No Silver Bullet

-Esencia Y accidentes en Ingeniería del Software

Frederick P. Brooks, Jr.

Universidad de Carolina del Norte en Chapel Hill

No existe un único desarrollo, ya sea en la tecnología o la gestión técnica, que por sí mismo promete aún un orden de magnitud mejora dentro de una década en la productividad, en la fiabilidad, en la simplicidad.

Abstracto 1

Toda la construcción de software implica tareas esenciales, la confección del complejo estructuras conceptuales que componen la entidad de software abstracto y tareas accidentales, la representación de estas entidades abstractas en lenguajes de programación y la asignación de éstas en los idiomas de la máquina dentro de las limitaciones de espacio y velocidad. La mayor parte de la gran pasado ganancias de productividad de software han venido de la eliminación de las barreras artificiales que tienen realizado las tareas accidentales excesivamente duro, tales como las limitaciones de hardware graves, lenguajes de programación difíciles, la falta de tiempo de máquina. ¿Cuánto de lo que el software los ingenieros ahora hacen es todavía dedican a lo accidental, en contraposición a lo esencial? A menos que es más de 9/10 de todo el esfuerzo, la reducción de todas las actividades accidentales a tiempo cero no lo hará dar un orden de magnitud de mejora.

Por lo tanto, parece que ha llegado el momento de abordar las partes esenciales de la tarea de software, los relacionados con la configuración de las estructuras conceptuales abstractas de gran complejidad. Sugiero:

- Explotar el mercado de masas para evitar la construcción de lo que se puede comprar.
- Uso de creación rápida de prototipos como parte de una iteración planificada en el establecimiento de software requisitos.
- Creciendo software de forma orgánica, añadiendo cada vez más la función de los sistemas, ya que son ejecutar, que se utiliza, y probado.
- La identificación y el desarrollo de los grandes diseñadores conceptuales de la nueva generación.

Introducción

De todos los monstruos que pueblan las pesadillas de nuestro folklore, ninguno aterrorice más hombres lobo, porque transforman inesperadamente de lo familiar en horrores. Por éstos, buscamos balas de plata que les puede poner mágicamente para descansar.

1 Reproducido de: Frederick P. *Brooks,* The *Mythical Man-Month, edición aniversario con 4 nueva capítulos,* Addison-Wesley (1995), reimpreso en sí de *las* Actas de la *IFIP Décimo Mundial Computing Conferencia,* H.-J. Kugler, ed., Elsevier Science BV, Amsterdam, NL (1986) pp. 1069-1076

El proyecto de software familiar tiene algo de este personaje (al menos como se ve por el gerente no técnica), por lo general inocente y sencilla, pero capaz de convertirse en un monstruo de las planificaciones perdidas, presupuestos soplado y productos defectuosos. Así escuchamos desesperada llora por una bala de plata, algo para que los costos de software caen tan rápidamente como equipo los costos de hardware hacen.

Pero, al mirar al horizonte de una década, por lo tanto, no vemos ninguna bala de plata. No hay el desarrollo individual, ya sea en tecnología o técnica de gestión, lo que por sí mismo promete incluso un orden de magnitud de mejora de la productividad, en la fiabilidad, en simplicidad. En este capítulo vamos a tratar de ver por qué, pero el examen tanto de la naturaleza de la problema de software y las propiedades de las balas proponen.

El escepticismo no es pesimismo, sin embargo. Aunque no vemos avances sorprendentes, y, de hecho, creer tal es incompatible con la naturaleza del software, muchas innovaciones alentadoras están en marcha. Una disciplina, esfuerzo constante para desarrollar, propagar, y explotarlos de hecho debe producir una mejora orden de magnitud.

No hay camino real, pero hay un camino.

El primer paso hacia el tratamiento de la enfermedad fue la sustitución de las teorías del demonio y humores teorías de la teoría de los gérmenes. Ese mismo paso, el comienzo de la esperanza, en sí mismo discontinua todas las esperanzas de soluciones mágicas. Se dijo a los trabajadores se haría el progreso paso a paso, con gran esfuerzo, y que una, la atención incesante persistente tendría que pagar a un disciplina de la limpieza. Lo mismo sucede con la ingeniería de software hoy.

¿Tiene que ser duro? - Dificultades Esenciales

No sólo no hay balas de plata ahora a la vista, la naturaleza misma de software hace que sea poco probable que no habrá ningún-no hay inventos que harán para la productividad de software, fiabilidad y simplicidad lo que la electrónica, transistores, y la integración a gran escala hicieron por hardware de la computadora. No podemos esperar que nunca para ver ganancias al doble cada dos años.

En primer lugar, hay que observar que la anomalía no es que el progreso software es tan lento, pero que el progreso material informático es tan rápido. Ninguna otra tecnología desde el inicio de la civilización ha visto seis órdenes de magnitud aumento de precio-rendimiento en 30 años. En ninguna otra tecnología puede un solo optar por tomar la ganancia, ya sea en un mejor rendimiento o reducido costes. Estas ganancias se derivan de la transformación de la fabricación de un ordenador industria de ensamblaje en una industria de proceso.

En segundo lugar, para ver qué tasa de progreso que podemos esperar en la tecnología de software, nos dejó examinar sus dificultades. Siguiendo a Aristóteles, yo las divido en esencia las dificultades inherente a la naturaleza de los programas y accidentes, aquellas dificultades que hoy asisten su producción, pero que no son inherentes.

Los accidentes que discuten en la siguiente sección. En primer lugar vamos a considerar la esencia.

La esencia de una entidad de software es una construcción de los conceptos entrelazados: los conjuntos de datos, las relaciones entre los elementos de datos, algoritmos y las invocaciones de funciones. Esta esencia es abstracto, en que el constructo conceptual es el mismo bajo diferentes representaciones. Sin embargo, es altamente preciso y rico en detalles.

Creo que la parte más difícil de la construcción de software para ser la especificación, diseño y pruebas de este constructo conceptual, no el trabajo de representarlo y probar la fidelidad de la representación. Todavía cometemos errores de sintaxis, sin duda; pero son fuzz en comparación con los errores conceptuales en la mayoría de los sistemas

Si esto es cierto, la creación de software siempre será difícil. No es inherentemente no plata bala.

Vamos a considerar las propiedades inherentes de esta esencia irreductible de software moderno sistemas: la complejidad, la conformidad, la mutabilidad, y la invisibilidad.

