Ninguna bala de plata

-Estado y accidente en la ingeniería del software

Frederick P. Brooks, Jr.
Universidad de Carolina del Norte en Chapel Hill

> *No hay ningún avance, ni en la tecnología ni en la técnica de gestión, que por sí solo prometa una mejora de un orden de magnitud en el plazo de una década en cuanto a productividad, fiabilidad y simplicidad.*

Resumen[1]

Toda la construcción de software implica tareas esenciales, como la creación de las complejas estructuras conceptuales que componen la entidad abstracta del software, y tareas accidentales, como la representación de estas entidades abstractas en lenguajes de programación y la asignación de éstas a lenguajes de máquina dentro de las limitaciones de espacio y velocidad. La mayor parte de los grandes avances del pasado en la productividad del software han venido de la mano de la eliminación de las barreras artificiales que han hecho que las tareas accidentales sean excesivamente difíciles, como las severas restricciones de hardware, los lenguajes de programación incómodos o la falta de tiempo de máquina. ¿Qué parte de lo que hacen ahora los ingenieros de software sigue dedicándose a lo accidental, en contraposición a lo esencial? A no ser que sea más de 9/10 de todo el esfuerzo, reducir todas las actividades accidentales a tiempo cero no supondrá una mejora de un orden de magnitud.

Por lo tanto, parece que ha llegado el momento de abordar las partes esenciales de la tarea del software, las que tienen que ver con la formación de estructuras conceptuales abstractas de gran complejidad. Yo sugiero:

- Explotar el mercado de masas para evitar construir lo que se puede comprar.
- Utilizar la creación rápida de prototipos como parte de una iteración planificada para establecer los requisitos del software.
- El software crece de forma orgánica, añadiendo más y más funciones a los sistemas a medida que se ejecutan, se utilizan y se prueban.
- Identificar y desarrollar los grandes diseñadores conceptuales de la nueva generación.

Introducción

De todos los monstruos que llenan las pesadillas de nuestro folclore, ninguno aterroriza más que los hombres lobo, porque se transforman inesperadamente de lo familiar en horrores. Para ellos, buscamos balas de plata que puedan hacerlos descansar mágicamente.

---

[1] Reproducido de: Frederick P. Brooks, *The Mythical Man-Month, Anniversary edition with 4 new chapters*, Addison-Wesley (1995), a su vez reimpreso de *Proceedings of the IFIP Tenth World Computing Conference*, H.-J. Kugler, ed., Elsevier Science B.V., Amsterdam, NL (1986) pp. 1069-76.

El conocido proyecto de software tiene algo de este carácter (al menos, visto por el gestor no técnico), normalmente inocente y sencillo, pero capaz de convertirse en un monstruo de plazos incumplidos, presupuestos disparados y productos defectuosos. Por eso se oyen gritos desesperados en busca de una bala de plata, algo que haga que los costes del software disminuyan tan rápidamente como los del hardware informático.

Pero, cuando miramos al horizonte de dentro de una década, no vemos ninguna bala de plata. No hay ningún avance, ni en la tecnología ni en la técnica de gestión, que por sí solo prometa una mejora de un orden de magnitud en la productividad, en la fiabilidad o en la simplicidad. En este capítulo intentaremos ver por qué, pero examinando tanto la naturaleza del problema del software como las propiedades de las balas propuestas.

Sin embargo, el escepticismo no es pesimismo. Aunque no vemos avances sorprendentes y, de hecho, creemos que son incompatibles con la naturaleza del software, hay muchas innovaciones alentadoras en marcha. Un esfuerzo disciplinado y coherente para desarrollarlas, propagarlas y explotarlas debería producir una mejora del orden de magnitud. No hay un camino real, pero hay un camino.

El primer paso hacia la gestión de las enfermedades fue la sustitución de las teorías del demonio y de los humores por la teoría de los gérmenes. Ese mismo paso, el comienzo de la esperanza, echó por tierra todas las esperanzas de soluciones mágicas. Indicó a los trabajadores que el progreso se haría por etapas, con gran esfuerzo, y que habría que prestar una atención persistente e incesante a una disciplina de limpieza. Lo mismo ocurre hoy con la ingeniería de software.

¿Tiene que ser difícil? - Dificultades esenciales

No sólo no hay balas de plata a la vista, sino que la propia naturaleza del software hace ιμπροβαβλε que haya inventos que hagan por la productividad, la fiabilidad y la simplicidad del software lo que la electrónica, los transistores y la integración a gran escala hicieron por el hardware informático. No podemos esperar que se dupliquen los avances cada dos años.

En primer lugar, debemos observar que la anomalía no es que el progreso del software sea tan lento, sino que el progreso del hardware informático es tan rápido. En ninguna otra tecnología desde el comienzo de la civilización se ha producido una ganancia de precio-rendimiento de seis órdenes de magnitud en 30 años. En ninguna otra tecnología se puede elegir entre la mejora del rendimiento o la reducción de los costes. Estas ganancias se derivan de la transformación de la fabricación de ordenadores, que ha pasado de ser una industria de ensamblaje a una industria de procesos.

En segundo lugar, para ver qué ritmo de progreso podemos esperar en la tecnología del software, examinemos sus dificultades. Siguiendo a Aristóteles, las divido en esenciales -las dificultades inherentes a la naturaleza del σοφτωαρε– ψ αχχιδενταλεσ –λασ dificultades que hoy en día acompañan a su producción pero que no son inherentes-.

Los accidentes los analizo en la siguiente sección. Primero consideremos la esencia.

La esencia de una entidad de software es una construcción de conceptos entrelazados: conjuntos de datos, relaciones entre elementos de datos, algoritmos e invocaciones de funciones. Esta esencia es abstracta, en el sentido de que la construcción conceptual es la misma bajo muchas representaciones diferentes. Sin embargo, es muy precisa y rica en detalles.

Creo que la parte más difícil de la creación de software es la especificación, el diseño y la comprobación de esta construcción conceptual, no el trabajo de representarla y comprobar la fidelidad de la representación. Seguimos cometiendo errores de sintaxis, sin duda, pero son insignificantes comparados con los errores conceptuales de la mayoría de

los sistemas.

Si esto es cierto, construir software siempre será difícil. No hay una bala de plata intrínseca.

Consideremos las propiedades inherentes a esta esencia irreductible de los sistemas de software modernos: complejidad, conformidad, cambiabilidad e invisibilidad.

