No Hay Balas de Plata

Esencia y Accidentes de a Ingeniería de Software*

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

www.apl.jhu.edu/Classes/635431/felikson/Publications/No_Silver_Bullet_Brooks.html

Traducido de IEEE Computer
Abril de 1987 por Pablo A. Straub

Índice

Índice 2
¿Tiene que ser difícil? Dificultades esenciales 3
Complejidad 3
Adaptabilidad 4
Modificabilidad 4
Invisibilidad 5
Los avances anteriores más importantes resolvieron dificultades accidentales 5
Lenguajes de alto nivel 5
Tiempo compartido. 5
Sistemas de programación unificados. 6
Esperanzas por las balas 6
Programación orientada a los objetos 6
Inteligencia artificial 7
Sistemas expertos 7
Programación "automática". 8
Programación gráfica 8
Verificación de programas 9
Ambientes y herramientas 9
Estaciones de trabajo 9
Ataques prometedores a la esencia conceptual 9
Comprar versus construir 10
Refinamiento de requerimientos y prototipos rápidos 10
Diseñadores brillantes 11
Agradecimientos 13
Referencias 14

De todos los monstruos que llenan las pesadillas de nuestro folklore, ninguno aterroriza más que los hombre lobo, porque se transforman inesperadamente de lo familiar al horror. Para ellos, buscamos balas de plata que puedan matarlos mágicamente.

El típico proyecto de software, por lo menos visto por el gerente no técnico, tiene algo de este carácter: usualmente es inocente y simple, pero puede transformarse en un monstruo de programas atrasados, presupuestos reventados y productos defectuosos. De modo que oímos gritos desesperados pidiendo balas de plata — algo que haga bajar los costos de software tan rápido como los costos de hardware—.

Pero, así como miramos el horizonte de aquí a una década, no vemos balas de plata. No hay ningún desarrollo, en técnicas tecnológicas ni administrativas, que prometa por si sólo ni siquiera una mejora de un orden de magnitud en productividad, en confiabilidad, en simplicidad. En este artículo trataré de explicar por qué, examinando tanto la naturaleza del problema de software como las propiedades de las balas propuestas.

Sin embargo, el escepticismo no es pesimismo. Si bien no vemos ningún cambio revolucionario —y de hecho, creo que son inconsistentes con la naturaleza del software— hay muchas innovaciones alentadoras en camino. Un esfuerzo disciplinado y consistente en desarrollar, propagar y explotar estas innovaciones puede dar de hecho una mejora de un orden de magnitud. No hay un Camino Real, pero hay un camino.

El primer paso hacia el control de las enfermedades fue el reemplazo de las teorías de los demonios por la teoría de los gérmenes. Ese sólo paso, el comienzo de la esperanza, en sí eliminó toda esperanza de soluciones mágicas. Se le dijo a los trabajadores que el progreso se haría paso a paso, con gran esfuerzo, y que un cuidado persistente e incesante debería ser pagado a una disciplina de limpieza. Así es con la ingeniería de software hoy día.

¿Tiene que ser difícil? Dificultades esenciales

No sólo no hay balas de plata a la vista, sino que la misma naturaleza del software hace poco probable que exista una: ningún invento podrá hacer para la productividad, confiabilidad y simplicidad del software, lo que la electrónica, transistores e integraciones a gran escala hicieron para el hardware.
No podemos esperar ganancias del doble cada dos años.
Primero, debemos observar que la anomalía no es que el progreso en el software sea muy lento, sino que el progreso en el hardware computacional es muy rápido. Ninguna tecnología desde el comienzo de la civilización ha visto una ganancia de seis órdenes de magnitud en 30 años. En ninguna otra tecnología uno puede escoger ganar en aumento del rendimiento o bien en reducción de costos. Estas ganancias se derivan de la transformación de la fabricación de computadores de una industria de armada a una industria de proceso.
Segundo, para ver la tasa de progreso que uno puede esperar de la tecnología de software, examinemos las dificultades de esa tecnología. Siguiendo a Aristóteles, yo las divido en esencia, las dificultades inherentes a la naturaleza del software, y accidente, aquellas dificultades que ahora existen en su producción, pero que no son inherentes.
La esencia de una entidad de software es una construcción de conceptos interrelacionados: grupos de datos, relaciones entre los datos, algoritmos e invocaciones de funciones. Esta esencia es abstracta en el sentido de que la construcción conceptual es la misma bajo muchas representaciones. No obstante es altamente precisa y ricamente detallada.
Yo creo que lo difícil de hacer software es la especificación, diseño y prueba de esta construcción conceptual, no el trabajo de representarla y verificar la fidelidad de la representación. Sí, aún cometemos errores de sintaxis; pero en la mayoría de los sistemas son insignificantes comparados con los errores conceptuales.
Si esto es cierto, la construcción de software va a ser siempre difícil. Inherentemente no hay balas de plata.
Consideremos ahora las propiedades de esta esencia irreducible de los sistemas de software modernos: complejidad, adaptabilidad, modificabilidad e invisibilidad.

Complejidad. Las entidades de software son más complejas para su tamaño que tal vez cualquiera otra obra humana, porque no hay dos partes iguales (por lo menos más arriba que al nivel de las instrucciones). Si las hay, hacemos de las dos partes similares en un subprograma, ya sea abierto o cerrado. En ese sentido, los sistemas de software difieren profundamente de los computadores, edificios

