No hay balas de plata: Lo esencial y lo accidental en la Ingeniería del Software

by Frederick P. Brooks, Jr.

De todos los monstruos que pueblan nuestras pesadillas, ninguno es tan terrorífico como el hombre lobo, porque pasa repentinamente de lo familiar al horror. Por eso, todos buscamos balas de plata que puedan acabar con ellos mágicamente.

El familiar proyecto de software, al menos tal como lo ve un gestor no técnico, tiene algo de ese carácter: suele ser inocente y sencillo, pero es capaz de convertirse en un monstruo de plazos incumplidos, objetivos fallados y productos defectuosos. Por eso escuchamos lamentos clamando por una bala de plata -- algo que haga que los costes del software caigan tan rápidamente como lo han hecho los del hardware.

Pero no se ve en ningún lugar una bala de plata. No hay ningún desarrollo, ni en tecnología ni en técnicas de gestión, que por si sólo prometa ni siquiera una mejora en un orden de magnitud en productividad, en fiabilidad, en simplicidad. En este artículo, intentaré mostrar el porqué, examinando la naturaleza del problema del software y las propiedades de las balas propuestas.

Pero ser escéptico no es lo mismo que ser pesimista. Aunque no se vea la luz al final del túnel - y, de hecho, creo que es inconsistente con la naturaleza del software- se están realizando muchas innovaciones. Un esfuerzo consistente y disciplinado para desarrollar, difundir y explotar estas innovaciones debería conducir a una mejora de un orden de magnitud. No hay un camino dorado, pero hay un camino.

El primer paso hacia la cura de las enfermedades fue reemplazar las teorías sobre demonios y las teorías sobre humores por la teoría de los gérmenes. Ese primer paso, el principio de la esperanza, destruyó todas las esperanzas de una solución mágica. Se le dijo a los trabajadores que el progreso se haría paso a paso, con gran esfuerzo, y que una vida saludable sería el pago por una disciplina de limpieza. Eso también es lo que ocurre con la ingeniería del software hoy.

¿Tiene que ser tan duro?--Dificultades esenciales: No sólo no hay balas de plata a la vista, sino que la misma naturaleza del software impide que las haya -- ningún invento de los que mejoraron la productividad, fiabilidad y simplicidad en el hardware, como la electrónica, los transistores y las altas escalas de integración (VLSI) harán lo mismo por el software. No podemos esperar ver doblarse las prestaciones cada dos años.

Lo primero, debemos darnos cuenta de que la anomalía no es que el progreso del software sea tan lento sino que la del hardware sen tan rápido. Ninguna otra tecnología desde que empezó la civilización ha visto una mejora en seis órdenes de magnitud en 30 años. En cualquier otra técnica debemos elegir mejorar *o* las prestaciones *o* reducir costes. Estas mejoras provienen de la transformación la fabricación de ordenadores de una industria de ensamblaje en una industria de procesos.

Segundo, para ver el ratio de progreso que podemos esperar en la tecnología del software, vamos a examinar las dificultades de esa tecnología. Emulando a Aristóteles, las dividiré en *esenciales*, dificultades inherentes a la naturaleza del software, y *accidentales*, aquellas dificultades que se encuentran hoy en día pero que no son inherentes al software.

La esencia de una entidad software es una construcción de conceptos entrelazados: conjuntos de datos, relaciones entre los datos, algoritmos y llamadas a funciones. Esta esencia nos indica que uno de estos conceptos abstractos construidos tiene muchas representaciones. Sin embargo es muy preciso y muy detallado.

Creo que la parte más dura de construir software es la especificación, diseño y prueba de este concepto construido, no el trabajo de representarlo y comprobar la fidelidad de la representación. Todavía tendremos errores de sintaxis, evidentemente; pero eso es trivial si lo comparamos con los errores conceptuales en la mayoría de los sistemas.

Si esto es cierto, hacer software siempre será algo duro. No hay ninguna bala de plata.

Vamos a considerar las propiedades inherentes a la esencia irreductible de los modernos sistemas de software: complejidad, conformidad, variabilidad e invisibilidad.

Complejidad. Las entidades de software son más complejas debido a su tamaña que, posiblemente, cualquier otra construcción humana porque no hay dos partes iguales (al menos por encima del nivel de instrucción). Si la hay, podemos convertir a las dos partes similares en una subrutina--abierta o cerrada. A este respecto, los sistemas de software difieren profundamente de los ordenadores, edificios o automóviles, donde abundan los elementos repetidos.

Los ordenadores digitales son en sí mismos más complejos que la mayoría de las cosas que la gente hace: Tienen un enorme número de estados. Esto hace que su concepción, descripción, y pruebas sean muy duras. Los sistemas de software tienen órdenes de magnitud de estados mayores que los que tienen los ordenadores

Además, un incremento en escala de una entidad software no consiste meramente en repetir algunos elementos a mayor escala, sino que es necesario incrementar el número de elementos diferentes. En la mayoría de los casos, el número de interacciones entre los elementos cambia de una forma no lineal con el número de elementos y la complejidad se incrementa a un ritmo mucho más que lineal.

La complejidad del software es una propiedad esencial, no accidental. Además, una descripción de una entidad de software que elimine su complejidad a menudo elimina su esencia. Durante tres siglos, los matemáticos y los físicos han hecho grandes avances construyendo modelos simplificados de fenómenos complejos, derivando propiedades de los modelos, y verificando esas propiedades experimentalmente. Este paradigma funcionó porque las complejidades ignoradas en los modelos no eran las propiedades esenciales del fenómeno. Pero eso no funciona cuando las complejidades son la esencia.

Muchos de los problemas clásicos del desarrollo de software derivan de esta complejidad esencial y su incremento no lineal con el tamaño. De la complejidad procede la dificultad de comunicación entre los miembros del equipo, que conduce a productos defectuosos, sobrecostes y retrasos. De la complejidad procede la dificultad de enumerar, y aún más comprender, todos los posibles estados del programa, y de ahí procede la falta de fiabilidad. De la complejidad de la función procede la dificultad de llamar a la función, lo que hace difícil de usar al programa. De la complejidad de la estructura procede la dificultad en extender el

programa para realizar nuevas funciones sin crear efectos laterales. De la complejidad de la estructura proceden los estados no previstos que se convierten en agujeros en la seguridad.