Complejidad. Las entidades de software son más complejas por su tamaño que quizás cualquier otra construcción humana, porque no hay dos partes iguales (al menos por encima del nivel de declaración). Si lo son, convertimos las dos partes similares en una sola, una subrutina, abierta o cerrada. En este sentido, los sistemas de software difieren profundamente de los ordenadores, los edificios o los automóviles, donde abundan los elementos repetidos.

Los ordenadores digitales son en sí mismos más complejos que la mayoría de las cosas que la gente construye; tienen un número muy grande de estados. Esto hace que sea difícil concebirlos, describirlos y probarlos. Los sistemas de software tienen órdenes de magnitud superiores a los de los ordenadores.

Del mismo modo, la ampliación de una entidad de software no es una mera repetición de los mismos elementos en un tamaño mayor; es necesariamente un aumento del número de elementos diferentes. En la mayoría de los casos, los elementos interactúan entre sí de alguna manera no lineal, y la complejidad del conjunto aumenta mucho más que linealmente.

La complejidad del software es una propiedad esencial, no accidental. De ahí que las descripciones de una entidad de software que abstraen su complejidad en a menudo abstraen su esencia. Las matemáticas y las ciencias físicas hicieron grandes progresos durante tres siglos construyendo modelos simplificados de fenómenos complejos, derivando propiedades de los modelos y verificando esas propiedades experimentalmente. Esto funcionó porque las complejidades ignoradas en los modelos no eran las propiedades esenciales de los fenómenos. No funciona cuando las complejidades son la esencia.

Muchos de los problemas clásicos del desarrollo de productos de software se derivan de esta complejidad esencial y de su aumento no lineal con el tamaño. De la complejidad surge la dificultad de comunicación entre los miembros del equipo, lo que provoca fallos en el producto, sobrecostes y retrasos en el calendario. De la complejidad surge la dificultad de enumerar, y mucho menos de comprender, todos los estados posibles del programa, y de ahí la falta de fiabilidad. De la complejidad de las funciones viene la dificultad de invocar esas funciones, lo que hace que los programas sean difíciles de usar. De la complejidad de la estructura viene la dificultad de extender los programas a nuevas funciones sin crear efectos secundarios. De la complejidad de la estructura surge el estado no visualizado que constituye trampas de seguridad.

La complejidad no sólo plantea problemas técnicos, sino también de gestión. Esta complejidad dificulta la visión de conjunto, lo que impide la integridad conceptual. Hace difícil encontrar y controlar todos los cabos sueltos. Crea una tremenda carga de aprendizaje y comprensión que hace que la rotación de personal sea un desastre.

Conformidad. Los programadores no son los únicos que se enfrentan a la complejidad. La física se enfrenta a objetos terriblemente complejos incluso a nivel de partículas "fundamentales". Sin embargo, el físico sigue trabajando con la firme convicción de que existen principios unificadores, ya sea en los quarks o en las teorías del campo unificado. Einstein sostuvo repetidamente que debe haber explicaciones simplificadas de la naturaleza, porque Dios no es caprichoso ni arbitrario.

Esta fe no reconforta al ingeniero de software. Gran parte de la complejidad que debe dominar es una complejidad arbitraria, forzada sin ton ni son por los muchos

instituciones y sistemas a los que sus interfaces deben confirmar. Estos difieren de una interfaz a otra, y de un tiempo a otro, no por necesidad, sino sólo porque fueron diseñados por diferentes personas, y no por Dios.

En muchos casos, el programa informático debe confirmarse porque es el que más recientemente ha entrado en escena. En otros, debe conformarse porque es percibido como el más conforme. Pero en todos los casos, gran parte de la complejidad proviene de la conformidad con otras interfaces; esto no puede simplificarse sólo con un rediseño del software.

Cambiabilidad. La entidad de software está constantemente sujeta a presiones de cambio. Por supuesto, también lo están los edificios, los coches y los ordenadores. Pero las cosas fabricadas no suelen cambiarse después de la fabricación; son sustituidas por modelos posteriores, o los cambios esenciales se incorporan en copias posteriores con número de serie del mismo diseño básico. Las devoluciones de automóviles son realmente infrecuentes; los cambios de campo de los ordenadores, algo menos. Ambas cosas son mucho menos frecuentes que las modificaciones de los programas informáticos de campo.

En parte, esto se debe a que el software de un sistema encarna su función, y la función es la parte que más siente las presiones del cambio. En parte se debe a que el software puede cambiarse más ϕ⟨χιλμεντε: es pura materia de pensamiento, infinitamente maleable. De hecho, los edificios se cambian, pero los elevados costes del cambio, comprendidos por todos, sirven para amortiguar el capricho de los cambiadores.

Todo el software que tiene éxito se modifica. Hay dos procesos en marcha. A medida que un producto de software resulta útil, la gente lo prueba en nuevos casos en el límite o más allá del dominio original. Las presiones para ampliar la función provienen principalmente de los usuarios que les gusta la función básica e inventan nuevos usos para ella.

En segundo lugar, el software de éxito también sobrevive más allá de la vida normal del vehículo de la máquina para la que se escribió por primera vez. Si no son nuevos ordenadores, al menos aparecen nuevos discos, nuevas pantallas, nuevas impresoras; y el software debe adaptarse a sus nuevos vehículos de oportunidad.

En resumen, el producto de software está inmerso en una matriz cultural de aplicaciones, usuarios, leyes y vehículos mecánicos. Todos ellos cambian continuamente, y sus cambios obligan inexorablemente a cambiar el producto de software.

Invisibilidad. Los programas informáticos son invisibles e invisibles. Las abstracciones geométricas son herramientas poderosas. La planta de un edificio ayuda al arquitecto y al cliente a evaluar los espacios, los flujos de tráfico y las vistas. Las contradicciones se hacen evidentes, las omisiones pueden detectarse. Los dibujos a escala de piezas mecánicas y los modelos de moléculas, aunque sean abstracciones, sirven para el mismo propósito. Una realidad geométrica se plasma en una abstracción geométrica.

La realidad del software no está intrínsecamente integrada en el espacio. Por lo tanto, no tiene una representación geométrica fácil, como la tierra tiene mapas, los chips de silicio tienen diagramas, los ordenadores tienen esquemas de conectividad. En cuanto intentamos diagramar la estructura del software, descubrimos que no constituye uno, sino varios gráficos generales dirigidos, superpuestos unos sobre otros. Los distintos gráficos pueden representar el flujo de control, el flujo de datos, los patrones de dependencia, la secuencia temporal, las relaciones entre los espacios de nombres. Por lo general, ni siquiera son planos, y mucho menos jerárquicos. De hecho, una de las formas de

El establecimiento de un control conceptual sobre dicha estructura consiste en imponer el corte de enlaces hasta que uno o varios de los gráficos se jerarquicen.[2]

A pesar de los avances en la restricción y simplificación de las estructuras del software, éstas siguen siendo intrínsecamente invisualizables, lo que priva a la mente de algunas de sus herramientas conceptuales más poderosas. Esta carencia no sólo impide el proceso de diseño dentro de una mente, sino que dificulta gravemente la comunicación entre mentes.