o automóviles, en donde abundan los elementos repetidos.
Las computadoras digitales son a su vez más complejas que la mayoría de las cosas que construye la gente: tienen un número muy grande de estados. Esto hace que sea difícil concebirlas, describirlas y probarlas. Los sistemas de software tienen un orden de magnitud de estados mayor que las computadoras.
De igual manera, la construcción de software a gran escala no es simplemente una repetición de los mismos elementos en tamaños mas grandes, sino que se necesita aumentar el número de elementos diferentes. Casi siempre los elementos interactúan en alguna forma no lineal, y la complejidad del todo aumenta mucho más que linealmente.
La complejidad del software es una propiedad esencial, no accidental. Por lo tanto, las abstracciones de una entidad de software que hacen abstracción de su complejidad a menudo abstraen su esencia. Durante tres siglos, las ciencias físicas y matemáticas hicieron avances construyendo modelos simplificados de fenómenos complejos, derivando propiedades de los modelos y verificando experimentalmente esas propiedades. Este paradigma funcionó porque las complejidades que se ignoraban en el modelo no eran las complejidades esenciales de los fenómenos. No funciona cuando las complejidades son la esencia.
Muchos de los problemas clásicos del desarrollo de software se derivan de esta complejidad esencial y de su aumento no lineal en el tamaño. De la complejidad viene la dificultad de comunicación entre los miembros del equipo, que lleva a deficiencias del producto, exceso de costo, atrasos. De la complejidad viene la dificultad de enumerar todos los estados posibles de un programa; y de ahí viene la poca confiabilidad. De la complejidad de una función viene la complejidad de invocar la función, lo que hace a los programas difíciles de usar. De la complejidad de la estructura viene la dificultad de extender los programas con nuevas funciones sin crear efectos laterales. De la complejidad de la estructura vienen los estados no visualizados que constituyen problemas de seguridad.
No sólo problemas técnicos, sino también administrativos vienen de la complejidad. Esta hace difíciles las revisiones, impidiendo integridad conceptual. Hace difícil encontrar y controlar todos los cabos sueltos. Crea un problema tremendo de aprendizaje y entendimiento que hace un desastre de cada cambio en el personal.

Adaptabilidad. La gente de software no es la única que encara la complejidad. La física manipula objetos sumamente complejos al nivel "fundamental" de las partículas. Sin embargo, los físicos trabajan sobre la firme base de la existencia de principios unificadores, ya sea en quarks o en teorías de campo unificadas. Einstein afirmaba que deben haber explicaciones simples de la naturaleza, porque Dios no es caprichoso o arbitrario.
Ninguna fe similar apoya al ingeniero de software. La mayoría de la complejidad que él debe manejar es arbitraria, forzada sin razón alguna por la multiplicidad de instituciones y sistemas humanos cuyas interfaces deben ser respetadas. Ellas difieren de interfaz en interfaz y cambian con el tiempo, no debido a necesidad, sino sólo debido a que fueron diseñadas por diferentes personas en lugar de Dios.
En muchos casos, el software debe adaptarse porque es el último en llegar. En otros, debido a que se percibe como el más adaptable. Pero en todos los casos la adaptación a otras interfaces crea mucha complejidad; esta complejidad no se puede eliminar sólo rediseñando el software.

Modificabilidad. La entidad de software está constantemente bajo presión para cambiar. Por supuesto, esto también pasa con los edificios, autos y computadoras. Pero las cosas armadas raramente se modifican después de manufacturadas: se reemplazan por modelos nuevos, o bien cambios esenciales se incorporan en las nuevas copias del mismo diseño básico. Las modificaciones de los automóviles ya vendidos son muy poco frecuentes; los cambios en las computadoras son un poco más frecuentes. Ambos son mucho menos frecuentes que las modificaciones del software ya instalado.
Esto es en parte porque el software de un sistema incluye su función, y la función es la parte que siente más las presiones de cambio. Esto es en parte porque el software se puede modificar más fácilmente: es un producto en materia de pensamiento puro, infinitamente maleable. Los edificios también se modifican, pero el alto costo de las modificaciones, entendido por todos, sirve para amortiguar los ánimos de quienes quieren cambios.
Todo software exitoso se cambia. Hay dos procesos simultáneos. Primero, en la medida en que el software prueba ser útil, la gente usa casos que estaban en el límite o más allá del campo de aplicación inicial. Las presiones para extender la funcionalidad vienen principalmente de los usuarios que les gusta la funcionalidad básica e inventan nuevos usos para ella.
Segundo, el software exitoso sobrevive la vida normal de la máquina para la cual fue escrito. Si no son nuevas computadoras, por lo menos son nuevos discos, nuevas pantallas, nuevas impresoras; y el software debe adaptarse a estos nuevos vehículos.
En resumen, el producto de software está inmerso en una matriz cultural de aplicaciones, usuarios, leyes y máquinas en los cuales están embebidos. Todos estos elementos cambian continuamente y sus cambios inexorablemente fuerzan modificaciones al producto de software.

Invisibilidad. El software es invisible y no visualizable. Las abstracciones geométricas son herramientas poderosas. El plano de planta de un edificio le ayuda al arquitecto y al cliente a evaluar espacios, flujos de tránsito, vistas. Las contradicciones y omisiones se hacen evidentes.
Los dibujos a escala de partes mecánicas y los modelos de palitos para las moléculas, si bien son abstracciones, sirven para lo mismo. Una realidad geométrica se captura con una abstracción geométrica.
La realidad del software es inherentemente no espacial. Por consiguiente, no hay representaciones geométricas claras, del modo que la tierra tiene mapas, los circuitos de silicio tienen diagramas, las computadoras tienen esquemas de conexión. Apenas intentamos hacer un diagrama de la estructura del software, nos damos cuenta de que no constituye uno, sino muchos grafos dirigidos generales superpuestos. Estos grafos pueden representar el flujo de control, el flujo de datos, dependencias, secuencias de tiempo, relaciones nombre-espacio. Estos grafos ni siquiera son planos, y mucho menos jerárquicos. De hecho, una manera de obtener control conceptual sobre la estructura es el corte forzoso de arcos hasta que uno o más de los grafos sean jerárquicos.
A pesar del progreso en las restricciones y simplificaciones de las estructuras del software, éstas son inherentemente no visualizables, y por ende impiden usar una de las herramientas conceptuales más poderosas de la mente. Esto no sólo dificulta el proceso de diseño en nuestra mente, sino que dificulta severamente la comunicación entre las mentes.

Los avances anteriores más importantes resolvieron dificultades accidentales

Si examinamos los tres pasos más fructíferos en la tecnología de desarrollo de software, nos damos cuenta de que cada uno atacó una dificultad importante en la construcción de software, pero que todas esas dificultades han sido accidentales, no esenciales. Además podemos ver los límites naturales que hay al extrapolar esos ataques.

