Scientific American, noviembre, 1994 LA CRISIS CRÓNICA DE LA PROGRAMACIÓN (1)

W. Wayt Gibbs

A pesar de 50 años de progresos, a la industria del software le faltan años -quizás décadas- para convertirse en la disciplina de ingeniería madura que la sociedad de la información requiere.

El nuevo aeropuerto internacional de Denver, en los EE.UU., iba a ser la maravilla de la ingeniería moderna, el orgullo de la región. Su enorme tamaño, como diez veces el londinense Heathrow, permite el aterrizaje simultáneo de tres jets, incluso con mal tiempo. Por colosales que sean sus dimensiones, aun es más impresionante el sistema subterráneo de traslado de equipajes con que cuenta. 4.000 “telecarros” independientes, lanzados a toda velocidad por 33 kilómetros de vía férrea, transportan y distribuyen los equipajes entre los mostradores, las puertas de acceso y las áreas para retirar de 20 líneas aérea distintas. Un sistema nervioso central de 100 computadores conectados entre sí y a 5.000 ojos eléctricos, a 400 receptores de radio y a 56 lectores de código de barras se encarga de orquestar la llegada segura y a tiempo de cada maleta y de cada bolsa de esquí (2). Eso, al menos, era lo previsto. Este Gulliver ha estado nueve meses prisionero de los liliputienses, léase, de los errores del software que controla el sistema automático de equipajes. El aeropuerto debía “despegar” el día de Halloween (3) del año pasado, pero su gran inauguración hubo que posponerse hasta diciembre, para permitir que los técnicos de la empresa BAE Automated Systems exorcizasen de espíritus malignos su sistema, que había costado 193 millones de dólares. De diciembre hubo que pasar a marzo. Marzo trocóse en mayo. En junio del presente año, las acciones del aeropuerto se cotizaban al precio del papel de embalar y el presupuesto de sus promotores sufría una hemorragia de tinta roja que alcanzaba 1,1 millones de dólares diarios en intereses y gastos operativos, viéndose obligados a admitir que no podían predecir en qué momento el sistema de equipajes adquiriría la estabilidad necesaria para poder abrir el aeropuerto.

Para los desarrolladores de software más veteranos, lo único inusual del desastre de Denver es su notoriedad. Hay estudios que señalan que por cada seis nuevos sistemas de software de gran tamaño que entran en servicio, otros dos quedan cancelados. Los proyectos de desarrollo de software de tamaño medio suelen consumir vez y media el tiempo previsto, situación que empeora en los grandes. Y alrededor de tres cuartas partes de todos los sistemas de gran tamaño son “fracasos operativos”, que no funcionan como se quería o no se utilizan para nada. El arte de la programación ha requerido 50 años de incesante perfeccionamiento para alcanzar este estadio. Al cumplir los 25, las dificultades que planteaba la construcción de software grande cobraron tanta importancia que en el otoño (4) de 1968 el Comité de Ciencias de la OTAN convocó a un grupo de unas 50 personas, formado por programadores de primera

categoría, científicos de la computación y empresarios, para que trazasen un rumbo que permitiera salir de lo que se ha dado en denominar “la crisis del software”.

Aunque no fueron capaces de elaborar una ruta que guiase la nave hacia tierra firme, sí acuñaron un nombre para esa lejana meta: ingeniería de software, ahora definida formalmente como “la aplicación de métodos sistemáticos, disciplinados y cuantificables al desarrollo, funcionamiento y mantenimiento del software”. Otro cuarto de siglo después, la ingeniería de software sigue siendo una aspiración. La mayor parte del código de computadoras se confecciona a mano por artesanos que utilizan lenguajes de programación bastante burdos técnicas que ni se miden ni pueden repetirse con consistencia. Para el profesor universitario Brad J. Cox “Es algo así como la fabricación de arcabuces antes de Eli Whitney” (5). Dice Brad J. Cox, un profesor de la Universidad George Mason, “Antes de la revolución industrial no existían métodos especializados de manufactura de productos, lo que entrañaba un máximo de destreza artesanal y un mínimo de intercambiabilidad. Si queremos vencer esta crisis del software ha de acabarse con el secretismo y con los métodos preindustriales, en los que cada programador ha de construirlo todo desde cero”. Pero hay esperanzas. La intuición va cediendo paso lentamente al análisis y los programadores comienzan a utilizar medidas cuantitativas de la calidad del software que producen para mejorar la forma en que los producen. Los fundamentos matemáticos de la programación se están consolidando merced a la investigación de métodos para la expresión algebraica de los diseños de programas, lo que ayuda a evitar la comisión de errores graves. Los académicos de ciencias de la computación empiezan a tratar de remediar su fracaso en la generación de profesionales de software competentes. Y, lo que quizás es más importante, muchas personas de la industria centran su atención hacia la invención de las tecnológicas y estructuras mercantiles necesarias para soportar las partes del software que sean intercambiables y reutilizables. “Lamentablemente, la industria no aplica uniformemente lo que es reconocido como mejor práctica”, lamenta Larry E. Druffel, director de Software Engineering Institute de la Universidad Carnegie Mellon. De hecho, una innovación que genera la investigación requiere 18 años en abrirse paso hasta el repertorio de técnicas standard de programación. Es posible que la combinación de esfuerzos de la universidad, las empresas y la administración pública logren elevar el desarrollo de software hasta el nivel de una disciplina ingenieril propia de la era industrial en este decenio. Si no se consigue, la alocada carrera de la sociedad hacia la era de la información se tornará, en el mejor de los casos, espasmódica e impredecible.