De la complejidad no sólo se derivan problemas técnicos, sino también problemas de gestión. Hace difícil tener una visión de conjunto, impidiendo la integridad conceptual. Dificulta encontrar y controlar todos los cabos sueltos. Crea los enormes agobios en aprender y comprender que lleva al personal hacia el desastre.

Conformidad. La gente del Software no es la única que encara la complejidad. Los físicos tratan con objetos terriblemente complejos incluso a los niveles "fundamentales" de la física de partículas. Las labores del físico descansan, sin embargo, en una profunda fe de que hay principios unificadores que deben encontrarse, o en los quarks o en las teorías de campo unificado. Einstein argumentaba que debe haber explicaciones simplificadas de la naturaleza, porque Dios no es caprichoso ni arbitrario.

Nada de esa fe conforta al ingeniero de software. La mayoría de la complejidad que el debe controlar es arbitraria, forzada sin ritmo ni razón por las muchas instituciones humanas y sistemas a los que debe ajustarse. Estos difieren de interfaz a interfaz, y de un momento a otro, no por necesidad sino sólo porque fueron diseñados por gente distinta, en lugar de por Dios.

En la mayoría de los casos, el software debe ajustarse porque es lo más reciente que ha llegado a la escena. En otros casos, debe hacerlo porque parece lo más correcto. Pero en casi todos los casos, la mayoría de la complejidad viene del tener que ajustarse a otros interfaces; esta complejidad no puede simplificarse solamente con un rediseño del software.

Variabilidad. El software está constantemente sometido a presiones para cambiarlo. Por supuesto, esto también afecta a los edificios, coches u ordenadores. Pero las cosas manufacturadas no cambian casi nunca una vez fabricadas; son sobrepasadas por modelos posteriores, o cambios esenciales se incorporan en copias de número de serie superior del mismo diseño básico. Es raro que un automóvil deba ser revisado por un defecto de diseño; cambios en los ordenadores una vez que están funcionando, son algo más frecuentes. Pero son muchísimo menos frecuentes que cambios en el software ya en funcionamiento.

En parte, esto se debe a que el software de un sistema se adapta a su función, y la función es la parte que más siente la presión para cambiar. En parte es porque el software puede cambiarse más fácilmente, es puro pensamiento, infinitamente maleable. Los edificios pueden cambiarse, pero los altos costos de hacerlo, comprendidos por todos, sirven para que la gente que desea los cambios se contenga.

Todo el software que triunfa cambia, debido a dos procesos. Primero, cuando un programa resulta útil, la gente intenta usarlos para nuevas aplicaciones en el límite o más allá del dominio original para el que se diseño. La presión para extender las funcionalidades procede de usuarios a los que les encantan las funciones básicas e inventan nuevos usos *para* ellas.

Segundo, el software exitoso sobrevive más allá de la vida del hardware para el que fue diseñado. Si no son nuevos computadores, son nuevos discos,

pantallas, impresoras...; y el software debe adaptarse para estas nuevas oportunidades.

En resumen, el software vive dentro de un entorno cultural de aplicaciones, usuarios, leyes y hardware. Todo este entorno cambia continuamente, y esos cambios inexorablemente fuerzan a que se produzcan cambios en el software.

Invisibilidad. El software es invisible. Las abstracciones geométricas son herramientas potentes. El plano de la planta de un edificio ayuda tanto al arquitecto como al cliente a evaluar espacios, flujos de tráfico, vistas. Las contradicciones y omisiones salen a la luz. Dibujos a escalas de partes mecánicas y modelos de palos de moléculas, aunque son abstracciones, sirven al mismo propósito. Una realidad geométrica es capturada dentro de una abstracción geométrica.

La realidad del software no es un algo espacial. Aquí, no hay representaciones geométricas como las que que tiene la tierra en forma de mapas, los chips de silicio en forma de diagramas o los computadores en esquemas de conexiones. En el momento que intentamos crear diagramas de las estructuras de software, descubrimos que no está constituido por uno, sino por varios gráficos en distintas direcciones, superpuestos unos sobre otros. Los diversos gráficos pueden representar el flujo de control, el flujo de datos, patrones de dependencia, secuencias temporales, relaciones entre los nombres. Estos gráficos no suelen ser planos y mucho menos jerárquicos. De hecho, una de establecer control conceptual sobre estas estructuras es forzar que todo encaje sobre uno (o más de uno) gráficos jerárquicos. [1]

A pesar del progreso en limitar y simplificar las estructuras del software, es inherente a las mismas el ser no-visualizables, y esto no permite a nuestras mentes usar algunas de las más potentes herramientas conceptuales. Esto no sólo dificulta que nuestras mentes piensen sobre el diseño, sino que también hace difícil que nuestras mentes se comunican sobre el diseño.

Los avances del pasado solucionaron dificultades accidentales. Si examinamos los tres pasos el desarrollo de la tecnología del software que han sido más fructíferos, descubriremos que cada uno atacó una dificultad de envergadura en el desarrollo del software, pero que estas dificultades eran accidentales, no esenciales. Podemos también descubrir los límites naturales a la extrapolación de cada uno de estos ataques.

Lenguajes de Alto Nivel. Posiblemente la mayor mejora en productividad, fiabilidad y simplicidad ha venido del uso de lenguajes de alto nivel para la programación. La mayoría de observadores dirán que proporciona un incremento en un factor cinco de la productividad, y ganancias sustanciales en fiabilidad, simplicidad y comprensibilidad.

¿ Que es lo que consigue un lenguaje de alto nivel ? Libera a un programa de mucha de su dificultad accidental. Un programa abstracto consiste de construcciones conceptuales: operaciones, tipos de datos, secuencias y comunicación. El programa para una máquina concreta está relacionado con bits, registros, condiciones, bifurcaciones, canales, discos y demás. Al extender los lenguajes de alto nivel sus construcciones a lo que uno busca en el programa abstracto y evitar los niveles inferiores, elimina un nivel entero de complejidad que no es inherente en absoluto al programa.

