MONOGRÁFICO
MÈTODE SCIENCE STUDIES JOURNAL (2020). Universitat de València.
DOI: 10.7203/metode.11.16112
Enviado: 18/11/2019. Aceptado: 10/03/2020.

EL LARGO Y SINUOSO CAMINO

Accidentes y bricolaje en la estandarización del *software*

SERGI VALVERDE

El *software* se basa en principios universales, pero no así su desarrollo. Relacionar el *software* con el *hardware* no es nunca automático ni sencillo. Los intentos de optimizar la producción de *software* y reducir su coste (como ocurre con el *hardware*) han sido muy limitados. El éxito de un proyecto suele depender de las acciones de individuos altamente cualificados y experimentados. El largo y complicado camino hacia un *software* fiable y útil suele estar plagado de accidentes idiosincrásicos y de una complejidad emergente. Se esperaba que la estandarización del *software* eliminara estas fuentes no deseadas de diversidad para alcanzar procesos de desarrollo controlables, lenguajes de programación universales y componentes reutilizables. Sin embargo, la adopción limitada de estándares de desarrollo sugiere que todavía no comprendemos las razones por las cuales es tan difícil producir *software*. La estandarización de *software* se ha visto obstaculizada por nuestra limitada comprensión del papel de los humanos en el origen de la diversidad tecnológica.

Palabras clave: estándares de *software*, desarrollo de *software*, lenguaje de programación, complejidad, evolución tecnológica.

Imaginemos que, al escribir una carta de amor, nos viéramos obligados a escribir oraciones con un número fijo de caracteres. Olvidemos la idea de utilizar una prosa compleja o de mencionar a Shakespeare en nuestra obra maestra: si nos pasamos del límite, la frase se cortará, hubiera o no terminado. Esto es lo que vivimos todos y cada uno de los estudiantes de programación informática en las décadas de los ochenta y noventa del siglo pasado. En aquel momento era necesario romper las cadenas de más de ochenta caracteres en pedazos más pequeños para ajustarse a las limitaciones de los editores de texto. Esto era aún más evidente al escribir cálculos complejos en un respetado lenguaje de programación llamado Fortran. En ocasiones, una ecuación no se podía escribir en una sola línea, y era necesario partir esa larga expresión matemática en múltiples trozos. La inserción de molestos saltos de línea que interrumpían nuestros pensamientos parecía una complicación injustificada para una tarea –la programación informática– que ya era (y sigue siendo) intrínsecamente compleja. ¿Por qué 80 y no 166 o cualquier otro número de caracteres?

«La tendencia exponencial del *hardware* no se ha visto reflejada con innovaciones paralelas en *software*»

El origen de esta cuestión es anterior a las pantallas de tamaño fijo; radica en el tamaño de las tarjetas perforadas con las que se procesaba el censo estadounidense en la década de 1890. Estas tarjetas con ochenta caracteres por línea se introducían en las primeras computadoras comerciales de IBM, en los años cincuenta del siglo xx. Nuestros ordenadores personales heredaron el formato de columna de ochenta caracteres, que se convirtió en el estándar *de facto* para muchos de nosotros. Incluso hoy en día, en un momento en el que las pantallas de alta resolución son tan comunes, los editores de texto siguen siendo compatibles con reliquias de *hardware* que ya no volveremos a utilizar.

Como hemos visto en el ejemplo anterior, una casualidad histórica puede dejar una honda huella en la evolución tecnológica (Arthur, 1994). Algunas pueden ser inofensivas, pero otras provocan ineficiencias asociadas a tareas subóptimas. ¿Cómo podemos garantizar la idoneidad de una elección tecnológica? Una forma de hacerlo es comprobar la lista de buenas prácticas y recomendaciones. Un estándar tecnológico elimina los elementos innecesarios de las

prácticas de la ingeniería y conserva «lo que vale la pena». Estos estándares han ahorrado mucho esfuerzo asegurándose de que los «materiales, productos, procesos y servicios sean los adecuados para su propósito», tal como predica la Organización Internacional para la Estandarización (ISO). Podemos encontrar muchos ejemplos de estándares útiles en la industria y en ingeniería, donde las comunidades de expertos definen y actualizan sus consistentes recomendaciones. En lo esencial, la calidad de estos estándares depende de la capacidad de los expertos para decidir si las normas e innovaciones asociadas siguen siendo útiles o no. La estandarización aumenta la eficiencia tecnológica, pero también puede prolongar más de la cuenta la vida de las tecnologías existentes al inhibir cualquier inversión en innovación (Tassey, 1999). Alcanzar un equilibrio óptimo entre la eficiencia y la innovación es extremadamente difícil, y las predicciones sobre innovación tecnológica han sido hasta ahora poco fiables. En particular, las innovaciones complejas se enfrentan a más obstáculos que las simples a la hora de lograr el éxito comercial (véase Figura 1) (Schnaars y Wymbs, 2004).