Los avances del pasado resolvieron dificultades accidentales

Si examinamos los tres pasos de la tecnología del software que han sido más fructíferos en el pasado, descubrimos que cada uno de ellos atacó una dificultad importante diferente en la construcción de software, pero han sido las dificultades accidentales, no las esenciales. También podemos ver los límites naturales de la extrapolación de cada uno de esos ataques.

Lenguajes de alto nivel. Sin duda, el golpe más fuerte para la productividad, la fiabilidad y la simplicidad del software ha sido el uso progresivo de lenguajes de alto nivel para la programación. La mayoría de los observadores atribuyen a este desarrollo al menos un factor de cinco en la productividad, y con ganancias concomitantes en la fiabilidad, la simplicidad y la comprensibilidad. ¿Qué consigue un lenguaje de alto nivel? Libera a un programa de gran parte de su complejidadaccidental. Unprograma abstracto consiste en construcciones conceptuales: operaciones, tipos de datos, secuencias y comunicación. El programa concreto de la máquina se ocupa de los bits, los registros, las condiciones, las ramas, los canales, los discos y demás. En la medida en que el lenguaje de alto nivel encarna las construcciones deseadas en el programa abstracto y evita todas las inferiores, elimina todo un nivel de complejidad que era no es en absoluto inherente al programa.

Lo máximo que puede hacer un lenguaje de alto nivel es proporcionar todas las construcciones que el programador imagine en el programa abstracto. No cabe duda de que el nivel de sofisticación de nuestro pensamiento sobre las estructuras de datos, los tipos de datos y las operaciones aumenta constantemente, pero a un ritmo cada vez menor. Y el desarrollo del lenguaje se acerca cada vez más a la sofisticación de los usuarios.

Además, en algún momento la elaboración de un lenguaje de alto nivel se convierte en una carga que aumenta, no reduce, la tarea intelectual del usuario que rara vez utiliza las construcciones esotéricas.

Tiempo compartido. La mayoría de los observadores atribuyen a la multiplicidad de tiempos una importante mejora de la productividad de los programadores y de la calidad de sus productos, aunque no tan grande como la que aportan los lenguajes de alto nivel.

El tiempo compartido ataca una dificultad claramente diferente. El tiempo compartido preserva la inmediatez y, por tanto, nos permite mantener una visión general de la complejidad. La lentitud de la programación por lotes hace que inevitablemente olvidemos las minucias, si no la esencia misma, de lo que estábamos pensando cuando dejamos de programar y pedimos la compilación y la ejecución. Esta interrupción de la conciencia es costosa en tiempo, ya que debemos refrescarla. El efecto más grave puede ser la pérdida de comprensión de todo lo que ocurre en un sistema complejo.

---

[2] Parnas, D.L., "Designing software for ease of extension and contraction", *IEEE Trans. on SE*, 5, 2 (marzo, 1979), pp. 12-138.

La lentitud de los plazos, al igual que las complejidades del lenguaje de la máquina, es una dificultad accidental y no esencial del proceso de software. Los límites de la contribución del tiempo compartido se derivan directamente. El efecto principal es acortar el tiempo de respuesta del sistema. A medida que se va reduciendo, en algún momento pasa el umbral humano de perceptibilidad, unos 100 milisegundos. Más allá de eso no cabe esperar ningún beneficio.

Entornos de programación unificados. Se considera que Unix e Interlisp, los primeros entornos de programación integrados que se han generalizado, han mejorado la productividad por factores integrales. ¿Por qué?

Atacan las dificultades accidentales del uso conjunto de programas, proporcionando bibliotecas integradas, formatos de archivo unificados y pilas y filtros. Como resultado, las estructuras conceptuales que en principio siempre podrían llamarse, alimentarse y utilizarse mutuamente pueden hacerlo fácilmente en la práctica.

Este avance estimuló a su vez el desarrollo de bancos de herramientas completos, ya que cada nueva herramienta podía aplicarse a cualquier programa utilizando los formatos estándar.

Debido a estos éxitos, los entornos son el objeto de gran parte de la investigación actual en ingeniería del software. En la siguiente sección analizaremos sus promesas y limitaciones.

Esperanzas para la plata

Consideremos ahora los avances técnicos que más a menudo se presentan como posibles balas de plata. ¿Qué problemas abordan? ¿Son los problemas de la esencia, o son restos de nuestras dificultades accidentales? ¿Ofrecen avances revolucionarios o incrementales?

Ada y otros avances del lenguaje de alto nivel. Uno de los avances recientes más cacareados es el lenguaje de programación Ada, un lenguaje de alto nivel de propósito general de la década de 1980. De hecho, Ada no sólo refleja las mejoras evolutivas en los conceptos del lenguaje, sino que incorpora características que fomentan los conceptos modernos de diseño y modularización. Tal vez la filosofía de Ada sea un avance mayor que el lenguaje Ada, ya que es la filosofía de la modularización, de los tipos de datos abstractos, de la estructuración jerárquica. Ada es quizás demasiado rico, el producto natural del proceso por el que se establecieron los requisitos para su diseño. Esto no es fatal, ya que los subconjuntos de vocabularios de trabajo pueden resolver el problema del aprendizaje, y los avances del hardware nos darán los MIPS baratos para pagar los costes de compilación. El avance en la estructuración de los sistemas de software es, de hecho, un muy buen uso para el aumento de MIPS que comprarán nuestros dólares. Los sistemas operativos, criticados a bombo y platillo en la década de los 60 por sus costes de memoria y de ciclo, han demostrado ser una forma excelente de utilizar algunos de los MIPS y bytes de memoria baratos de la pasada oleada de hardware.

Sin embargo, Ada no será la bala de plata que mate al monstruo de la productividad del software. Al fin y al cabo, no es más que otro lenguaje de alto nivel, y los mayores beneficios de este tipo de lenguajes provienen de la primera transición, desde las complejidades accidentales de la máquina hasta el enunciado más abstracto de las soluciones paso a paso. Una vez que se han eliminado esos accidentes, los restantes son más pequeños, y la recompensa de su eliminación será seguramente menor.