Lo máximo que un lenguaje de alto nivel puede hacer es abarcar todas las construcciones que el programador imagina en el programa abstracto. Para asegurarnos de eso, el nivel de nuestro pensamiento sobre estructuras de datos, tipos de datos y operaciones sigue subiendo, pero a un ratio cada vez menor. Y el desarrollo de los lenguajes se aproxima cada vez más a la sofisticación de los usuarios.

Pero, finalmente, se llega al punto que el desarrollo del lenguaje de alto nivel crea una herramienta nueva que incrementa en vez de reducir el trabajo intelectual del usuario, el cual raramente usa esa esotérica construcción.

Tiempo compartido. El Tiempo Compartido consiguió una gran mejora en la productividad de los programadores y en la calidad de su producto, aunque no tan grande como los lenguajes de alto nivel.

El tiempo compartido ataca una dificultad bastante diferente. Permite la inmediatez, y esto nos permite tener una visión de conjunto de la complejidad. El lento desarrollo del trabajo por lotes implicaba que uno se olvidaba de los detalles que tenía en mente en el momento que se dejaba de programa y se preparaba para compilar y ejecutar. El efecto más serio era el dejar de tener en mente todo lo que está ocurriendo en un sistema complejo.

Esos retrasos, igual que las complejidades del lenguajes máquina, es algo más accidental que esencial en el proceso del software. Los límites de la potencial contribución del tiempo compartido se derivan directamente. El efecto principal es acortar el tiempo de respuesta. Conforme este tiempo de respuesta se aproxima a cero, llega un momento que supera el umbral de percepción humana, que es de unos 100 milisegundos. Más allá de ese límite, no habrá beneficios.

Entornos de desarrollo unificados. Unix e Interlisp, los primeros entornos de programación integrados que se difundieron, parecieron mejorar la productividad por factores propios a ellos. ¿Por qué ?

Atacaban a las dificultades accidentales que se producen al usar programas individuales de forma conjunta, proporcionando librarías integradas, formatos de fichero unificados, y colas y filtros. Como resultado, estructuras conceptuales que en principio consistían en llamarse, alimentarse y utilizarse entre si podía de hecho ser implementadas fácilmente en la práctica.

Este avance estimuló el desarrollo de 'bancos de herramientas', puesto que cada nueva herramienta podía aplicarse a cualquiera de los programas que utilizaban los formatos estándar.

Debido a estos éxitos, los entornos son el objetivo de mucha de la investigación de hoy en día en la ingeniería del software. Echaremos un vistazo a sus promesas y limitaciones en la siguiente sección.

Esperanzas de descubrir la plata Ahora vamos a considerar los desarrollos técnicos que han ido presentándose como potenciales balas de plata. ¿Que tipo de problema encaran? ¿Los esenciales o los accidentales? ¿Ofrecen avances revolucionarios o sólo mejoras incrementales?

Ada y otros avances en lenguajes de alto nivel. Uno de los desarrollos recientes que más nos han vendido es Ada, un lenguaje de alto nivel de propósito general de los años 80. Ada no sólo refleja mejoras revolucionarias en conceptos del lenguaje, sino que abarca características para animar a utilizar diseño

moderno y modularidad. Quizás la filosofía de Ada sea un avance mayor que el lenguaje, a causa de su énfasis en modularidad, tipos de datos abstractos o estructura jerárquica. Ada es excesivamente prolijo, un resultado natural de su proceso de desarrollo, donde los requisitos fueron más importantes que el diseño. Esto no es algo fatal, ya que subconjuntos de su vocabulario pueden solventar el problema de aprenderlo, y avances en el hardware los MIPS baratos necesarios para pagar los costes de compilar. Los avances en estructura de los sistemas de software son, de hecho, una excelente forma de gastar los MIPS que nuestro dinero compra. Los sistemas operativos, a los que se denostaba en los años 60 por su consumo de memoria y ciclos de CPU, han demostrado ser una forma excelente para utilizar los MIPS y memoria baratos que han surgido del desarrollo del hardware.

Sin embargo, Ada no ha demostrado ser la bala de plata que acabe con el monstruo de la productividad del software. Es, después de todo, tan sólo otro lenguaje de alto nivel, y el mayor beneficio de estos lenguajes vino de la primera vez que se adoptaron: la transición de las complejidades accidentales de la máquina en las instrucción más abstractas de las soluciones paso-a-paso. Uno vez eliminados estos accidentes, los que quedan son más pequeños, y el premio por eliminarlos seguramente es menor.

Predigo que en una decada, cuando la efectividad de Ada sea valorada, se verá que ha conseguido algo de mejora, pero no a causa de alguna particularidad del lenguaje, ni siguiera a causa de todas ellas combinadas. Ni será el nuevo entorno de Ada la causa de las mejoras. La mayor contribución de Ada será la transición de los programadores que lo usen, aunque sea ocasionalmente, a las modernas técnicas de diseño de software.

Programación orientada a objeto. Muchos estudiantes del estado del arte tienen más esperanzas en la programación orientada a objetos que en cualquier otra técnica en desarrollo. [2] Yo entre ellos.. Mark Sherman de Dartmouth nos dice en CSnet News que debemos ser cuidadosos distinguiendo dos ideas separadas que aparecen bajo el mismo nombre: *tipos de datos abstractos* y *tipos jerárquicos*. El concepto del tipo de datos abstracto es que el tipo del objeto debería ser definido por un nombre, un conjunto de valores y un conjunto de operaciones propios más que por su estructura almacenada, la cual debería ocultarse. Ejemplos son los paquetes Ada (con tipos privados) y los módulos de Modula.

Los tipos jerárquicos, como las clases de Simula-67, nos permiten definir interfaces generales que pueden ser refinadas posteriormente suministrando tipos subordinados. Los dos conceptos son ortogonales: podemos tener jerarquía sin ocultación u ocultación sin jerarquía. Ambos conceptos representan avances reales en el arte de construir software.

Cada uno elimina otra dificultad accidental del proceso, permitiendo al diseñador expresar la esencia del diseño sin tener que escribir grandes conjuntos de material sintáctico que no añade información sobre el contenido. Para ambos tipos de abstracción, el resultado es eliminar un tipo de dificultad accidental de un mayor nivel y permitir un mayor nivel de expresión en el diseño.