Figura 1. Desde la aparición del teléfono, sus inventores pronosticaron la llegada de interacciones visuales a larga distancia. En 1924, Alexander Graham Bell dijo que llegaría el día en el que la persona al teléfono podría ver a la persona con la que estaba hablando. Sin embargo, el intento por parte de AT&T de comercializar el *picturephone* en los años sesenta (en la imagen) fue un fracaso comercial. AT&T invirtió tanto en esta tecnología que, si el éxito comercial dependiera únicamente de la convicción, el videófono se habría convertido en algo tan común como el teléfono hace muchos años.

EL CUELLO DE BOTELLA DEL *SOFTWARE*

En el último siglo hemos presenciado una espectacular aceleración en el rendimiento computacional, la capacidad de almacenamiento digital y las comunicaciones electrónicas a escala global. La famosa ley de Moore ha marcado la evolución de las tecnologías de la información. Esto es consecuencia de unos principios teóricos sólidos: los fundamentos conceptuales de la computación siguen siendo los mismos desde la publicación de los textos clásicos de Alan Turing. Por otro lado, la tendencia exponencial del *hardware* no se ha visto reflejada con innovaciones paralelas en *software*, que se siguen midiendo en escalas de tiempo humanas. Aunque tanto *hardware* como *software* han crecido en complejidad, su evolución presenta una importante asimetría (Valverde, 2016). El actual cuello de botella en tecnologías de la información no es el coste del *hardware*, sino la obtención del *software* necesario para hacerlo funcionar (Ensmenger, 2010). El *software* dicta la utilidad de las tecnologías de la información y su demanda ha creado un enorme problema económico. Muchos proyectos de *software* están plagados de errores, accidentes y decisiones idiosincráticas (Brooks, 1975). El fracaso recurrente de proyectos llevó al sector a definir enfoques fiables para asegurar el desarrollo de *software* de alta calidad (Charette, 2005).

«Muchos proyectos de *software* están plagados de errores, accidentes y decisiones idiosincráticas»

Desde la invención de la tecnología de computación en la década de 1950, la estandarización ha sido la aspiración de muchas asociaciones de usuarios y profesionales de la informática. Como en toda disciplina emergente, los profesionales estaban ansiosos por demostrar su utilidad social. En particular, los primeros pasos de los programadores en la estandarización se centraron en la interoperabilidad del *software* informático; es decir, en la posibilidad de intercambiar y reutilizar *software* entre diferentes ordenadores. Por ejemplo, el estándar del sistema operativo MS-DOS dio lugar a la adopción generalizada de los ordenadores personales (PC), que a su vez condujeron a muchas innovaciones posteriores, como la aparición de Internet y la World Wide Web. El crecimiento exponencial del mercado de los ordenadores y la proliferación de ordenadores incompatibles aumentó la competencia en el mundo del *software*. Se esperaba que la estandarización facilitara la aparición

de *software* no solo compatible sino también más económico. Para ello, los estándares se centraron principalmente en dos aspectos: escribir y mantener código (o «desarrollo de *software*») y las herramientas para apoyar a ese proceso (por ejemplo, los lenguajes de programación y los sets de componentes de *software* reutilizables). Pero si bien la interoperabilidad de los programas informáticos tuvo un gran éxito, la adopción tan escasa de estándares en desarrollo de *software* sugiere la presencia de límites que todavía no comprendemos lo suficiente.