Lenguajes de alto nivel. Ciertamente el ataque más efectivo a la productividad, confiabilidad y simplicidad del software ha sido el uso progresivo de lenguajes de alto. La mayoría de los observadores le adjudican por lo menos un factor de cinco en productividad, con ganancias adicionales en confiabilidad, simplicidad y comprensibilidad.
¿Qué logran los lenguajes de alto nivel? Liberan a un programa de la mayoría de su complejidad accidental. Un programa abstracto consta de construcciones conceptuales: operaciones, tipos de datos, secuencias y comunicación. El programa concreto de la máquina se preocupa de bits, registros, condiciones, saltos, canales, discos, etc. En la medida en que los lenguajes de alto nivel representan las construcciones que uno desea en el programa abstracto y evitan los de bajo nivel, eliminan un nivel completo de complejidad que nunca fue inherente al programa.
Lo más que un lenguaje de alto nivel puede hacer es proporcionar todas las construcciones que un programador pueda imaginarse su programa abstracto. Ciertamente el nivel de nuestro pensamiento acerca de las estructuras de datos, tipos de datos y las operaciones aumenta constantemente, pero con una velocidad decreciente. Y los desarrollos en los lenguajes se acercan más y más a la sofisticación de los usuarios.
Además, llega un momento en que la elaboración de los lenguajes de alto nivel crea un problema de manejo de la herramienta que aumenta, en lugar de disminuir, el esfuerzo intelectual del usuario que raramente usa las construcciones esotéricas.

Tiempo compartido. El uso de sistemas de tiempo compartido trajo una mejora importante en la productividad de los programadores y en la calidad de sus productos, claro que no tan grande como la de los lenguajes de alto nivel.
El tiempo compartido ataca una dificultad muy diferente. El tiempo compartido mantiene la inmediatez y por ende permite mantener control de la complejidad. El tiempo de respuesta muy lento de los sistemas de programación batch implica que uno invariablemente se olvida de los detalles, si es que no de lo más importante, de lo que uno estaba pensando cuando dejó de programar e invocó al compilador y a la ejecución. Esta interrupción cuesta tiempo, porque uno debe refrescarse la memoria. El efecto más grave puede ser la decadencia de la comprensión de todo lo que está sucediendo en un sistema complejo.
Un tiempo de respuesta muy bajo, así como las complejidades del lenguaje de máquina, es una dificultad accidental y no esencial del proceso de software. Los límites de la contribución potencial del tiempo compartido se deducen directamente. El principal efecto del tiempo compartido es reducir el tiempo de respuesta. En la medida que el tiempo de respuesta se acerca a cero, éste pasa el nivel de percepción humana, alrededor de los 100 milisegundos. Más allá de ese punto, no deben esperarse

beneficios.

Sistemas de programación unificados. Unix e Interlisp, los primeros sistemas integrados de programación usados ampliamente, parecen haber mejorado la productividad por algún factor entero ¿Por qué?
Ellos atacan las dificultades accidentales que resultan de usar programas individuales en conjunto, entregando bibliotecas integradas, y tuberías y filtros.
Como resultado, las estructuras conceptuales que en principio pueden siempre llamar, alimentar y usar otras estructuras, pueden de hecho hacerlo fácilmente en la práctica.
Este importante avance, a su vez, estimuló el desarrollo de juegos de herramientas completos, desde que cada nueva herramienta puede ser aplicada a cualquier programa que use los formatos estándares.
Debido a su éxito, los ambientes son objeto de una gran parte de la investigación actual en la ingeniería de software. Veremos sus promesas y limitaciones en la sección siguiente.

Esperanzas por las balas

Examinemos ahora los desarrollos técnicos que más frecuentemente se consideran potenciales balas de plata. ¿Qué problemas atacan? ¿Los problemas de esencia, o los problemas accidentales restantes? ¿Ofrecen avances revolucionarios, o avances incrementales?
Uno de los desarrollos recientes más promocionados es Ada, un lenguaje de alto nivel de uso general de los años ochenta. Ada no sólo refleja mejoras evolutivas en conceptos de lenguajes, sino que de hecho incluye elementos que incentivan el diseño moderno y la modularización. Tal vez la filosofía de Ada es más avanzada que el lenguaje Ada, porque es la filosofía de la modularización, de los tipos abstractos de datos, de la estructuración jerárquica. Ada es excesivamente rico, como resultado natural del proceso por el que se determinaron los requerimientos de su diseño. Esto no es terrible, porque el uso de subconjuntos puede resolver el problema de aprendizaje, y los avances en hardware no darán MIPS baratos para pagar los costos de compilación. El avance de la estructuración del software es de hecho un excelente uso de los MIPS que nuestros dólares pueden comprar. Los sistemas operativos, altamente criticados en los sesenta por su costo en memoria y ciclos de procesador, han demostrado ser una excelente manera de usar algunos de los ciclos y bytes baratos[1].
Sin embargo, Ada no demostrará ser la bala de plata que puede matar al monstruo de la productividad de software. Es, después de todo, un lenguaje de alto nivel más, y la mayor rentabilidad de estos lenguajes viene de la primera transición: la transición desde las complejidades accidentales de la máquina hacia el planteamiento más abstracto de soluciones paso a paso. Una vez que estos accidentes se han removido, los restantes serán más pequeños, y la rentabilidad de su remoción ciertamente será menor.
Yo pronostico que de aquí a una década, cuando se evalúe la efectividad de Ada, se verá que ha hecho una diferencia sustancial, pero no debido a ningún atributo del lenguaje en particular, ni siquiera debido a todos ellos combinados. Tampoco los nuevos ambientes de Ada probarán ser la causa de las mejoras. La mayor contribución de Ada será que el cambio a Ada habrá ocasionado la capacitación de los programadores en técnicas modernas de diseño de software.