Preveo que dentro de una década, cuando se evalúe la eficacia de Ada, se verá que ha supuesto una diferencia sustancial, pero no por ninguna característica concreta del lenguaje, ni tampoco por todas ellas combinadas. Tampoco el nuevo Ada

entorno demuestran ser la causa de las mejoras. La mayor contribución de Ada será que el cambio a este sistema ha permitido formar a los programadores en técnicas modernas de diseño de software.

Programación orientada a objetos. Muchos estudiosos del arte tienen más esperanzas en la programación orientada a objetos que en cualquiera de las otras modas técnicas del momento.[3] Yo me encuentro entre ellos. Mark Sherman, de Dartmouth, señala que debemos tener cuidado de distinguir dos ideas distintas que se agrupan bajo ese nombre: los tipos de datos abstractos y los tipos jerárquicos, también llamados clases. El concepto de tipo de datos abstracto es que el tipo de un objeto debe ser definido por un nombre, un conjunto de valores propios y un conjunto de operaciones propias, en lugar de su estructura de almacenamiento, que debe estar oculta. Algunos ejemplos son los paquetes de Ada (con tipos privados) o los módulos de Modula.

Los tipos jerárquicos, como las clases de Simula-67, permiten la definición de interfaces generales que pueden refinarse aún más proporcionando tipos subordinados. Los dos conceptos son ortogonales: puede haber jerarquías sin ocultación y ocultación sin jerarquías. Ambos conceptos representan verdaderos avances en el arte de construir software.

Cada una de ellas elimina una dificultad accidental más del proceso, permitiendo al diseñador expresar la esencia de su diseño sin tener que expresar grandes cantidades de material sintáctico que no añaden ningún contenido informativo nuevo. Tanto para los tipos abstractos como para los tipos jerárquicos, el resultado es eliminar una clase de dificultad accidental de orden superior y permitir una expresión de diseño de orden superior.

Sin embargo, estos avances no pueden hacer más que eliminar todas las dificultades accidentales de la expresión del diseño. La complejidad del diseño en sí misma es esencial, y estos ataques no suponen ningún cambio. La programación orientada a objetos sólo puede suponer una ganancia de un orden de magnitud si la innecesaria maleza de la especificación de tipos que queda hoy en día en nuestro lenguaje de programación es en sí misma responsable de las nueve décimas partes del trabajo que implica el diseño de un producto de programa. Lo dudo.

La inteligencia artificial. Mucha gente espera que los avances de la inteligencia artificial proporcionen el avance revolucionario que dará ganancias de orden de magnitud en la productividad y la calidad del software.[4] Yo no lo espero. Para ver por qué, debemos diseccionar lo que se entiende por "inteligencia artificial" y luego ver cómo se aplica.

Parnas ha aclarado el caos terminológico:

Hoy en día se utilizan dos definiciones bastante diferentes de la IA. IA-1: el uso de ordenadores para resolver problemas que antes sólo podían resolverse aplicando la inteligencia humana. IA-2: El uso de un conjunto específico de técnicas de programación conocido como programación heurística o basada en reglas. En este enfoque se estudia a los expertos humanos para determinar qué heurística o reglas empíricas utilizan en la resolución de problemas. ..... El sitio web
El programa está diseñado para resolver un problema de la forma en que los humanos parecen resolverlo.

[3] Booch, G., "Object-oriented design", en *Software Engineering with Ada*. Menlo Park, California: Benjamin Cummings, 1983.

[4] Mostow, J., ed., Special Issue on Artifical Intelligence and Software Engineering, *IEEE Trans. on SE*, **11**, 11 (nov. 1985).

La primera definición tiene un significado deslizante. Algo puede encajar en la definición de AI-1
hoy en día, pero, una vez que veamos cómo funciona el programa y entendamos el problema,ya no pensará en él como IA. Por desgracia, no puedo identificar un cuerpo de tecnología
que es único en este campo. La mayor parte del trabajo es específico para cada problema, y algunos
abstracción o creatividad es necesario para ver cómo transferirlo.[5]

Estoy completamente de acuerdo con esta crítica. Las técnicas utilizadas para el reconocimiento del habla parecen tener poco en común con las utilizadas para el reconocimiento de imágenes, y ambas son diferentes de las utilizadas en los sistemas expertos. Me cuesta ver cómo el reconocimiento de imágenes, por ejemplo, va a suponer una diferencia apreciable en la práctica de la programación. Lo mismo ocurre con el reconocimiento del habla. Lo difícil de construir software es decidir qué decir, no decirlo. Ninguna facilitación de la expresión puede dar más que ganancias marginales.

La tecnología de los sistemas expertos, AI-2, merece una sección propia.

Sistemas expertos. La parte más avanzada del arte de la inteligencia artificial, y la que más se aplica, es la tecnología para construir sistemas expertos. Muchos científicos del software están trabajando duro para aplicar esta tecnología al entorno de la construcción de software.[6] ¿Cuál es el concepto y cuáles son las perspectivas?

Un sistema experto es un programa que contiene un motor de inferencia generalizado y una base de reglas, diseñado para tomar datos de entrada y suposiciones y explorar las consecuencias lógicas a través de las inferencias derivadas de la base de reglas, arrojando conclusiones y consejos, y ofreciendo la explicación de sus resultados mediante el seguimiento de su razonamiento para el usuario. Los motores de inferencia suelen poder manejar datos y reglas difusas o probabilísticas, además de la lógica puramente determinista.

Estos sistemas ofrecen algunas ventajas claras sobre los algoritmos programados para llegar a las mismas soluciones de los mismos problemas:

- La tecnología de los motores de inferencia se desarrolla de forma independiente de la aplicación y luego se aplica a muchos usos. Se puede justificar un esfuerzo mucho mayor en los motores de inferencia. De hecho, esa tecnología está muy avanzada.
- Las partes modificables de los materiales específicos de la aplicación se codifican en la base de reglas de manera uniforme, y se proporcionan herramientas para desarrollar, cambiar, probar y documentar la base de reglas. Esto regulariza gran parte de la complejidad de la propia aplicación.

Edward Feigenbaum afirma que la potencia de estos sistemas no proviene de mecanismos de inferencia cada vez más sofisticados, sino de bases de conocimiento cada vez más ricas que reflejan el mundo real con mayor precisión. Creo que el avance más importante que ofrece la tecnología es la separación de la complejidad de la aplicación del propio programa.

¿Cómo se puede aplicar esto a la tarea del software? De muchas maneras: sugiriendo reglas de interfaz, aconsejando sobre estrategias de prueba, recordando frecuencias de tipos de peros, ofreciendo pistas de optimización, etc.