DESARROLLO DE "SOFTWARE" IMPREDECIBLE

Las iniciativas del Departamento de Defensa de los Estados Unidos son un buen ejemplo de los obstáculos a los que se enfrentaba el desarrollo de estándares de *software*. Desde la década de los setenta hasta los ochenta, el Departamento trató de hacer que sus contratas cumplieran unos estándares en este sentido (McDonald, 2010). Su objetivo era reducir los altos costes del desarrollo de *software*. Detrás de ello (y de otras iniciativas paralelas) estaba la aspiración (nunca realizada) de reemplazar el componente humano con un proceso de desarrollo de *software* completamente automatizado y a prueba de errores. En 1978 el Departamento de Defensa publicó una serie de normas de desarrollo que todas las contratas debían seguir. En primer lugar, el *software* se debería desarrollar siguiendo un proceso de diseño descendente, desde la definición global del sistema a sus componentes funcionales (véase Figura 2). Por otra parte, para mejorar la legibilidad del código de programación (reduciendo en consecuencia las posibilidades de error), había un tamaño máximo para los componentes individuales de *software* y se prohibieron instrucciones «perjudiciales» (como la orden «GO TO», que rompe la secuencia lógica de las operaciones de *software*). Y, por último, todo el código programado debería escribirse utilizando una serie aprobada de lenguajes de programación de alto nivel. Sorprendentemente, el Departamento se enfrentó a una dura oposición cuando intentó que las contratas incorporaran estas normas. Aunque el estándar reflejaba las convenciones sobre cómo escribir buen código, muchos programadores pensaban que era innecesario y que había quedado obsoleto, por lo que suponía una carga innecesaria que limitaba su libertad. Debido a las crecientes críticas y a la presión social, en los años noventa se abandonó cualquier intento de imponer estándares de *software* por contrato.

Un obstáculo fundamental era la suposición poco realista por parte del modelo descendente de que se puede planificar la trayectoria de los proyectos de *software*. El código real (como muchos otros proyectos de ingeniería complejos) suele implicar costosas correcciones en etapas posteriores del proyecto; es decir, algunas de las elecciones iniciales de diseño se convierten en accidentes históricos (McDonald, 2010). La experiencia con los proyectos de *software* sugiere lo difícil que resulta cumplir los requisitos funcionales; esto es, determinar el conjunto de tareas que el programa debe realizar. En particular, cualquier requisito no especificado en el diseño inicial se traduce en costosas modificaciones en etapas posteriores del proyecto (Boehm, 1976). Por ejemplo, los usuarios tienen intuiciones sobre cómo deberían funcionar los sistemas operativos (como Microsoft Windows), pero les cuesta mucho más describir funciones de *software* que todavía no han utilizado.

Un enfoque más pragmático conceptualiza el *software* como un proceso incremental e iterativo, no muy distinto a la evolución natural. Con ello se espera minimizar la cantidad de decisiones de diseño redundantes. Por ejemplo, en el modelo basado en prototipos, los usuarios y programadores cooperan activamente construyendo un sistema de *software* grande y estable. Aquí, los cambios de los programadores son una fuente de variación natural en el proyecto. Los usuarios actúan como el entorno del prototipo de *software* seleccionando características según sus necesidades. Repitiendo este proceso iterativo, los usuarios y los programadores coevolucionan en un sistema que se ajusta a las especificaciones. El prototipado de *software* acepta que, en un entorno cambiante, la adaptabilidad rápida y poco refinada es preferible a una planificación descendente inadecuada.

«Un enfoque más pragmático conceptualiza el *software* como un proceso incremental e iterativo, no muy distinto a la evolución natural»

UNA TORRE DE BABEL ELECTRÓNICA

Las herramientas utilizadas en el desarrollo de *software* tampoco han alcanzado la uniformidad de otras disciplinas tecnológicas como la ingeniería eléctrica. La diversidad tecnológica se encuentra en los repositorios públicos de *software* de código abierto, donde no existe un marco único de desarrollo, sino una multitud de ellos entre los que podemos encontrar muchas maneras de resolver el mismo problema en un programa para diferentes plataformas (por ejemplo, Windows, Mac OSX o Linux) escrito en lenguajes de programación incompatibles (como C++, Python o Java) utilizando una mezcla de librerías registradas de *software* (OpenGL o DirectX). Estas so-