Complejidad. Entidades de software son más complejos para su tamaño que quizás cualquier otra construcción humana, porque no hay dos piezas iguales (al menos por encima del nivel de los estados). Si son, hacemos las dos partes similares en una sola, una subrutina, abierta o cerrada. En esto sistemas de software respecto difieren profundamente de los ordenadores, edificios o automóviles, donde abundan los elementos repetidos.

Ordenadores digitales son a su vez más complejo que la mayoría de cosas que las personas construyen; ellos tener un gran número de estados. Esto hace concebir, describiendo, y prueba de ellos difícil. Los sistemas de software tienen órdenes de magnitud más estados que las computadoras hacen.

Asimismo, una ampliación de una entidad de software no es más que una repetición de los mismos elementos en tamaño más grande; es necesariamente un aumento en el número de diferentes elementos.

En la mayoría de los casos, los elementos interactúan entre sí de alguna manera no lineal, y el la complejidad de la totalidad aumenta mucho más que linealmente.

La complejidad de software está en propiedad esencial, no un accidente. Por lo tanto descripciones de una entidad software que abstraer su complejidad menudo abstracta de distancia de su esencia. Matemáticas y ciencias físicas hicieron grandes avances durante tres siglos por construir modelos simplificados de fenómenos complejos, derivando inmuebles de las modelos, y la verificación de estos inmuebles experimentalmente. Esto funcionó porque las complejidades ignoradas en los modelos no eran las propiedades esenciales de los fenómenos. Ello no funciona cuando las complejidades son la esencia.

Muchos de los problemas clásicos de desarrollar productos de software derivados de esta complejidad esencial y su no lineal aumentó con el tamaño. De la complejidad viene la dificultad de la comunicación entre los miembros del equipo, lo que conduce a defectos del producto, costo sobrecostos, retrasos en el programa. De la complejidad viene la dificultad de enumerar, mucho menos comprensión, todos los posibles estados del programa, y desde ese viene la falta de fiabilidad. A partir de la complejidad de las funciones viene la dificultad de invocar aquellos funciones, lo que hace que los programas de difícil de usar. De la complejidad de la estructura viene la dificultad de extender los programas a las nuevas funciones sin crear efectos secundarios. Desde la complejidad de la estructura viene el estado unvisualized que constituyen la seguridad trampillas.

No sólo problemas, pero los problemas de gestión técnica y venir del complejidad. Esta complejidad hace panorama duro, impidiendo así la integridad conceptual.

Esto hace que sea difícil de encontrar y controlar todos los cabos sueltos. Se crea el tremendo aprendizaje y la comprensión de carga que hace que el personal de facturación de un desastre.

Conformidad. La gente de software no están solos para afrontar la complejidad. Ofertas de Física con objetos tremendamente complejos, incluso a nivel de partículas "fundamentales". Los trabajos físico, sin embargo, en una fe firme que hay principios unificadores que se encuentran, ya sea en quarks o en las teorías del campo unificado. Einstein argumentó reiteradamente que debe haber explicaciones simplificadas de la naturaleza, porque Dios no es caprichoso o arbitrario.

Sin tal fe consuela el ingeniero de software. Gran parte de la complejidad que debe principal es la complejidad arbitraria, forzada sin ton ni son por el que muchos humanos instituciones y sistemas a los que sus interfaces deben confirmar. Estos difieren de interfaz para interactuar, y de vez en cuando, no por necesidad, sino sólo porque eran diseñada por diferentes personas, en lugar de Dios.

En muchos casos, el software debe confirmar porque ha llegado más recientemente a la escena. En otros, debe cumplir porque se percibe como el más adaptable. Pero en todos los casos, tanto la complejidad proviene de conformación a otras interfaces; esto no puede ser simplificado por cualquier rediseño del software por sí solo.

Mutabilidad. La entidad de software es constantemente objeto de presiones para el cambio. De

Por supuesto, también lo son los edificios, automóviles y computadoras. Pero las cosas son fabricados con poca frecuencia cambiado después de la fabricación; que se sustituya por los modelos posteriores, o cambios esenciales son incorporados en copias del número de serie posteriores del mismo diseño básico. Devoluciones de llamada de los automóviles son realmente muy poco frecuente; cambios en el campo de los ordenadores un poco menos.

Ambos son mucho menos frecuentes que las modificaciones de software fielded.

En parte esto se debe a que el software en un sistema encarna su función, y la función es la parte que más se siente las presiones del cambio. En parte se debe a que el software puede ser cambiado más fácilmente: es puro pensamiento-cosas, infinitamente maleable. Edificios de hecho conseguir cambiado, pero los altos costos del cambio, entendida por todos, sirven para amortiguar el caprichode los cambiadores.

Todo el software exitoso se cambia. Dos procesos están en el trabajo. Como un software producto se encuentra para ser útil, la gente trata de él en nuevos casos en el borde de, o más allá, la dominio original. Las presiones para la función ampliada venir principalmente de los usuarios que les gusta la función básica e inventar nuevos usos para ella.

En segundo lugar, el software de éxito también sobrevive más allá de la vida normal de la máquina vehículo para el que está escrito primero. Si no las computadoras nuevas, entonces al menos discos nuevos, nuevo pantallas, impresoras nuevas vienen a lo largo; y el software debe ser conformado a su nueva vehículos de ocasión.

En resumen, el producto de software está incrustado en una matriz cultural de las aplicaciones, los usuarios, leyes, y los vehículos de la máquina. Todos ellos cambian continuamente, y sus cambios inexorablemente fuerza de cambio en el producto de software.

Invisibilidad. El software es invisible y unvisualizable. Abstracciones geométricas son herramientas de gran alcance. La planta de un edificio de ayuda tanto arquitecto y el cliente evalúan espacios, flujos de tráfico, y vistas. Las contradicciones se hacen evidentes, omisiones pueden ser atrapados. Dibujos a escala de partes mecánicas y modelos de figuras de palo de moléculas, aunque abstracciones, tienen el mismo propósito. Una realidad geométrica es capturado en un la abstracción geométrica.