Sin embargo, tales avances pueden hacer poco más que eliminar todas las dificultades accidentales de la expresión del diseño. La complejidad del diseño en si mismo es esencial, y tales ataques no hacen que cambie en eso. La programación orientada a objeto puede conseguir una ganancia de un orden de

magnitud sólo si la innecesaria especificación de tipos de nuestro lenguaje consigue acabar con las nueve decimas partes del trabajo de diseño de un programa. Permítanme que lo dude.

Inteligencia Artificial. La mayoría de la gente espera que avances en inteligencia artificial nos proporcionen el avance revolucionario que nos dará ganancias de varios órdenes de magnitud en la productividad del software y su calidad. [3] Yo no. Para ver porqué, debemos aclarar lo que entendemos por "inteligencia artificial."

D.L. Parnas ha aclarado el caos terminológico: [4]

Dos tipos bastante diferentes de AI son de uso común hoy en día. AI-1: el uso de ordenadores para solucionar problemas que previamente sólo podían solventarse utilizando inteligencia humana. AI-2: El uso de un conjunto específico de técnicas de programación conocidas como heurísticas o programación basada en reglas. En esta aproximación, se estudia a expertos humanos para determinar que heurísticas o reglas utilizan para solucionar problemas... El programa se diseña para solucionar un problema de la misma forma que los humanos parecen hacerlo.

La primera definición tiene un significado que va cambiando... Algo puede encajar con la definición de AI-1 hoy, pero una vez que sabemos como trabaja el programa y comprendemos el problema, no lo veremos como AI nunca más... Desafortunadamente no puedo identificar un único cuerpo de tecnología en este campo... La mayoría del trabajo es específico a un problema, y se requiere algo de abstracción o creatividad para ver la forma de transferirlo.

Coincido completamente con esta crítica. Las técnicas usadas para reconocimiento de voz parecen tener poco que ver con las usadas para reconocimiento de imágenes, y ambas son muy diferentes de las usadas en los sistemas expertos. Se me hace difícil ver como el reconocimiendo de imágenes, por ejemplo, producirá una diferencia apreciable en la práctica de la programación. Lo mismo ocurre con el reconocimiento del habla. Lo difícil al escribir software es decidir lo que uno quiere decir, no decirlo. Ninguna facilidad en la expresión puede dar más que ganancias marginales.

A la tecnología de sistemas expertos, AI-2, le reservo una sección entera.

Sistemas expertos. El cmapo más avanzado dentro de la inteligencia artificial, y el usado en mayor grado, es la tecnología para construir sistemas expertos. Muchos científicos del software están trabajando intensamente para aplicar esta tecnología en el ámbito del desarrollo de software. [3, 5] ¿ De que va todo esto, y cuáles son las expectativas ?

Un *sistema experto* es un programa que contiene un motor de inferencias generalista y una base de reglas, que coge datos de entrada y suposiciones, explorar las inferencias derivadas de la base de reglas, consigue conclusiones y consejos, y ofrece explicar sus resultados mostrando su razonamiento al usuarios. Los motores de inferencia normalmente tratan con datos y reglas probabilísticos o de tipo fuzzy (lógica difusa), además de usar lógica puramente determinista.

Estos sistemas ofrecen ventajas claras sobre algoritmos programados diseñados para conseguir las mismas soluciones para los mismos problemas:

- La tecnología de motor de inferencia se desarrolla de forma independiente de la aplicación y, a continuación, se aplica para muchos usos. Se puede justificar un mayor esfuerzo en los motores de inferencia. De hecho, esa tecnología es muy avanzada.
- Las partes cambiables de las peculiaridades de una aplicación en particular se codifican dentro de la base de reglas, de una forma uniforme, y se suministran herramientas para desarrollar, cambiar, probar y documentar la base de reglas. Esto regulariza gran parte de la complejidad de la aplicación.

La potencia de estos sistemas no proviene de maravillosos mecanismos de inferencia sino más bien de sobre-prolijas bases de conocimiento que reflejan el mundo real de forma muy precisa Creo que el avance más importante ofrecido por la tecnología es la separación de la complejidad de la aplicación de la la complejidad del programa.

Cómo puede aplicarse esta tecnología al trabajo de la ingeniería del software? De muchas formas: Esos sistemas pueden sugerir las reglas para la interfaz, aconsejar sobre estrategias de test, recodar las frecuencias de tipos de bugs, y mostrar claves para la optimización.

Piense en un asesor sobre pruebas. En su forma más rudimentaria, el sistema experto de diagnóstico es poco más que el checklist de un polito, tan sólo un listado enumerado de sugerencias como causas posibles de dificultad. Conforme la estructura del sistema sea más y más captada por la base de reglas, y conforme la base reglas tenga una vista más sofisticada de los síntomas de problemas que se informen, el asesor sobre tests será más detallado sobre las hipótesis que genera y los tests que recomienda. Tal sistema experto se separa radicalmente de un sistema convencional en tanto en cuanto que su base de reglas puede ser modularizada de forma jerárquica en la misma forma que el software correspondiente, porque si el software se cambia modularmente, las reglas de diagnóstico pueden asimismo modificarse modularmente.

El trabajo requerido para general las reglas de diagnóstico es un trabajo que debería de ser hecho de todas formas generando el conjunto de casos de prueba para los módulos y para el sistema. Si se hace de una forma generalista, tanto con una estructura uniforme para las reglas como un buen motor de inferencia, puede reducir realmente el trabajo total de generar los casos de test y ayudar también en el mantenimiento durante el ciclo de vida del producto y las pruebas para las modificaciones. De la misma forma, podemos defender otros asesores, posiblemente muchos y simples, para otras partes de las tareas de la producción de software.

Hay muchas dificultades en el camino de comprender la utilidad de los asesores basados en sistemas expertos en el desarrollo de programas. Una parte crucial de nuestro escenario imaginario es el desarrollo de formas sencillas de pasar desde la especificación de la estructura del programa a generación de reglas de diagnóstico de forma automática o semiautomática. Incluso más difícil e importante es la tarea doble de la adquisición de conocimientos: encontrar expertos auto-analíticos que articulen el porqué hacen las cosas que hacen y desarrollar técnicas eficientes para extraer lo que ellos saben y destilarlo dentro de

bases de reglas. El prerrequisito esencial para construir un sistema experto es tener un experto.