Arenas movedizas

“A lo largo de los próximos años presenciaremos cambios enormes [en el uso de los computadores] que en comparación harán que la revolución inicial causada por el ordenador personal resulte insignificante”, concluían el pasado mes de abril veintidós especialistas en el desarrollo de software, procedentes de la universidad, de las empresas y de los laboratorios de investigación. Estos expertos se reunieron en Hedson Parkn un lugar cerca de Londres para conmemorar la conferencia de la OTAN y para

considerar las direcciones del software en el futuro. “En 1968 sabíamos lo que queríamos construir, pero no podíamos hacerlo”, rememora Cliff Jones, profesor de la universidad de Manchester. “Hoy pisamos arenas movedizas.” Las bases en que se asientan las prácticas tradicionales de programación son erosionadas rápidamente conforme los ingenieros de hardware van lanzando equipos cada vez más rápidos, más pequeños y más baratos. Muchos de los supuestos fundamentales de los programadores (por ejemplo, su aceptación de que sus productos siempre tendrán defectos) deben de cambiar en consecuencia. Como indica la profesora Mary S. Shaw, de Carnegie Mellon, “cuando los interruptores de la luz incorporen computadoras es preciso que el software funcione la primera vez, porque no habrá posibilidades de revisarla”. “La cantidad de código instalada en la mayoría de los productos de consumo se está duplicando cada dos años”, según Remi H. Bourgonjon, director de tecnología de software del Philips Research Laboratory, en Eindhoven. Un televisor actual puede incorporar hasta 500 Kb de software y una afeitadora eléctrica, 2 Kb. Las transmisiones de los nuevos automóviles de General Motors ejecutan de 30.000 líneas de código de computadora. Lograr que el software funcione bien la primera vez les resulta difícil incluso a quienes procurar conseguirlo. El Departamento de Defensa de los Estados Unidos (DoD) aplica normas de ensayo muy estrictas- y costosas- para garantizar la confiabilidad de software del que depende una misión. Esos standards fueron las utilizados en la certificación de Clementine, un satélite que el DoD junto con la NASA, querían situar en órbita lunar la primavera pasada. Un aspecto principal de la misión consistía en ensayar software de selección de blanco, aptos para ser utilizados algún día en un sistema defensivo antimisiles, instalado en el espacio. Pero cuando se hizo girar al satélite para que situara a la Luna en su punto de mira, un bug del programa provocó que, en lugar de lo solicitado, la nave mantuviera ininterrumpidamente encendidos sus motores de maniobra durante 11 minutos. El satélite, agotado su combustible y girando alocadamente sobre sí, mismo no pudo efectuar el encuentro previsto con el asteroide Geographos. La detección de errores en sistemas de tiempo real, como el Clementine, resulta de una dificultad diabólica porque, lo mismo que pasa con ese ruidito sospechoso del motor de nuestro coche, sólo suelen manifestarse en circunstancias muy determinadas [véase “Los riesgos de la programación”, por Bev Littlewood y Lorenzo Strigini; INVESTIGACIÓN Y CIENCIA, enero de 1993]. “No está claro que los métodos utilizados actualmente para la producción de software en los que la seguridad tiene importancia decisiva -caso de los reactores nucleares o de los automóviles- evolucionen y se amplíen en la medida necesaria para satisfacer nuestras expectativas futuras”, advertía Gilles Kahn, director científico del laboratorio de investigación INRIA de Francia, en la reunión de Hedson Park. “Me parece, por el contrario, que en el caso de estos sistemas estamos en un punto de quiebre.” La software sufre también las presiones de la inexorablemente creciente demanda de “sistemas distribuidos”, esto es, programas que operen conjuntamente en muchos computadores integrados en una red. Las empresas están volcando recursos a manos llenas en sistemas de información distribuidos, que confían en poder utilizar como armas estratégicas. La inconstancia del desarrollo del software pueden hacer de tales proyectos una ruleta rusa. Son muchas las empresas seducidas por metas de apariencia sencilla. Algunas tratan de reencarnar

en forma distribuida software obsoleto basado en mainframe. Otros desean conectar entre sí los sistemas de que ya disponen, o conectarlos a sistemas nuevos mediante los que se pueda compartir datos y disponer de una interfaz más amistosa para el usuario. En la jerga técnica, este tipo de interconexión de programas suele denominarse "integración de sistemas". Pero Brian Randell un científico de la computación de la Universidad de Newcastle en Tyne, sugiere "que hay una palabra más adecuada que integración, del slang de la RAF: nominalmente 'to graunch', que significa 'hacer que algo encaje por aplicación de una fuerza excesiva'." El asunto es riesgoso, pues aunque el software pueda parecer cosa maleable, la mayoría de los programas son en realidad complicadas redes de estructuras lógicas muy frágiles, a través de los cuales solamente pueden pasar datos de naturaleza específica. Al igual que los arcabuces hachos a mano, diversos programas pueden realizar funciones semejantes y ser, sin embargo, de diseño exclusivo. De aquí que sean difíciles de modificar y de reparar, y de aquí también que las tentativas de embutirlos unos en otros acaben a menudo de mala manera. Por poner un ejemplo. El Departamento de Vehículos Automotor de California decidió en 1987 hacerles la vida más fácil a los conductores, para lo que resolvió fundir en uno los sistemas de permiso de conducción para el personal y de circulación de los vehículos. Tarea sencilla, en apariencia. Los responsables confiaban en que en 1993 tendrían en servicio quioscos donde se efectuasen cómodamente las renovaciones en una sola parada. En vez de ello, lo que vieron era cómo se multiplicaba por 6,5 el costo previsto y se posponía hasta 1998 la fecha de inauguración. El pasado diciembre cerraron el grifo y perdieron la inversión efectuada en los siete años: 44,3 millones de dólares. Hay veces que nada es tan catastrófico como el éxito. En el decenio de 1970, la compañía American Airlines construyó el SABRE, un sistema de reserva de pasajes de autentico virtuosismo, que costo 2000 millones de dólares y llegó a formar parte de la infraestructura de la industria de viajes. "SABRE constituyó el ejemplo más luminoso de sistema estratégico de información, porque hizo que American Airlines se convirtiera en la mayor línea aérea del mundo", recuerda Bill Curtis, consultor del Software Engineering Institute. Decidida a valerse del software con igual efectividad en este decenio, American Airlines trató de acoplar sus sistema de reserva de pasajes con los de reserva de hotel y de alquiler de automóviles de Marriott, Hilton y Budget. El proyecto colapsó en 1992, en medio de un montón de pleitos. "Fue un fracaso aplastante", dice Curtis. "American Airlines tuvo que pasar a pérdidas 165 millones de dólares por esta causa".