---

[5] Parnas, D.L., "Software aspects of strategic defense systems", *Communications of the ACM*, **28**, 12 (dic., 1985), pp. 1326-1335. También en *American Scientist*, **73**, 5 (Sept.-Oct., 1985), pp. 432-440.

[6] Balzer, R., "A 15-year perspective on automatic programming", en Mostow, op. cit.

Consideremos un asesor de pruebas imaginario, por ejemplo. En su forma más rudimentaria, el sistema experto de diagnóstico se parece mucho a la lista de comprobación de un piloto, ofreciendo fundamentalmente sugerencias sobre las posibles causas de las dificultades. A medida que se desarrolla la base de reglas, las sugerencias se vuelven más específicas, teniendo en cuenta de forma más sofisticada los síntomas de los problemas notificados. Se puede visualizar un asistente de depuración que ofrezca sugerencias muy generalizadas al principio, pero que a medida que la estructura del sistema se incorpora a la base de reglas, se vuelve más particular en las hipótesis que genera y las pruebas que recomienda. Un sistema experto de este tipo puede apartarse más radicalmente de los convencionales en que su base de reglas probablemente debería modularse jerárquicamente de la misma manera que el producto de software correspondiente, de modo que a medida que el producto se modifique modularmente, la base de reglas de diagnóstico también pueda modificarse modularmente.

El trabajo necesario para generar las reglas de diagnóstico es un trabajo que tendrá que hacerse de todos modos al generar el conjunto de casos de prueba para los módulos y para el sistema. Si se hace de una manera convenientemente general, con una estructura uniforme para las reglas y un buen motor de inferencia disponible, puede realmente reducir el trabajo total de generación de casos de prueba de tracción, así como ayudar en el mantenimiento de por vida y las pruebas de modificación. Del mismo modo, podemos postular otros asesores, probablemente muchos y probablemente sencillos, para las demás partes de la tarea de construcción de software.

Muchas dificultades se interponen en el camino de la pronta realización de asesores expertos útiles para el desarrollador de programas. Una parte crucial de nuestro escenario imaginario es el desarrollo de formas fáciles de pasar de la especificación de la estructura del programa a la generación automática o semiautomática de reglas de diagnóstico. Aún más difícil e importante es la doble tarea de adquisición de conocimientos: encontrar expertos articulados y autoanalíticos que sepan por qué hacen las cosas; y desarrollar técnicas eficaces para extraer lo que saben y destilarlo en bases de reglas. El requisito esencial para construir un sistema experto es tener un experto.

La contribución más poderosa de los sistemas expertos será seguramente poner al servicio del programador inexperto la experiencia y la sabiduría acumulada de los mejores programadores. Esta contribución no es pequeña. La brecha entre la mejor práctica de la ingeniería del software y la práctica media es muy αμπλια, θυιζ(más que en cualquier otra disciplina de la ingeniería. Una herramienta que difunda las buenas prácticas sería importante.

Programación "automática". Hace casi 40 años que se anticipa y se escribe sobre la "programación automática", la generación de un programa para resolver un problema a partir de un enunciado de las especificaciones del mismo. Hoy en día, algunas personas escriben como si esperaran que esta tecnología proporcionara el siguiente avance.[7]

> Parnas da a entender que el término se utiliza por el glamour y no por el contenido semántico, afirmando,
>
> *En resumen, la programación automática siempre ha sido un eufemismo para referirse a la programación con un lenguaje de mayor nivel que el disponible en ese momento para el programador.*[8]

Sostiene, en esencia, que en la mayoría de los casos es el método de solución, y no el problema, el que debe ser especificado.

[7] Mostow, op. cit.
[8] Parnas, 1985, op. cit.

Se pueden encontrar excepciones. La técnica de construcción de generadores es muy poderosa, y se utiliza de forma rutinaria con buen provecho en los programas de ordenación. Algunos sistemas de integración de ecuaciones diferenciales también han permitido la especificación directa del problema. El sistema evaluaba los parámetros, elegía entre una biblioteca de métodos de solución y generaba los programas.

- Estas aplicaciones tienen propiedades muy favorables:
- Los problemas se caracterizan fácilmente con relativamente pocos parámetros.
- Hay muchos métodos conocidos de solución para proporcionar una biblioteca de alternativas.
- Un extenso análisis ha dado lugar a reglas explícitas para seleccionar las técnicas de solución, dados los parámetros del problema.

Es difícil ver cómo estas técnicas se generalizan al mundo más amplio del sistema de software ordinario, en el que los casos con propiedades tan nítidas son la excepción. Es difícil incluso imaginar cómo podría producirse este avance en la generalización.

Programación gráfica. Uno de los temas favoritos de las tesis doctorales en ingeniería de software es la programación gráfica o visual, la aplicación de los gráficos por ordenador al diseño de software.[9] A veces, la promesa de este enfoque se postula a partir de la analogía con el diseño de chips VLSI, donde los gráficos por ordenador desempeñan un papel tan fructífero. A veces, el enfoque se justifica considerando los diagramas de flujo como el medio ideal para el diseño de programas, y proporcionando potentes facilidades para construirlos.

Todavía no ha surgido nada convincente, y mucho menos emocionante, de tales esfuerzos. Estoy convencido de que nada lo hará.

En primer lugar, como he argumentado en otro lugar, el diagrama de flujo es una abstracción muy pobre de la estructura del software.[10] De hecho, la mejor manera de verlo es como el intento de Burks, von Neumann y Goldstine de proporcionar un lenguaje de control de alto nivel, desesperadamente necesario, para su propuesta de ordenador. En la lamentable forma de recuadro de conexiones de varias páginas en la que se ha elaborado hoy el diagrama de flujo, ha demostrado ser esencialmente inútil como herramienta de diseño: los programadores dibujan los diagramas de flujo después, no antes, de escribir los programas que describen.

En segundo lugar, las pantallas actuales son demasiado pequeñas, en píxeles, para mostrar tanto el alcance como la resolución de cualquier diagrama de software detallado serio. La llamada "metáfora del escritorio" de la estación de trabajo actual es, en cambio, una metáfora de "asiento de avión". Cualquiera que haya revuelto un regazo lleno de papeles mientras está sentado en un vagón entre dos pasajeros corpulentos reconocerá la διφερενχια: υνο sólo puede ver unas pocas cosas a la vez. El verdadero escritorio ofrece una visión general y un acceso aleatorio a una veintena de páginas. Además, cuando los arrebatos de creatividad son fuertes, se sabe que más de un programador o escritor abandona el escritorio por el piso más espacioso. La tecnología de hardware tendrá que avanzar bastante antes de que el alcance de nuestros ámbitos sea suficiente para la tarea de diseño de software.