La contribución más poderosa de los sistemas expertos posiblemente será poner al servicio del programado inexperto la experiencia y visión acumulada por los mejores programadores. Esto no es precisamente poco. El salto entre la práctica del mejor ingeniero de software y la del ingeniero medio es enorme, quizás la mayor que se encuentre entre las distintas ingenierías existentes. Una herramienta que disemine buenas prácticas sería importante.

Programación "Automática". Durante casi 40 años, la gente ha estado anticipando y escribiendo sobre "programación automática", o la generación de un programa para resolver un problema a partir de las especificaciones del problema. Hay quien escribe hoy como si esperaran que esta técnica proporcionara el siguiente salto. [5]

Parnas [4] implica que el término se usa por su glamour, no por la semántica de su palabra, aseverando,

Resumiendo, la programación automática siempre ha sido un eufemismo para la programación en un lenguaje de alto nivel que estaba disponible en ese momento para el programador.

Esencialmente argumenta que en la mayoría de los casos las especificaciones que deben tenerse no son las del problema sino las del método para su solución.

Hay excepciones. La técnica de construir generadores es muy potente, y se usar rutinariamente en programas para ordenar. Algunos sistemas para integrar ecuaciones diferenciales también han permitido la especificicación directa del problema, y los sistemas han ajustado los parámetros, elegido los metodos de solución entre los de una librería y generado los programas.

Estas aplicaciones tienen propiedades muy favorables:

- Los problemas son claramente caracterizados por (relativamente) muy pocos parámetros.
- Hay muchos métodos conocidos de solución que proporcionan una librería de alternativas.
- Un análisis extensivo ha conducido a reglas explicitas para seleccionar técnicas de solución, a partir de parámetros dados por el problema.

Resulta difícil ver la forma en que estas técnicas se generalizan al mundo enormemente más amplio del software ordinario, donde los casos donde esas bonitas propiedades aparecen son la excepción. Incluso más difícil es imaginar como este avance en la generalización pudiera producirse.

Programación gráfica. Uno de los temas favoritos en las disertaciones para PhD es la programación gráfica o visual -- la aplicación de los gráficos de ordenador al diseño de software. [6, 7] A veces la promesa hecha por esa aproximación se defiende como análoga al diseño de los chips VLSI, en los cuales los gráficos por ordenador juegan un rol muy fructífero. A veces los teóricos, justifican la aproximación considerando los flujogramas como el medio ideal para el diseño de programas y proporcionando poderosos medios para construirlos.

Nada convincente, y mucho menos excitante, ha surgido de esos esfuerzos. Estoy convencido de nunca lo hará.

En primer lugar, como he defendido antes [8], el flujograma es una pobre abstracción de la estructura del software. De hecho, es mejor verlo como el intento de Burks, von Neumann, y Goldstine de proporcionar un lenguaje de control de alto nivel que necesitaban desesperadamente para su propuesta de ordenador. En la forma lastimosa, multi-página, de caja de conexiones en la que ha terminado elaborada el flujograma hoy en día, ha demostrado ser inútil como una herramienta de diseño: los programadores dibujan los flujogramas después, no antes de haber escrito los programas que describen.

Segundo, las pantallas de hoy son demasiado pequeñas, en pixeles, para mostrar tanto el conjunto como la resolución de cualquier diagrama de software seriamente detallado. La conocida "metáfora de escritorio" de las estaciones de trabajo actuales es, de hecho, una metáfora de "sillón de avión". Cualquiera que haya abierto un maletín lleno de papeles mientras estaba sentado entre dos pasajeros reconocerá las diferencias: uno puede sólo ver unas pocas cosas a la vez. El escritorio real proporciona tanto acceso a un montón de páginas como una visión de conjunto. Más aún, cuando la creatividad sube al límite, más de un programador o escritor abandona el escritorio por el mucho más espacioso suelo. La tecnología de hardware tendrá que avanzar bastante más antes de que la amplitud de nuestros monitores sea suficiente para la tarea de diseñar software.

Aún más fundamental, como defendí antes, es que el software sea muy difícil de visualizar. Si uno hace diagrama de flujo de control, el anidado de los ámbitos de las variables, las referencias cruzadas entre las variables, el flujo de datos, estructuras jerárquicas de datos o cualquier otra cosa, uno percibe sólo una dimensión del elefante de estructuras intricadamente interconectas que es el software. Si se superponen todos los diagramas generados por las muchas vistas relevantes, resulta difícil extraer cualquier visión de conjunto. La analogía VLSI es fundamentalmente errónea: un diseño de un chip es una descripción bi-dimensional cuya geometría refleja su construcción en tres dimensiones. Un software no.

Verificación del programa. Gran parte del esfuerzo de la programación consiste en el testeo y la corrección de errores. ¿Es quizás donde pudiéramos esperar una bala de plata, eliminando los errores del software en la fase de diseño del sistema? ¿Puede mejorar tanto la productividad como la fiabilidad radicalmente la estrategia enormemente diferente de probar la corrección de los diseños antes del inmenso esfuerzo realizado en implementarlos y comprobarlos?

No creo que podamos encontrar la magia de la productividad aquí. La verificación de programas es un concepto muy potente y será muy importante en cosas tales como kernels de sistemas operativos seguros. La tecnología no promete, sin embargo, ahorrar en trabajo. Las verificaciones cuestan tanto trabajo que sólo unos pocos programas son verificados.

La verificación del programa no implica programas a prueba de errores. No hay magia aquí. Las pruebas matemáticas pueden estar llenas de fallos. Aunque la verificación podría reducir la carga en el testeo del programa, no puede eliminarla.

Más en serio, incluso la verificación de programa perfecta sólo podría establecer que un programa cumple con sus especificaciones. La tarea más dura en el software es conseguir unas especificaciones completas y consistentes, y la

parte más esencial de construir un programa es, de hecho, la depuración de las especificaciones.