Mal puede decirse que la compañía aérea se encuentre sola en sus sufrimientos. El Consulting Gropu de IBM hizo públicos en junio los resultados de un estudio efectuado en 24 compañías líderes que habían desarrollado grandes sistemas distribuidos. Las cifras eras inquietantes: el 55 por ciento de los proyectos costó más de lo esperado, el 68 por ciento incumplió los plazos previstos y el 88 por ciento tuvo que rediseñarse a fondo. El informe no mencionaba, empero, un dato estadístico de la mayor importancia: la confiabilidad del funcionamiento de los programas terminados. Los sistemas informáticos suelen estrellarse o quedarse colgados debido a su incapacidad de enfrentar lo inesperado. Las redes amplían el problema, "Los sistemas distribuídos pueden consistir en un conjunto muy grande de puntos individuales interconectados, en

lo que cabe que se produzcan fallas y muchos de los cuales no están identificados de antemano", explica Randell. "La complejidad y la fragilidad de estos sistemas plantean un problema de primera magnitud." El desafío de la complejidad no sólo es grande sino que va en aumento. Lo que recibe uno por cada unidad monetaria invertida en ordenadores se duplica cada 18 meses, más o menos. Lo que origina, entre otras cosas, "un crecimiento de un orden de magnitud en el tamaño de los sistemas por decenio y, en ciertos sectores, cada cinco años", según Curtis. Para no verse superados por semejante demanda, los programadores tendrán que cambiar la forma en que trabajan. "No se pueden construir rascacielos con carpinteros", bromea.

Auxilio!! Socorro!

Cuando un sistema se vuelve tan complejo que ningún gerente lo comprende individualmente por completo, los procesos tradicionales de desarrollo se vienen abajo. La Administración Federal de Aviación estadounidense (FAA, Federal Aviation Administration) ha tenido que afrontar este problema a lo largo de su intento- que ya dura diez años- de sustituir el sistema de control de tráfico aéreo del país, cada vez más anticuado. En su sustituto, denominado Sistema Avanzado de Automación (AAS, Advanced Automation Systems), se combinan todos los problemas de la computación de nuestro decenio. El programa, cuyo tamaño supera el millón de líneas, está distribuido en cientos de ordenadores y empotrado en aparatos nuevos y muy complejos. Todo el sistema debe responder instantáneamente a acontecimientos impredecibles durante las veinticuatro horas del día. Incluso un ligero bug puede constituir una amenaza para la seguridad pública. Para llevar a la práctica sus sueño tecnológico, la FAA seleccionó a Federal Systems Company, de IBM, entidad de muy buena reputación y líder en el desarrollo de software, posteriormente adquirida por Loral. Los responsables de la FAA esperaban (aunque no lo exigieron) que IBM aplicase las últimas técnicas disponibles para estimar el costo y la duración del proyecto, al igual que dieron por supuesto que revisaría cuidadosamente requerimientos y el diseño del sistema, para detectar los errores cuanto antes, cuando es posible corregirlos en horas y no en días. Y aceptaron - conservadoramente - que tendrían que pagar unos 500 dólares por línea de programa producida, unas cinco veces más que el costo medio normal para procesos de desarrollo bien gestionados.

Según un informe sobre el proyecto AAS, publicado en mayo de este año por el Centro de Análisis Naval estadounidense, "las estimaciones de costos y el seguimiento del proceso de desarrollo" hechos por IBM "se efectuaron con datos inadecuados y se realizaron de forma inconsistente, y fueron rutinariamente ignoradas por los gerentes del proyecto. El resultado es que la FAA ha estado pagando la programación para el sistema AAS a razón de 700 a 900 dólares por línea código. Una de las razones de precio tan exorbitante es que, "por término medio, cada una de dichas líneas ha de volver a escribirse", según se lamentaba un documento de la FAA. Alarmado por la estratosférica elevación de costos y porque los ensayos efectuados sobre el sistema semiterminado mostraban que no era confiable, David R. Hinson, administrador de la FAA, decidió en

junio pasado cancelar dos de las cuatro grandes partes del proyecto y reducir la escala de una tercera. Los 144 millones de dólares dilapidados en estos programas fallidos son sólo una gota de agua frente a los 1400 millones invertidos en la pieza cuarta y central: el nuevo software de las estaciones de trabajo de los controladores de tráfico aéreo. También este proyecto va camino de irse a pique. El programa lleva cinco años de retraso, ha engullido más de 1000 millones de dólares sobre lo presupuestado y está lleno de bugs. Expertos informáticos de la Carnegie Mellon y del Instituto de Tecnología de Massachusetts (MIT, Massachusetts Institute of Technology) están tratando de depurarlo y de averiguar si todavía es posible salvarlo o si debe cancelarse sin más. Los desastres serán una parte crecientemente común y disruptiva del desarrollo de software, a menos que la programación adopte muchas de las características de las ramas de la ingeniería, que están firmemente entroncadas en la ciencia y en las matemáticas (ver el cuadro "Progreso hacia el profesionalismo"). Por fortuna, esa tendencia ha comenzado ya. A lo largo de la última década, empresas líderes de la industria han comprendido bastante sobre cómo medir cuantitativa y consistentemente, el caos de sus procesos de desarrollo, la densidad de errores de sus productos y el estancamiento de la productividad de sus programadores. Los investigadores están dando el paso siguiente: encontrar soluciones prácticas y reproducibles para estos problemas.

