

Para quienes estudiamos ingeniería en computación en el TEC, estas preguntas no son un debate filosófico lejano. Desde el inicio de la carrera, trabajamos con sistemas basados en Linux, entornos como LibreOffice y lenguajes cuyas especificaciones son públicas. Rápidamente se hace evidente que el conocimiento técnico se acumula y no surge de la nada: aprender a programar implica apoyarse en herramientas que otras personas diseñaron, mantuvieron y pusieron a disposición. Sería contradictorio formarse en ese ambiente y luego sostener que el software es una propiedad estrictamente individual e intransferible.

Eso no significa que la responsabilidad individual desaparezca. Hay una diferencia clara entre usar el trabajo de otros con atribución honesta y apropiarse de código ajeno sin reconocimiento. La diferencia no está en el origen del código, sino en la honestidad con que se usa. Knuth (1974) argumentaba que programar

bien tiene una dimensión estética genuina, lo que implica un compromiso real con la calidad del propio trabajo. Ese compromiso no es compatible con el plagio, pero sí encaja perfectamente con aprender de la comunidad y construir sobre lo que otros han hecho.

En la ingeniería de computación, el programador se encuentra de forma permanente entre su propio esfuerzo y el trabajo colectivo: cada solución propia se apoya en lo que otros construyeron antes, y cada aporte propio puede convertirse en la base sobre la que trabaje quien venga después. Como señalaba Maeda (1999), la herramienta digital forma parte activa de la creación sin llegar a reemplazar al creador. Admitir esa interdependencia no le resta valor ni mérito al trabajo propio; simplemente lo sitúa en el contexto real en el que ocurre.

La pregunta sobre quién es el dueño del software no tiene una respuesta única, y esa dificultad no es una falla del análisis sino un reflejo de la complejidad del objeto que se estudia. El software contemporáneo es una creación acumulativa: cada programa incorpora decisiones, convenciones y código producido por quienes trabajaron antes. Los marcos legales que intentan reducirlo a la posesión de un individuo o empresa logran capturar solo una parte de esa realidad.

Tanto el modelo libre como el privado tienen lugar dentro del ecosistema tecnológico, y probablemente la respuesta más honesta no sea elegir uno y descartar el otro, sino entender en qué contextos cada uno sirve mejor y qué condiciones deben existir para que ambos puedan coexistir sin que ninguno elimine al otro. Lo que sí parece claro, retomando a Stallman (2020), es que la posibilidad de revisar el código que se ejecuta en los propios dispositivos no debería ser un privilegio técnico, sino una condición accesible para cualquier usuario.

La inteligencia artificial no suprime la autoría: la redefine. Franganillo (2023) señala que su llegada plantea preguntas sobre autoría y propiedad que todavía no

tienen respuesta consolidada. Pero si se usa con criterio, el autor sigue siendo quien decide, quien evalúa y quien asume la responsabilidad del resultado. Al final, la pregunta por el código es también una pregunta por nosotros como comunidad técnica: si el software es, como plantean Cox y McLean (2013) y como sugería Knuth (1974), una forma de expresión colectiva, entonces las decisiones sobre quién puede acceder a él y modificarlo son también decisiones culturales. Y esas decisiones merecen debatirse con la misma seriedad con que se discute cualquier otra política que afecte el acceso al conocimiento.

Referencias

Cox, G., & McLean, A. (2013). *Speaking code: Coding as aesthetic and political expression.* MIT Press.

Franganillo, J. (2023). La inteligencia artificial generativa y su impacto en la creación de contenidos mediáticos. *methaodos. Revista de Ciencias Sociales,* 11(2), m231102a10. https://doi.org/10.17502/mrcs.v11i2.710

Knuth, D. E. (1974). Computer programming as an art. *Communications of the ACM,* 17(12), 667-673. https://doi.org/10.1145/361604.361612

Maeda, J. (1999). *Design by numbers.* MIT Press.

McCullough, M. (1996). *Abstracting craft: The practiced digital hand.* MIT Press.

Paul, C. (2015). *Digital art* (3.a ed.). Thames & Hudson.

Stallman, R. M. (2020). La definición de software libre. *Communiars. Revista de Imagen, Artes y Educación Crítica y Social,* (3), 151-154. https://dx.doi.org/10.12795/Communiars.2020.i03.09

Por un mundo nuevo,
uno nuestro y para todas y todos.