La realidad de software no está incrustada inherentemente en el espacio. Por lo tanto, no tiene ninguna listo representación geométrica de la manera que la tierra contiene mapas, chips de silicio tienen diagramas, computadoras tienen esquemas de conectividad. Tan pronto como se intenta diagramar software estructura, encontramos a no constituye una, sino varias, en general dirigida gráficos, superpuestas una sobre otra. Los varios gráficos pueden representar el flujo de control, el flujo de datos, patrones de dependencia, secuencia de tiempo, relaciones nombre en el espacio. Estas son por lo general ni siquiera plana, mucho menos jerárquica. De hecho, una de las formas de

establecer el control conceptual sobre dicha estructura es hacer cumplir el corte de enlace hasta la una o más de los gráficos se convierte jerárquica.2

A pesar del progreso en la restricción y la simplificación de las estructuras de software, que siendo inherentemente unvisualizable, privando así a la mente de algunos de sus más poderosos herramientas conceptuales. Esta falta no sólo impide el proceso de diseño dentro de un mismo sentir, que dificulta gravemente la comunicación entre mentes.

Los avances anteriores Resuelto Dificultades accidental

Si examinamos los tres pasos en la tecnología de software que han sido más fructífera en el pasado, descubrimos que cada atacaron una de las principales dificultades diferentes en la construcción de software, pero que han sido las dificultades, no las esenciales, accidentales. También podemos ver lo natural límites a la extrapolación de cada uno de esos ataques.

Los lenguajes de alto nivel. Seguramente la carrera más poderoso para la productividad de software, fiabilidad y simplicidad ha sido la progresiva utilización de lenguajes de alto nivel para programación. La mayoría de los observadores acreditan que el desarrollo con al menos un factor de cinco en productividad, y con ganancias concomitantes en fiabilidad, simplicidad y comprensibilidad.

¿Qué quiere lograr un lenguaje de alto nivel? Libera un programa de gran parte de su complejidad accidental. Un programa de resumen consiste en construcciones conceptuales: operaciones, tipos de datos, secuencias, y la comunicación. El programa de la máquina hormigón es trate con trozos, registros, condiciones, sucursales, canales, discos, y tal. A la medida en que el lenguaje de alto nivel encarna las construcciones quería en abstracto programa y evita todos los inferiores, se elimina todo un nivel de complejidad que era

Nunca inherente en el programa en absoluto.

El más un lenguaje de alto nivel puede hacer es presentar todas las construcciones del programador imagina en el programa abstracto. Sin duda, el nivel de sofisticación en nuestro pensamiento sobre estructuras de datos, tipos de datos y operaciones está aumentando de manera constante, pero aun en constante tasa decreciente. Y el desarrollo del lenguaje se acerca más y más a la sofisticación de los usuarios.

Por otra parte, en algún momento de la elaboración de un lenguaje de alto nivel se convierte en una carga que aumenta, no disminuye, la tarea intelectual del usuario que rara vez se utiliza lo esotérico construye.

Tiempo compartido. Observadores de crédito La mayor parte de tiempo compartido con una mejora importante en el la productividad de programadores y en la calidad de su producto, aunque no tan grande como la ejercitada por lenguajes de alto nivel.

Tiempo de intercambio de ataques una dificultad claramente diferente. Conservas de tiempo compartido inmediatez, y por lo tanto nos permite mantener una visión general de la complejidad. El lento cambio de la programación por lotes significa que inevitablemente nos olvidamos de los pequeños detalles, si no el muy empuje, de lo que estábamos pensando cuando paramos programación y llamamos para compilación y ejecución. Esta interrupción de la conciencia es costoso en tiempo, ya que debe actualizar. El efecto más grave puede ser la decadencia de comprensión de todo lo que está pasando en un sistema complejo.

2Parnas, DL, "software de diseño para facilitar la extensión y *contracción," IEEE Trans. en SE,* 5, 2(Marzo, 1979), pp. 12-138

Vuelco lenta, como las complejidades de lenguaje de máquina, es un accidente y no un dificultad esencial del proceso de software. Los límites de la contribución de tiempo compartido derivar directamente. El efecto principal es reducir el tiempo de respuesta del sistema. Como se va a cero, en algún momento se pasa el umbral de perceptibilidad humana, alrededor de 100 milisegundos.

Más allá de eso no hay beneficios son de esperar.

Entornos de programación unificado.

Unix y Interlisp, el primero integrado programación entornos próximos al uso generalizado, se perciben haber mejorado la productividad por factores integrales. ¿Por qué?

Ellos atacan a las dificultades accidentales de uso de programas juntos, proporcionando bibliotecas integradas, formatos de archivo unificadas, y pilas y filtros. Como resultado, conceptual estructuras que, en principio, siempre se podría llamar, alimentación, y el uso de los otros puede de hecho fácil hacerlo en la práctica.

Este avance a su vez estimuló el desarrollo de toolbenches enteros, ya que cada nueva herramienta podría aplicarse a todos los programas mediante el uso de los formatos estándar.

Debido a estos éxitos, los entornos son objeto de gran parte de software de hoy investigación en ingeniería. Vamos a mirar a su promesa y limitaciones en la siguiente sección.

Las esperanzas de la Plata

Ahora vamos a considerar los desarrollos técnicos que más a menudo avanzado como potenciales balas de plata. ¿Qué problemas se dirigen? ¿Son los problemas de esencia, o son restos de nuestras dificultades accidentales? ¿Ofrecen avances revolucionarios, o los incrementales?

Ada y otros avances del lenguaje de alto nivel.

Uno de los más promocionado reciente desarrollos es el lenguaje de programación Ada, una de propósito general, lenguaje de alto nivel de la década de 1980. Ada de hecho no sólo refleja mejoras evolutivas en el lenguaje conceptos, pero encarna características para fomentar el diseño moderno y la modularización conceptos. Tal vez la filosofía Ada es más de un anticipo que el lenguaje Ada, por es la filosofía de la modularización, de tipos de datos abstractos, de estructuración jerárquica.

Ada es quizás sobre-rico, el producto natural del proceso por el cual requisitos eran puesto en su diseño. Eso no es fatal, para vocabularios de trabajo de subconjuntos pueden resolver el aprendiendo problema, y los avances de hardware nos darán las MIPS baratas para pagar la la compilación de los costos. Avanzando la estructuración de sistemas de software es de hecho una muy buena utilizar para el aumento de MIPS nuestros dólares comprarán. Sistemas operativos, en voz alta denunciado en el 1960 por sus costos de memoria y de ciclo, han demostrado ser una excelente forma en la que utilizar algunos de los MIPS y bytes de memoria baratas del pasado oleada de hardware.

Sin embargo, Ada no resultará ser la bala de plata que mata el software monstruo de la productividad. Es, después de todo, sólo otro lenguaje de alto nivel, y el mayor pago de tales lenguas vino de la primera transición, por encima de lo accidental complejidades de la máquina en el estado más abstracto de soluciones paso a paso.