Los frutos del proceso

Así, por ejemplo, el SEI, un vivero de ideas sobre el tema, financiado por los militares, reveló en 1991 su "modelo de capacidad y madurez" (CMM, Capability Maturity Model). "El CMM proporciona una visión de la excelencia de la ingeniería de software y de su gestión", dice David Zubrov, quien dirige un proyecto sobre métodos empíricos en el Instituto. Al menos el CMM, ha persuadido a muchos programadores de concentrarse en la medición del proceso por el que producen software, un requisito previo en cualquier disciplina de ingeniería industrial. Sirviéndose de entrevistas, de cuestionarios y del CMM como sistema de comparación, puede calificarse la capacidad de un equipo de programación para crear software predecible y que solucionen las necesidades de sus clientes. El CMM aplica una escala de cinco niveles, que van desde el caos en el nivel 1, hasta el parangón de una buena gestión en el nivel 5. Las organizaciones evaluadas hasta ahora son 261. "La aplastante mayoría- alrededor del 75 por ciento- siguen todavía empantanadas en el nivel 1", nos cuenta Curtis. "Carecen de procesos formales, no miden lo que hacen y no tienen forma de saber si van por el mal camino o si han descarrilado del todo." (El Centro de Análisis Naval concluyó que el nivel del proyecto AAS en Federal Systems, de IBM, como "1 bajo".) El 24 por ciento restante de los proyectos se encuentra entre los niveles 2 y 3. Tan sólo dos grupos de elite han merecido la máxima puntuación CMM, el nivel 5. El equipo de programación que Motorola tiene en Bangalore, India, ostenta uno de esos títulos. El otro ha sido el proyecto de software de a bordo para el transbordador espacial, realizado por Loral (que antes pertenecía a IBM). Tan perfectamente ha aprendido el equipo de Loral a controlar los errores, que es

capaz de pronosticar correctamente cuántos van a descubrirse en cada nueva versión del software. Se trata de una hazaña notable, habida cuenta de que la mayoría de los programadores ni siquiera llevan la cuenta de los bugs que descubren, de acuerdo a Capers Jones, Chairman de Software Productivity Research. Y entre quienes lo hacen, dice, son pocos quienes detectan más de la tercera parte de los existentes. El director del proyecto de software para el transbordador espacial, Tom Peterson, atribuye su éxito a "una cultura que se esfuerza por enmendar no sólo los errores del programa sino también las fallas del proceso de comprobación, que permitieron que los primeros pasaran desapercibidos". No obstante, siempre se cuela alguno. El primer lanzamiento del transbordador espacial en 1981 tuvo que interrumpirse y posponerse dos días debido a que una señal errónea impidió que los cinco ordenadores de a bordo quedaran debidamente sincronizados. Otro fallo, esta vez en el programa de rendezvous del transbordador, puso en peligro la misión de rescate del satélite Intelsat-6, en 1992. Aunque el modelo CMM no es la panacea, la promoción que de él ha realizado el SEI ha persuadido a algunas de las principales empresas de software de que, a largo plazo, el control cuantitativo de la calidad puede resultar rentable. Así, por ejemplo, la división de equipos de Raytheon elaboró en 1988 un "iniciativa de ingeniería de software" tras haber fallado en una evaluación del CMM. Empezaron a invertir un millón de dólares anuales para refinar las guías de inspección y de verificación rigurosa y para formar a sus 400 programadores para que las siguieran. En el plazo de tres años ya habían escalado dos niveles. El mes de junio pasado casi todos los proyectos, entre ellos complejos sistema de radar y de control de tráfico aéreo, se estaban concluyendo antes de plazo y a costo inferior del presupuestado. La productividad se ha duplicado con holgura. Los análisis de costos de retrabajo han indicado ahorros de 7,80 dólares por cada dólar invertido en la iniciativa. Impresionada por semejantes éxitos, la Fuerza Aérea de los Estados Unidos ha establecido que quienes desarrollen programas para ella han de alcanzar el nivel 3 de CMM en 1998. Se tienen noticias de que la NASA piensa actuar de forma parecida.

Re- creaciones matemáticas

En tanto los humanos creen programas, siempre se producirán errores; hasta los diseños más cuidados pueden desviarse. No obstante, los bugs detectados en los estadios iniciales raramente amenazan los plazos y presupuestos de un programa. Los que provocan efectos devastadores son casi siempre deslices cometidos en el diseño inicial que llegan intactos hasta el producto definitivo. Los fabricantes de programas para mercados masivos, que no han de complacer a un cliente individual, pueden adoptar un enfoque a posteriori y de fuerza bruta para eliminar los defectos: hacen circular el producto deficiente con el nombre de versión "beta" y dejan que multitudes de usuarios se encarguen de descubrir las fallas. Según Charles Simonyi, un Arquitecto Jefe de Microsoft, la nueva versión del sistema operativo Windows será objeto de tests "beta" por veinte mil voluntarios. Tal sistema es muy eficaz, pero también oneroso, ineficiente y poco práctico, dado que los productos masivos para computadores personales sólo