Entornos y herramientas. Que ganancia podemos esperar de las mejoras e investigaciones en los entornos de programación? Una reacción instintiva es que los problemas con mayor retribución (sistemas de ficheros jerárquicos, formatos de ficheros uniformes para hacer posibles interfaces de programa uniformes y herramientas generalizadas) fueron atacados hace ya mucho, y han sido resueltos. Los editores y entornos de desarrollo inteligentes específicos a un lenguaje no se usan todavía ampliamente en la práctica, pero la mayoría de ellos prometen libarnos de los errores sintácticos y los errores semánticos simples.

Quizás la mayor ganancia que deba conseguirse por los entornos de programación es el uso de sistemas de base de datos integrados para realizar el seguimiento de la miríada de detalles que deben ser seguidos de forma precisa por el programador individual y mantenidas uniformemente por el grupo de colaboradores en un único sistema.

Seguramente este trabajo se realice y arrojará alguna fruta de productividad y fiabilidad, pero será marginal.

Estaciones de trabajo. Que ganancia puede esperar el arte del software del incremento rápido y seguro de la potencia y memoria de las estaciones de trabajo individuales? Bien, cuantos MIPS puede uno usar fructíferamente? La composición y edición de programas y documentos es soportada totalmente por las máquinas de hoy en día. La compilación significa un retraso, pero un factor de 10 en la velocidad de la máquina podría llevar el tiempo necesario para la velocidad de pensamiento de la actividad diaria del programador. De hecho, eso parece que ocurra hoy en día.

Seguro que unas estaciones de trabajo más potentes serán bienvenidas. Pero no podemos esperar mágicos avances gracias a ellas.

Ataques prometedores sobre la Esencia. Incluso aunque ningún avance tecnológico prometa dar el tipo de resultado mágico al que estamos acostumbrados en el hardware, actualmente hay abundancia de buenos trabajos, y la promesa de avances firmes, aunque no espectaculares.

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

Si, como creo, los componentes conceptuales de la tarea están ocupando ahora la mayoría del tiempo, entonces ninguna actividad sobre las partes necesarias del trabajo que consistan simplemente en la expresión de los conceptos dará grandes ganancias de productividad.

Deberíamos centrarnos en aquellos ataques sobre la esencia del problema del software, la formulación de estructuras conceptuales complejas. Afortunadamente, algunos de estos ataques son prometedores.

Comprar vs construir. La solución más radical para construir software es no hacerlo.

Cada día esto es más fácil, ya que más y más vendedores ofrecen más y mejor software de una asombrosa variedad de aplicaciones. Mientras los ingenieros de software hemos trabajo en la metodología de producirlo, la revolución del PC ha creado no uno, sino muchos mercados masivos para el software. Cada revista mensual, ordenada por tipo de máquina, anuncia y revisa

docenas de productos a precios por debajo de los cientos de Euros. Las fuentes más especializadas ofrecen productos muy potentes para los mercados de las estaciones de trabajo y el mercado Unix. Incluso herramientas de software y entornos pueden comprarse sin problemas. Yo mismo he propuesto un mercado para módulos individuales [9]

Cualquiera de estos productos es más barato de comprar que de hacer. Incluso a un costo de 100.000 Euros, un software comprado cuesta lo mismo que un año de programador. Y la entrega es inmediata! Bueno, inmediata siempre que el producto exista, productos que harán feliz a un usuario. Más aún, tales productos tienden a estar mucho mejor documentados y mejor mantenidos que el producto propio.

El desarrollo del mercado de masas es, creo, el mayor avance en la ingeniería del software. El coste del software ha sido siempre el de desarrollo, no el de copia. Al compartir el coste simplemente entre unos pocos usuarios reduce radicalmente el coste por usuario. Otra forma de verlo es que el uso de *n* copias de un software multiplica de forma efectiva la productividad de sus desarrolladores por *n*. Eso implica una mejora de la productividad enorme.

El factor clave, por supuesto, es la aplicabilidad. Puedo usar un paquete ya hecho para implementar mi tarea? Aquí ha ocurrido algo sorprendente. Durante los años 50 y 60, estudio tras estudio mostraban que los usuarios no utilizarían paquetes generalistas para gestión de nominas, control de inventarios, contabilidad y cosas así. Los requisitos eran demasiado especializados, las variaciones de un caso a otro excesivas. Durante los años 80, nos encontramos que la gente demandaba estos productos y se difundían de forma masiva. ¿ Que es lo que cambio ?

Ciertamente, no los paquetes. Podían ser más generalistas y algo más ajustables que antes, pero no mucho. Ni las aplicaciones. En todo caso, las necesidades de los negocios y ciencias de hoy son mayores y más complicadas que las de hace 20 años.

El gran cambio ha venido del ratio de costo hardware/software. En 1960, el comprador de una máquina de dos millones de dólares sentía que él podía afrontar otros 250.000 más por un programa de nóminas a la medida, uno que encajara más fácilmente y sin problemas en el entorno hostil de la sala de ordenadores. Hoy, el comprador de una máquina para la oficina de 50.000 dólares no concibe afrontar un programa de nominas a la medida, por lo que se adapta su gestión de nóminas a los paquetes disponibles. Los ordenadores son algo tan ubicuo, aunque no tan amado, que las adaptaciones son aceptadas como algo inevitable.

Hay excepciones dramáticas a mi argumento de que los paquetes de software no han cambiado mucho en estos años: las hojas de cálculo y los sistemas simples de base de datos. Estas poderosas herramientas, tan obvias retrospectivamente y tan tardías en aparecer, conducen ellas mismas a miles de usos, algunos bastante poco ortodoxos. Artículos e incluso libros abundan ahora mismo sobre como realizar tareas insospechadas con la hoja de cálculo. Un gran número de aplicaciones que podrían haberse escrito como programas a la medida en Cobol o RPG son ahora realizadas rutinariamente con estas herramientas.

Muchos usuarios operan ahora sus propios ordenadores con diversas aplicaciones sin haber escrito nunca un programa. De hecho, muchos de estos

usuarios no pueden escribir nuevos programas para sus máquinas, pero son sin embargo adeptos a solucionar nuevos problemas con ellos.