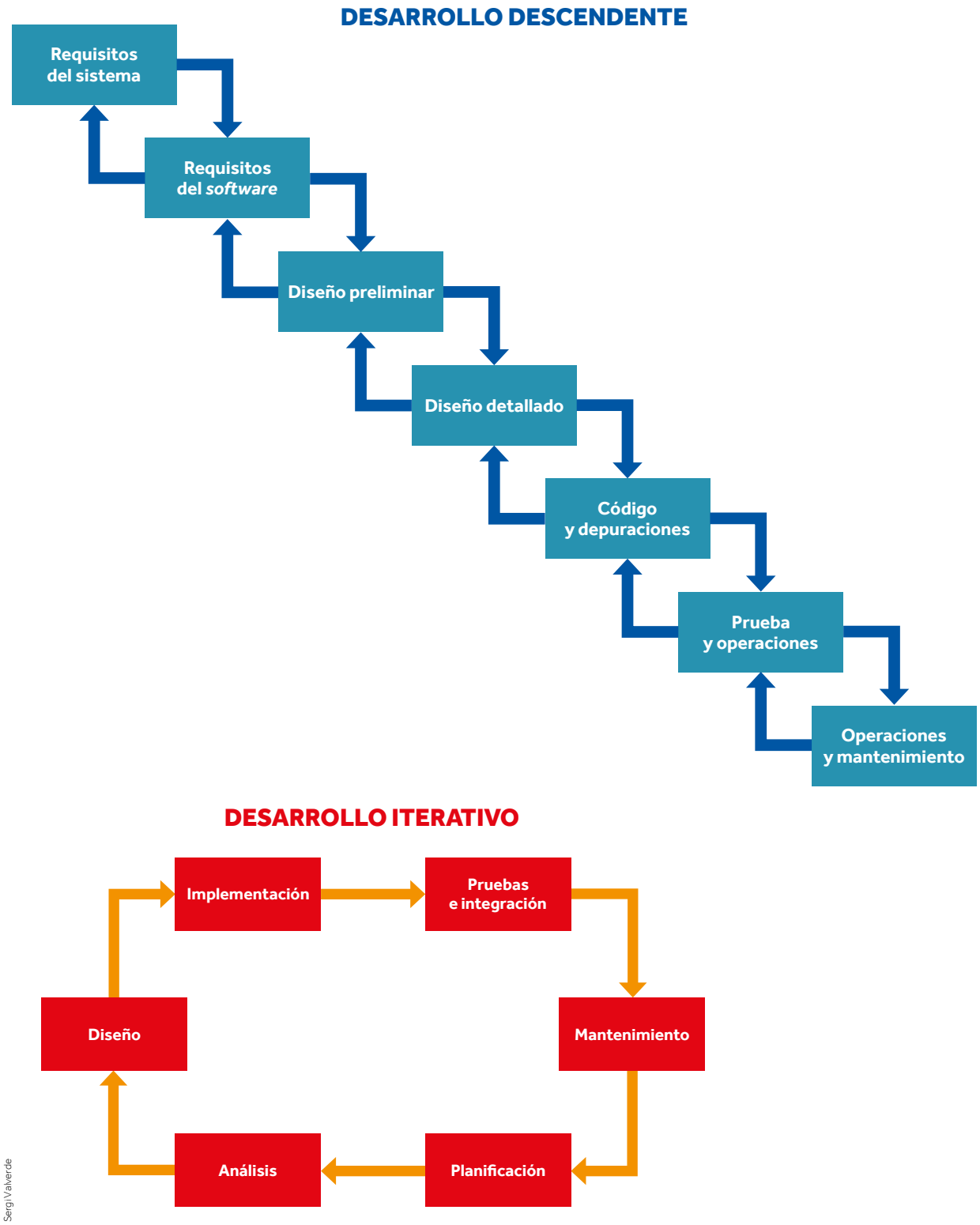


Figura 2. Los estándares dividen el desarrollo de *software* en diferentes etapas, como el diseño, la construcción, las pruebas y el mantenimiento. En los años sesenta, el Departamento de Defensa de los Estados Unidos intentó imponer a sus contratas un modelo secuencial (o descendente) de desarrollo de *software*, con escaso éxito. El desarrollo iterativo de *software* es más flexible y puede reducir los malentendidos entre los usuarios y los programadores.

