El arte de programar con identidad

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