Más fundamentalmente, como he argumentado anteriormente, el software es muy difícil de visualizar. Ya sea que diagramemos el flujo de control, el anidamiento del alcance de las variables, las referencias cruzadas de las variables, el golpe de datos, las estructuras de datos jerárquicas, o lo que sea, sólo sentimos una dimensión del intrincado elefante del software. Si superponemos todos los diagramas generados por las numerosas vistas relevantes, es difícil extraer una visión global. El VLSI

[9] Raeder, G., "A survey of current graphical programming techniques", en R. B. Grafton y T. Ichikawa, eds., Special Issue on Visual Programming, *Computer*, **18**, 8 (agosto, 1985), pp. 11-25
[10] Brooks 1995, op. cit., capítulo 15.

El diseño de un chip es un objeto bidimensional estratificado cuya geometría refleja su esencia. Un sistema de software no lo es.

Verificación de programas. Gran parte del esfuerzo de la programación moderna se dedica a la comprobación y reparación de errores. ¿Acaso se puede encontrar una bala de plata eliminando los errores en su origen, en la fase de diseño del sistema? ¿Pueden mejorarse radicalmente tanto la productividad como la fiabilidad del producto siguiendo la estrategia profundamente diferente de probar que los diseños son correctos antes de dedicar el inmenso esfuerzo a implementarlos y probarlos?

No creo que encontremos la magia aquí. La verificación de programas es un concepto muy potente , y será muy importante para cosas como los núcleos de sistemas operativos seguros. Sin embargo, la tecnología no promete ahorrar trabajo. Las verificaciones suponen tanto trabajo que sólo se han verificado unos pocos programas importantes.

La verificación de programas no significa programas a prueba de errores. Tampoco hay magia en esto. Las pruebas matemáticas también pueden ser defectuosas. Así que, aunque la verificación puede reducir la carga de pruebas de los programas, no puede eliminarla.

Y lo que es más grave, incluso la verificación perfecta de un programa sólo puede establecer que éste cumple su especificación. La parte más difícil de la tarea del software es llegar a una especificación completa y coherente, y gran parte de la esencia de la construcción de un programa es, de hecho, la depuración de la especificación.

Entornos y herramientas. ¿Cuánto más se puede esperar de la explosión de investigaciones sobre mejores entornos de programación? La reacción instintiva de uno es que los problemas más importantes fueron los primeros que se atacaron y se han resuelto: sistemas de archivos jerárquicos, formatos de archivo uniformes para tener interfaces de programa uniformes y herramientas generalizadas. Los editores inteligentes específicos para cada lenguaje son desarrollos que todavía no se utilizan mucho en la práctica, pero lo máximo que prometen es la ausencia de errores sintácticos y de errores semánticos simples.

Tal vez la mayor ventaja que queda por conseguir en el entorno de la programación sea el uso de sistemas de bases de datos integradas para hacer un seguimiento de las miríadas de detalles que deben ser recordados con precisión por el programador individual y mantenidos al día en un grupo de colaboradores en un único sistema.

Seguramente este trabajo merece la pena, y seguramente dará algún fruto tanto en productividad como en fiabilidad. Pero, por su propia naturaleza, el rendimiento a partir de ahora debe ser marginal.

Estaciones de trabajo. ¿Qué ganancias se pueden esperar para el arte del software a partir de un aumento seguro y rápido de la potencia y la capacidad de memoria de la estación de trabajo individual? ¿Cuántos MIPS se pueden utilizar de forma fructífera? La composición y la edición de programas y documentos son totalmente compatibles con las velocidades actuales. La compilación podría soportar un impulso, pero un factor de 10 en la velocidad de la máquina seguramente dejaría el tiempo de pensamiento como la actividad dominante en el día del programador. De hecho, así parece ser ahora.

Las estaciones de trabajo más potentes son sin duda bienvenidas. No podemos esperar mejoras mágicas de ellas.

Ataques prometedores a la esencia conceptual

Aunque ningún avance tecnológico promete dar los resultados mágicos que conocemos en el ámbito del hardware, se está trabajando mucho y bien y se promete un progreso constante, aunque no espectacular.

Todos los ataques tecnológicos a los accidentes del proceso de software están fundamentalmente limitados por la ecuación de la productividad:

$$\textit{Tiempo de la tarea} = (\textit{Frecuencia})_i \; x \; (\textit{Tiempo})_i$$

Si, como creo, los componentes conceptuales de la tarea se llevan la mayor parte del tiempo, ninguna actividad en los componentes de la tarea que son simplemente la expresión de los conceptos puede dar grandes ganancias de productividad.

De ahí que debamos considerar aquellos ataques que abordan la esencia del problema del software, la formulación de estas complejas estructuras conceptuales. Afortunadamente, algunos de ellos son muy prometedores.

Comprar frente a construir. La solución más radical posible para la construcción de software es no construirlo en absoluto.

Cada día es más fácil, ya que cada vez más proveedores ofrecen más y mejores productos de software para una vertiginosa variedad de aplicaciones. Mientras nosotros, los ingenieros de software, nos afanamos en la metodología de producción, la revolución de los ordenadores personales ha creado no uno, sino m ás, mercados de masas para el software. En todos los quioscos hay revistas mensuales que, clasificadas por tipo de máquina, anuncian y reseñan docenas de productos a precios que van desde unos pocos dólares hasta unos cientos de dólares. Otras fuentes más especializadas ofrecen productos muy potentes para las estaciones de trabajo y otros mercados de Unix. Incluso los peajes y entornos de software pueden comprarse listos para usar. En otro lugar he propuesto un mercado de módulos individuales.

Cualquier producto de este tipo es más barato de comprar que de construir de nuevo. Incluso con un coste de 100.000 dólares, una pieza de software comprada sólo cuesta lo mismo que un año de programación. Y la entrega es inmediata. Inmediata al menos para los productos que realmente existen, productos cuyo desarrollador puede remitir a un usuario satisfecho. Además, este tipo de productos suelen estar mucho mejor documentados y se mantienen algo mejor que los programas informáticos creados en casa.

El desarrollo del mercado de masas es, en mi opinión, la tendencia más profunda a largo plazo en la ingeniería del software. El coste del software siempre ha sido el coste de desarrollo, no el de replicación. Compartir ese coste entre unos pocos usuarios reduce radicalmente el coste por usuario. Otra forma de verlo es que el uso de n copias de un sistema de software multiplica efectivamente la productividad de sus desarrolladores por n. Esto supone una mejora de la productividad de la disciplina y de la nación.