Programación orientada a los objetos. Muchos estudiantes del arte tienen más esperanza en la programación orientada a los objetos que en cualquiera de las otras modas tecnológicas del día [2]. Yo soy uno de ellos. Mark Sherman del Dartmouth College indica a través de la red de noticias CSnet que uno debe distinguir dos ideas distintas que se identifican con ese nombre: tipos abstractos de datos y tipos jerárquicos. El concepto de tipo abstracto de datos es que el tipo de un objeto debe ser definido por un nombre, un conjunto de valores y un conjunto de operaciones, en lugar de su estructura de almacenamiento la cual debe ocultarse. Ejemplos de esto son los paquetes de Ada (con tipos privados) y los módulos de Modula.
Los tipos jerárquicos, como las clases de Simula-7, permiten definir interfaces generales que pueden ser refinadas creando tipos subordinados. Estos dos conceptos son ortogonales —uno puede tener jerarquías sin ocultación y ocultación sin jerarquías—. Ambos conceptos representan avances reales en el arte de la construcción de software.
Cada uno remueve otra dificultad accidental más, permitiendo al diseñador expresar la esencia del diseño sin tener que expresar gran cantidad de material sintáctico que no agrega información. Tanto para los tipos abstractos como para los tipos jerárquicos, el resultado es remover una dificultad accidental de alto nivel y permitir una expresión del diseño de alto nivel.
Sin embargo, estos avances no pueden hacer más que remover todas las dificultades accidentales en la expresión del diseño. La complejidad del diseño mismo es esencial, y estos ataques no cambian eso. La programación orientada a objetos sólo puede producir ganancias de un Orden de magnitud si el lastre

de especificación de tipos que hay aún en nuestro lenguaje de programación es nueve décimos del trabajo de diseño de un programa. Yo lo dudo.

Inteligencia artificial. Mucha gente espera que los avances en la inteligencia artificial proveerán el cambio revolucionario que dará ganancias de órdenes de magnitud en la productividad y calidad del software [3]. Yo no. Para ver por qué, debemos diseccionar qué se entiende por "inteligencia artificial". D. L. Parnas ha clarificado el caos terminológico [4]:

> Hay dos definiciones bien distintas de la inteligencia artificial en uso común. IA-1: El uso de computadores para resolver problemas que antes sólo se podían resolver usando inteligencia humana. IA-2: El uso de un conjunto específico de técnicas de programación conocido como heurísticas o programación basada en reglas. En este enfoque estudian a los humanos expertos para determinarse qué heurística o reglas utilizan en solucionar problemas... El programa se diseña para resolver un problema como lo hacen los humanos.
> La primera definición tiene un significado movedizo... Algo puede calzar en la definición IA-1 hoy día, pero una vez que vemos como funciona el programa y entendemos el problema, pensamos que ya no es más inteligencia artificial... Desgraciadamente, yo no puedo identificar una tecnología que sea propia de este campo... La mayoría del trabajo es específico de cada problema, y se necesita abstracción o creatividad para transferirlo.

Estoy completamente de acuerdo con esta crítica. Las técnicas usadas para el reconocimiento del lenguaje hablado parecen tener poco en común con las usadas para el reconocimiento de imágenes, y ambas difieren de las usadas en los sistemas expertos. Me parece difícil ver cómo, por ejemplo, el reconocimiento de imágenes hará alguna diferencia apreciable en la práctica de la programación. Lo mismo pasa con el reconocimiento del lenguaje hablado. Lo más difícil en la construcción de software es decidir lo que uno quiere decir, no decirlo. Ninguna ayuda de expresión puede dar más que ganancias marginales.
La tecnología de sistemas expertos, IA-2, merece una sección por sí sola.

Sistemas expertos. La parte más avanzada del arte de la inteligencia artificial, y la más aplicada, es la tecnología para construir sistemas expertos. Muchos científicos de software están trabajando duro para aplicar esta tecnología a los ambientes de construcción de software [3] [5] ¿Cuál es el concepto y cuáles son las expectativas?
Un sistema experto es un programa que contiene un motor de inferencias generalizada y una base de reglas, obtiene datos de entrada y supuestos, explora las inferencias deducibles de la base de reglas, entrega conclusiones y consejos, y ofrece explicar sus resultados rehaciendo su razonamiento para el usuario. Los motores de inferencia típicamente pueden usar datos y reglas poco precisos o probabilísticos, 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 independientemente de la aplicación y luego es aplicada en muchos casos. De hecho, esta tecnología está bastante avanzada.
- Las partes intercambiables particulares a cada aplicación se codifican uniformemente en una base de reglas, y se proporcionan herramientas para desarrollar, cambiar, verificar y documentar la base de reglas. Esto regulariza una buena parte de la complejidad de la aplicación misma.

El poder de estos sistemas no proviene de mecanismos de inferencia cada vez más intrincados, sino de bases de conocimientos cada vez más ricas que reflejan el mundo real con más precisión. Yo creo que el avance más importante ofrecido por esta tecnología es la separación de la complejidad de la aplicación del programa mismo.
¿Cómo se puede aplicar esta tecnología a la ingeniería de software? De muchas maneras: Estos sistemas pueden sugerir reglas de interfaz, aconsejar sobre estrategias de prueba, recordar frecuencias de los tipos de errores, y dar ideas para optimizar.
Por ejemplo, considere un consejero de pruebas imaginario. En su forma más rudimentaria, el sistema experto de diagnóstico es como la lista de chequeo de un piloto, que sólo enumera sugerencias de posibles causas de los problemas. En la medida en que se incluye más estructura del sistema en la base de reglas y ésta considera los síntomas de problemas en forma más sofisticada, el consejero de pruebas se hace más específico en las hipótesis que genera y en las pruebas que recomienda. Este sistema experto puede alejarse radicalmente de los más convencionales porque su base de reglas probablemente debería estar modularizada jerárquicamente del mismo modo que el producto de software lo está, así que si el producto se modifica en forma modular, la base de reglas de diagnóstico también debe modificarse en forma modular.