suponen el diez por ciento del mercado de software en los Estados Unidos, que mueve 92.800 millones de dólares al año. En consecuencia, los investigadores formulan diversas para atacar los errores tempranamente, e incluso para impedir que ocurran. Una ideas consiste en tener en cuenta que el problema que se supone resuelve el sistema, siempre cambia a medida que se lo va construyendo. Las planificaciones del aeropuerto de Denver cargaron sobre las espaldas de BAE modificaciones por valor de 20 millones de dólares en el diseño del sistema de equipajes mucho después de haberse iniciado su construcción. IBM sufrió de igual manera las indecisiones de los directivos de la FAA. Ambas empresas habían dado por supuesto, ingenuamente, que una vez aprobado su diseño, se las dejaría en paz para construirlo. Algunos desarrolladores están abandonando esa ilusión y repensando al software como algo que, más que construirse, se cultiva. En un primer paso, se hilvana rápidamente un prototipo merced a componentes estándar de interfases gráficas. Lo mismo que las maquetas utilizadas en arquitectura, el prototipo de un sistema puede servir para aclarar malentendidos entre el programador y su cliente antes de empezar a establecer los cimientos lógicos. Pero los prototipos sirven de poco en la detección de inconsistencias lógicas del diseño, pues sólo remedan el aspecto externo de su comportamiento. “La gran mayoría en software de gran envergadura son errores de omisión”, nota Laszlo A. Belady, director de Mitsubishi Electric Research Laboratory. Y una vez escrito el programa los modelos en nada facilitan la detección de bugs. Cuando la exigencia de corrección del programa es absoluta, dice Martyn Thomas, gerente general de Paxis, una compañía británica de software, los ingenieros recurren a análisis matemáticos para predecir cuál será el comportamiento de sus creaciones en el mundo real. Por desdicha, las matemáticas tradicionales, aptas para la descripción de sistemas físicos, no son aplicables al universo sintético binario de un programa de computador; es la matemática discreta, una especialidad mucho menos madura, la que gobierna este campo. Aun así, valiéndose del instrumental, no muy amplio todavía, de la teoría de conjuntos y del cálculo de predicados, los científicos de la computación se las han arreglado para traducir especificaciones y programas al lenguaje matemático, donde pueden analizarse con los instrumentos teóricos denominados métodos formales. Praxis ha usado recientemente métodos formales en un proyecto de control de tráfico aéreo para el órgano administrativo responsable de la aviación civil en Gran Bretaña. Aunque su programa era mucho más pequeño que el de la FAA, ambos afrontaban un problema similar: la necesidad de mantener sincronizados sistemas redundantes, de manera que si uno de ellos fallase, el otro pudiera entrar en servicio instantáneamente. “Lo difícil consistía en garantizar que los mensajes se entregaran en el orden debido a las dos redes gemelas”, según Anthony Hall, un consultor principal de Praxis. “Tratamos de conseguir demostraciones matemáticas de nuestro enfoque, que fallaron, porque el diseño era erróneo. La detección de errores en ese estadio inicial tiene enormes ventajas”. El sistema quedó concluido a tiempo y entró en servicio en octubre de 1993. Praxis sólo utilizó notaciones formales solamente en las secciones más críticas del software, pero otras firmas de software han aplicado el rigor matemático a la totalidad del desarrollo de un sistema. En París, GEC Alsthom está utilizando un método formal llamado “B”, al tiempo que invierte 350 millones de dólares en perfeccionar los programas de control

de velocidad y de guiado de los 6000 trenes eléctricos del sistema nacional de ferrocarriles francés, la SNCF. El sistema puede ahorrar miles de millones de dólares si permite aumentar la velocidad de los trenes y su frecuencia, el sistema puede ahorrar miles de millones de dólares evitando los nuevos tendidos de líneas necesarios de otro modo. La seguridad es una preocupación obvia. Por eso lo que los desarrolladores de GEC escribieron el diseño completo y el programa final en notación formal y luego usaron matemáticas para demostrar la consistencia de ambos. “Los test funcionales siguen siendo necesarios por dos razones”, indica Fernando Mejía, director de la sección de desarrollo formal de GEC. En primer lugar, a veces los programadores cometen errores en las demostraciones. En segundo lugar, los métodos formales solamente pueden garantizar que el software cumple con su especificación, no que sean capaces de manejarse con las sorpresas que depara el mundo real. No son éstos los únicos problemas de los métodos formales. Ted Ralston, director de planeamiento estratégico de Odyssey Research Associates, en Ithaca, Nueva York, hace notar que la lectura de página tras página de fórmulas algebraicas resulta más soporífera todavía que la revisión de código de compujtadora. Odyssey es una de las empresas que están tratando de automatizar los métodos formales para que les resulten menos onerosos a los programadores. En Francia, GEC colabora con Digilog para comercializar herramientas de programación para el método B. Siete compañías e instituciones, Aerospace entre ellas,así como el organismo responsable de la energía atómica y el ministerio de defensa franceses, están probando la versión beta. Del otro lado del Atlántico, los métodos formales por sí mismos tienen todavía que abrirse paso. “Yo soy escéptico que los estadounidenses sean lo suficientemente disciplinados para aplicar métodos formales en alguna forma generalizada”, dice David A. Fisher del National Institute of Standards and Technology (NIST). Existen excepciones, no obstante, sobre todo en el círculo cada vez más amplio de las empresas que experimentan una metodología de programación denominada “enfoque de sala limpia” (clean-room approach). Este método trata de conjugar las notaciones formales, las demostraciones de validez y los controles estadísticos de calidad con un enfoque evolutivo del desarrollo de software. Al igual que la técnica de fabricación de circuitos integrados de la que toma su nombre, el desarrollo de “sala limpia” trata de aplicar consistentemente técnicas rigurosas de ingeniería para fabricar productos que funcionen perfectamente la primera vez. Los programadores desarrollan el sistema de a una función por vez y se aseguran de certificar la calidad de cada unidad antes de integrarla en el la arquitectura. Software creciendo en esta forma, requiere toda una nueva metodología de testeo. Tradicionalmente, los desarrolladores testean un programa ejecutándolo del modo en que ellos entienden que debe funcionar, modos que, muchas veces, apenas si guardan leve parecido con los del mundo exterior. En un proceso de sala limpia, los programadores tratan de asignar una probabilidad a cada camino de ejecución -correcto e incorrecto- que los usuarios puedan tomar. Ellos pueden derivar casos de prueba de estos datos estadísticos, de forma que las rutas más frecuentemente utilizadas se analicen más profundamente. Después se hace funcionar el programa en cada una de estos casos de prueba y se cronometra cuánto tarda en fallar. Estos tiempos se le introducen a continuación a un modelo matemático, que calcula la confiabilidad del programa, a la manera más puramente ingenieril. Los