Una vez que se han eliminado los accidentes, los restantes son más pequeños, y la recompensa de su eliminación seguramente será menor.

Mi predicción es que una década a partir de ahora, cuando se evalúa la eficacia de Ada, será visto que han hecho una diferencia sustancial, pero no a causa de cualquier idioma en particular función, ni tampoco porque de todos ellos juntos. Tampoco lo hará el nuevo Ada entorno de llegar a ser la causa de las mejoras. La mayor contribución de ADA sea que el cambio a ocasionó programadores de formación en diseño de software moderno técnicas.

Programación orientada a objetos. Muchos estudiantes de la técnica tienen más esperanza para la programación orientada a objetos, que para cualquiera de las otras modas técnicas del día. 3 Yo estoy entre ellos. Marcos Sherman de Dartmouth señala que hay que tener cuidado de distinguir de dos ideas separadas que van con ese nombre: tipos abstractos de datos y tipos jerárquicos, también llamadas clases. El concepto de tipo abstracto de datos es que el tipo de un objeto debe ser definido por un nombre, un conjunto de valores propios, y un conjunto de operaciones apropiadas, en lugar de su estructura de almacenamiento, que debe ser ocultado. Ejemplos de ello son los paquetes de Ada (con privada tipos) o módulos de Modula.

Tipos jerárquicas, como las clases de Simula-67, permiten la definición de generales interfaces que pueden ser refinados aún más al ofrecer tipos subordinados. Los dos conceptos son ortogonales-puede haber jerarquías sin esconder y ocultar sin jerarquías.

Ambos conceptos representan avances reales en la técnica de la construcción de software.

Cada elimina una dificultad más accidental del proceso, permitiendo que el diseñador para expresar la esencia de su diseño sin tener que expresar grandes cantidades de sintáctica material que añadir ningún nuevo contenido de información. Para ambos tipos abstractos y jerárquica tipos, el resultado es para eliminar una orden superior tipo de dificultad accidental y permitir que un expresión de orden superior del diseño.

Sin embargo, estos avances pueden hacer más que para eliminar todo lo accidental dificultades de la expresión del diseño. La complejidad del diseño en sí mismo es esencial; y este tipo de ataques no hacen el cambio alguno en eso. Una ganancia de orden de magnitud se puede hacer por la programación orientada a objeto sólo si la maleza innecesaria de tipo especificación restante hoy en nuestro lenguaje de programación es en sí mismo responsable de nueve décimas de trabajo involucrado en el diseño de un producto de programa. Lo dudo.

Inteligencia artificial. Muchas personas esperan avances en inteligencia artificial para proporcionar el avance revolucionario que dará orden de magnitud ganancias en software productividad y la calidad. 4 Yo no. Para ver por qué, debemos diseccionar lo que se entiende por "Inteligencia artificial" y luego ver cómo se aplica.

Parnas ha aclarado el caos terminológico:

Dos definiciones muy diferentes de AI son de uso común en la actualidad. AI-1: El uso de ordenadores para resolver problemas que antes sólo podían ser resueltos mediante la aplicación humana inteligencia. AI-2: El uso de un conjunto específico de técnicas de programación sabe cómo programación heurística o basada en reglas. En este enfoque expertos humanos son los estudios realizados hasta determinan lo heurísticas o reglas de oro que utilizan en la solución de problemas. . . . El programa está diseñado para resolver un problema de la forma en que los seres humanos parecen resolverlo.

3 Booch, G., "diseño orientado a objetos", *en* Ingeniería de Software *con Ada*. Menlo Park, Calif .: Benjamin Cummings, 1983.

4 Mostow, J., ed., Edición Especial sobre Inteligencia Artificial e Ingeniería del *Software,* IEEE *Trans. en SE,* **11,** 11 (. 11 1985).

La primera definición tiene un significado de deslizamiento. . . . Algo puede ajustarse a la definición de AI-1 hoy, pero, una vez que veamos cómo funciona el programa y entender el problema, lo haremos no pensar en ella como la IA más. . . . Lamentablemente no puedo identificar un cuerpo de la tecnología que es único a este campo. . . . La mayor parte del trabajo es específico del problema, y algunos abstracción o la creatividad es requiere para ver cómo transferirlo. 5

Estoy completamente de acuerdo con esta crítica. Las técnicas utilizadas para el reconocimiento de voz parecen tener poco en común con los utilizados para el reconocimiento de imágenes, y ambos son diferentes de los utilizados en sistemas expertos. Tengo un tiempo difícil ver cómo la imagen reconocimiento, por ejemplo, hará ninguna diferencia apreciable en la práctica de programación.

Lo mismo es tratar de reconocimiento de voz. La cosa difícil sobre la construcción de software es decidir qué decir, no decirlo. Sin facilitación de expresión puede dar más de lo marginal ganancias.

Tecnología de sistemas expertos, AI-2, merece un apartado propio.

Los sistemas expertos. La parte más avanzada de la técnica de inteligencia artificial, y el más ampliamente aplicable, es la tecnología para la construcción de sistemas expertos. Muchos científicos de software están trabajando duro en la aplicación de esta tecnología para el entorno de creación de software. 6

¿Que es el concepto, y cuáles son las perspectivas?

Un sistema experto es un programa que contiene un motor de inferencia generalizada y una reglan base, diseñado para tomar los datos y supuestos y explorar las consecuencias lógicas a través de las inferencias derivables de la base de reglas, conclusiones rendimiento y asesoramiento, y ofreciendo para explicar sus resultados por volver sobre su razonamiento para el usuario. La inferencia motores normalmente pueden hacer frente a los datos y reglas difusas o probabilísticos, además de puramente lógica determinista.

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

- Tecnología de motor de inferencia se desarrolla de una manera independiente de la aplicación, y luego aplicada a muchos usos. Uno puede justificar un esfuerzo mucho mayor en los motores de inferencia. De hecho, de que la tecnología está muy avanzada.
- Las piezas cambiables de los materiales de aplicación-peculiar se codifican en la base de reglas de una manera uniforme, se proporcionan y herramientas para el desarrollo, el cambio, las pruebas y documentación de la base de reglas. Este regulariza gran parte de la complejidad de la aplicación en sí.

Edward Feigenbaum dice que el poder de este tipo de sistemas no viene de en constante mecanismos más elegantes de inferencia, sino desde siempre más ricas bases de conocimiento que reflejan el mundo real con mayor precisión. Creo que el avance más importante que ofrece el tecnología es la separación de la complejidad de la aplicación desde el propio programa.