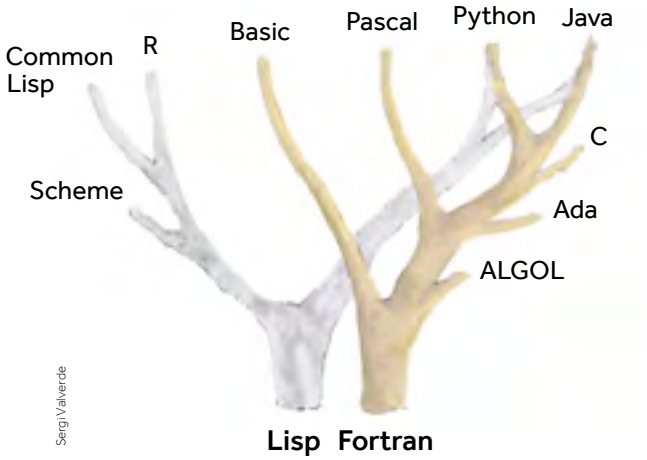


Figura 3. Este diagrama ilustra la evolución de los lenguajes de programación en forma de árbol. El objetivo de los primeros lenguajes en los sesenta era principalmente la industria y los negocios. Los lenguajes de esa época, como ALGOL, fueron diseñados por comités privados de expertos. Sin embargo, la adopción general de las tecnologías de la información aceleró la diversificación de los lenguajes de programación. Algunos de los más populares, como C o Python, han sido desarrollados por comunidades distribuidas de programadores. Diferentes ramas del árbol influyen en la evolución continua de los lenguajes de programación.

luciones se basan en las mismas ideas y conceptos, pero sus tecnologías de base son incompatibles, lo que limita la reutilización. En este contexto, integrar *software* con éxito sigue dependiendo de la buena voluntad y la colaboración desinteresada entre programadores. La situación de los lenguajes de programación es particularmente reveladora. En 1936, Alan Turing demostró que no existen barreras teóricas para la unificación de los lenguajes de programación. Es decir, que existe una computadora universal capaz de realizar cualquier tarea. Sin embargo, la realidad es que tenemos miles de lenguajes de programación distintos a nuestra disposición (véase Figura 3). ¿Por qué no podemos simplemente crear un lenguaje universal con muchas funcionalidades?

Ha habido múltiples intentos de estandarizar los lenguajes de programación, pero ninguno de ellos se ha aceptado universalmente. Por ejemplo, el objetivo del lenguaje de programación Ada era reemplazar a la miríada de lenguajes utilizados en los proyectos de *software* del Departamento de Defensa (en la década de los setenta, se habían utilizado más de 450 lenguajes diferentes en proyectos del ejército). A diferencia de muchos lenguajes diseñados por consorcios privados (como COBOL), la creación de Ada fue el resultado de un concurso internacional sujeto a revisión externa (a cargo de expertos académicos en lenguajes de programación). Uno de los objetivos de Ada era evitar los errores humanos en el desarrollo de *software*, lo cual requiere ser estrictos con la seguridad y la concurrencia de una forma que rara vez se da en otros lenguajes. A pesar de estos beneficios, Ada nunca alcanzó la popularidad de otros lenguajes menos robustos como C++, que apareció en 1985. Unos veinte años después, ISO estandarizó por primera vez el lenguaje C++ (que se convirtió en un estándar *de facto*). Esto muestra, de nuevo, que el éxito no se puede predecir, sin importar el esfuerzo realizado en el diseño inicial. Muchos factores influyen en el éxito de los lenguajes de programación, incluyendo la popularidad, la complejidad y la economía.

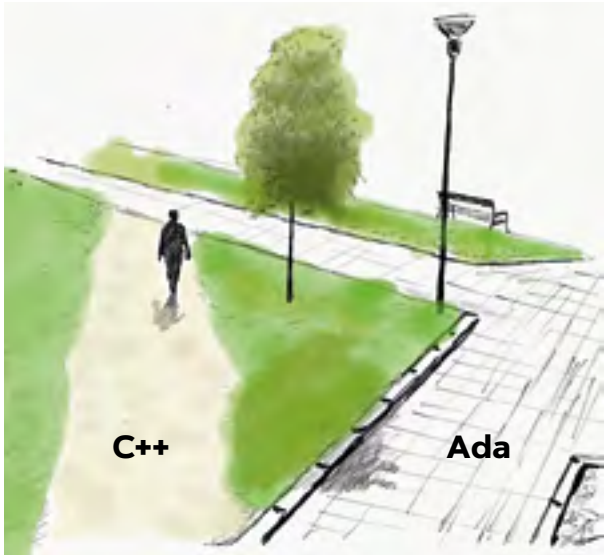
Desde una perspectiva histórica, parece que el camino desde el lenguaje C a C++ fue más fácil que la adopción de un estándar de alta calidad (pero relativamente desconocido) (Figura 4). El compromiso con la opción «menor» no se tradujo en un bloqueo del mercado ni en la falta de innovación porque los lenguajes de programación evolucionan continuamente y se influyen mutuamente. En algún momento la gran comunidad de usuarios de C++ se benefició de las innovaciones propuestas por Ada sin perder la compatibilidad con la tecnología existente. Parece que los programadores prefieren vivir con un *software* imperfecto a reconstruirlo todo desde cero.