La cuestión clave, por supuesto, es la aplicabilidad. ¿Puedo utilizar un paquete disponible en el mercado para realizar mi tarea? En este punto ha sucedido algo sorprendente. Durante las décadas de 1950 y 1960, un estudio tras otro demostró que los usuarios no utilizaban los paquetes disponibles para las nóminas, el control de inventarios, las cuentas por cobrar, etc. Los requisitos eran demasiado especializados, la variación entre casos era demasiado elevada. En la década de los 80, nos encontramos con paquetes de este tipo muy demandados y de uso generalizado. ¿Qué ha cambiado?

En realidad, los paquetes no. Puede que sean algo más generalizados y algo más personalizables que antes, pero no mucho. Tampoco las aplicaciones. Si

cualquier cosa, las necesidades empresariales y científicas de hoy son más diversas, más complicadas que las de hace 20 años.

El gran cambio se ha producido en la relación de costes entre el hardware y el software. El comprador de una máquina de 2 millones de dólares en 1960 consideraba que podía permitirse 250.000 dólares más por un programa de nóminas personalizado, que se integraba fácilmente y sin problemas en el entorno social hostil a los ordenadores. Hoy en día, los compradores de máquinas de oficina de 50.000 dólares no pueden permitirse programas de nóminas personalizados, por lo que adaptan sus procedimientos de nóminas a los paquetes disponibles. Los ordenadores son ahora tan comunes, si no tan queridos, que las adaptaciones se aceptan como algo natural.

Hay excepciones dramáticas a mi argumento de que la generalización de los paquetes de software ha cambiado poco a lo largo de los años: las hojas de cálculo electrónicas y los sistemas simples de bases de datos. Estas poderosas herramientas, tan obvias en retrospectiva y sin embargo tan tardías en aparecer, se prestan a innumerables usos, algunos bastante poco ortodoxos. Ahora abundan los artículos e incluso los libros sobre cómo abordar tareas inesperadas con la hoja de cálculo. Un gran número de aplicaciones que antes se habrían escrito como programas personalizados en Cobol o en el Generador de Programas de Informes se realizan ahora de forma rutinaria con estas herramientas.

En la actualidad, muchos usuarios manejan sus propios ordenadores día tras día en diversas aplicaciones sin haber escrito nunca un programa. De hecho, muchos de estos usuarios no pueden escribir nuevos programas para sus máquinas, pero sin embargo son expertos en resolver nuevos problemas con ellas.

Creo que la estrategia de productividad de software más poderosa para las organizaciones humanas hoy en día es equipar a los trabajadores intelectuales ingenuos en la línea de fuego con ordenadores personales y buenos programas generalizados de escritura, dibujo, archivos y hojas de cálculo, y darles rienda suelta. La misma estrategia, con capacidades de programación sencillas, también funcionará para cientos de científicos de laboratorio.

Perfeccionamiento de los requisitos y creación rápida de prototipos. La parte más difícil de la construcción de un sistema de software es decidir precisamente qué construir. Ninguna otra parte del trabajo conceptual es tan difícil como establecer los requisitos técnicos detallados, incluyendo todas las interfaces con las personas, con las máquinas y con otros sistemas de software. Ninguna otra parte del trabajo paraliza tanto el sistema resultante si se hace mal. Ninguna otra parte es más difícil de rectificar después.

Por lo tanto, la función más importante que realizan los creadores de software para sus clientes es la extracción y el perfeccionamiento iterativo de los requisitos del producto. Porque la verdad es que los clientes no saben lo que quieren. Normalmente no saben a qué preguntas hay que responder, y casi nunca han pensado en el problema con el detalle que hay que especificar. Incluso la respuesta simple "Hacer que el nuevo sistema de software funcione como nuestro antiguo sistema de procesamiento de información manual∀ es, de hecho, demasiado simple. Los clientes nunca quieren exactamente eso. Los sistemas informáticos complejos son, además, cosas que actúan, que se mueven, que funcionan. La dinámica de esa acción es difícil de imaginar. Por eso, a la hora de planificar cualquier actividad de software, es necesario permitir una amplia iteración entre el cliente y el diseñador como parte de la definición del sistema.

Yo iría un paso más allá y afirmaría que es realmente imposible que los clientes, incluso los que trabajan con ingenieros de software, especifiquen de forma completa, precisa y correcta los requisitos exactos de un producto de software moderno antes de haber construido y probado algunas versiones del producto que están especificando.

Por ello, uno de los esfuerzos tecnológicos más prometedores de la actualidad, y que ataca la esencia, no los accidentes, del problema del software, es el desarrollo de enfoques y herramientas para la creación rápida de prototipos de sistemas como parte de la especificación iterativa de requisitos.

Un sistema de software prototipo es aquel que simula las interfaces importantes y realiza las funciones principales del sistema previsto, sin estar necesariamente sujeto a las mismas limitaciones de velocidad, tamaño o coste del hardware. Los prototipos suelen realizar las tareas principales de la aplicación, pero no intentan manejar las excepciones, responder correctamente a las entradas no válidas, abortar limpiamente, etc. El objetivo del prototipo es hacer realidad la estructura conceptual especificada, de modo que el cliente pueda probar su coherencia y usabilidad

Gran parte de los procedimientos actuales de adquisición de software se basan en la suposición de que se puede especificar un sistema satisfactorio de antemano, obtener ofertas para su construcción, mandarlo construir e instalarlo. Creo que esta suposición es fundamentalmente errónea, y que muchos problemas de adquisición de software se derivan de esa falacia. Por lo tanto, no pueden solucionarse sin una revisión fundamental, que contemple el desarrollo iterativo y la especificación de prototipos y productos.

Ελ δεσαρρολλο **incremental hace crecer, no construir, el software.** Todavía recuerdo la sacudida que sentí en 1958 cuando escuché por primera vez a un amigo hablar de *construir* un programa, en lugar de escribirlo. En un instante se amplió toda mi visión del proceso de software. El cambio de metáfora fue poderoso y preciso. Hoy en día entendemos que la construcción de software se parece a otros procesos de construcción y utilizamos libremente otros elementos de la metáfora, como las *especificaciones*, el *montaje de componentes* y el *andamiaje*.