¿Cómo se puede aplicar a la tarea de software? En muchos sentidos: Interfaz sugiriendo normas, asesoramiento sobre estrategias de ensayo, recordando las frecuencias, pero de tipo, oferta consejos de optimización, etc.

5 Parnas, DL, "aspectos de software de sistemas de defensa estratégica", *Communications of the ACM,* **28,** 12 (diciembre, 1985), pp. 1.326-1.335. También en *American Scientist,* **73,** 5 (septiembre-octubre de 1985), pp. 432-440.

6 Balzer, R., "Una perspectiva de 15 años en la programación automática," en Mostow, op. cit.

Considere la posibilidad de un asesor de pruebas imaginario, por ejemplo. En su forma más rudimentaria, el sistema experto de diagnóstico es muy como lista de verificación de un piloto, que ofrece fundamentalmente sugerencias en cuanto a las posibles causas de dificultades. A medida que se desarrolla la base de reglas, la sugerencias se hacen más específica, teniendo en cuenta más sofisticado de los problemas síntomas reportados. Se puede visualizar un asistente de depuración que ofrece muy generalizada sugerencias al principio, pero a medida que más y más la estructura del sistema está incorporado en la base de reglas, viene más y más en particular en las hipótesis es genera y de las pruebas, recomienda. Tal sistema experto puede apartarse más radical de lo convencional en los que su base de reglas probablemente deben ser modularizados jerárquicamente de la misma manera el producto de software correspondiente es, por lo que a medida que el producto es modificado de forma modular, la base de reglas de diagnóstico se puede modular modificado también.

El trabajo necesario para generar las reglas de diagnóstico es un trabajo que tendrá que hacer de todos modos en la generación del conjunto de casos de prueba para los módulos y para el sistema. Si se hace de una manera adecuada en general, con una estructura uniforme de las normas y una buena inferencia motor disponible, en realidad puede reducir la mano de obra total de la generación de casos de prueba traen en marcha, así como ayudar en el mantenimiento de toda la vida y las pruebas de modificación. De la misma manera, hemos puede postular otros consejeros probablemente muchos de ellos y los probablemente simples para la otras partes de la tarea de construcción de software.

Muchas dificultades se interponen en el camino de la realización precoz de útiles asesores expertos al creador del programa. Una parte crucial de nuestro escenario imaginario es el desarrollo de fácil maneras de conseguir a la especificación de la estructura del programa para la automática o semi-automática generación de reglas de diagnóstico. Aún más difícil e importante es la doble tarea de la adquisición de conocimientos: articular la búsqueda, los expertos independientes de análisis que saben por qué lo hacen las cosas; y el desarrollo de técnicas eficientes para la extracción de lo que saben y destilándolo en bases de reglas. El requisito previo esencial para la construcción de un sistema experto es tener un experto.

La más poderosa contribución de los sistemas expertos que seguramente será poner al servicio del programador sin experiencia la experiencia y la sabiduría acumulada de los mejores programadores. Esto no es una pequeña contribución. La brecha entre el mejor software práctica de la ingeniería y de la práctica media es muy amplia, tal vez más amplio que en cualquier otra disciplina de la ingeniería. Una herramienta que difunde buenas prácticas sería importante.

Programación "automática". Durante casi 40 años, la gente ha estado anticipando y escribiendo sobre "programación automática", la generación de un programa para resolver un problema desde una declaración de las especificaciones del problema. Algunas personas hoy escriben como si se espera que esta tecnología para proporcionar la siguiente avance. 7

Parnas implica que el término se utiliza para el glamour y no contenido semántico, afirmando,

En resumen, la programación automática ha sido siempre un eufemismo para la programación con un lenguaje de alto nivel que era actualmente disponible en el programador. 8

Sostiene, en esencia, que en la mayoría de los casos es el método de solución, no el problema, cuya especificación tiene que ser dada.

7 Mostow, op. cit.

8 Parnas, 1.985, op. cit

Las excepciones se pueden encontrar. La técnica de los generadores de construcción es muy potente, y que se usa rutinariamente para buena ventaja en los programas para la clasificación. Algunos sistemas para la integración de las ecuaciones diferenciales también han permitido la especificación directa del problema.

El sistema evalúa los parámetros, escogió de una biblioteca de métodos de solución, y generado los programas.

- Estas aplicaciones tienen propiedades muy favorables:
- Los problemas se caracterizan fácilmente por relativamente pocos parámetros.
- Hay muchos métodos conocidos de solución para proporcionar una biblioteca de alternativas.
- Análisis extensivo ha llevado a reglas explícitas para la selección de técnicas de solución, teniendo en cuenta parámetros del problema.

Es difícil ver cómo tales técnicas generalizan al resto del mundo de lo ordinario sistema de software, donde los casos con tales propiedades ordenadas son la excepción. Es difícil incluso imaginar cómo podría ocurrir posiblemente este avance en la generalización.

Programación gráfica. Un tema favorito de PH.D. Disertaciones en el software ingeniería es gráfico, o visual, la programación, la aplicación de gráficos de ordenador para diseño de software. 9

A veces la promesa de un enfoque de este tipo se postula desde el analogía con el diseño de chips VLSI, donde los gráficos de computadora juega un papel tan fructífera.

A veces, el enfoque se justifica considerando diagramas de flujo como el programa ideal diseñar medio y proporcionar instalaciones de gran alcance para la construcción de ellos.

Nada siquiera convincente, mucho menos emocionante, sin embargo, ha surgido de tales esfuerzos. Yo estoy convencido de que nada lo hará.

En primer lugar, como ya he dicho en otro lugar, el diagrama de flujo es una abstracción muy pobre de la estructura de software. 10

De hecho, se ve mejor como Burks, von Neumann y

El intento de Goldstine proporcionar un lenguaje de control de alto nivel necesita desesperadamente para su equipo propuesto. En varias páginas, forma lastimosa, conexión en caja a la que el flujo carta ha sido elaborado hoy, se ha demostrado ser esencialmente inútil como una herramienta de diseño programadores dibujar diagramas de flujo después, no antes, escribiendo los programas que describen.