El trabajo necesario para generar las reglas de diagnóstico se haría de todas maneras al generar los casos de prueba para los módulos y el sistema. Si esto se hace en forma adecuadamente general, con una estructura uniforme para las reglas y una buena máquina de inferencias, se puede reducir el trabajo total de generar casos de prueba, ayudando además en las pruebas de mantenimiento y modificaciones durante la vida del sistema. Del mismo modo, uno puede proponer otros consejeros, probablemente muchos y simples, para otras partes del proceso de construcción de software.
Hay muchos factores que dificultan realizar pronto sistemas expertos consejeros útiles para los desarrolladores de software. Una parte crusial de nuestro escenario imaginario es el desarrollo de métodos sencillos para pasar de una especificación de la estructura de un programa a la generación automática o semiautomática de reglas de diagnóstico. Más difícil e importante aún es el trabajo doble de adquisición de conocimiento: encontrar expertos introspectivos que saben por qué hacen las cosas, y desarrollar técnicas eficientes para extraer lo que saben y representarlo en la base de reglas. El requisito esencial para construir un sistema experto es tener un experto.
La contribución más poderosa de los sistemas expertos sin duda será poner la experiencia y sabiduría acumulada de los mejores programadores al servicio de los programadores sin experiencia. Esta no es una contribución pequeña. La diferencia entre la mejor práctica de la ingeniería de software y la promedio es muy grande —tal vez más grande que en cualquiera otra disciplina de la ingeniería—. Una herramienta que disemina buena práctica sería importante.

Programación "automática". Por más de 40 años la gente ha estado anticipando y escribiendo acerca de la "programación automática", o la generación de un programa para resolver un problema a partir de una especificación del mismo. Algunos escriben hoy día como si esperaran que esta tecnología fuera a proveer el siguiente cambio revolucionario [5].
Parnas [4] indica que el término se ha usado más por su encanto que por su contenido semántico.
En resumen, "programación automática" ha sido siempre un eufemismo para la programación en un lenguaje de más alto nivel que el disponible actualmente por el programador.
Él afirma, en esencia, que en la mayoría de los casos es el método de solución, no el problema, lo que debe especificarse.
Uno puede encontrar excepciones. La técnica de construcción de generadores es muy poderosa, y se usa rutinariamente con buenos resultados en programas para ordenamiento. Algunos sistemas para integrar ecuaciones diferenciales también permiten especificar el problema directamente, y los sistemas evalúan los parámetros, eligen el método de solución de una biblioteca, y generan los programas.
Estas aplicaciones tienen muchos atributos favorables:

- Los problemas se pueden caracterizar por relativamente pocos pará metros.
- Hay varios métodos conocidos de solución, de modo de tener una biblioteca de opciones.
- Un análisis extensivo ha creado reglas explícitas para elegir el método de solución, dados los parámetros del problema.

Es difícil ver cómo se pueden generalizar estas técnicas al amplio campo de los sistemas de software, en donde los casos con esas bellas propiedades son la excepción. Es difícil incluso imaginar cómo podría ocurrir una gran generalización.

Programación gráfica. Un tema favorito para las tesis de doctorado en ingeniería de software es la programación gráfica o visual —la aplicación de gráficas computacionales al diseño de software. [6][7]
A veces se muestra la promesa de este enfoque por analogía al diseño de circuitos de integración en gran escala, donde las gráficas computacionales juegan un papel tan fructífero. Algunas veces el teórico justifica su enfoque considerando los diagramas de flujo como el medio ideal para diseñar programas y entregando poderosas herramientas para construirlos.
Nada convincente, mucho menos excitante, ha salido aún de esos esfuerzos. Estoy convencido de que nada saldrá.
En primer lugar, como yo ya he sostenido, [8] el diagrama de flujo es una representación muy pobre de la estructura del software. De hecho, se ve mejor como los intentos de Burks, von Neumann y Goldatine de proveer un lenguaje de control de alto nivel desesperadamente necesario para su computador propuesto. En la forma penosa, de muchas páginas de recuadros conectados, que tienen los diagramas de flujo actuales han probado ser inútil como una herramienta de diseño —los programadores dibujan los diagramas de flujo después, no antes, de escribir los programas que éstos describen—.
Segundo, las pantallas de hoy en día son muy pequeñas, en pixeles, para mostrar tanto el alcance como la resolución de cualquier diagrama de software seriamente detallado. La llamada "metáfora del escritorio" de las estaciones de trabajo es más bien un metáfora de "asiento de avión" - Cualquiera que haya revuelto un montón de papeles en sus rodillas sentado entre dos pasajeros corpulentos reconocerá la diferencia: uno puede ver sólo unas pocas cosas a la vez. Un escritorio de verdad permite mirar y escoger aleatoriamente entre muchas páginas. La tecnología de hardware deberá avanzar substancialmente antes de que la vista en nuestras ventanas sea suficiente para diseñar software.

Más importante, como ya indiqué, el software es muy difícil de visualizar. Independientemente de que uno haga diagramas de flujo de control, anidación del alcance de las variables, referencias de variables, flujo de datos, estructuras de datos jerárquicas, o lo que sea, uno siente sólo una dimensión del elefante de software intrincadamente interrelacionado. Si uno superpone todos los diagramas generados por muchas de estas vistas relevantes, es difícil extraer una visión global. La analogía de los circuitos de integración en gran escala es fundamentalmente conducente a error —el diseño de un circuito integrado es una descripción bidimensional en niveles cuya geometría refleja su realización tridimensional—. El de un sistema de software no.

Verificación de programas. Una buena parte del esfuerzo de la programación moderna se dedica a la prueba de programas y a la corrección de errores. ¿Es posible encontrar una bala de plata eliminando los errores en su origen, en la fase de diseño del sistema? ¿Es posible mejorar radicalmente la productividad y la confiabilidad de los productos siguiendo la estrategia profundamente diferente de demostrar que los diseños están correctos antes de hacer grandes esfuerzos en implementarlos y luego probarlos?
Yo no creo que vayamos a encontrar aquí productividad mágica. La demostración de programas es un concepto poderoso, y va a ser muy importante en cosas como núcleos de sistemas operativos seguros. Sin embargo, la tecnología no promete ahorrar trabajo. Las demostraciones son tan trabajosas que sólo unos pocos programas substanciales han sido demostrados.
La demostración de programas no significa programas libres de error. Aquí tampoco hay magia. Las demostraciones matemáticas también pueden ser erróneas. De modo que si bien las demostraciones pueden reducir el trabajo de prueba de programas, no lo pueden eliminar.
Peor aún, incluso una demostración de programas perfecta sólo puede establecer que un programa satisface su especificación. La parte más difícil del trabajo de software es llegar a una especificación completa y consistente; de hecho, una buena parte de la esencia de la construcción de un programa es el corregir la especificación.