informes de los que adoptaron inicialmente el enfoque alientan a utilizarlo. Ericsson Telecom, el gigante europeo de las telecomunicaciones, aplicó procesos de sala limpia en un proyecto de 70 programadores destinado a elaborar un sistema operativo para computadores de conmutación telefónica. Sus informes hablan de que los errores se redujeron a uno por cada 1000 líneas de código de programa, cuando el promedio de la industria es unas veinticinco veces mayor. Y, lo que puede ser más importante, se comprobó que la productividad del desarrollo aumentó un 70 por ciento y la de las pruebas se duplicó.

No hay balas de plata (6)

En esta profesión se ha oído hablar ya muchas veces de “balas de plata” presuntamente capaces de hacer desaparecer los bugss que plagan los proyectos. Desde los años sesenta se han anunciado docenas de innovaciones técnicas destinadas a incrementar la productividad; muchos de sus creadores han presentado incluso proyectos “demostrativos” de la veracidad de sus pretensiones. Los partidarios del análisis y de la programación orientada a objetos, uno de los comodines verbales del presente, proclaman que su método constituye un cambio de paradigma capaz de generar “una mejora de productividad de 14 a 1”, junto con superior calidad y mantenimiento más fácil, todo ello a menor costo. No faltan motivos para el escepticismo. Curtis recuerda que “En los años 70 se proclamó que la programación estructurada era un cambio de paradigma y otro tanto ocurrió luego con CASE [computer-assisted software engineering, ingeniería de software asistida por ordenador], repitiéndose la historia con los lenguajes de tercera, cuarta y quinta generación. Nosotros hemos oído grandes promesas tecnológicas, muchas de las cuales jamás se han cumplido”. Mientras tanto, la productividad en el desarrollo de software se ha rezagado con respecto a la de disciplinas más maduras, sobre todo con respecto a la ingeniería de computadoras. “Yo pienso del software como un objeto de culto”, dice Cox. “Nuestros principales logros fueron importados desde la cultura extranjera de la ingeniería de hardware - máquinas cada vez más rápidas y dotadas de más memoria”. Fisher tiende a acordar: ajustado por inflación “el valor agregado por trabajador en la industria ha sido de US$ 40,000 por dos décadas”, asevera. “Nosotros no estamos viendo ningún incremento.” “Yo no pienso así”, replica Richard A. DeMillo, un profesor de la Universidad de Purdue y principal directivo del Software Engineering Research Consortium. “Ha habido mejoras, pero cada uno usa diferentes definiciones de productividad”. Un estudio reciente publicado por Capers Jones -pero basado necesariamente en datos históricos dudosos- establece que los programadores de Estados Unidos de América generan el doble de código que en 1970. El hecho es que nadie sabe lo productivos que son quienes desarrollan software, y ello por tres razones. La primera es que menos del diez por ciento de las compañías norteamericanas mide de forma coherente y sistemática la productividad de sus programadores. La segunda es que no se ha establecido aún una unidad de medida standard y útil en el sector. La mayoría de los estudios, incluyendo los publicados en las revistas sujetas a referato de ciencias de la computación, expresan la productividad en

términos de líneas de código por trabajador por mes. Pero los programas están escritos en una gran variedad de lenguajes y varían enormemente en la complejidad de su operación. Comparando el número de líneas escritas por un programador japonés usando C con el número producido por un estadounidense usando Ada, es como comparar sus salarios sin convertir de yen a dólares. En tercer lugar, dice Fisher, "usted puede caminar en una empresa típica y encontrar a dos personas que comparten la misma oficina, perciben el mismo salario y poseen calificaciones equivalentes y usted puede detectar, sin embargo, diferencias en un factor cien en el número de instrucciones por día que producen". Tan enormes diferencias individuales tienden a anular los efectos, mucho menos acusados, de las mejoras tecnológicas o de procesos.