«Ha habido múltiples intentos de estandarizar los lenguajes de programación, pero ninguno de ellos se ha aceptado universalmente»

COMPLEJIDAD EMERGENTE

En los procesos complejos de ingeniería, como los que tienen que ver con la programación, los ingenieros no siempre pueden decidir el mejor curso de acción. Más bien les «impulsa» la complejidad emergente de sus creaciones. Ni la evolución de la tecnología ni la de la biología pueden evitar los accidentes y el bricolaje cuando la complejidad aumenta demasiado.

A principios de los noventa, la heterogénea diversidad de *hardware*, sistemas operativos y lenguajes de programación suponía un obstáculo para la interoperabilidad del *software*. Los ingenieros y los gestores vieron en la reutilización de componentes la solución natural a este problema. En lugar de construir un programa desde cero, se podría utilizar un repositorio de bloques (o componentes) que incluyera las funciones de *software* más comunes. Este enfoque requiere definir un estándar de interoperabilidad de *software* como CORBA (*Common Object Request Broker Architecture*; en español, "Arquitectura Intermediaria de Petición de Componentes Comunes"). En CORBA, los componentes de *software* escritos en diferentes lenguajes, como Java y C, también pueden inter-



Sergi Valverde

Figura 4. El camino hacia la adopción de tecnologías depende de muchos factores, incluyendo el coste. Y el hecho de que algunos lenguajes de programación estén diseñados cuidadosamente por comités de expertos no significa automáticamente que se vayan a adoptar de forma generalizada. El lenguaje estándar Ada (derecha) disfruta de un reconocimiento como lenguaje de calidad (con muchas características únicas que no suelen estar presentes en otros lenguajes de programación). Sin embargo, la popularidad del sistema operativo Unix catalizó la adopción del lenguaje C, así como su sucesor orientado a componentes, C++ (izquierda). Estos dos lenguajes de programación –ambos estandarizados mediante normas ISO– han acabado dominando el mercado, mientras que Ada se ha mantenido como una solución especializada.

cambiar información adhiriéndose a un protocolo de *software* común. Se definió este estándar como «la siguiente generación de tecnología para el comercio electrónico», y ganó mucha popularidad al principio. Sin embargo, las deficiencias técnicas y los errores de diseño dificultaron que se consolidara en el mercado, y finalmente CORBA quedó desplazado por tecnologías web como XML (Henning, 2008).

El fracaso de CORBA se ha asociado principalmente a cuestiones de calidad. En un examen más profundo, ejemplifica la difícil tarea de definir una interfaz estándar entre componentes de *software*. El desarrollo de *software* basado en componentes es muy similar a la idea de las piezas de Lego. Los bloques de Lego son interoperables gracias a un sistema de ensamblaje patentado (Figura 5). Esto es, podemos conectar cualquier par de bloques, independientemente de su forma, color o función. Pero esto no ocurre con la programación. Existen muchos ejemplos catastróficos de interacciones no deseadas entre componentes de *software*. Para evitarlo, nos hemos visto obligados a probar (y depurar) cualquier interacción, lo cual requiere mucho tiempo y esfuerzo. Es inevitable e impone un límite estricto al desarrollo escalable de *software*.

«La incertidumbre afecta a la evolución de muchas tecnologías, pero es incluso más grave en el caso del *software* debido a la ausencia de un entorno físico»

EVOLUCIÓN EN EL FUTURO

En un espacio de tiempo muy corto, hemos pasado de una visión de ciencia ficción con computadoras todopoderosas a un producto básico que está al alcance de todo el mundo. Esta amplia adopción de la tecnología de la información convirtió el *software* en un componente clave de nuestra sociedad. Tenemos una necesidad apremiante de conseguir formas más fiables y económicas de desarrollar *software* a un ritmo más rápido. Se ha propuesto la estandarización del *software* como solución a este problema, asimilándola a los miles de estándares básicos necesarios para el funcionamiento de nuestra sociedad.