Ambientes y herramientas. ¿Cuánta ganancia adicional se puede esperar de las muchas investigaciones en ambientes de programación? Nuestra reacción instintiva es que los problemas con mayor rendimiento -sistemas jerárquicos de archivos, formatos de archivos uniformes para permitir interfaces entre programas, y herramientas generalizadas- ya fueron atacados y fueron resueltos. Editores inteligentes orientados a un lenguaje son desarrollos no muy usados en la práctica, pero lo más que prometen es liberarse de los errores sintácticos y de los errores semánticos simples.
Tal vez la ganancia más grande que aún no se ha obtenido de los ambientes de programación es el uso de bases de datos integradas para controlar la gran cantidad de detalles que necesita saber un programador individual y que deben mantenerse al día por un grupo de colaboradores en un sistema.
Con certeza este trabajo vale la pena y dará frutos tanto en productividad como confiabilidad. Pero por su propia naturaleza, las utilidades de aquí en adelante deben ser marginales.

Estaciones de trabajo. ¿Qué ganancias pueden esperarse para el arte del software a partir del aumento cierto y rápido en la capacidad de las estaciones de trabajo individuales? Bueno, ¿cuántos millones de instrucciones por segundo puede uno usar con provecho? La composición y edición de programas se hace perfectamente bien con las velocidades actuales. La compilación podría aumentar de velocidad, pero un factor de 10 en la velocidad hará que con certeza el pensar sea la actividad dominante de los programadores. De hecho, ya parece serlo.
Claro que le damos la bienvenida a las estaciones de trabajo más poderosas. Pero no podemos esperar encantos mágicos de ellas.

Ataques prometedores a la esencia conceptual

Si bien ningún cambio tecnológico promete el tipo de resultados mágicos con los que estamos familiarizados en el área de hardware, hay tanto una abundancia de buenos trabajos llevándose a cabo, y una promesa de progreso constante, si bien no espectacular.
Todos estos ataques tecnológicos a los accidentes del software están limitados por la ecuación de la productividad:

$$\text{Tiempo\_de\_ejecución} = \sum_i (\text{frecuencia})_i \times (\text{tiempo})_i$$

Si, como yo creo, los componentes conceptuales del trabajo ya toman la mayoría del tiempo, entonces ninguna cantidad de actividad en los componentes del trabajo que son meramente la expresión de los conceptos dará grandes ganancias en la productividad.
Por consiguiente, debemos considerar aquellos ataques que se concentran en la esencia del problema

del software, la formulación de esas estructuras conceptuales complejas. Afortunadamente, algunos de esos ataques son muy prometedores.

Comprar versus construir. La solución más radical para la construcción de software es no construirlo en su totalidad.
Esto resulta cada día más fácil, mientras más y más empresas ofrecen más y mejores productos de software para una mareadora variedad de aplicaciones. Mientras los ingenieros de software hemos trabajado en las metodologías de producción, la revolución de las computadoras personales ha creado no uno, sino muchos mercados masivos de software. Cada kiosco vende revistas mensuales específicas para cada tipo de máquina, los cuales publicitan y revisan docenas de productos a precios desde unos pocos dólares a unos cientos de dólares. Fuentes más especializadas ofrecen productos muy poderosos para las estaciones de trabajo y otros mercados de Unix. Incluso es posible comprar herramientas y ambientes de programación. Yo he propuesto en otro artículo que se cree un mercado de módulos individuales. [9]
Cualquiera de esos productos es más barato comprarlo que hacerlo. Incluso si cuesta cien mil dólares, una pieza de software comprada es sólo como un año-programador ¡Y con entrega inmediata! Inmediata, por lo menos para los productos que en realidad existen. Es mas, estos productos tienden a estar mucho mejor documentados y algo mejor mantenidos que el software hecho en casa.

En mi opinión, el desarrollo del mercado masivo es la tendencia de largo plazo más profunda en la ingeniería de software. El costo del software siempre ha sido el costo de desarrollo, no el costo de duplicación. El compartir ese costo, incluso entre unos pocos usuarios, baja radicalmente el costo por usuario. Otra manera de verlo es que el uso de *n* copias de un sistema de software realmente multiplica la productividades de sus desarrolladores por *n*. Este es un aumento de la productividad de la disciplina y del país.
La clave, por supuesto que es la aplicabilidad. ¿Puedo usar un paquete disponible en el mercado para hacer mi trabajo? En esto sucedió algo sorprendente. En los años cincuenta y sesenta, estudio tras estudio demostraba que los usuarios no comprarían paquetes hechos para planillas de pagos, control de inventarios, cuentas por cobrar, etc. Los requerimientos eran muy especializados, las variaciones caso a caso eran muy grandes. En los ochenta encontramos que estos paquetes tienen gran demanda y uso. ¿Qué ha cambiado?
En realidad, no los paquetes. Estos podrán ser algo más generalizados y algo más adaptables al usuario que antes, pero no mucho más. Tampoco las aplicaciones. Si algo ha cambiado, son las necesidades de los negocios y la ciencia de ahora son más diversas y complejas que hace 20 años.
El gran cambio ha sido en la razón de costo hardware/software. En 1960, el comprador de una máquina de dos millones de dólares sentía que podía pagar US$250.000 más por un programa de planillas de pago hecho a la medida; uno que se deslizara fácilmente y sin interrupciones en un ambiente social hostil a los computadores. Hoy en día, el comprador de una máquina de US$50.000 no puede darse el lujo de un sistema de planillas de pago a la medida, de modo que él adapta el procedimiento de pagos a los paquetes disponibles. Las computadoras son ya tan comunes, si no tan amados, que las adaptaciones son simplemente aceptadas.
Hay excepciones dramáticas a mi argumento de que la generalización del software ha cambiado poco a través de los años: las hojas de cálculo electrónicas y sistemas simples de bases de datos. Estas poderosas herramientas, que ahora parecen obvias pero que demoraron tanto en aparecer, se prestan para una multiplicidad de usos, algunos bastante poco ortodoxos. Abundan artículos e incluso libros de cómo resolver problemas inesperados con una hoja de cálculo. Grandes cantidades de aplicaciones que antes se habrían escrito en Cobol o Report Program Generator se hacen ya rutinariamente con esas herramientas.
Muchos usuarios usan sus propias computadoras todo el día en varias aplicaciones sin escribir nunca un programa. De hecho, muchos de esos usuarios no pueden escribir programas para sus máquinas, pero aún así son capaces de resolver nuevos problemas con ellas.
Yo pienso que la estrategia más poderosa por si sola para la productividad del software es darle a todos los trabajadores intelectuales a punto de ser despedidos y que no saben computación, computadoras personales con buenos programas para escribir, dibujar, archivar, hacer hojas de cálculo, etc., y luego soltarlos. La misma estrategia, llevada a cabo con paquetes matemáticos y estadísticos generalizados y con algunas capacidades de programación, puede funcionar con cientos de científicos de laboratorio.