La metáfora del edificio ha dejado de ser útil. Es hora de cambiar de nuevo. Si, como creo, las estructuras conceptuales que construimos hoy en día son demasiado complicadas para ser especificadas con precisión de antemano, y demasiado complejas para ser construidas sin fallos, entonces debemos adoptar un enfoque radicalmente diferente.

Volvamos a la naturaleza y estudiemos la complejidad de los seres vivos, en lugar de las obras muertas del hombre. Aquí encontramos construcciones cuya complejidad nos sobrecoge. Sólo el cerebro es más complejo que un mapa, más poderoso que una imitación, más rico en diversidad, más autoprotegido y más renovable. El secreto es que se cultiva, no se construye.

Así debe ser con nuestros sistemas de software. Hace algunos años, Harlan Mills propuso que cualquier sistema de software debería crecer mediante un desarrollo incremental.[11] Es decir, primero hay que hacer que el sistema funcione, aunque no haga nada útil salvo llamar al conjunto adecuado de subprogramas ficticios. A continuación, se va ampliando poco a poco, y los subprogramas se convierten en acciones o llamadas a subprogramas vacíos en el nivel inferior.

He visto los resultados más espectaculares desde que empecé a insistir en esta técnica a los constructores de proyectos en mi clase de laboratorio de ingeniería de software. Nada en la última década ha cambiado tan radicalmente mi propia práctica, ni su eficacia. El enfoque requiere un diseño descendente, ya que se trata de un crecimiento descendente del software. Permite retroceder fácilmente. Se presta a los primeros prototipos. Cada función añadida y cada nueva disposición para datos o circunstancias más complejas crecen orgánicamente a partir de lo que ya existe.

[11] Mills, H. D., "Top-down programming in large systems", *Debugging Techniques in Large Systems*, R. Rustin, ed., Englewood Cliffs, N.J., Prentice-Hall, 1971.

Los efectos en la moral son sorprendentes. El entusiasmo se dispara cuando hay un sistema en funcionamiento, aunque sea sencillo. Los esfuerzos se redoblan cuando aparece en la pantalla la primera imagen de un nuevo sistema de software gráfico, aunque sólo sea un rectángulo. Uno siempre tiene, en cada etapa del proceso, un sistema que funciona. Me parece que los equipos pueden *hacer crecer* entidades mucho más complejas en cuatro meses que las que pueden *construir*.

Se pueden obtener los mismos beneficios en los proyectos grandes que en los pequeños.[12]

Grandes diseñadores. La cuestión central de cómo mejorar el arte del software se centra, como siempre, en las personas.

Podemos conseguir buenos diseños siguiendo buenas prácticas en lugar de malas. Las buenas prácticas de diseño se pueden enseñar. Los programadores se encuentran entre la parte más inteligente de la población, por lo que pueden aprender buenas prácticas. Por ello, uno de los principales objetivos de Estados Unidos es promulgar las buenas prácticas modernas. Nuevos planes de estudio, nueva literatura, nuevas organizaciones como el Instituto de Ingeniería de Software, todo ha surgido para elevar el nivel de nuestra práctica de pobre a buena. Esto es totalmente correcto.

Sin embargo, no creo que podamos dar el siguiente paso hacia arriba de la misma manera. Mientras que la diferencia entre los malos diseños conceptuales y los buenos puede residir en la solidez del método de diseño, la diferencia entre los buenos diseños y los grandes seguramente no. Los grandes diseños provienen de los grandes diseñadores. La construcción de software es un proceso creativo. Una metodología sólida puede potenciar y liberar la mente creativa, pero no puede enardecer ni inspirar a los aburridos.

Las diferencias no son μενορεσ, es más bien como Salieri y Mozart. Un estudio tras otro demuestra que los mejores diseñadores producen estructuras más rápidas, más pequeñas, más sencillas, más limpias y con menos esfuerzo. Las diferencias entre los grandes y la media se acercan a un orden de magnitud.

Una pequeña retrospección muestra que, aunque muchos sistemas de software útiles y de calidad han sido diseñados por comités y construidos por proyectos multipartitos, los sistemas de software que han entusiasmado a los apasionados son los productos de una o unas pocas mentes diseñadoras, grandes diseñadores. Consideremos Unix, APL, Pascal, Modula, la interfaz Smalltalk, incluso Fortran; y contrastémoslo con Cobol, PL/I, Algol, MVS/370 y MS-DOS (fig. 1)

Sí	No
Unix	Cobol
APL	PL/1
Pascal	Algol
Modula	MVS/370
Smalltalk	MS-DOS
Fortran	

Fig. 1 Productos interesantes

Por eso, aunque apoyo firmemente los esfuerzos de transferencia de tecnología y desarrollo de planes de estudio que se están llevando a cabo, creo que el esfuerzo más importante que podemos hacer es desarrollar formas de hacer crecer a los grandes diseñadores.

[12] Boehm, B. W., "A spiral model of software development and enhancement", *Computer*, **20**, 5 (mayo,

1985), pp. 43-57.

Ninguna organización de software puede ignorar este reto. Los buenos gestores, por muy escasos que sean, no son más escasos que los buenos diseñadores. Tanto los grandes diseñadores como los grandes gestores son muy raros. La mayoría de las organizaciones dedican un esfuerzo considerable a encontrar y cultivar las perspectivas de gestión; no conozco ninguna que dedique el mismo esfuerzo a encontrar y desarrollar los grandes diseñadores de los que dependerá en última instancia la excelencia técnica de los productos.

Mi primera propuesta es que cada organización de software debe determinar y proclamar que los grandes diseñadores son tan importantes para su éxito como los grandes gestores, y que se puede esperar que se les nutra y recompense de forma similar. No sólo el salario, sino también los beneficios del ρεχονοχιμιεντο – tamaño de la οφιχινα, mobiliario, equipo técnico personal, fondos para viajes, αποψο al personal – δεβεν ser totalmente equivalentes.

¿Cómo hacer crecer a los grandes diseñadores? El espacio no permite una larga discusión, pero algunos pasos son obvios:

- Identifique sistemáticamente a los mejores diseñadores lo antes posible. Los mejores no suelen ser los más experimentados.
- Asigne a un mentor de carrera que se responsabilice del desarrollo del prospecto, y mantenga un cuidadoso archivo de carrera.
- Diseñar y mantener un plan de desarrollo profesional para cada candidato, que incluya aprendizajes cuidadosamente seleccionados con los mejores diseñadores, episodios de educación formal avanzada y cursos cortos, todo ello intercalado con tareas de diseño en solitario y liderazgo técnico.
- Proporcionar oportunidades para que los diseñadores en crecimiento interactúen y se estimulen mutuamente.

■