¿Podemos definir normas universales para controlar el desarrollo de *software*? La historia reciente demuestra que el desarrollo de *software* fiable sigue siendo un objetivo difícil de alcanzar. Uno de los principales problemas es la incertidumbre presente en cada etapa del proceso de desarrollo de *software*. Al diseñar un sistema de *software*, existen muchas opciones diferentes. Decidir cuál es la mejor en cada etapa no es en absoluto fácil. No podemos estar seguros de las funcionalidades de la tecnología a largo plazo porque el éxito se basa en cambios impredecibles del entorno. La incertidumbre afecta a la evolución de muchas tecnologías (Petroski, 1992), pero es incluso más grave en el caso del *software* debido a la ausencia de un entorno físico. El *software* no se descompone ni se estropea como otras tecnologías. Conocer los principios que subyacen a los sistemas físicos permitió a la ingeniería y la industria realizar avances espectaculares. Pero la situación es muy diferente en este caso, dado que el enfoque científico del desarrollo de *software* está muy retrasado en comparación con la práctica. Es necesario que el desarrollo de *software* madure, y eso depende de la disponibilidad de los resultados científicos (Glass, 2009).

Dominio público

Los obstáculos para la estandarización de *software* sugieren la existencia de un problema recurrente: el ingenio humano no se puede reemplazar por piezas estandarizadas. Por el momento, el cerebro es un componente esencial para traducir los requisitos humanos al lenguaje de programación. Todavía no entendemos del todo la forma en la que los humanos programan los ordenadores. Desarrollar programas útiles y fiables requiere ingenio y una experiencia considerable. Los programadores inexpertos no pueden confiar

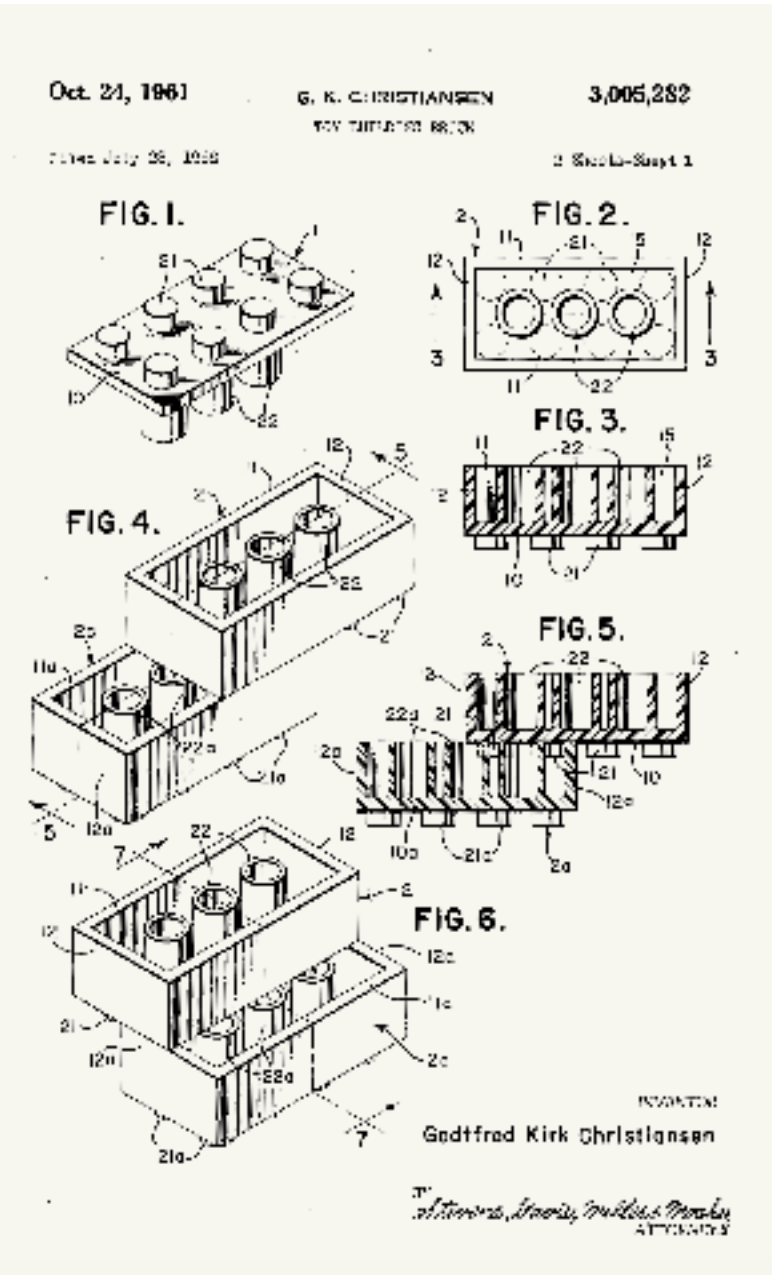


Figura 5. En 1961, la patente estadounidense 3005282A propuso el diseño de un juguete con «bloques de construcción», conocidos popularmente como *legos*. Este documento describe un ingenioso mecanismo de ensamblaje que permite unir muchas estructuras diferentes. En el desarrollo de *software* nunca se ha conseguido crear una interfaz universal como esta.