En segundo lugar, las pantallas de hoy son demasiado pequeños, en píxeles, para mostrar el alcance y la resolución de cualquier diagrama detallado grave software. El llamado "metáfora del escritorio" de estación de trabajo de hoy es más bien una metáfora "avión-asiento". Cualquier persona que ha barajado un lapful de papeles, mientras que sentado en un coche entre dos pasajeros corpulentos reconocerá el diferencia se puede ver sólo un par de cosas a la vez. El verdadero escritorio proporciona visión general y el acceso aleatorio a una veintena de páginas. Además, cuando un ataque de creatividad corren fuerte, más de un programador o escritor se ha conocido a abandonar el escritorio el piso más amplio. La tecnología de hardware tendrá que avanzar bastante sustancialmente antes de que el alcance de nuestros telescopios es suficiente para la tarea de diseño de software.

Más fundamentalmente, como he argumentado anteriormente, el software es muy difícil de visualizar.

Ya sea que diagrama de flujo de control, anidación alcance variables, referencias cruzadas variables, los datos soplar, estructuras de datos jerárquicos, o lo que sea, nos sentimos sólo una dimensión de la intrincadamente entrelazados elefante software. Si superponemos todos los diagramas generados por los muchos puntos de vista relevantes, es difícil extraer cualquier visión global. El VLSI analogía es fundamentalmente engañoso-un diseño de chip es un objeto bidimensional en capas cuya geometría refleja su esencia. Un sistema de software no lo es.

9 Raeder, G., "Un estudio de las técnicas de programación gráfica actual," en RB Grafton y T. Ichikawa,

eds., Edición Especial de Programación Visual, *ordenador,* **18, 8 (agosto, 1985), pp. 11-25**

10 Brooks 1,995, op. cit., capítulo 15.

La verificación de programas. Gran parte del esfuerzo en la programación moderna entra en la prueba y reparación de errores. ¿Hay quizá una bala de plata que se encuentran al eliminar los errores en la fuente, en la fase de diseño del sistema? ¿Pueden la productividad y la fiabilidad del producto es mejorar radicalmente siguiendo el profundamente diferente estrategia de diseños que demuestren correcta antes de que el inmenso esfuerzo se vierte en la implementación y prueba de ellos?

No creo que vayamos a encontrar la magia aquí. Programa de verificación es un muy potente concepto, y será muy importante para cosas tales como granos de sistema operativo seguro.

La tecnología no promete, sin embargo, para ahorrar mano de obra. Las verificaciones son tanto trabajo que se han verificado siempre sólo unos pocos programas sustanciales.

Programa de verificación no significa programas de prueba de error. No hay magia aquí, ya sea. Pruebas matemáticas también pueden estar defectuosos. Así que mientras que la verificación podría reducir el programa de pruebas de carga, no puede eliminarlo.

Más en serio, incluso la verificación programa perfecto sólo puede establecer que un programa cumple con su especificación. La parte más difícil de la tarea de software está llegando a una completa y especificación consistente, y gran parte de la esencia de la construcción de un programa es de hecho el la depuración de la especificación.

Entornos y herramientas. ¿Cuánto más la ganancia se puede esperar de The Exploding investiga en mejores entornos de programación? Uno de reacción instintiva es que el problemas grandes Payoff fueron los primeros atacados, y se han resuelto: archivos jerárquico sistemas, formatos de archivo de uniformes de manera que tienen interfaces de programa de uniformes, y generalizada herramientas. Editores inteligentes específicos del idioma son desarrollos aún no se utiliza ampliamente en la práctica, pero el más que prometen es la libertad de los errores sintácticos y errores semánticos simples.

Tal vez la mayor ganancia aún no se ha realizado en el entorno de programación es el uso de los sistemas de bases de datos integradas para realizar un seguimiento de las miríadas de detalles que deben ser recordó con precisión por el programador individual y mantenido al día en un grupo de colaboradores en un solo sistema.

Seguramente este trabajo vale la pena, y seguramente llevará un poco de fruta, tanto en la productividad y fiabilidad. Pero por su propia naturaleza, el regreso a partir de ahora debe ser marginal.

Las estaciones de trabajo. ¿Qué beneficios se esperan para el arte software del seguro y rápido aumento en la capacidad de potencia y la memoria de la estación de trabajo individual? Bien, cuántos MIPS se puede utilizar con provecho? La composición y edición de los programas y documentos es totalmente compatible con velocidades de hoy. Compilar podía soportar un impulso, sino un factor de 10 en velocidad de la máquina seguramente dejar pensar a tiempo la actividad dominante en el día del programador. De hecho, parece ser lo que ahora.

Estaciones de trabajo más potentes que sin duda dan la bienvenida. Mejoras mágicas de ellos no podemos esperar.

Ataques prometedores en la esencia conceptual

A pesar de que ningún avance tecnológico promete dar la suave mágica resultados con los que nos son tan familiares en el área de hardware, no es a la vez una gran cantidad de buen funcionamiento pasando ahora, y la promesa de constante, si el progreso espectacular.

Todos los ataques tecnológicos sobre los accidentes del proceso de software son fundamentalmente limitada por la ecuación de la productividad:

Tiempo de trabajo = Σ (Frecuencia) x (Tiempo)

Si, como creo, los componentes conceptuales de la tarea están tomando la mayor parte del tiempo, entonces ninguna cantidad de la actividad en los componentes de tareas que no son más que la expresión de los conceptos pueden dar grandes ganancias de productividad.

Por lo tanto debemos tener en cuenta los ataques que abordan la esencia del software problema, la formulación de estas estructuras conceptuales complejas. Afortunadamente, algunos de estos son muy prometedores.

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

Cada día esta se vuelve más fácil, ya que cada vez más vendedores ofrecen más y mejor productos de software para una variedad vertiginosa de aplicaciones. Mientras que los ingenieros de software que han trabajado en la metodología de la producción, la revolución del ordenador personal ha creado no uno, sino Muchas, los mercados de masas para el software. Cada quiosco lleva mensual revistas que, ordenados por tipo de máquina, anunciar y revisar decenas de productos en precios desde unos pocos dólares a unos pocos cientos de dólares. Las fuentes más especializadas ofrecen muy productos de gran alcance para la estación de trabajo y otros mercados de Unix. Incluso los peajes y de software ambientes se pueden comprar fuera de la plataforma. He propuesto un mercado a otro lugar paran módulos individuales.

Cualquier producto es más barato comprar que construir de nuevo. Incluso a un costo de $ 100.000, una pieza comprada de software está costando sólo alrededor tanto como un programador años.

Y la entrega es inmediata! Inmediato al menos para los productos que realmente existen, productos cuyo desarrollador puede referirse a la perspectiva de un usuario feliz. Además, tales productos tienden a ser mucho mejor documentados y algo mejor mantenidos que el software de cosecha propia.