Tras veinticinco años de desengaños, debido a aparentes innovaciones que resultaron ser no reproducibles o imposibles de adoptar a mayor escala, son muchos los investigadores que admiten que la ciencia de la computación requiere una rama experimental destinada a separar los resultados generales de los meramente accidentales. "Siempre se ha dado por supuesto que, si yo enseño un método, éste es correcto sólo porque yo lo digo", denuncia Víctor R. Basili, un profesor en la Universidad de Maryland. "Se están produciendo todo genero de cosas y resulta verdaderamente escalofriante lo malas que son algunas de ellas," dice. Mary Shaw, de Carnegie Mellon puntualiza que las ingenierías maduras han codificado en manuales las soluciones bien probadas, de forma que incluso novicios poco experimentados puedan manejar consistentemente los diseños rutinarios, dejando libres para los proyectos avanzados a los profesionales de mayor talento. No existen manuales semejantes para el software, por lo que los errores se repiten año tras año, en un proyecto tras otro. DeMillo sugiere que el gobierno debería tomar un papel más activo. "La National Science Foundation debería estar interesada en financiar investigación orientada a verificar experimentalmente resultados que ha sido establecidos por otra gente", dice. "Actualmente si no es investigación hecha por primera vez y de gran creatividad, los administradores de los programas de la NSF tienden a discontinuar el trabajo." DeMillo conoce de qué habla. Desde 1989 a 1991 dirigió la división de investigación en computadoras y computación de la NSF. Además, "si la ingeniería de software es una ciencia experimental, esto significa que necesita ciencia de laboratorio. Dónde están los laboratorios?" pregunta Basili. Puesto que el esfuerzo de llevar las tecnologías prometedoras a la escala de la proporciones industriales a menudo falla, los laboratorios pequeños son de utilidad limitada. "Necesitamos tener instalaciones en las que obtener datos y tratar de sacar las cosas," dice DeMillo. "La única forma de hacer esto es tener una organización real de desarrollo de software como socio." Ha habido sólo unos pocas de esas asociaciones. Quizás la más exitosa es el Software Engineering Laboratory, un consorcio del Goddard Space Flight Center de la NASA, Computer Sciences Corporation y la Universidad de Maryland. Basili ayhudó a fundar el laboratorio en 1976. Desde entonces, estudiantes de posgrado y programadores de la NASA han colaborado en "más de 100 proyectos," dice Basili, la mayoría vinculados con la construcción de software para soporte terrestre para satélites.

Basta echarle agua

Los fabricantes de arcabuces no lograron mejorar su productividad hasta que Eli Whitney concibió la forma de producir piezas intercambiables, que pudieran ser ensambladas por cualquier operario hábil. De forma análoga una vez standarizadas, las partes del software podrían reutilizarse en diferentes escalas. Hace decenios que los programadores vienen utilizando bibliotecas de subrutinas para no tener que escribir una y otra vez el mismo código. Pero estos componentes dejan de funcionar al ser trasladados a un lenguaje de programación diferente, o a una plataforma de hardware o entorno operativo de distinto tipo. “La tragedia es que, al devenir obsoleto el hardware, resulta necesario volver a escribir lo que era una excelente muestra de un algoritmo de clasificación conseguida en los años sesenta”, observa Simonyi de Microsoft. Fisher ve una tragedia de diferente tipo. “El verdadero precio que nosotros pagamos es que como especialista en cualquier tecnología de software usted no puede reflejar en un producto esa peculiar destreza que posee. Y si tal cosa no es posible, también es imposible ser especialista.” Y no es que falten quienes lo hayan intentado. Antes de trasladarse a NIST el año pasado, Fisher fundó y actuó de máximo responsable de Incremental Systems. “Sin ninguna duda eramos “world-class” en tres de las técnicas que intervienen en los compiladores, pero no alcanzábamos el mismo nivel de calidad en las otras siete o así”, declara. “Y descubrimos que no había ninguna forma práctica de vender componentes de compilador; era preciso que vendiéramos compiladores completos.” Así que ahora está tratando de ponerle remedio a esa situación. NIST anunció en abril que estaba preparando un Programa de Tecnología Avanzada que contribuyera a generar un mercado para software basado en componentes. En su calidad de director del programa, Fisher distribuirá 150 millones de dólares en subvenciones de investigación a compañías de software dispuestas a afrontar los obstáculos técnicos que actualmente hacen inviable producir partes de software. El máximo desafío consiste en encontrar formas de romper los lazos que ligan indisolublemente los programas a computadores específicos y a otros programas. Los investigadores están estudiando varios métodos prometedores, entre ellos un lenguaje común que podría servir para describir partes de software; programas capaces de reformar los componentes, adaptándolos a un ambiente cualquiera; y componentes provistos de múltiples opciones, que el usuario podría activar o no. Fisher favorece la idea que las componentes deberían ser sintetizadas en la acción. Los programadores deberían “básicamente capturar cómo hacer en lugar de haciéndolo,” produciendo una reformulación que cualquier computadora pueda comprender. “Luego cuando usted quiere ensamblar dos componentes, usted podría tomar esta reformulación y derivar versiones compatibles añadiendo elementos adicionales a sus interfaces. Todo podría ser automatizado,” explica. Incluso con un incentivo de 150 US$ millones y presiones del mercado forzando a las empresas a encontrar caminos más baratos para producir software, no es inminente una revolución industrial en el software. “Nosotros esperamos ver solamente ejemplos aislados de estas tecnologías en cinco a siete años - y eso en el caso que no fracase todo,” se resguarda Fisher. E incluso cuando la tecnología esté a punto, serán pocos quienes adopten los componentes a menos que éstos puedan

producirse a un precio ventajoso. Y el costo de los módulos dependerá menos de la tecnología utilizada que del tipo de mercado que pueda surgir para producirlos y consumirlos.