«Todavía no entendemos del todo la forma en la que los humanos programan los ordenadores»

en su intuición para evaluar si un *software* es demasiado grande o demasiado pequeño, simple o complejo, porque es invisible. Solo después de pasar mucho tiempo programando (y sobre todo depurando), podemos comenzar a comprender la enorme complejidad del *software*. Más allá de nuestra inteligencia y nuestras habilidades, solo podemos desarrollar tecnología compleja gracias a todo el conocimiento generado por nuestra sociedad con el paso de los años (Basalla, 1988; Messoudi, 2011). La evolución de tecnologías complejas como el lenguaje de programación C++ ha sido el resultado de la información acumulada por mucha gente a lo largo de muchos años en comunidades de código abierto. Hay quien defiende la idea de que el desarrollo de *software* pronto quedará obsoleto, y que la inteligencia artificial reemplazará a comunidades enteras de programadores humanos. Esto parece poco probable sin una comprensión completa de la forma en que los humanos programan las computadoras. En cualquier caso, una cosa parece cierta: el *software* del futuro no se diseñará, sino que evolucionará.

REFERENCIAS

Arthur, W. B. (1994). *Increasing returns and path dependence in the economy*. Ann Arbor: Michigan University Press. doi: 10.3998/mpub.10029

Basalla, G. (1988). *The evolution of technology*. Cambridge: Cambridge University Press. doi: 10.1017/CBO9781107049864

Boehm, B. W. (1976). Software engineering. *IEEE Transactions on Computers*, *25*(12), 1226–1241. doi: 10.1109/TC.1976.1674590

Brooks, F. (1975). *The mythical man-month: Essays on software engineering*. Boston: Addison-Wesley.

Charette, R. N. (2005, 2 de septiembre). Why software fails. *IEEE Spectrum*. Consultado en https://spectrum.ieee.org/computing/software/why-software-fails

Ensmenger, N. L. (2010). *The computer boys take over. Computers, programmers, and the politics of technical expertise*. Cambridge: The MIT Press.

Glass, R. L. (2009). Doubt and software standards. *IEEE Software*, *26*(5), 104. doi: 10.1109/MS.2009.126

Henning, M. (2008). The rise and fall of CORBA. *Communications of the ACM*, *51*(8), 52–57. doi: 10.1145/1378704.1378718

McDonald, C. (2010). From art form to engineering discipline? A history of US military software development standards, 1974-1998. *IEEE Annals of the History of Computing*, *32*(4), 32–47. doi: 10.1109/MAHC.2009.58

Messoudi, A. (2011). *Cultural evolution: How Darwinian theory can explain human culture and synthesize the social sciences*. Chicago: University of Chicago Press.

Petroski, H. (1992). *To engineer is human: The role of failure in successful design*. Nueva York: Vintage Books.

Schnaars, S., & Wymbs, C. (2004). On the persistence of lackluster demand: The history of the video telephone. *Technological Forecasting and Social Change*, *71*(3), 197–216. doi: 10.1016/S0040-1625(02)00410-9

Tassey, G. (1999). Standardization in technology-based markets. *Research Policy, 29*(4-5), 587–602. doi: 10.1016/S0048-7333(99)00091-8

Valverde, S. (2016). Major transitions in information technology. *Philosophical Transactions of the Royal Society B*, *371*(1701), 20150450. doi: 10.1098/rstb.2015.0450

SERGI VALVERDE. Experto en sistemas complejos, doctor en Física Aplicada e investigador del Instituto de Biología Evolutiva (UPF-CSIC) de Barcelona (España), donde lidera el Laboratorio de Evolución de la Tecnología (ETL). Su grupo de investigación es pionero en el estudio de las grandes transiciones evolutivas mediante la comparativa de sistemas biológicos y artificiales. Su investigación multidisciplinar integra diversas áreas de conocimiento, des de la teoría de redes hasta la ecología teórica y la simulación computacional de los elementos evolutivos. Es miembro de la junta de la Red Catalana para el Estudio de los Sistemas Complejos (complexitat.cat). ✉ sergi.valverde@ibe.upf-csic.es