El desarrollo del mercado de masas es, creo, la más profunda tendencia a largo plazo en ingeniería de software. El costo del software siempre ha sido el coste de desarrollo, no costo de replicación. Compartiendo ese costo incluso entre algunos usuarios reduce radicalmente el usuario al-costo. Otra forma de verlo es que el uso de n copias de un sistema de software multiplica efectivamente la productividad de sus desarrolladores por n. Eso es una mejora de la productividad de la disciplina y de la nación.

La cuestión clave, por supuesto, es de aplicación. ¿Puedo utilizar un paquete off-the-shelf disponibles para hacer mi tarea? Una cosa sorprendente ha sucedido aquí. Durante los años 1950 y 1960, el estudio después de un estudio mostró que los usuarios no utilizar off-the-shelf paquetes para la nómina, inventario control, cuentas, etc. Los requisitos eran demasiado especializado, el caso por al-cobrar variación demasiado alto. Durante la década de 1980, nos encontramos con este tipo de paquetes de alta demanda y uso generalizado. ¿Lo que ha cambiado?

En realidad, no los paquetes. Ellos pueden ser algo más generalizada y algo más personalizable que en otro tiempo, pero no mucho. En realidad, no las aplicaciones, tampoco. Si nada, el de las necesidades científicas de hoy en día los negocios y son más diversos, más complicado que los de hace 20 años.

El gran cambio ha sido en el ratio de eficiencia de hardware / software. El comprador de un $ 2- millones de máquinas en 1960 sintió que podía pagar $ 250.000 más por una nómina personalizada programa, que se deslizó con facilidad y sin interrupciones en el ordenador hostil sociales ambiente. Los compradores de $ 50.000 máquinas de oficina de hoy no puede permitirse el lujo concebible programas de nómina personalizados; para que adapten sus procedimientos de nómina a los paquetes disponible. Computadoras ahora son tan comunes, si no es sin embargo tan querida, que las adaptaciones se aceptan como algo natural.

Hay excepciones dramáticas a mi argumento de que la generalización del software paquetes ha cambiado poco en los últimos años: las hojas de cálculo electrónicas y base de datos simple sistemas. Estas herramientas de gran alcance, tan obvias en retrospectiva y sin embargo, tan tarde que aparecen, se prestan sí mismos para infinidad de usos, algunos bastante poco ortodoxa. Artículos e incluso libros abundan ahora sobre cómo abordar tareas inesperadas con la hoja de cálculo. Un gran número de aplicaciones eso sería antes se han escrito los programas personalizados en Cobol o Programa Reportar

Generador ahora se realizan de forma rutinaria con estas herramientas.

Muchos usuarios ahora operan su propio día computadoras a día de variada aplicaciones sin tener que escribir un programa. De hecho, muchos de estos usuarios no pueden escribir nuevos programas para sus máquinas, pero son, sin embargo, expertos en resolución de nuevo problemas con ellos.

Creo que la estrategia de la productividad de software más poderosa para el hombre organizaciones a día es dotar a los trabajadores intelectuales en computadoras ingenuo en la línea de fuego con ordenadores personales y buena generalizada escritura, dibujo, archivo y hoja de cálculo programas, y los convierten suelto. La misma estrategia, con una programación sencilla capacidades, también trabajarán para cientos de científicos de laboratorio.

Requisitos refinamiento y prototipado rápido. La parte más difícil de la única construcción de un sistema de software es decidir exactamente qué construir. Ninguna otra parte del trabajo conceptual es difícil, ya que se establecen los requisitos técnicos detallados, incluyendo toda la interfaces para las personas, a las máquinas, y de otros sistemas de software. Ninguna otra parte del trabajo de modo paraliza el sistema resultante si se hace mal. Ninguna otra parte es ir más difícil rectificar más tarde.

Por lo tanto, la función más importante que los constructores de software hacen para sus clientes es la extracción y refinamiento iterativo de los requisitos del producto. Pues la verdad es que el los clientes no saben lo que quieren. Por lo general, no saben qué preguntas debe ser contestadas, y que casi nunca han pensado en el problema en el detalle que debe ser especificado. Incluso el simple respuesta- "Hacer que el nuevo sistema funcione software como nuestro viejo sistema de información-procesamiento manual ", es, de hecho, demasiado simple. Los clientes nunca quieren exactamente eso. Sistemas de software complejos son, por otra parte, las cosas que actúan, que se mueven, que trabajo. La dinámica de la acción son difíciles de imaginar. Así que en la planificación de cualquier software la actividad, es necesario para permitir una extensa iteración entre el cliente y el diseñador como parte de la definición del sistema.

Me gustaría ir un paso más allá y afirmar que es realmente imposible para los clientes, incluso aquellos trabajar con los ingenieros de software, para especificar por completo, precisa y correctamente el exacto requisitos de un producto de software moderno antes de haber construido y trataron algunas versiones del producto que están especificando.

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

Un sistema de software prototipo es una que simula las interfaces importantes y realiza las principales funciones del sistema previsto, mientras que no siendo necesariamente vinculado por las mismas limitaciones de velocidad de hardware, el tamaño, o el costo. Los prototipos suelen realizar la tareas de la línea principal de la aplicación, pero no intentan manejar las excepciones, responden correctamente a las entradas no válidas, abortar limpiamente, etc. El propósito del prototipo es hacer real de la estructura conceptual especifica, de manera que el cliente puede probar para la consistencia y usabilidad

Gran parte de los procedimientos de adquisición de software actual se basa en la suposición de que se puede especificar un sistema satisfactorio de antelación, obtener ofertas para su construcción, lo tienen construcción, e instalarlo. Creo que esta suposición es fundamentalmente equivocada, y que muchos problemas de adquisición de software surgen de esa falacia. Por lo tanto no pueden ser fijos sin revisión fundamental, que prevé el desarrollo iterativo y especificación de los prototipos y productos.

Desarrollo incremental - crecer, no construye, software. Todavía recuerdo la sacudida que sentí en 1958, cuando escuché por primera vez a un amigo hablar de *la construcción de* un programa, en lugar de *escribir* una.

En un instante ampliarse todo mi punto de vista del proceso de software. El cambio metáfora era poderoso y preciso. Hoy entendemos cómo al igual que otro edificio procesa la construcción de software es, y usamos libremente otros elementos de la metáfora, como *especificaciones, montaje de componentes* y *andamios*.