Refinamiento de requerimientos y prototipos rápidos. Lo más difícil en la construcción de un sistema de software es decidir exactamente qué construir. Ninguna otra parte del trabajo conceptual es tan difícil como establecer los requerimientos técnicos detallados, incluyendo todas las interfaces con las personas, las máquinas y otros sistemas de software. Ninguna otra parte inutiliza tanto al sistema resultante si se hace mal. Ninguna otra parte es más difícil de corregir después.

Por consiguiente, la labor más importante que hace un constructor de software para su cliente es la extracción iterada y el refinamiento de los requerimientos del producto. Porque la verdad es que el cliente no sabe lo que quiere. El cliente usualmente no sabe qué respuestas debe responder, y casi nunca ha pensado en el problema con el nivel de detalle necesario para especificarlo. Incluso la simple respuesta "Haga que nuestro nuevo programa funcione como nuestro antiguo sistema manual de procesamiento de información"— es de hecho muy simple. Uno nunca quiere exactamente eso. Además, los sistemas de software complejos son entes que actúan, se mueven, trabajan. De modo que al planificar cualquier actividad de diseño de software es necesario considerar una larga iteración entre el cliente y el diseñador, como parte de la definición del sistema.
Yo daría un paso más y afirmaría que en realidad es imposible que el cliente, aún trabajando con un ingeniero de software, especifique completa, precisa y correctamente los requerimientos exactos de un producto de software antes de probar algunas versiones del mismo.
Por consiguiente, uno de los esfuerzos tecnológicos actuales más importantes, y uno que ataca la esencia en lugar de los accidentes del problema de software, es el desarrollo de enfoques y herramientas para hacer prototipos rápidamente, porque la construcción de prototipos es parte de la especificación iterativa de requerimientos.
Un sistema de software prototipo es uno que simula las interfaces más importantes y ejecuta las funciones más importantes del sistema deseado, sin que esté necesariamente restringido por la misma velocidad, tamaño y costo. Usualmente los prototipos hacen el trabajo más típico, pero no intentan manejar los casos excepcionales, ni responder correctamente a los errores de ingreso, ni terminar la ejecución limpiamente. El objetivo del prototipo es hacer real la estructura especificada de modo que el cliente pueda probar su consistencia y usabilidad.
La mayoría de los procedimientos de adquisición de software actuales se basan en el supuesto de que uno puede especificar de antemano un sistema satisfactorio, conseguir ofertas para su construcción, mandarlo a hacer e instalarlo. Yo pienso que este supuesto está fundamentalmente equivocado y que muchos de los problemas de la adquisición de software se originan en esa falacia. Entonces, éstos no pueden cambiarse sin una revisión fundamental —una revisión que incluya desarrollo iterativo y especificación de prototipos y productos—.

Desarrollo incremental: Cultive, no construya, el software. Aún recuerdo la sorpresa que sentí en 1958 la primera vez que oía un amigo hablar de construir un programa, en lugar de escribirlo. Instantáneamente, él me amplió mi visión global del proceso de software. El cambio de metáfora fue poderoso y preciso. Ahora entendemos cuánto se parece la construcción de software a otros procesos de edificación, y usamos libremente otros elementos de la metáfora, como especificación, armado de componentes y andamiaje.
La metáfora del edificio ha sobrevivido su utilidad. Es hora de volver a cambiar. Si, como yo creo, las estructuras conceptuales son muy complicadas para ser especificadas de antemano y muy complejas para ser construidas sin errores, entonces debemos adoptar un enfoque radicalmente diferente.
Veamos ahora la naturaleza y estudiemos la complejidad de los seres vivientes, en vez de las obras inertes del hombre. Allí encontramos construcciones cuya complejidad nos emocionan de admiración. El cerebro por sí solo es más complejo de lo imaginable, más poderoso de lo imitable. rico en diversidad, autoprotegido y autorenovado. El secreto es que se cultiva, no se construye.
Así debe ser con nuestros sistemas de software. Algunos años atrás, Harían Milis propuso que el software debe cultivarse por desarrollo incremental [10]. Es decir, primero el sistema debe hacerse funcionar, aún si no hace nada útil excepto llamar a un conjunto adecuado de subprogramas vacíos, aún no implementados. Luego, paso a paso, se le debe dar forma, desarrollando los subprogramas —en acciones o llamados a los subprogramas vacíos de nivel inferior—.
He visto resultados muy espectaculares desde que empecé a insistir en el uso de esta técnica en mi curso de Laboratorio de Ingeniería de Software. Nada en la última década ha cambiado tan radicalmente mi propia práctica o su efectividad. Este enfoque necesita un diseño de arriba hacia abajo, porque es un crecimiento del software de arriba hacia abajo; facilita la marcha atrás en las decisiones: se presta para tener prototipos desde el comienzo. Cada función adicional y cada adaptación para obtener datos más complejos crece orgánicamente de lo que ya existe.
Los efectos morales son sorprendentes. El entusiasmo aflora cuando hay un sistema ejecutable, incluso uno simple. Los esfuerzos se redoblan cuando aparece la primera figura en un nuevo software gráfico, incluso si es sólo un rectángulo. Uno siempre tiene, en cada etapa del proceso, un sistema funcionando. Yo encuentro que los equipos pueden cultivar (o hacer crecer) mucho más software del que pueden construir.
Los mismos beneficios de mis sistemas pequeños se pueden obtener en uno grande [11].