Creo que la más potente estrategia de productividad del software para muchas organizaciones de hoy es equipar a los trabajadores intelectuales que están en primera linea con PCs y programas generalistas de hoja de cálculo, ficheros, dibujo y proceso de textos y dejar que se las apañen. La misma estrategia, con paquetes de estadística y matemáticas y algunas capacidades simples de programación, funcionarían con centenares de científicos de laboratorio.

Refinamiento de requisitos y prototipado rápido. La parte más dura de hacer software es, precisamente, decidir lo que hacer. Ninguna otra de las partes en que se divide este trabajo es tan difícil como establecer los requisitos técnicos detallados, incluyendo todas las interfaces con la personas, las máquinas y los otros sistemas de software. Ninguna otra parte del trabajo destroza tanto el resultado final si se hace mal. Ninguna otra parte resulta tan díficil de rectificar a posteriori.

Por tanto, la función más importante que realiza el diseñador de software para el cliente es la extracción iterativa y el refinamiento de los requisitos del producto. La verdad es que el cliente no sabe lo que quiere. El cliente normalmente no sabe que preguntas deben ser respondidas, y casi nunca ha pensado en los detalles del problema necesarios para la especificación. Incluso la simple respuesta ("Haz que el nuevo software trabaje como nuestro viejo sistema de procesado manual de la información") es de hecho demasiado simple. Uno nunca quiere exactamente eso. Es más, los sistemas de software complejos son cosas que actúan, se mueven, que trabajan. La dinámica de esa acción son difíciles de imaginar. Por tanto, al planificar cualquier actividad de diseño de software, es necesario permitir un trabajo iterativo extensivo entre el cliente y el diseñador como parte de la definición del sistema.

Quisiera ir un paso más allá y asegurar que es realmente imposible para un cliente, trabajando con un ingeniero de software, especificar completa, precisa y correctamente los requisitos exactos de un sistema de software moderno sin probar algunas versiones de prueba del sistema.

Por tanto, una de las mayores promesas de los esfuerzos tecnológicos actuales, y una que ataca a la esencia, no los accidentes, del problema del software, es el desarrollo de aproximación y herramientas para el prototipado rápido de sistemas ya que el prototipado es parte de la especificación iterativa de requisitos.

Un *prototipo de software* es uno que simula las interfaces más importantes e implementa las funciones principales del sistema deseado, aunque no se ajuste a las limitaciones de velocidad del hardware, tamaño o coste. Los prototipos normalemente implementan las tareas principales de la aplicación, pero no intentan manipular las tareas excepcionales, responder correctamente a entradas erroneas, o abortar de forma limpia. El proposito del prototipo es hacer real la estructura del concepto del software especificado, para que el cliente pueda comprobar su consistencia y su usabilidad.

Gran parte del procedimiento de adquisición de software de hoy en día descansa en asumir que uno puede especificar de forma satisfactoria un sistema por anticipado, pagar lo necesario para que se construya, tenerlo construido e

instalarlo. Creo que asumir esto es algo fundamentalmente erróneo, y que muchos problemas relativos a la adquisición de software se deben a esa falacia. A partir de ese momento, no podrán ser solucionados sin revisiones de sus fundamentos, revisiones que implican desarrollo iterativo y especificación de prototipos y productos.

Desarrollo incremental: hacer crecer en vez de construir software. Todavía recuerdo la sacudida que sentí en 1958 cuando escuche por primera vez a un amigo hablando de *construir* un programa, como algo opuesto a *escribirlo*. Como en un fogonazo, el cambió mi forma de ver el proceso del software. La metáfora era potente y precisa. Hoy comprendemos mejor como se asemeja la construcción del software a otros procesos de construcción y usamos libremente otros elementos de la metáfora, tales como *especificaciones*, *ensamblaje de componentes*, y *andamiaje (scaffolding)*.

La metáfora de la construcción ha sobrevivido a su utilidad. Es el momento de cambiar una vez más. Si, como yo creo, las estructuras conceptuales que construimos hoy son demasiado complicadas para ser especificadas de forma precisa de forma temprana, y demasiado complejas para ser construidas sin fallos, debemos tomar una aproximación radicalmente diferente

Deberíamos mirar a la naturaleza y estudiar la complejidad de las cosas vivas, en vez de las cosas muertas que hace el hombre. Aquí encontramos construcciones cuya complejidad nos estremece. El mismo cerebro es tan intrincado que no se le puede sacar un mapa, poderoso más allá de la imitación, rico en diversidad, auto-protegido y auto-regenerativo. El secreto es que creció, no se construyó.

Lo mismo debería ocurrir con nuestro software. Hace algunos años Harlan Mills propuso que cualquier software debería crecer mediante desarrollo incremental. [10] Esto significa que el sistema debería primero conseguir que se ejecutara, incluso sin hacer nada útil excepto llamar al conjunto apropiado de subprogramas que no hacen nada. Entonces, paso a paso, debería ser rellenado, desarrollando los subprogramas, como acciones o llamadas a subrutinas vacías en el nivel inferior.

He visto resultados impresionantes desde que empecé a instigar a usar esta técnica a los constructores de software en mi clase (Software Engineering Laboratory). Nada en la década anterior ha cambiado tanto mi propia práctica, o su efectividad. La aproximación necesita el diseño de arriba hacia abajo (top-down), lo que representa un crecimiento desde arriba hacia abajo del software. Permite un fácil rastreo (backtracking). Permite prototipos tempranos. Cada función añadida y cada nueva provisión para datos o circunstancias más complejos crece orgánicamente a partir de lo ya existente.

Los efectos en la moral son asombrosos. El entusiasmo salta cuando hay un sistema en marcha, aunque sea uno simple. Los esfuerzos se redoblan cuando el primer dibujo de un software de gráficos aparece en la pantalla, aunque sea sólo un rectángulo. Uno siempre tiene, en cada paso del proceso, un sistema en marcha. Descubro que los equipos de desarrollo pueden hacer *crecer* entidades mucho más complejas en cuatro meses de lo que podían *construir*.

Los mismos beneficios se obsevan tanto en proyectos enormes como en los más pequeños. [11]

Grandes diseñadores. La cuestión central sobre como mejorar el arte del software se centra, como siempre, en la gente.