La metáfora edificio ha sobrevivido a su utilidad. Es el momento de cambiar de nuevo. Si, como yo creer, las estructuras conceptuales que construimos hoy son demasiado complicados para ser precisa especificada de antemano, y demasiado complejos para ser construida sin errores, entonces tenemos que tomar una enfoque radicalmente diferente.

Volvamos a la naturaleza y estudiamos la complejidad de los seres vivos, en lugar de los muertos obras del hombre. Aquí nos encontramos con construcciones cuyas complejidades emocionarnos con asombro. El cerebro solo es complicado más allá de la cartografía, poderoso más allá de la imitación, rico en diversidad, auto-protección y auto-renovación. El secreto es que se cultiva, no construido.

Y así debe ser con nuestros sistemas de software. Hace algunos años Harlan Mills propuso que cualquier sistema de software debe ser cultivado por desarrollo incremental. 11 Es decir, el sistema de primero debe ponerse a correr, a pesar de que no hace nada útil, excepto llamada el conjunto adecuado de subprogramas ficticios. Luego, poco a poco se enriquezca con los subprogramas a su vez, está desarrollando en acciones o llamadas a vaciar los talones en el nivel inferior.

He visto los resultados más dramáticos desde que comencé instando a esta técnica en los constructores de proyectos de software de clase de laboratorio de ingeniería. Nada en la última década ha cambiado tan radicalmente mi propia práctica, o su eficacia. Las requiere

enfoque diseño descendente, ya que es una de arriba hacia abajo cada vez mayor del software. Permite fácil retroceso. Se presta a los primeros prototipos. Cada función de agregado y nueva disposición para los datos más complejos o circunstancias crecidas orgánicamente de lo que ya existe

11 Mills, de alta definición, la "programación de arriba hacia abajo en grandes sistemas," *técnicas de depuración en sistemas* grandes, R. Rustin, ed., Englewood Cliffs, NJ, Prentice-Hall, 1971

Los efectos de moral son alarmantes. El entusiasmo salta cuando hay un sistema en funcionamiento, aunque sea simple. Redoblar los esfuerzos cuando la primera imagen de un nuevo software de gráficos sistema aparezca en la pantalla, incluso si es sólo un rectángulo. Uno siempre tiene, en cada etapa en el proceso, un sistema de trabajo. Me parece que los equipos pueden *crecer* mucho más complejo entidades en cuatro meses lo que pueden *construir*.

Los mismos beneficios se pueden realizar en grandes proyectos como en mis pequeños. 12

Los grandes diseñadores. La cuestión central de cómo mejorar los centros de arte de software, ya que siempre, en las personas.

Podemos conseguir buenos diseños siguiendo las buenas prácticas en lugar de los pobres. Bien prácticas de diseño se pueden enseñar. Los programadores se encuentran entre la parte más inteligente de la población, para que puedan aprender las buenas prácticas. Por lo tanto un empuje importante en los Estados Unidos es promulgar buena práctica moderna. Nuevos planes de estudio, la nueva literatura, nuevas organizaciones tales como el Instituto de Ingeniería de Software, todos han llegado a ser con el fin de elevar el nivel de nuestra práctica de mala a buena. Esto es totalmente adecuado.

Sin embargo, no creo que podamos dar el siguiente paso hacia arriba de la misma manera.

Mientras que la diferencia entre pobres diseños conceptuales y buenos puede estar en el solidez del método de diseño, la diferencia entre los buenos diseños y grandes sin duda no. Grandes diseños provienen de grandes diseñadores. La construcción de software es un creativo proceso. Metodología de sonido pueden potenciar y liberar la mente creativa; no puede inflamar o inspirar el esclavo.

Las diferencias no son menores, es más bien como Salieri y Mozart. Estudio tras estudio muestra que los mejores diseñadores producen estructuras que son más rápidos, más pequeños, más simple, limpiador, y producido con menos esfuerzo. Las diferencias entre el grande y el promedio acercarse a un orden de magnitud.

Un poco de retrospectiva muestra que aunque muchos bien, los sistemas de software útiles tienen sido diseñado por los comités y construido por los proyectos de varias partes, los sistemas de software que han emocionado aficionados apasionados son los que son los productos de una o unas pocas diseño mentes, grandes diseñadores. Considere Unix, APL, Pascal, Modula, la interfaz de Smalltalk, incluso Fortran; y el contraste con Cobol, PL / I, Algol, MVS / 370 y MS-DOS (fig. 1)

Sí	Sin
Unix	Cobol
APL	PL / 1
Pascal	Algol
Modula	MVS / 370
Smalltalk	MS-DOS
Fortran	

Fig. 1 productos emocionantes

Por lo tanto, aunque yo apoyo firmemente la transferencia de tecnología y currículo los esfuerzos de desarrollo en marcha, creo que el esfuerzo más importante que podemos montaje es desarrollar formas de hacer crecer grandes diseñadores

12 Boehm, BW, "Un modelo espiral de desarrollo de software y mejora," *Computer,* **20, 5 (mayo,** 1985), pp. 43-57

Ninguna organización software puede ignorar este reto. Los buenos gerentes, aunque escasa que sean, no son más escasos que los buenos diseñadores. Grandes diseñadores y gerentes son a la vez muy raro. La mayoría de las organizaciones gastan considerable esfuerzo en la búsqueda y el cultivo de la perspectivas de gestión; No conozco ninguno que gasta el mismo esfuerzo en la búsqueda y desarrollo los grandes diseñadores a quienes la excelencia técnica de los productos en última instancia, dependerá.

Mi primera propuesta es que cada organización debe determinar el software y proclamar que grandes diseñadores son tan importantes para su éxito como los grandes gerentes son,

y que pueden ser espera que se nutre y recompensado de manera similar. No sólo el sueldo, pero las gratificaciones de

Tamaño reconocimiento oficina, mobiliario, equipo técnico personal, fondos para viajes, el personal apoyo debe ser totalmente equivalente.

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

- Sistemáticamente identificar a los mejores diseñadores tan pronto como sea posible. Lo mejor suele ser el son más experimentados.
- Asignar un mentor profesional a ser responsable para el desarrollo de la perspectiva, y mantenerlo un archivo de carrera cuidado.
- Elaborar y mantener un plan de desarrollo de carrera para cada prospecto, incluyendo cuidado aprendizajes seleccionados con los mejores diseñadores, episodios de la educación formal avanzada, y cursos cortos, todos entremezclados con un diseño individual y el liderazgo técnico asignaciones.
- Proporcionar oportunidades para el crecimiento de los diseñadores para interactuar y estimular entre sí.