Brad Cox, como Fisher también fue director de una compañía de componentes de programación y le resultó difícil sacarla adelante. Cree saber dónde está el problema...y su solución. Su empresa trataba de vender componentes de programas de bajo nivel, algo análogo a los microcircuitos de los ordenadores. “La diferencia entre los circuitos integrados de silicio y los ‘circuitos integrados’ para software estriba en que los primeros están formados por átomos, y perduran por conservación de la masa; en consecuencia la gente sabe como comprarlos y venderlos con seguridad,” dice Cox. “Pero este proceso de intercambio, que se encuentra en el corazón de todo el comercio, sencillamente no funciona para cosas que se pueden copiar en nanosegundos.” Cuando trató de vender los módulos creados por sus programadores, el precio que los clientes estaban dispuestos a pagar era demasiado bajo para resarcirle de los costos de desarrollo. Las razones eran dobles. La primera, que la adaptación manual de los módulos a las necesidades de cada cliente era una operación que consumía mucho tiempo; NIST confía en salvar esta barrera con su Programa de Tecnología Avanzada. El otro factor no era tanto técnico como cultural: lo que quieren los compradores es pagar por el componente una vez y luego hacer copias gratis. “La industria de la música ha tenido cerca de un siglo de experiencia con este grave problema,” observa Cox. “Ellos solían vender objetos tangibles, como partituras y rollos de pianola, pero llegaron la radio y la televisión y todo aquello se fue al garete.” Las compañías musicales se adaptaron a la radiodifusión estableciendo agencias encargadas de percibir los derechos de autor cada vez que se emite una melodía y de encauzar esos fondos hacia los artistas y los productores. Cox propone que se les pase un cargo a los usuarios cada vez que utilicen un componente de programación. “De hecho,” dice, “en el caso del software tal modelo podría funcionar mejor que con la música, gracias a las ventajas de infraestructura que proporcionan los ordenadores y las comunicaciones. Los reproductores de grabaciones no disponen de enlace a redes de alta velocidad que registren su utilización, peor los ordenadores, si.” O, por lo menos, la tendrán en el futuro. Escudriñando en el tiempo en que casi todas las computadoras estén conectadas, Cox visualiza la distribución de software de todos los tipos a través de redes que vinculen productores de componentes, usuarios finales e instituciones financieras. “Es análogo a una operación de tarjeta de crédito pero con tentáculos que penetran en cada PC,” dice. Aunque a algunos pueda sonarle ominoso, Cox argumenta que “ahora Internet es más un depósito de desperdicios que un mercado para comprar. Necesitamos una gran infraestructura que pueda soportar la distribución de todo tipo de productos.” Reconociendo la envergadura del cambio cultural que propone, Cox espera empujar su causa por años a través de la Coalition for Electronic Markets, de la que es presidente. La combinación de control industrial de procesos, herramientas técnicamente avanzadas y partes intercambiables promete transformar no sólo la forma de realizar la programación, sino también a los encargados de efectuarla. Muchos de los expertos que convergieron en Hedsor Park

acuerdan con Belady que “en el futuro, los profesionales de muchos campos usarán la programación como una herramienta de trabajo, pero ellos no querrán llamarse a sí mismos programadores ni creerán estar dedicando su tiempo a la programación. Pensarán, en cambio, estar haciendo arquitectura, o control de tráfico, o películas de cine”. Esta posibilidad suscita la pregunta de quiénes están capacitados para construir sistemas importantes. En la actualidad cualquiera puede anunciarse como ingeniero de software. “Pero cuando se tengan cien millones de programadores-usuarios, será frecuente que se hagan cosas críiticas para la vida, como, por ejemplo, construir programas para extender recetas médicas,” hace notar Barry W.Boehm , director del Center for Software Engineering de la Universidad de Southern California. Boehm es uno del creciente número de quienes recomiendan que los ingenieros de software deben certificarse, como se hace en otros campos de la ingeniería. Tal certificación sólo servirá de algo si los programadores reciben la formación adecuada. Actualmente sólo 28 universidades ofrecen programas de posgrado en ingeniería de software; cinco años atrás había sólo 10. Ninguna ofrece títulos de grado. Incluso académicos tales como Shaw, DeMillo y Basili acuerdan que en general la curricula de computer science ofrece por lo general una preparación deficiente para el desarrollo industrial de software. “Pues aspectos fundamentales, como el diseño de inspecciones del código, la producción de documentación para el usuario y el mantenimiento de los programas que van quedándose anticuados, no son cubiertos en la academia”, se lamenta Capers Jones. Los ingenieros, la infantería de toda revolución industrial, no se generan espontáneamente. Reciben una formación tendiente a evitar los malos hábitos de los artesanos que les precedieron. En tanto las lecciones de la ciencia de la computación no inculquen no sólo el deseo de construir mejores cosas, sino el de construir mejor las cosas, lo mejor que podemos esperar es que el desarrollo del software vaya pasando por una evolución industrial lenta y probablemente dolorosa.

Cita bibliográfica:

Gibbs, Wayt. (1994). Software's Chronic Crisis. Scientific American - SCI AMER. 271. 86-95. 10.1038/scientificamerican0994-86.

Bibliografía complementaria:

Encyclopedia of Software Engineering. Editada por John J. Marciniak. John Wiley & Sons, 1994. Software 2000: A View of the Future. Editado por Brian Randell, Gill Ringland y Bill Wulf. ICL y la Comisión de Comunidades Europeas, 1994. Formal Methods: A Virtual Library. Jonathan Bowen. Disponible en hipertexto en Wide World Web, con la referencia http://www.comlab.ox.ac.uk/archive/formal-methods.html

1 Traducción del Scientific American aparecida en Investigación y Ciencia, noviembre de 1994. Revisada por Alejandro Oliveros. En lo que sigue las notas son comentarios de AO a fin de aclarar la traducción.

2 Denver, Colorado, es un gran centro de esquí invernal de los Estados Unidos de América

3 Día de Brujas, 2 de noviembre

4 Del hemisferio norte

5 El promotor de las piezas de recambio

6 Alude a un artículo clásico de la Ingeniería de Software, de Fred Brooks. El título No silver bullets, recupera una leyenda según la cual al hombre lobo sólo se lo puede matar con una bala de plata en el medio del corazón.