Podemos conseguir buenos diseños siguiendo prácticas buenas en lugar de pobres. Las buenas prácticas de diseño pueden enseñarse. Los programadores están entre los más inteligentes entre la población, por lo que pueden aprender buenas prácticas. A partir de ahora, uno de los mayores obligación en USA debería ser promulgar prácticas buenas y modernas. Nuevos curricula, nueva literatura, nuevas organizaciones tales com el Software Engineering Institute, todas deberían enfocarse en subir el nivel de nuestras prácticas de uno pobre a otro bueno. Esto es completamente apropiado.

No obstante, no creo que podamos dar el siguiente paso adelante en la misma forma. Mientras que la diferencia entre diseños conceptuales pobres y buenos debe descansar en lo saludable del método de diseño, la diferencia entre diseños buenos y excelentes seguramente no. Los diseños excelentes proceden de diseñadores excelentes. La construcción de software es un proceso *creativo*. Una metodología sana puede potenciar y liberar la mente creativa; pero no puede inflamar o inspirar al esclavo.

Las diferencias no son menores: son similares a las diferencias entre Salieri y Mozart. Estudio tras estudio muestran que los diseñadores excelentes producen estructuras que son más rápidas, pequeñas, simples, claras y producen más con menos esfuerzo. [12] Las diferencias entre los excelentes y la media se aproxima a un orden de magnitud.

Una pequeña mirada retrospectiva muestra que aunque mucho software bueno y útil ha sido diseñado por comités y fabricado como parte de proyectos con varios apartados, aquellos sistemas de software que tienen fans excitados y apasionados son aquellos que son el producto de uno o sólo unos pocos diseñadores excelentes. Pensemos en Unix, APL, Pascal, Modula, Smalltalk, incluso Fortran; y comparémoslos con Cobol, PL/I, Algol, MVS/370, y MS-DOS. (Vea la Tabla 1.)

Por tanto, aunque apoyo fuertemente los esfuerzos actuales para la transferencia de tecnología y la formación, creo que el esfuerzo individual más importante que podemos hacer es desarrollar las formas de que surgan grandes diseñadores.

Ninguna organización dedicada al software puede ignorar este desafío. Los buenos gestores, aunque escasos, no lo son tanto como los buenos diseñadores. Los diseñadores y gestores excelentes son incluso más raros. La mayoría de las organizaciones realizan esfuerzos considerables buscando y cultivando a sus gestores; no conozco a ninguna que realice esfuerzos equivalentes para encontrar y desarrollar a los diseñadores excelentes de los que depende en última instancia la excelencia técnica de los productos.

Tabla 1. Software Excitante Vs. Útil pero No excitante. Excitante No Excitante Unix Cobol APL PL/I Pascal Algol Modula MVS/370 Smalltalk MS-DOS Fortran -

Mi primera propuesta es que las organizaciones de software debe determinar y proclamar que los diseñadores excelentes son tan importantes para su éxito como los gestores excelentes, y que deben esperar ser recompensados y arropados de la misma forma. No sólo en el salario, sino el resto de los requisitos -- tamaño de la oficina, mobiliario, equipamiento, gastos para viajes, personal de apoyo -- deben ser equivalentes.

Como desarrollar a los diseñadores excelentes? El espacio no me permite una discusión detallada, pero algunos pasos son obvios:

- Identificar sistemáticamente a los mejores diseñadores tan pronto como sea posible. Los mejores a menudo no son los que tienen más experiencia.
- Asignar un mentor de carrera responsable del desarrollo de su futuro, y realizar un seguimiento cuidadoso de su carrera.
- Idear y mantener un plan de desarrollo de carrera para cada una de las jovenes promesas, incluyendo aprendizaje junto a diseñadores excelentes cuidadosamente seleccionados, episodios de educación formal avanzada y cursillos, todo entrelazado con asignación de diseños solitarios y liderazgo técnico.
- Dar oportunidades para que los diseñadores en desarrollo interactúen y se estimulen con otros.

AcknowledgmentsI thank Gordon Bell, Bruce Buchanan, Rick Hayes-Roth, Robert Patrick, and, most especially, David Parnas for their insights and stimulating ideas, and Rebekah Bierly for the technical production of this article.

REFERENCES

[1] D.L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, Vol. 5, No. 2, March 1979, pp. 128-38.

[2] G. Booch, "Object-Oriented Design," *Software Engineering with Ada*, 1983, Menlo Park, Calif.: Benjamin/ Cummings.

[3] *IEEE Transactions on Software Engineering* (special issue on artificial intelligence and software engineering), I. Mostow, guest ed., Vol. 11, No. 11, November 1985.

[4] D.L. Parnas, "Software Aspects of Strategic Defense Systems:" *American Scientist*, November 1985.

[5] R. Balzer, "A 15-year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering* (special issue on artificial intelligence and software engineering), J. Mostow, guest ed., Vol. 11, No. 11 (November 1985), pp. 1257-67.

[6] *Computer* (special issue on visual programrning), R.B. Graphton and T. Ichikawa, guest eds., Vol. 18, No. 8, August 1985.

[7] G. Raeder, "A Survey of Current Graphical Programming Techniques," *Computer* (special issue on visual programming), R.B. Graphton and T. Ichikawa, guest eds., Vol. 18, No. 8, August 1985, pp. 11-25.

[8] F.P. Brooks, *The Mythical Man Month*, Reading, Mass.: Addison-Wesley, 1975, Chapter 14.

[9] Defense Science Board, *Report of the Task Force on Military Software* in press.

[10] H.D. Mills, "Top-Down Programming in Large Systems," *in Debugging Techniques in Large Systems*, R. Ruskin, ed., Englewood Cliffs, N.J.: Prentice-Hall, 1971.

[11] B.W. Boehm, "A Spiral Model of Software Development and Enhancement," 1985, TRW Technical Report 21-371-85, TRW, Inc., 1 Space Park, Redondo Beach, Calif. 90278.

[12] H. Sackman, W.J. Erikson, and E.E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Communications of the ACM*, Vol. 11, No. 1 (January 1968), pp. 3-11.

Brooks, Frederick P., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Vol. 20, No. 4 (April 1987) pp. 10-19.