Diseñadores brillantes. La cuestión central de cómo mejorar el arte del software se centra, como siempre, en la gente.
Podemos obtener buenos diseños siguiendo métodos buenos en lugar de malos. El buen diseño siempre

puede enseñarse. Los programadores están entre las personas más inteligentes de la población, de modo que pueden aprender buenos métodos. Por consiguiente, un impulso importante en los Estados Unidos es la promulgación de buenos métodos modernos. Nuevos programas de estudio, nueva literatura, nuevas organizaciones, como el Instituto de Ingeniería de Software (SEI), todos ellos se han creado para elevar el nivel de nuestra práctica de malo a bueno. Esto es completamente apropiado.
Sin embargo, yo no creo que podamos dar el siguiente paso hacia adelante de la misma forma. Mientras que la diferencia entre diseños conceptuales malos y buenos puede deberse a lo adecuado del método de diseño, la diferencia entre los diseños buenos y los brillantes ciertamente no. Los diseños brillantes vienen de los diseñadores brillantes. La construcción de software es un proceso creativo. Los métodos adecuados pueden darle poder y libertad a la mente creativa; pero no pueden estimular o inspirar a un simple afanado.
Las diferencias no son menores —son más bien como las diferencias entre Salieri y Mozart—. Estudio tras estudio demuestra que los mejores diseñadores producen estructuras que son más rápidas, más pequeñas, más simples, más limpias y producidas con menor esfuerzo [12]. Las diferencias entre el brillante y el término medio se acercan a un orden de magnitud.
Un poco de retrospección nos muestra que si bien muchos sistemas de software buenos y útiles han sido diseñados por comités y construidos como partes de proyectos más grandes, aquellos sistemas de software que han excitado a fans apasionados son aquellos que son el producto de unos pocos diseñadores: diseñadores brillantes. Considere a Unix, APL, Pascal, Modula, la interfaz de Smalltalk, incluso Fortran; y contrástelos con Cobol, PL/I, Algol. MVS/370 y MS-DOS. (Vea la Tabla 1.)
Por consiguiente, si bien yo apoyo fuertemente los esfuerzos actuales de transferencia tecnológica y desarrollo de planes de estudio, pienso que el esfuerzo individual más importante que podemos hacer es desarrollar maneras para cultivar a diseñadores brillantes.
Ninguna organización de software puede ignorar este desafío. Los gerentes brillantes, siendo escasos, no son más escasos que los diseñadores brillantes. La mayoría de las organizaciones invierten un esfuerzo considerable en encontrar y cultivar a los candidatos a gerente; yo no sé de ninguna que invierta el mismo esfuerzo en encontrar y desarrollar a los diseñadores brillantes de quienes va a depender la excelencia técnica de los productos.

Tabla 1: Productos excitantes versus productos útiles pero no excitantes.

Productos Excitantes

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

Mi primera proposición es que cada organización de software debe determinar y proclamar que para su éxito los diseñadores brillantes son tan importantes como los gerentes brillantes, y se puede esperar que ellos sean cuidados y recompensados en forma similar. No sólo el sueldo, sino que los beneficios adicionales de reconocimiento —tamaño de oficina, muebles, equipamiento técnico, viáticos, apoyo de personal administrativo— deben ser completamente equivalentes.
¿Cómo uno puede cultivar diseñadores excelentes? El espacio no permite una larga discusión, pero algunos pasos resultan obvios:

- Identificar sistemáticamente a los mejores diseñadores tan pronto como sea posible. Los mejores suelen no ser los más experimentados.
- Asignar a un seguidor de la carrera como responsable del desarrollo del candidato y mantener cuidadosamente un archivo de su carrera.
- Crear y mantener un plan de desarrollo de carrera para cada candidato, incluyendo aprendizajes cuidadosamente seleccionados con los mejores diseñadores, etapas de educación formal avanzada y cursos cortos, todo esto mezclado con trabajos de diseño individual y de liderazgo técnico.
- Dar oportunidades a los diseñadores que se están desarrollando para interactuar y estimularse mutuamente.

Agradecimientos

Agradezco a Gordon Belí, Bruce Buchanan, Rick Hayes-Roth. Robert Patrick y especialmente a David Parnas por su entendimiento e ideas estimulantes, y a Rebekah Bierly por la producción técnica de este artículo.

Referencias

1. D. L. Parnas, "Designing Software for Ease of Extension and Contraction", IEEE Transactiona on Software Engineering. Volumen 5, Número 2, marzo de 1979.

2. G. Booch, "Object Oriented Design", Software Engineering with Ada, 1983, Benjamin-Cummings, Menlo Park, California.

3. IEEE Transactions on Software Engineering (número especial sobre inteligencia artificial e ingeniería de software), editor invitado. J. Mostow, volumen 11, número 11. noviembre de 1985.

4. D. L. Parnas, "Software Aspecta of strategic Defense systems", American Scientist, noviembre de 1985.

5. R. Balzer. "A 15-Year Perspective on Automatic Programming", IEEE Transactions on Software Engineering, volumen 11, número 11, noviembre de 1985.

6. IEEE Computer (número especial sobre programación visual), tares invitados, R. B. Graphton y 1. Ichikawa, agosto de 1985.

7. G. Raeder, "A Survey of Current Graphical Programming Techniques", IEEE Computer, agosto de 1985.

8. F. P. Brooks, The Mythical Man-Month, 1975, Addison-Wesley, Reading, Massachussetts, capítulo 14.

9. Defense Science Board, Report of the Task Force On Military Software, en imprenta.

10. H. D. Milis, "Top-Down Programming ín Large Systems", en Debugging Techniques in Large Systems, editor, R. Ruskin, Prentice-Hall, Englewood Clifis, New Jersey, 1971.

11. B. W. Boehm, "A Spiral Approach of Software Development and Enhancement", 1985, Informe Técnico TRW 21-371-85, TRW, Inc., 1 Space Park, Redondo Beach, California, 90278, EE.UU.

12. H. Sackman, W. J. Erickson y E. E. Gra nt, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance", CACM, volúmen 11, número 1, enero de 1968.